OpenQM

Copyright © Zumasys Inc., 2020

All rights reserved. No parts of this work may be reproduced in any form or by any means - graphic, electronic, or mechanical, including photocopying, recording, taping, or information storage and retrieval systems - without the written permission of the publisher.

Products that are referred to in this document may be either trademarks and/or registered trademarks of the respective owners. The publisher and the author make no claim to these trademarks.

While every precaution has been taken in the preparation of this document, the publisher and the author assume no responsibility for errors or omissions, or for damages resulting from the use of information contained in this document or from the use of programs and source code that may accompany it. In no event shall the publisher and the author be liable for any loss of profit or any other commercial damage caused or alleged to have been caused directly or indirectly by this document.

Publisher
Zumasys Inc.
1050 Calle Amanecer
San Clemente
CA 92673
USA

Technical Editor
Martin Phillips

Special thanks to:

Users of the OpenQM product who have contributed topics and suggestions for this manual.

Such information is always very much appreciated so please continue to send comments to support@openqm.com.
### Table of Contents

<table>
<thead>
<tr>
<th>Part</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Part 1</td>
<td>Introduction to the QM Database</td>
<td>7</td>
</tr>
<tr>
<td>Part 2</td>
<td>The Command Environment</td>
<td>63</td>
</tr>
<tr>
<td>Part 3</td>
<td>The QM File System</td>
<td>121</td>
</tr>
<tr>
<td>Part 4</td>
<td>QM Commands</td>
<td>249</td>
</tr>
<tr>
<td>Part 5</td>
<td>Query Processing</td>
<td>639</td>
</tr>
<tr>
<td>Part 6</td>
<td>QMBasic</td>
<td>795</td>
</tr>
<tr>
<td>Part 7</td>
<td>QMClient API</td>
<td>1529</td>
</tr>
<tr>
<td>Part 8</td>
<td>System Administration</td>
<td>1679</td>
</tr>
<tr>
<td>Part 9</td>
<td>Migrating to OpenQM</td>
<td>1727</td>
</tr>
<tr>
<td>Part 10</td>
<td>System Limits</td>
<td>1739</td>
</tr>
<tr>
<td>Part 11</td>
<td>Glossary of Terms</td>
<td>1743</td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>1751</td>
</tr>
</tbody>
</table>
Part 1

Introduction to the QM Database
1 Introduction to the QM Database

OpenQM is a database management system that allows you to develop and run applications for your business or personal use. It includes a wide range of advanced tools and features for complex applications whilst still allowing relatively painless construction of simpler applications.

OpenQM is a member of a family of database products known as multivalue databases, a term that relates to how the system stores your data. If you have experience of products such as Access or Oracle, you may find the architecture of OpenQM to be alien to what you have learnt in the past. It's not wrong; it's just a different way to work. Experience over many years shows that application development for a multivalue database is often many times quicker than for other methodologies, resulting in lower development costs and simpler maintenance.

The name OpenQM is often abbreviated to QM and it is this shorter name (which is the operating system command used to enter the product) that is used in most places within this documentation.

QM has a high degree of compatibility with other multivalue databases systems such as UniVerse, PI/open, Prime Information, Unidata, D3, Reality and many more.

Facilities are provided to create data files, enter, modify and retrieve data, produce reports and, where the data processing operation required cannot be achieved using the supplied tools, to construct powerful programs with the minimum of effort.

The major components of QM are:

The command processor

This includes a comprehensive command set to create, modify, copy and delete files and data stored in them as well as many commands to control processing. Developers and system administrators use the command processor directly. End users of QM applications do not usually have access to the command processor but the application software will make use of many of its features.

The query processor

This provides facilities to produce reports from stored data using an English-like command syntax. The query processor also provides tools to select data which meets particular criteria and to perform operations on this data. End users who need to produce reports may be able to make direct use of the query processor or they may access it via application interfaces.

QMBasic

For those occasions where QM does not provide the desired functions, QMBasic is a very easy to use programming language with powerful screen manipulation and data handling functions. QM extends the language found in other multivalue products to add modern features such as object oriented programming.

OMClient API

The QMClient API is a set of functions that can be used to access QM from applications such as web servers or graphical user interface (GUI) applications. There are variants of this API library for use with many languages, including access from inside QMBasic to allow programs to call subroutines or execute commands on other servers.
Specific topics:

- Application level security
- Security issues
- Printing
- A summary of QM’s printing system.

Document Conventions

The QM documentation uses a simple set of conventions in descriptions of command lines or language elements. For example

```plaintext
DELETE.FILE (DATA | DICT) file.name {FORCE}
```

Items in bold type (DELETE.FILE for example) are keywords that must be entered as they appear in the description except that in most instances they may be in either upper or lower case.

Items in italics (file.name) represent places in commands or language statements where some variable data is required. In this case it is the name of a file.

Except when explicitly stated otherwise, items enclosed in curly brackets (e.g. {FORCE}) are optional parts of a command or statement. The curly brackets are not part of the syntax and should not be typed. The descriptions will explain when the item should be used and what effect it has.

Lists of alternative keywords are shown separated by vertical bars (e.g. {DATA | DICT}).

Items that may be repeated are followed by ellipsis (...). The text explains the rules governing related items.

The mark characters (discussed with the data model) are represented by IM, FM, VM, SM and TM.
1.1 What is a Multivalue Database?

There are many different databases available but they all fall into a small number of basic types. One of these is the relational database such as Oracle or Access. A relational database holds data in the form of tables in just the same way that we could store information as tables written on paper.

Consider an order processing system. We need to hold information about the orders that each customer has placed. Keeping things very simple, at a minimum we might need a table such as that shown below.

<table>
<thead>
<tr>
<th>Order no</th>
<th>Date</th>
<th>Customer</th>
<th>Product</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>12 Jan 05</td>
<td>1728</td>
<td>107</td>
<td>4</td>
</tr>
<tr>
<td>1002</td>
<td>12 Jan 05</td>
<td>3194</td>
<td>318</td>
<td>2</td>
</tr>
<tr>
<td>1003</td>
<td>13 Jan 05</td>
<td>7532</td>
<td>220</td>
<td>1</td>
</tr>
<tr>
<td>1004</td>
<td>13 Jan 05</td>
<td>1263</td>
<td>318</td>
<td>2</td>
</tr>
</tbody>
</table>

In this simple table, each row represents an order and each column holds data associated with that order.

Relational databases are built following a set of rules known as the Laws of Normalisation [E. Codd: "A Relational Model of Data for Large Shared Data Banks", Communications of the ACM, June 1970]. The process of transforming data to fit the rules of a relational database is called normalisation and the steps in this process are referred to as first normal form, second normal form, and so on.

The First Law of Normalisation states that we may not have repeating data. In practical terms this means that we cannot add extra columns to the right of the table to allow a customer to order more than one item at the same time.

<table>
<thead>
<tr>
<th>Order no</th>
<th>Date</th>
<th>Customer</th>
<th>Product</th>
<th>Quantity</th>
<th>Product</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>1001</td>
<td>12 Jan 05</td>
<td>1728</td>
<td>107</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1002</td>
<td>12 Jan 05</td>
<td>3194</td>
<td>318</td>
<td>2</td>
<td>452</td>
<td>3</td>
</tr>
<tr>
<td>1003</td>
<td>13 Jan 05</td>
<td>7532</td>
<td>220</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1004</td>
<td>13 Jan 05</td>
<td>1263</td>
<td>318</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Clearly this restriction is not acceptable in the real world.

There are many reasons why the Laws of Normalisation do not allow this, mostly based on the way in which the data might be stored by the computer system. If we are to observe the First Law of Normalisation, we must reconstruct our data in some way that removes the additional columns. One way would be to split an order that has multiple item across several rows of our table.
Although we can now store as many items in an order as we wish, things have become more complicated. Firstly, the details of a single order are now split across multiple rows of our table. Secondly, we have been forced to add an extra column so that we can know how many lines there are in the order. Also, we have duplicated some information, a step which actually breaks another of the Laws of Normalisation. To avoid this last complication, a typical implementation of this sort of data in a fully normalised system (e.g. Oracle or Access) would break the order into two separate tables, one containing the basic information about the order and the other containing the details of the items ordered.

Multivalue database products avoid this complication by removing the need to adhere to the First Law of Normalisation. We allow a single cell of our table to hold more than one value (hence "multivalue").

If you have spent many years working with fully normalised databases, you are probably shaking your head and saying that we cannot do this. Yes, we can do it; it's just a different way to hold our data.

Think about the advantages: The entire order is all held as a single record; there is no redundant duplication of data; we do not need an item counter.

The end result of this is that our multivalue view of the world is typically much faster than its fully normalised counterpart though there will always be situations where this model is not ideal. In such cases, you can freely revert to using the fully normalised approach. Notice that fully normalised data can be stored in a multivalue database. The opposite tends not to be true.
The time has come to introduce some terminology. A typical application will have many tables, perhaps hundreds or even thousands though the multivalue model usually results in far fewer tables than in other data models. Each table is stored as a file. The rows of our table are known as records and the columns as fields (some users refer to these as attributes). The data stored in a field may be made up of multiple values.

Note how in our multivalued implementation of the above example, the values in the product and quantity columns are related together. For any particular order, the first product number belongs with the first quantity, the second product number belongs with the second quantity and so on. A typical realistic table may have several separate sets of fields that are linked in this way. The relationship between the values in different fields (e.g. product and quantity above) is referred to as an association.

By adopting this data model instead of using additional columns, the data model imposes no limit to the number of items that may be included in an order.

This extended form of the relational database model is at the heart of the QM database. You may also see it referenced as post-relational, nested table or NF2 (non-first normal form). They all mean the same thing.

In a multivalue database, the tables can gain a fourth dimension (subvalues). Continuing with the above example, perhaps we need to record the serial number of each item that we sell. Thus each value line in the table depicted above would have subvalues containing the serial numbers for each item supplied. Order 1002 might become

<table>
<thead>
<tr>
<th>Order no</th>
<th>Date</th>
<th>Customer</th>
<th>Product</th>
<th>Quantity</th>
<th>Serial</th>
</tr>
</thead>
<tbody>
<tr>
<td>1002</td>
<td>12 Jan 05</td>
<td>3194</td>
<td>318</td>
<td>2</td>
<td>21222 21223</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>452</td>
<td>3</td>
<td>41272 41723 41728</td>
</tr>
</tbody>
</table>

The model described above leads to a very flexible design in which a database record may have any number of fields (table columns). The entire record and the constituent fields are of variable length, there being no restriction applied by QM. A record may exist in the database with no data or with many megabytes of data.

Every record in a data file has to have a unique record id or primary key that identifies that record as distinct from all other records in the same table and can be used to retrieve the record. In the above example, the order number would serve this purpose. Although it may be useful to show the record id as a column in the tabular representation of the data, the record id is not part of the actual database record but is instead a handle by which it is accessed. Thus the record above has id 1002 and contains five fields representing the date, customer number, product numbers, quantities and serial numbers.

Internally, a record is stored as a simple character representation of the data. Because the fields are of variable length, it is necessary to mark where one field ends and the next begins. This is done by placing a special marker character called a field mark between the fields. A field may be divided into values by use of value mark characters and values may be further divided by use of subvalue mark characters.

Record 1002 depicted above would be stored as

13527_FM3194_FM318_VM452_FM2_VM3_FM21222_SM21223_VM41272_SM41723_SM41728
where the FM, VM and SM items represent the mark characters. The way to read a record structure such as that shown above is to dismantle it layer by layer. First find the field marks to separate out each field, then look for the values within the fields, and finally the subvalues within the values.

The data model defines two additional mark characters. The text mark is typically used to mark points in text data where newlines should be inserted. This mark character is often inserted by programs manipulating data in memory rather than being stored in the database. The item mark is defined mainly for compatibility with other database systems. Its only reserved use within QM is to separate items in the DATA queue.

When the multivalue data model was originally invented, the upper half of the ASCII character set was not defined and the last five of the unused character values were adopted for the internal representation of the mark characters:

<table>
<thead>
<tr>
<th>Mark Character</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item mark</td>
<td>char(255)</td>
</tr>
<tr>
<td>Field mark</td>
<td>char(254)</td>
</tr>
<tr>
<td>Value mark</td>
<td>char(253)</td>
</tr>
<tr>
<td>Subvalue mark</td>
<td>char(252)</td>
</tr>
<tr>
<td>Text mark</td>
<td>char(251)</td>
</tr>
</tbody>
</table>

The memory representation of a record containing mark characters for use in QMBasic programs is known as a dynamic array and there are many specialised program operations for working on this data.

Fields, values within a field and subvalues within a value are numbered from one upwards. By convention the record key is sometimes referred to as field zero though it is not strictly part of the dynamic array and references to field zero are only recognised by QM in certain contexts.

The History of Multivalue Databases

The original multivalue database is usually attributed to Dick Pick (hence the frequently used term "Pick databases") back in 1968 though their origins can be tracked back further. The current D3 database from Rocket Software is a direct descendant of the original Pick product but there have been many other players along the way, some large, some small. Some of these are significant to the way in which QM works.

The Reality database, previously implemented on McDonnell Douglas systems but now owned by Northgate Information Solutions, closely follows the Pick style of operation. The long defunct Prime Information database from Prime Computer retained the same data model and general principles but made some fairly significant changes to the command and programming languages.

In the mid-1980's the various companies with multivalue products hit a problem. The world was standardising on the Unix operating system but these products did not run on Unix. As a result of this, McDonnell Douglas developed an "open systems" version of Reality (Reality X) and Prime Computer developed the PL/open database. At the same time, two start up companies appeared each with their own Unix based multivalue implementation, VMark (UniVerse) and Unidata (Unidata). These companies set out to capture users from the existing products as well as taking on new users. The history is long and complex but to bring it up to date in one step, D3, mvBase, UniVerse and Unidata are now all owned by Rocket Software.

The UniVerse and Unidata products (usually referred to collectively as U2) follow the Information style of implementation by default but have features that allow them to look more like the Pick style if required.

QM was originally developed in 1993 for use as an embedded database and was released as a product in its own right in 2001. Like the U2 products, it is an Information style database but has options to make it more like Pick for those who need it.
1.2 Installation

If you are going to try things out as you read this manual, the first thing we need to discuss is how to install your own version of QM. This section relates only to the commercial QM product. If you are planning to use the open source version of QM and build your own system, none of what follows in this section applies to you. Instead, you must download and build the system from its source code.

In this section, you will find details on how to install QM on

- Windows
- A USB memory stick under Windows
- Linux, FreeBSD, AIX, Solaris 10, Raspberry Pi
- Mac OS X

See Multi-Tenanting for details of how to install multiple separate versions of QM on the same server.

Although QM can be supplied on CD, users normally download the software from the OpenQM website, www.openqm.com, which ensures that you have the latest version of this rapidly developing product.

On most platforms, you can also use QM in its single user "Personal Version" mode. This is exactly the same as the commercial product but is restricted for use in non-commercial activities, typically as a learning environment, and has a low limit on the size of database file that it will support. The Personal Version comes with no support beyond any help necessary to get it installed.

You will probably not want to install every revision that is released. The product web site includes a "What's new in recent releases" page that can be used to help decide when an upgrade is desirable.

To download the software, follow the link to the download page and select the appropriate version for your platform. Right click on the Download link and select Save Target As to copy the install file to your system. If you need to move the file from the system on which it is downloaded to a different system for installation, be sure to use a binary mode copy tool.

If you have a commercial licence or are using an evaluation copy of QM, you can also download the AccuTerm terminal emulator by following the link from the QM download page. The activation code is included with your QM licence.

The installation process for QM is exactly the same for a new installation and for an upgrade. The following sections describe the process for each platform.

Installation on Windows

You must have administrator rights on the PC to install QM as it updates restricted system files. The self-extracting install file has a name of the form qm_4-0-0.exe (32-bit) or qm64_4-0-0.exe (64-bit), where the numeric components identify the release. Execute this file. The first screen confirms that you are about to install QM. Click on the Next button to continue.

The install process now displays the software licence. Tick the box to say that you accept the terms of this licence and click on the Next button.

QM can be installed in any convenient location. The default is C:\QMSYS but this can be changed. An upgrade installation will offer the directory used for the previous installation as the default.
Having selected the installation directory, you will be asked to specify the program group folder name in the Start menu. This defaults to QM and is probably best left unchanged.

You will be asked to choose between the 8-bit (ASCII) and Extended Character Set (Unicode) versions. This will depend on what languages you need to support.

The final step before installation commences is to select the components to be installed. The components offered are:

- **QM Database**: The QM database itself.
- **QM Help**: This document as a Windows help (.chm) file.
- **QMTerm**: A simple terminal emulator.
- **QM Online Documentation**: Adobe Acrobat style pdf documentation.
- **QMClient**: The client API.

After the main installation has been performed, the install process displays a screen in which the authorisation data can be entered as discussed below.

If this is an upgrade installation, you will be asked if the VOC file should be updated in all accounts. Although this is probably a good idea, users who have access to the QM command prompt will be asked about upgrading when they enter QM if it is left until later.

Finally, the installer offers to show the readme file.

If not already present, the installation will add the bin subdirectory of the QMSYS account location to the Windows PATH environment variable.

After installation is complete, you may need to make changes to configuration parameters. Although the default values work well for most systems, installations with many users or a large number of database tables may require changes. You can always make further changes later.

The self-extracting archive file of the standard install includes the user documentation as a set of pdf files and a compiled HTML help file for use on the QM server or on other Windows clients. Individual pdf manuals and a zip file containing a browser based help package are also available on the download page.

The performance of disk i/o intensive applications on Windows systems is massively affected by the settings of the Windows memory usage performance options. See the QM KnowledgeBase on the openqm.com web site for more details.

### Installation on a USB memory stick under Windows

This mode of installation allows you to carry a complete Windows based QM system on a USB memory stick and use it on any compatible PC without installing any software on the PC itself. The stick must comply with the USB 2 standards or later. Older USB 1 sticks cannot be used.

The installer will attempt to create or update a file named memstick in the top level directory of the USB device. This file contains the unique id codes for that stick and is only used during licence application, however, you should not amend this file as this may cause QM to fail at a later upgrade. If the installer is unable to create this file for any reason, the process can be done manually by running the USBCONFIG tool that can be downloaded from the OpenQM website.

To install QM, run the standard Windows installation program as described above, ensuring that the "USB memory device" check box is ticked and the pathname of the target device is correctly entered.
on the destination directory screen. The remainder of the installation process is as above. The drive letter assigned to the USB device may change each time it is inserted. This will not affect use of QM.

To use QM from a USB memory device:
1. Open a Command Prompt window.
2. Use CD to make the account directory on the USB device the current directory.
3. Type "\qmsys\bin\qm" to enter QM where the actual pathname will depend on where you installed the software.

Although the USB installation of QM is intended for single user use via the QMConsole interface activated by the process described above, it is possible to run network sessions with a USB version of QM. Because the USB installation is all about not needing to install anything on the host PC, this cannot use the QMSvc network service. Instead, a USB installation of QM includes the QMUSBSrvr network management program to allow telnet and QMClient connections. To start this, open a Command Prompt window and execute the \qmsys\bin\qmusbsrvr program from the USB device. Network connections run as the user currently logged in on the Windows system but, if QM's security features are enabled, the @LOGNAME variable will contain the user name supplied when entering QM.

When installing on a USB device, the QMClient.dll file is installed in the bin subdirectory of the QMSYS account instead of in the SYSTEM32 subdirectory of Windows on the hard drive. To use QMClient on a USB installation, it will be necessary either to use full pathnames for QM executable items of to ensure that the PATH environment variable includes the QMSYS\bin directory. Typically, this can be achieved by a line such as

```
PATH=\qmsys\bin;%PATH%
```

in the .bat file that is used to start QM on the USB device.

**Installation on Linux, FreeBSD, AIX and Solaris**

The installation is controlled by a shell script that has an embedded tar archive.

You must have superuser access rights to install QM as it updates restricted system files. The self-extracting install file has a name of the form:

- Linux: qm_4-0-0
- FreeBSD: qmf_4-0-0
- AIX: qmr_4-0-0
- Solaris: qms_4-0-0
- Raspberry Pi: qmp_4-0-0

where the numeric components identify the release. Where a 64-bit version is available, the leading part of the 64-bit installation file includes "64". Ensure that you have execute permission for this file and then execute it.

The installer confirms that you are about to install QM. Note that any existing installation of QM must have been shut down before installation of a new version.

The compressed install file is unpacked and the software licence is displayed. You must confirm that you agree with this licence to continue.

QM can be installed in any convenient location. The default is /usr/qmsys but this can be changed in response to a prompt from the installer. An alternative default location can be set using the optional QMSYS environment variable.

After the main installation has been performed, the install process displays a screen in which the authorisation data can be entered as discussed below. If this screen appears incorrectly formatted,
check that the operating system TERM variable references the correct terminal type and repeat the install.

If this is an upgrade installation, you will be asked if the VOC file should be updated in all accounts. Although this is probably a good idea, users will be asked about upgrading when they enter QM if it is left until later.

The installation process does not add QM to the operating system PATH environment variable. Depending on how you plan to operate your system it may be worth adding the bin subdirectory of the QMSYS account to the PATH variable.

The self-extracting archive file of the standard install does not include the user documentation. This can be downloaded separately from the openqm.com web site as individual pdf manuals, a zip file of all the manuals, a compiled HTML help file for use on Windows clients, or a zip file containing a browser based help package for use on all platforms.

The default password encryption mode of Solaris is weak and not accepted by network connections to QM. The encryption mode can be changed by editing the /etc/security/policy.conf file to uncomment the line that reads

```
CRYPT_ALGORITHMS_DEPRICATE=__unix__
```

and change the CRYPT_DEFAULT line to be

```
CRYPT_DEFAULT=2a
```

After making this change, any user who will directly access QM over a network must reset their password.

Some Unix systems (e.g. AIX) have a kernel configuration parameter that sets the limit on the number of files that may be opened simultaneously by one process. The name of this parameter differs across operating systems but is often NOFILES. QM will automatically handle reaching this limit but there will be a performance degradation caused by needing to close and reopen files to stay within the limit. There may also be a system wide file limit kernel parameter, often called NFILE which also needs to be set carefully.

**Installation on an Intel Mac**

The Mac install is performed using a variant of the Linux install process described above.

The self-extracting install file has a name of the form qmmi_4-0-0, where the numeric components identify the release.

Open a terminal window and set the terminal type to vt102.

Execute the downloaded file using sudo:

```
sudo bash qmmi_4-0-0
```

The installer confirms that you are about to install QM. Note that any existing installation of QM must have been shut down before installation of a new version.

The compressed install file is unpacked and the software licence is displayed. You must confirm that you agree with this licence to continue.
QM can be installed in any convenient location. The default is /var/qmsys as the Mac does not allow installation in /usr/qmsys. The installer will prompt for the pathname to be used. An alternative default can be set using the QMSYS environment variable.

After the main installation has been performed, the install process displays a screen in which the authorisation data can be entered as discussed below.

If this is an upgrade installation, you will be asked if the VOC file should be updated in all accounts. Although this is probably a good idea, users will be asked about upgrading when they enter QM if it is left until later.

From OS X 10.6, the installation process adds the pathname of the QM binaries to the /etc/paths.d directory which is used to construct the PATH environment variable on login. This will take effect from the next login. On older versions of the operating system, the installer does not add QM to the operating system PATH environment variable. Depending on how you plan to operate your system it may be worth adding the bin subdirectory of the QMSYS account to the PATH variable in a profile script.

The self-extracting archive file of the standard install does not include the user documentation. This must be downloaded separately from the web site as individual pdf manuals, a zip file of all the manuals, a compiled HTML help file for use on Windows clients or a zip file containing a browser based help package for use on all platforms.

**Licence Authorisation**

QM will request licence authorisation data entry as part of the installation process described above. A new licence can also be applied at any time by use of the UPDATE.LICENCE command in the QMSYS account or from the operating system command prompt by executing QM with the -L option (case insensitive).

![Licence Details](image)

You need to enter the details in the boxes surrounded by square brackets as given on your licence paperwork. If you have been supplied with a ten digit licence number, enter the following details:
Licence number  The unique ten digit number identifying this licence.
Max users  The maximum number of concurrent processes including Windows GUI processes such as QMClient.
Expiry date  The last date on which this licence is valid. Leave blank for a permanent licence.
Authorisation code  A case insensitive sequence required to validate your licence details.
Security number  A number required to further validate your licence details.
Site text  This must be entered exactly as on your licence document.

Alternatively, if you have been supplied with a profile licence code (two groups of four letters), ensure that the system being licensed is connected to the internet and enter this code in the licence number field. Your system will connect to the OpenQM licensing server to fetch the licence details. You may be asked to enter the user limit or the site text.

The system id is used to tie a licence to a specific machine (or USB memory device for a Windows USB installation). The normal licensing procedure starts with a short term licence that will install on any system. During the life of this licence, you should supply the system id to your QM reseller who will then send you the final permanent licence.

If you subsequently move the QM software to a new system, you will need to arrange with your dealer to receive a new licence. There will normally be no charge for this so long as you undertake to remove the old installation. Your licence document will also show how to obtain a short term emergency licence if you need to move your QM installation to a new system at a time when your reseller is unavailable.

When installing a new release of QM over an existing version, the previous licence details are displayed as the defaults. To preserve these either press the return key in each field in turn or use ctrl-X to exit from the screen.

When updating the licence on a system that uses data encryption, the install process will ask for entry of the master key if the licence number or system id code has changed.

**Post-Installation Options**

On completion of a new installation (not an upgrade), the installer will ask whether QM’s internal security system is to be enabled. For more information on this topic, see Application Level Security. The setting of this option can be amended later using the SECURITY command if necessary.

**Setting Configuration Parameters**

After a new installation, you may need to set the value of some configuration parameters. In particular, the values of NUMFILES (the maximum number of files that can be open at the application level simultaneously) and NUMLOCKS (the maximum number of concurrent record locks) need to be appropriate to your use of the system.
Many operating systems have a limit on the number of files that can be open simultaneously either per-process or system wide. It may be necessary to increase this to match the setting of the NUMFILES parameter. Note that opening a dynamic hashed file in QM requires two operating system files to be opened. Each alternate key index adds a further operating system file.

Upgrades

A commercial QM licence has an "upgrade entitlement period" during which you are free to download and install new versions. Prior to 1 January 2015, this period was initially either one or ten years but could be extended. From 1 January 2015, the upgrade period is linked to the maintenance agreement expiry date. In either case, after the upgrade period has expired, only versions of QM released before the expiry date may still be installed.

To install an upgrade:
1. Download the software from the OpenQM web site.
2. All QM users must log off.
3. On Windows, use the QM Network Control tool from the QM program group to shutdown the QMSvc service. On other platforms, use qm -stop to shutdown QM.
4. Follow the installation process described above.

Several files within the QMSYS account will be updated but all user accounts will remain unchanged except for optional updating of the VOC file for any new items. Note that the NEWVOC file which contains the items to go into the VOC of a new account is completely overwritten by an upgrade. Where there are site specific items that should be added to the VOC of all new accounts, it is recommended that a file named NEWVOC.MODS is created in the QMSYS account. The installer and the CREATE.ACCOUNT and UPDATE.ACCOUNT commands look for this file and, if found, copy its content into the account VOC.

Licence Updates

The upgrade entitlement period is encoded into the QM licence authorisation code and will change when the maintenance agreement is renewed. Although reaching the upgrade entitlement period expiry date will stop QM running, it will not be possible to install a new version of QM that was released after the expiry date. The revised licence will be supplied on renewal and should be kept safely if not applied immediately.

For QM systems with internet access, the GET.LICENCE.UPDATE (or GET.LICENSE.UPDATE) command in the QMSYS account will connect to the QM licensing server to check for and apply any updated licence. No application data is transmitted by use of this command.

It is also possible to automate this process. If the AUTOLIC configuration parameter is set to a non-zero value, the QM system will automatically poll for a licence update shortly after starting the QMSvc service or qmlnxed process and then at daily intervals.

Compatibility with Other Environments

The various multivalue database products implement some features differently. This results in syntactic or semantic differences in some commands or programming statements. If you are migrating an application to QM from another multivalue product, there are facilities to give closer compatibility without needing to make extensive modifications to the application.
The **OPTION** command can be used to enable features that are mostly concerned with the command environment. This command is typically embedded in the LOGIN paragraph that runs automatically when a user enters the system.

The QMBasic **$MODE** directive enables features that affect programming language syntax or semantics. Although developers could put this directive into every program, it is usually simpler to create a record named $BASIC.OPTIONS in the program file to apply mode settings (and other features) to every program in that file. Alternatively, this record can be put in the VOC file from where it will affect all programs in files that do not have their own $BASIC.OPTIONS record. For more details of this record, see the **BASIC** command.

It is recommended that after migration to QM, developers should spend some time reading the documentation to discover features of QM that were not in the previous environment so that they can take advantage of these.
Licensing Overview

Use of the QM multivalue database software is controlled by licence from Zumasys. This section provides an overview of the licensing policy.

What does the licence control?

A QM licence is issued to a named company or individual and is normally for use on a specific system identified by a system id code. The licence sets the maximum number of simultaneous users of the software, the platform (Windows, Linux, etc), the site text that will be displayed on entry to QM, and the availability of optional add-on components. There may also be an expiry date. Use within cloned virtual machine environments requires that each clone is separately licensed.

Each user of a QM system runs as a separate operating system process which falls into one of two broad classifications, interactive or phantom. Interactive processes are users connecting to a QM server via a terminal emulator, directly from the operating system command prompt (a console user), via the QMClient API or other route where the client side is controlling the actions of the server process. Phantom processes are background activity that has no direct interaction with the user. There is a special case of an interactive phantom (iphantom) which is a phantom process that has accepted an incoming network connection, effectively becoming an interactive process.

The user limit set by the licence determines the maximum number of interactive processes, including interactive phantoms. Attempting to start a new interactive process when this limit has been reached will fail.

There is also a limit on the number of simultaneous phantom processes. This limit is normally eight plus a further one for each two licensed users after the first four. Some licences may have different rules for determination of the phantom limit. Attempting to start a new phantom process when the limit has been reached will check whether there are spare interactive users available and, if so, borrow one of these.

QMNet

The QMNet subsystem that allows access to data on remote QM servers is built on an extension to the QMClient API. Opening a connection creates a QM process on the remote system that handles all QMNet activity for the same QM process on the local system. This consumes one licensed user on the remote system in addition to the licence used by the connecting process on the local system. This process terminates when the last QMNet file is closed although the NETDELAY configuration parameter can be used to keep the connection open briefly. Use of QMNet has no additional impact on licensing on the local system.

Web Services

Each child processes created by the WEBSVC command to process incoming web requests consume a licensed user, releasing it on termination.

Device Licensing

The device licensing option allows multiple interactive connections from the same client device to share a licence. It has no effect on phantom processes. Device licensing is typically used to allow both a terminal emulator connection and a QMClient connection from the same PC. Device licensing is available in sizes of two, four or eight sessions per user. In the two session per user mode, for
example, the first two users connecting from the same client device consume only a single licensed user. A third connection from the same device will consume a second licensed user. The system maintains these counters on a per-device basis such that if any one of these three users disconnects, the remaining two processes share the one licensed user.

Device licensing requires support in the client device. It is available with console users, AccuTerm, Winnix, QMClient and the largely obsolete QMTerm. Other terminal emulators will not take part in the device licensing system. Also, the licence terms state that device licensing may not be used with an intermediate layer that effectively makes multiple client devices appear to be just one device.

**Connection Pooling**

The **connection pooling** system allows a QMClient or phantom process to go into an idle state, waiting for new work to arrive. Except for non-interactive phantoms, the process continues to consume a licensed user whilst in the idle state. The pool timeout can be used to terminate the idle process if no new work arrives in a set time.

**See also:**
[Tutorial: Installation](#), QMNet, [WEBSVC](#), Connection pooling
Multi-Tenanting

QM supports the option to install multiple separate copies that each run in an isolated environment on the same server. This concept is referred to as **multi-tenanting**.

Multi-Tenanting on Linux, FreeBSD, AIX or Solaris

On Linux systems, this mode of install does not necessarily require root access though a non-root installation has weaker security. Non-root mode installations are referred to as "user mode".

A multi-tenant installation is initiated by using the -u command line option with the install script.

Each installation is identified by a numeric value automatically formed from the device and inode values of a hidden empty .qmid file. In the unlikely event of QM failing to start because two user mode installations have the same qmid value, use the Linux/Unix touch command to create a new empty file, delete the existing .qmid file and rename the new file to replace it.

Multi-Tenanting on Windows

Windows installation requires administrator rights to allow the QMSvc service to be installed. A multi-tenanted Windows system is installed by use of a command line option to the installer program /Tenant=n where n is an integer value in the range 1 to 65535 identifying the specific tenant installation.

The name of the QMSvc service is modified to become QMSvc-n. and it is this modified name that will appear in the Windows services administration screen.

Note that the Windows installer offers the location of the most recent installation as its default QMSYS pathname.

The QM Network Server Control tool has a browsable selection for the QMSYS pathname of the instance of QM to be monitored or updated. This defaults to the instance from which the QMNetworkCtrl executable is run.

Licensing and Configuration

Each installation must be covered by a separate licence.

An application can access the qmid tenant number by using the QMBasic **SYSTEM()** function with key value 1022. Zero indicates a standard install, non-zero is the qmid value referenced above.

There are some important rules to observe when using a multi-tenant installation. Firstly, the configuration parameters must be set such that each installation is listening on different port numbers for incoming network connections (see the PORT, CLIPORT and REPLPORT configuration parameters). The installer will set PORT and CLIPORT to zero for a new installation, effectively disabling network connections until valid port numbers are set. Secondly, it is essential that no files are simultaneously open to more than one installed instance of QM unless this is via QMNet. For directory files, failure to observe this rule may result in data integrity problems. For hashed files, the file will almost certainly become corrupted and unusable.

It is also essential that the qm command is executed from the bin subdirectory of the QMSYS account in the correct instance of QM. Users should take care to ensure that the search path list in the PATH environment variable does not cause the incorrect program to be run. Similarly, if the
optional QMSYS environment variable is defined, it is essential that it points to the correct installation.

System administrators should set operating system permissions to ensure that a user cannot see other instances of QM on the same server.

Because a user connecting to a multi-tenant installation directly over a network is routed to different installations based on the port number, it is necessary to ensure that a user can only connect to instances of QM for which they should have access. The easiest way to achieve this is to use QM’s user security features to restrict login based on user name. For users entering a Linux system via the shell, operating system permissions should provide adequate enforcement of access restrictions.

When running a Linux system in user mode, user authentication is handled differently depending on whether the qmlnxd daemon that manages network connections is run as the root user or as a non-root user.

- If qmlnxd is running as root, the user name and password entered when starting a QM process must be known to the underlying operating system and the QM process will run as this user.

- If qmlnxd is running as a non-root user and QM’s user security features are enabled, the user name and password must have been defined using the CREATE.USER or ADMIN.USER commands. The QM process will run as the same user as the qmlnxd process but the @LOGNAME variable will contain the user name entered on connection.

- If qmlnxd is running as a non-root user and QM’s user security features are not enabled, no prompt for the user authentication details will appear. The QM process will run as the same user as the qmlnxd process and the @LOGNAME variable will contain the qmlnxd user name. On a QMClient connection, the user name and password supplied in the QMConnect() function will be ignored.

User authentication can be disabled entirely by use of the SECURITY command but this is not recommended.
1.3 **Startup and Shutdown of QM**

QM maintains some persistent data in shared memory that is accessed by all QM users. This includes the locking tables, user tables, configuration data and other information that must be visible to all QM user processes.

**Windows Systems**

On a Windows system except when using a USB installation of QM, the shared memory is created by the QMSvc service. This service must be running for QM to be usable.

USB installations of QM use the QMUbsSrvr program in place of QMSvc for network access. QMConsole sessions on these systems do not require that this program is running.

QMSvc can be started and stopped from the QM Network Control program group item. In addition, QMSvc can be started, stopped or restarted from the Windows Control Panel Services screen or by use of the commands shown below from a Command Prompt window when logged in to Windows with administrator rights.

```
qmsvc -start
qmsvc -stop
qmsvc -restart
```

This assumes that the bin subdirectory of the QMSYS account is in the program search path. If not, the full pathname of the qmsvc executable must be used. These three commands normally pause briefly to allow time for the status message to be read. This pause can be eliminated, perhaps for use in a script, by an additional command line option:

```
qmsvc -start -nowait
```

Shutting down the QMSvc service will automatically logout all QM users.

**Other Platforms**

On other platforms, the QM shared memory must be explicitly loaded before users can enter QM. It may be manually discarded if required.

On some platforms, the installation process will add system startup and shutdown scripts to start QM when the system is booted and to take it down gracefully when the system is shutdown. On other platforms, it may be necessary to do this manually, if desired. QM may be started, stopped or restarted at any time by typing the commands shown below from the operating system shell.

```
qm -start
qm -stop
qm -restart
```

If you are not logged in as root, you may need to prefix these commands with sudo.

**Executing a Startup Script**
Sometimes it is useful to execute a paragraph or other command script when QM starts. This can be achieved using the STARTUP configuration parameter to specify the command to be executed. Multiple STARTUP parameters may be present, each command running as a separate QM process.

On Windows, the command is run when QMSvc starts. On other platforms, the command is run when the qm -start command is used.

The STARTUP parameter has two formats:

**STARTUP=command**

This command will be run as the SYSTEM (Windows) or root (Linux) user in the QMSYS account and would typically be the name of a VOC paragraph. The command may not include double quotes.

Because the SYSTEM and root users have privilege levels that may be higher than is appropriate for the tasks performed by processes initiated by the STARTUP configuration parameter, these processes can use the AUTHENTICATE command to authenticate user credentials and reduce the privilege level to that of some other user known to the operating system.

Alternatively, the parameter may include an account name, user name and password:

**STARTUP=command; account username password**

This command will be run using the specified user name and account. Again, the command may not include double quotes. Because the configuration parameter file is readable by all users, it is useful to encrypt the password using the AUTHKEY command (described with AUTHENTICATE). In this case the encrypted password is prefixed by "ENCR:", for example

**STARTUP=PHANTOM MAILSERVER;MAIL GSMITH ENCR:2CB51EFC17832221**

A startup process will run the MASTER.LOGIN and LOGIN paragraphs in the same way as any other QM session. A QM process can recognise that it is a startup command by testing the value of the @TTY variable which will contain "startup". Phantoms started from the startup process will have this variable set to "phantom".
1.4 Deinstallation

Should it be necessary to uninstall the QM database, the following steps are required:

**Windows**

Execute the QM Uninstaller from the QM program group.

**Other Platforms**

1. Login with superuser rights and type "qm -stop".
2. Run the uninstall program in the qmsys/bin directory.
1.5 Accounts

An account is a place to work, typically corresponding to an application. Physically, an account is represented by an operating system directory in which files private to that application are stored. As well as one or more user accounts representing different applications or versions of a single application, there is always a system account named QMSYS which contains all of the components of the QM database product itself. You should not use this account for your own applications as parts of it are overwritten when a new version is installed. You may also want to restrict access to some files in this account for improved system security.

A new account may be created from any other account by use of the CREATE.ACCOUNT command. Alternatively, use the relevant operating system command to make a new directory in a suitable position and invoke QM in that directory. You will be prompted to confirm that you wish to make this directory into a new account.

Whichever method you use, QM will create a VOC file in this directory and it is then ready for use. Other system files may be created subsequently by some commands.

QM maintains a register of account names and their corresponding operating system pathnames in a file named ACCOUNTS in the QMSYS account. This file is visible from all accounts on the system but, because ACCOUNTS is the sort of name that might well be used as an application file, the alternative name QM.ACCOUNTS is used. Account names are mapped to uppercase in QM. They must start with a letter or digit, may not contain spaces and are limited to 32 characters. On ECS mode systems, account names must contain only characters from the 8-bit character set. The format of records in the accounts register is:

- **Id** Account name
- **F1** Account pathname
- **F2** Account description
- **F3** Account group name (allows grouping of accounts for some operations). This may be multivalued to allow an account to be in multiple groups.
- **F4-9** Reserved for future use
- **F10-19** Reserved for user use. These fields are not accessed by QM.

The standard files present in an account are shown below.

- **VOC** The vocabulary, a file that controls all aspects of command processing within QM.
- **BP** Application programs are written using the QM Basic programming language. The BP file (Basic Programs) is the default place to keep application programs. This file must be created when first needed and is usually a directory file. The compiler output is placed in a file of the same name as the source file but with a suffix of .OUT added (e.g. BP.OUT). The output file is created automatically when first required and must be a directory file.
- **PVOC** The optional private vocabulary file. This is a multifile with subfiles corresponding to the uppercase form of the user’s login name.
- **$ACC** This is the account directory viewed as a QM directory file.
- **$COMO** QM provides a facility to record output that is displayed at the user’s screen in a file. This file is known as a como (command output) file for compatibility with other systems. The $COMO file is
automatically created as a directory file when the COMO ON command is first used. The command also specifies the record name to be used to store the output.

This file also contains the log files generated by background (phantom) processes.

$ED Created automatically by the ED editor when first required, this file holds pre-stored edit command sequences.

$FORMS This VOC entry points to a file in the QMSYS account that is shared by all accounts as a repository for Pick style form queue definitions created using the SET.QUEUE command and used by the SP.ASSIGN command.

$HOLD This is a directory file used to receive output sent to a print unit by a program or standard command that has been set into mode 3 (output to hold file).

$SAVEDLISTS This is a directory file used to store saved select lists. See the SAVE.LIST and GET.LIST commands for more information.

$SCREENS This is a dynamic file used to hold screen definitions that are to be shared between accounts. See the description of the SCRIB screen builder for more information.

The $COMO, $HOLD and $SAVEDLISTS files tend to collect redundant data as time goes by and may be cleared using the CLEAN.ACCOUNT command or some other process appropriate to your application.

Other files not directly visible from QM are:

cat A subdirectory under the account holding programs added to the private catalogue using the CATALOGUE verb. Users should not modify this file except by use of the associated QM commands.

The private catalogue can be moved by creating an X-type VOC entry named $PRIVATE.CATALOGUE in which field 2 contains the pathname of the alternative private catalogue directory. This only takes effect when QM is re-entered or on use of the LOGTO command. This feature is particularly useful where two or more accounts are to share a common private catalogue. The US spelling, $PRIVATE.CATALOG, may be used instead. If both are present, the British spelling takes priority.

stacks A subdirectory under the account used to store saved command stacks when a user exits from QM.

The following files are in the QMSYS account only:

ACCOUNTS The register of account names described above. This file is visible from all accounts as QM.ACCOUNTS. Field 1 contains the pathname of the account. Field 2 can be used to store a brief description of the account. Field 3 contains an optional multivalued list of account group names to which this account belongs.

bin A subdirectory, not visible from within QM, containing all the operating system level executable programs that form part of QM.
ERRMSG A file of standard Pick style message texts provided for compatibility with other multivalue products and used by the QMBasic STOP, ABORT and ERRMSG statements for programs compiled with Pick style message processing.

gcat Not directly visible from inside QM, this is the global catalogue directory. This file should only be accessed using the standard catalogue processing commands.

NEWVOC The template vocabulary file from which new accounts are created. This file should not be updated by users as it will be overwritten on upgrading to a new release.

$IPC This file, not visible from inside QM, is used to support inter-process communication and cannot be accessed directly by users.

$LOGINS A register of login names of users who may access QM. This file cannot be accessed by directly users. See Application Level Security for more details.

$MAP This file, visible from all accounts, is the default destination for a map of the system catalogue produced with the MAP command.

$SERVERS This file, not visible from inside QM, holds details of public QMNet server definitions.

SYSCOM The SYSCOM file holds standard definitions for use in QMBasic programs. It also contains versions of the QMClient include record for Visual Basic, C and other languages.

temp This subdirectory holds internal temporary files. All users must have full access to this directory.

terminfo A subdirectory containing definitions of control data for terminal devices.

termino.src The master source from which the terminfo definitions are built.

Accounts that are no longer needed can be deleted using the DELETE ACCOUNT command.
Moving an Account

Moving an account within the same QM environment usually requires nothing more than copying the entire account directory, changing the pathname in the QM.ACCOUNTS file and modifying any VOC records that reference the account pathname such as those describing locally catalogued subroutines for which the FULL_PATH option of the CATALOGUE command was used.

Moving an account to a different server is more complex and requires the steps below.

Note: It is essential that there are no QM sessions accessing a file while it is moved using an operating system level command. Failure to observe this rule may result in file corruption.

Linux to Linux or Windows to Windows

1. Copy the entire account directory to the new server, preferably using the same pathname as the original system.
2. Ensure that the new account location is referenced from the QM.ACCOUNTS file.
3. If the account pathname has changed, any reference to the old pathname in the VOC file must be changed.
4. Identify any files outside of the account directory that are used by the application and copy these, again preserving or modifying the pathname.
5. If the application uses globally catalogued subroutines, these must be recatalogued on the new system. If all application accounts are being moved, the easiest way to do this is to copy the global catalogue directory (the gcat subdirectory of the QMSYS account) from the original system to a temporary location on the new system and then use the *IMPGCAT tool to import the items that are not part of QM itself.
6. Ensure that any external software used by the application is available on the new system.

Windows to Linux

There are two additional problems when moving from Windows to Linux.

- Operating system pathnames are case insensitive in Windows but case sensitive in Linux.
- Directory files use a carriage return / linefeed pair as the newline in Windows and just a linefeed in Linux. The COPY command has a TEXT or BINARY option that may be useful.

The case sensitivity issue applies to all operating system files and directories within the account, including record ids in directory files though movement of these may be helped by use of the TO.UPPERCASE or TO.LOWERCASE options of the COPY command. It can only be resolved by examination of the application to ensure that casing is as expected.

The line terminator issue can be resolved by use of the *FIXDIR tool but it is essential that this is not applied to directories that hold binary data such as QMBasic object code.

Linux to Windows

The same two additional issues apply as for Windows to Linux but the case sensitivity issue is less of a problem moving in this direction. It is important to note that on a Lunix system, a directory file could have two records named, for example, "A" and "a" but on a Windows system these are both the same record and one will overwrite the other.
Copying Items from the QMSYS Account Directory

It is strongly recommended that users do not copy the QMSYS account in its entirety because this may result in system errors, especially if the two systems are running different releases of QM. Instead, selective copying/editing should be used.

$FORMS If the system uses Pick style form queue emulation, the relevant form queue definitions should be copied or recreated.

$LOGINS This file is inaccessible to QM sessions but holds information relating to the features controlled by the ADMIN.USER command and the application level security option. If all users are moving to the new system, simply use an operating system copy command to move the file. If only some users are moving, use ADMIN.USER to recreate the required user definitions.

$SERVERS This file is also inaccessible to QM sessions and holds server definitions for QMNet and the Virtual File System. Either copy it in its entirety with an operating system command or recreate it.

$VAULT If the encryption key vault is copied to the new system, the master key must be re-entered using the RESET.MASTER.KEY command.

ACCOUNTS Records may be copied selectively or the entire file may be moved using an operating system command.

ecs-maps ECS character maps can be copied individually or in their entirety.

gcat The global catalogue must not be copied in its entirety unless both systems are running the same version of QM. The *IMPGCAT tool mentioned above may be useful.

VOC The only item that should be copied is the MASTER.LOGIN paragraph, if present.

qmconfig This file should be examined carefully and items moved/edited as needed. The EDIT.CONFIG command may be useful.

terminfo.mods If used, this file may need to be copied to the new system.

Copying an Entire System to a New Server

As an alternative to the steps described above, moving an entire system to a new server of the same type (Windows/Linux/etc) can be done as follows:

Install QM on the new system. This will require a change to the licence and it may be useful to visit www.openqm.com/emergency.htm to generate a seven day "emergency licence" which will be replaced by a new permanent licence later.

1. Shutdown QM on both the old and new servers.

2. Use an operating system tool to copy all account directories, including QMSYS to the new server, using the same pathnames as on the old server.

3. Reinstall QM on the new server so that any standard parts of QM such as globally catalogued programs that were copied from the old server are replaced with the correct versions for the release of QM being used on the new server. If the old system uses encryption, the installer will request entry of the master key.
4. If the old server was a replication subscriber, the RPLSRVR configuration parameter may need updating.
1.6 Entering QM

The QM database can be accessed in a number of ways. The simplest is use of a console session. This is entry into QM directly from the operating system command prompt on the system on which it is installed. Other methods allow direct connection over a network or via a serial port and are discussed later in this section.

Entering QM from the Operating System Command Prompt

On Windows systems, once QM has been successfully installed, the program group chosen during the install (usually QM) will contain an item titled "QM Console". Clicking on this item will open a console window. You will see a copyright line and a site specific licence line. You will then be asked to enter the name of the account you wish to work in.

On all platforms you can login to the operating system and then type qm at the command prompt (this assumes that the operating system PATH environment variable has been set appropriately). If your current directory when you entered QM was not already a QM account, you will be asked if you wish to make it into one. Creation of new accounts in this way may be barred for specific users by the system administrator. See Application Level Security for more details. When entering QM in this way, the account name is determined by looking for an account register entry with the pathname of the current working directory, giving priority to an account name which is the uppercase form of the directory name.

When using a console session, you can force entry to a specific account using a command line of the form

```
qm -a xxx
```

where xxx is the account name.

On Linux systems, the operating system profile script can be used to take the user directly into QM when they login without them seeing an operating system command prompt.

Entering QM Directly via a Network

TCP/IP network technology assigns each computer on the network a unique address, usually written as four numbers separated by dots (e.g. 193.118.13.14) for an IPV4 address or a series of colon separated hexadecimal values for an IPV6 address. Alternatively, the network name of the server can be used and this will be looked up to translate it to a network address internally. When a connection is made to a network address, the caller also specifies a "port number" which identifies the service to which they wish to connect. If networking is new to you, it may help to consider the concept of network addresses and port numbers as being similar to telephone numbers and extensions.

With its default configuration, QM listens for users entering via a network connection on TCP/IP port 4242. This can be changed to an alternative port or disabled completely by amending the QM configuration parameters. Windows users who do not have any other telnet software running on their system may wish to change this to port 23, the default telnet port. Note that because this is a telnet style connection, data on the network is not encrypted. For best security, Linux users should use an SSH connection to login to the operating system and then enter QM as described above.

You can connect to QM using most terminal emulators. A licence for the AccuTerm emulator is bundled with a commercial QM licence. This emulator includes several features specifically for QM. Although the licence is bundled, you will need to download the latest version of the emulator software from the AccuTerm website.
On USB installations of QM, the installation process installs a server program, QMUSBSrvr, in the bin subdirectory of the account. This must be started manually, though this can be automated via the Startup folder. Due to a published defect in Windows, the server cannot detect a system shutdown and must be closed manually.

On other Windows installations, the QM installation process installs a Windows service (QMSvc) to manage the network. There should be no need to change anything as it will start and stop automatically as required.

On all Windows environments, there is a QM Network Control program in the QM program group that can be used to start and stop the appropriate network server.

On other platforms, the install process will make the necessary changes to the operating system files that control the network. There should be no need for any manual user intervention unless you decide to modify the default settings.

In all cases, a valid user name and password known to the operating system will be required to enter QM.

Port Mapping

Some software originating in other multivalue environments relies on being able to connect via multiple telnet ports, each leading to creation of a process with a fixed user number related to the port number. QM supports this capability via a feature known as port mapping. For more details, see the PORTMAP configuration parameter.

Port mapping is not available on USB installations of QM.

Entering QM Directly via a Serial Port

On Windows, the QMSvc service can monitor one or more serial ports for incoming QM connections. This allows entry from directly connected terminals or via dial-up lines. See the SERIAL configuration parameter of QMSvc for more details.

It is also possible to login a serial port from another QM process using the LOGIN.PORT command. This will skip the user authentication described below as the new process runs with the user name and access rights of the user who established the connection. This style of login can be useful when connecting to automated data collection devices. The LOGIN paragraph would typically be used to enter the application.

Logging In to QM

Users entering QM directly from a network connection or via a serial port must provide a valid user name and password for authentication purposes.

On Windows, the user name must also be known to the operating system. Many users of Windows XP choose to operate their systems with login at the server screen disabled, however, Windows enforces use of a valid user name on network connections, including “loop back” to the host system from a terminal emulator running on the same machine. User names can be set up using the User Administration area of the Windows Control Panel. The QM process will run as the specified user and with that user’s access rights.

When using domain style logins, the format is username@domain or the older domain\username.
USB installations of QM do not have access to a suitable user authentication system so QM provides its own. This can be disabled using the `SECURITY` command if required, leaving the system open for network users to connect with no authentication.

On all other installations, the user name must be known to the underlying operating system. The resultant QM process will run as this user and with the access rights of that user. Use the appropriate operating system administration tools to create and maintain user names.

**Suppressing the Copyright and Licence Lines**

The `-quiet` option to the QM executable suppresses display of the copyright and licence details. This is particularly useful in situations such as scripts using QM as part of a CGI web interface. The `LOGIN.PORT` command mentioned above, implies use of the `-quiet` option so that no data is sent to the port until the application starts execution.

**Device Licensing**

Device licensing is an option that allows multiple connections from a single client to share a QM licence. If your system has this feature enabled, you will see a reference to it in the first few lines of output from the `CONFIG` command. QM may be licensed to share a maximum of 2, 4 or 8 sessions per licensed user.

Device licensing needs support in the client software. This is included in Windows QMConsole sessions, AccuTerm 2k2 version 5.3c or later, Winnix version 3 or later, QMTerm and QMClient. With AccuTerm, device licensing must be enabled via a checkbox in the Connection page of the Tools, Settings menu. Note that the Linux Terminal command does not support device licensing.

When connecting to QM directly via a network (port 4242 as described above), device licensing is totally automatic. For entry from the operating system command prompt, the `-DL` option of the QM command must be used to initiate the negotiation between the client and QM. Alternatively, setting an environment variable named QMDL to a non-zero value will make QM behave as though this option is present. This option is necessary because the negotiation process may cause terminal emulators that do not support device licensing to behave erratically.

On Linux/Unix systems, device licensing for network connections is only available if the qmlnxd daemon is running as root.
1.7 The Login Process

There are two stages to login; user authentication and process initialisation.

User Authentication

On Windows, users connecting to QM via a network must enter a valid Windows user name and password. The new process runs as that user and with the associated access permissions.

QM implements a further layer of security on top of the Windows authentication by maintaining a register of user names allowed to use QM. A user name may be added to this register using the CREATE.USER or ADMIN.USER commands. The register entry determines:

1. whether the user is allowed to use QM at all. This check can be suppressed using the SECURITY command.
2. whether the user is to be granted administrator rights within QM.
3. the name of the account that the user should start in. If no account is specified, a prompt is displayed for the account name.
4. restrictions on which accounts they may access.

If security has been turned off and the user name does not appear in the user register, the user runs without administrator rights and an account name prompt is displayed.

On USB installations of QM, the above mechanism is extended such that QM performs the username and password validation using its own internal user register. The newly created process runs with the Windows user name and access permissions of the user that started the QMUSBSrvr process.

On other platforms, users connecting to QM via a network usually open telnet sessions as normal users and then enter QM, perhaps automatically via their profile script. It is, however, possible to connect directly to QM in which case the security mechanisms described above for Windows apply unless the qmlnxd process is running as a non-root user. In this situation, the user name must have been defined using the CREATE.USER or ADMIN.USER commands.

Process Initialisation

When a user successfully enters an interactive QM session, the following steps occur:

1. For users entering QM directly from a network connection, QM attempts to determine the terminal type by use of telnet negotiation commands. If the emulator in use does not support these, QM looks for an environment variable named TERM and, if this is found, uses it to set the default terminal type. If this also fails, vt100 is used by default.

   For QMConsole users on Windows, the terminal type is set to qmterm. For users entering QM from an operating command prompt on other platforms, QM looks for an environment variable named TERM and, if this is found, uses it to set the default terminal type. If this fails, vt100 is used by default.

   The terminal type can be changed later from within QM using the TERM command.

When using AccuTerm, it is strongly recommended that the terminal types with the -at
suffix (e.g. vt220-at) are used as these enable AccuTerm specific features such as the screen switching required for the full screen mode of the QMBasic debugger.

2. QM then looks for environment variables named LINES and COLUMNS and, if found and valid, uses these to set the initial size of the terminal window. When using a QMConsole session on Windows, the displayed window will be adjusted to be this size. On other connections, it is the user's responsibility to ensure that the terminal emulator screen dimensions match those expected by QM.

3. The system looks in the QMSYS account VOC file to find a paragraph named MASTER.LOGIN and, if this exists, executes it. This paragraph can be used for system wide initialisation such as setting European date format or standard printer associations.

4. The system checks in the user's account VOC file to find an executable (menu, paragraph, sentence, verb) item named LOGIN and, if this exists, executes it. The LOGIN item is typically used to perform account specific initialisation and the enter the application. Note that this happens for all QM processes, including phantoms and QMClient sessions. To exit from the LOGIN paragraph for a phantom process, insert a line

   IF @TTY = 'phantom' THEN STOP

at the relevant point in the paragraph. For a QMClient session, test for 'vbsrvr'. See @TTY for more details.

Some other multivalue products look for an item named the same as the user's login id. This can be emulated on QM by creating a LOGIN item that is simply

1: S
2: <<@LOGNAME>>

5. The break key is enabled. By running the MASTER.LOGIN and LOGIN paragraphs with the break key disabled, the user cannot quit out of any security checking done in these paragraphs. If a LOGIN paragraph is used to start the application, it may be necessary to enable the break key at this stage by including a BREAK ON command.

Step 4 above is also executed when the LOGTO command is used to move to a new account.

User specific process initialisation can be performed by testing the content of the @LOGNAME variable in the MASTER.LOGIN or LOGIN paragraphs. For example,

   IF @LOGNAME = 'ADMINISTRATOR' THEN ADMIN.STARTUP
1.8 **Command Scripts**

The QM VOC file normally contains one or more items that represent scripts of commands to be executed automatically at certain events. Although these are usually paragraphs, all except for the MASTER.LOGIN item may actually be any executable type of VOC record (verbs, menus, Procs, etc). None of these items need exist. They provide the means to perform a fixed sequence of commands at the events described below.

**LOGIN**

The LOGIN paragraph is executed on entry to QM and also when the LOGTO command is used to switch to a new account. The break key is inhibited until first execution of this paragraph has been completed. This paragraph is executed for interactive (terminal) users, phantom processes and QMClient connections. The @TTY variable can be tested to determine the user type and, as in this example, used to control flow through the paragraph. The LOGIN paragraph is typically used to set QM option flags, perform security checks, set up printers, set terminal characteristics and enter the application. The last step ensures that an application user does not see a QM command prompt.

Example

```
PA
DATE.FORMAT ON
IF @TTY IN 'phantom', 'vbsrvr' THEN STOP
PTERM CASE NOINVERT
BELL OFF
OPTION NO.USER.ABORTS
BREAK ON
RUN BP MAIN
```

**ON.LOGTO**

The ON.LOGTO paragraph is executed on use of the LOGTO command before switching to the new account. This paragraph might be used, for example, to clear down application specific data such as named common blocks.

Example

```
PA
DELETE.COMMON ALL
```

**ON.EXIT**

The ON.EXIT paragraph is executed on leaving QM by use of the QUIT command or any of its synonyms. The break key is inhibited during execution of this paragraph. An abort occurring in this paragraph will terminate the QM session immediately.

Example

```
PA
SELECT TEMP WITH UNO = <<@USERNO>>
IF @SELECTED THEN DELETE TEMP NO.QUERY
```

**ON.ABORT**

The primary role of the ON.ABORT paragraph is to prevent the user reaching a command prompt if the application fails. It may be useful to include logging of the cause of the abort.
The ON.ABORT paragraph is executed when QM aborts a program due to an internally detected error, a QMBasic program executes an **ABORT** statement or when the Abort response is chosen after use of the break key. The **@ABORT.CODE** and **@ABORT.MESSAGE** variables may be useful in determining the cause of the error. Aborts occurring in commands started using the QMBasic **EXECUTE** statement with the TRAPPING ABORTS option do not execute the ON.ABORT paragraph.

A further abort occurring whilst executing the ON.ABORT paragraph will cause a message to be displayed without re-entering the paragraph. If execution of the application is to be resumed, the ON.ABORT paragraph should execute a **CLEAR.ABORT** statement to inform the command processor that the abort condition has been handled and normal execution is to resume.

**Example**

```
PA
  RUN BP LOG.ABORT
QUIT
```

**MASTER.LOGIN** (QMSYS account)

This item, if present, must be a paragraph and is executed on initial entry to QM in any account before the LOGIN paragraph but not when the **LOGTO** command is used to switch to a new account. This paragraph is executed with the break key inhibited for terminal users and phantom processes. It is also executed for QMClient connections.

**Example**

```
PA
  DATE.FORMAT ON
  OPTION NO.USER.ABORTS
  OPTION DUMP.ON.ERROR
```
1.9 Connection Pooling

Connection pooling can improve performance of applications such as web servers that continually start short life phantom processes or QMClient sessions. Instead of terminating when the phantom completes its work or a QMClient connection is closed, the process enters an idle state waiting for a new connection to arrive, at which point it continues processing. This removes the need for start-up processing both in QM itself and, more significantly, in the application. For example, files opened in the server process do not need to be reopened and data in common blocks does not need to be reinitialised.

This feature is not currently available in the AIX (RS6000) version of QM. On this platform, the QMConnectPool() function will behave identically to QMConnect() and a pool name associated with a phantom process will be ignored.

Configuring a Connection Pool

A connection pool is defined by use of the POOL configuration parameter. This sets a case insensitive pool name of up to 15 characters and an associated pool size limit and timeout value.

POOL=name,limit,timeout

This parameter may be repeated to define multiple pools.

The limit value sets the maximum number of idle processes that are allowed to be waiting in the pool. If the limit is reached, a process attempting to go into the idle state will be terminated.

The timeout value sets the period in seconds for which an idle process will wait before terminating. A value of zero implies no timeout. A timeout of a few seconds is recommended.

Using Connection Pooling with Phantom Processes

A phantom process can be started as a member of a connection pool by use of the POOL option to the PHANTOM command or the QMBasic PHANTOM statement. If a suitable idle phantom process exists in the pool, it will be woken, otherwise a new process is started. The NEW.PROCESS qualifier to the POOL option can be used to force creation of a new process.

When it completes its work, the phantom process can use the POOL.IDLE statement to join the pool of idle processes.

Using Connection Pooling with QMClient

A client process connects to the server using the QMConnectPool() function which is similar to QMConnect() but takes an additional argument as the pool name.

QMConnectPool(host, port, username, password, account, poolname)

On first use, this function behaves exactly like QMConnect() except that, if the pool name is recognised by the server, the process started by this function becomes a member of that pool.

At the point when the client terminates the connection by using QMPoolIdle(), the server process will enter a waiting state and the client process connection to the server is closed. When a new QMClient connection arrives using QMConnectPool() for the same pool name, user name and account name, the waiting process resumes execution, processing the new connection. Note that the account name must match the account in which the idle process is currently operating which could
be different from the account specified when the session started. If no suitable idle process is found, a new process is started.

If the pool size limit has been reached, `QMPoolIdle()` will terminate the server process in the same way as `QMDisconnect()`. This parameter value does not limit the number of active processes in the pool.

Because the server process will be picked up by a new client that has no knowledge of what has happened before, when a process goes into the idle state, files opened using `QMOpen()` are closed and objects instantiated with `QMCreateObject()` are destroyed.

The `QMDisconnect()` function will force full termination of a server process when closing a connection in the same way as for sessions opened using `QMConnect()`.

From the client side, a connection pooling session looks identical to a non-pooled session. From the server side, the application must be written such that an executed series of commands or subroutines might not relate to the same client session. It may be necessary for the client to call some form of reset function prior to entering the idle state if there is session related data that must not be carried forward.

Although phantoms and QMClient sessions can exist in the same pool, only a process of the appropriate type will be woken from its idle state when new work arrives. It is recommended that different pool names are used for the two process types.

The `SYSTEM(1060)` function returns the pool name for a connection pooling process, a null string for other processes.
1.10 Extended Character Set Support

For many years, computer systems mostly used an 8-bit value (a byte) to represent a character. This allows 256 possible values which is adequate for most Western languages. The way in which these values relate to the characters that they represent is a matter of choice and two main standards exist, ASCII and EBCDIC, though the latter is largely redundant and is not considered further here.

Strictly, the ASCII character set defines only the first half of the available character values though the term is often misused to reference the entire 256 character set, known in QM as the 8-bit character set. Within this, the first 32 characters are defined to be the control characters such as backspace or linefeed. The next 96 characters are the letters, digits, punctuation, etc. The upper half of the 8-bit character set was originally not defined but subsequently became used in a number of different ways to provide accented letters, symbols and various graphical items. Because the actual characters represented by values 128 to 255 vary, it is important to know which character set is in use. In Windows systems, this is determined by the code page setting.

In today's business world, there is a need to be able to represent more than 256 different characters. Various schemes exist to do this but the underlying concept for most of these is the definition of the Unicode character set in which each character from a large range of languages is represented by a unique numeric value, the code point, conventionally written as a hexadecimal value in the form U+1234. Although the full definition of Unicode provides for more, the common usage of this (the Basic Multilingual Plane or BMP), allows 65536 character values which are stored internally as a 16-bit value (two bytes). The characters that lie outside this range are not usually required in business computer systems, however, if they are required, it is usually possible to move the ones that are needed into an area of the BMP that is reserved for application specific use (the Private Use Area, U+E000 to U+F8FF). QM reserves characters U+F880 to U+F8FF for internal use and developers should not map other characters into this area.

The standard version of QM works with 8-bit characters internally and is therefore limited to use of only the characters defined in this set with the appropriate interpretation of the upper half of the range. By default, sorting sequences data based on the character value which may not necessarily match local language conventions, especially regarding the placement of accented letters, and case conversion handles only the upper and lower case letters from the lower half of the character set and hence does not convert accented letters. Most of what follows relates to the Unicode version of QM but see below for details of the 8-bit character map feature.

The extended character set (ECS) version of QM uses 16-bit characters internally and provides support for the Unicode BMP with the exception of moving the five characters that are replaced on multivalue systems by the mark characters (251 to 255 or, in Unicode form, U+00FB to U+00FF) to an alternative location (U+F8FB to U+F8FF) in a part of the BMP that Unicode defines as the Private Use Area (PUA). This is done because many applications use, for example, CHAR(253) to reference a value mark and hence these characters must remain in their traditional locations.

Note that the term Extended Character Set in the context of QM is taken at its literal meaning of extending the character set beyond 8 bits. In other contexts, this term is sometimes used to refer to the upper half of the 8-bit character set.

The sorting sequence and case conversion rules are determined by a character map that can be selected to match local language conventions. Because these maps are set up to allow for the relocation of the five characters displaced by the marks, the only time an application developer needs to be concerned with this relocation is in the unlikely situation of needing to work directly with the code point value. For example, CHAR(252) refers to a subvalue mark. The German ü-umlaut that has been displaced by this character can be programmatically created as ECHAR(63740) or, using hexadecimal notation, ECHAR(0xF8FB). There is also a QMBasic function, SWAPMARKS(), that can be used to interchange the two groups of characters in an application though this should not
usually be required as all relevant external interfaces provide an option to exchange the relocated characters with their Unicode positions for compatibility with other software.

Outside of QM, most data storage and transmission systems are byte orientated and hence Unicode characters must be encoded in some way. QM provides a set of character encodings that can be applied to external interfaces such as directory files, socket connections, etc to transform the internal 16-bit characters to a form that is suitable for external use. All processing inside QM is performed with the 16-bit ECS representation.

The diagram below shows how ECS and encodings work. Everything inside the circle works with data in ECS 16-bit form, though hashed files will compact data to 8-bit form where possible. All the external interfaces shown outside the circle tend to use 8-bit data and require encoding if characters outside the 8-bit set are to be used. As described below, in some cases this encoding is automatic; in others it is up to the application to provide this encoding using conversion codes and other QM functions.

Characters and Bytes

In most cases, an application need not be concerned with the internal representation of characters. Programs that operate on the 8-bit version of QM will continue to operate unchanged on the ECS version. The QMBasic character manipulation functions that work with 8-bit characters in non-ECS systems simply work in exactly the same way with 16-bit characters in ECS systems.

Binary data such as images or data read using OSREAD will be manipulated within an application as a series of 16 bit characters that have values in the range 0 to 255. From the application’s viewpoint, nothing is different. The fact that internally each byte of the original data is stored internally as two bytes is irrelevant. QM refers to data stored in this form as a byte string.

The QMBasic CHAR() function is defined to work only for character values in the range 0 to 255. In QM and many other multivalue systems, only the low order eight bits of values outside this range are used. To allow creation of characters in the full ECS range, the ECHAR() function can be used. On a non-ECS system, this behaves exactly like CHAR().

Special Key Codes for Terminal Input

In a Windows console session, the QMBasic KEYIN() function recognises special keys such as the cursor keys and function keys and for each of these returns a character from the upper half of the 8-bit character set to represent that key. The KEYCODE() function uses the terminfo database to provide similar functionality, extended to include recognition of special characters when using a terminal emulator. In a non-ECS mode system with 8-bit characters, there is no alternative to the character positions used for these special keys clashing with other characters. The larger character
set available in ECS mode allows the special characters to be moved into the BMP Private Use Area in positions that do not clash with other characters.

In order to maintain compatibility with non-ECS applications and also to allow applications compiled on ECS mode systems to be run on non-ECS mode systems, the KEYIN() and KEYCODE() functions retain their non-ECS behaviour but are therefore unable to distinguish between the special keys and the characters that they replace. As an alternative, the KEYINV() and KEYCODEV() functions return a key value (character number) rather than the actual character. The special keys return values which, when considered as code points in the Unicode BMP, lie in the Private Use Area.

These functions are available on non-ECS mode systems but the returned values for the special keys cannot be encoded as a single character whereas on an ECS system they can. Also, they are likely to cause an application to behave incorrectly if one of the data characters replaced by the special keys is entered at the keyboard. It is, therefore, beneficial to modify programs that use KEYCODE() to use KEYCODEV() and the corresponding alternative set of character token names. This should not be a major task. Essentially, if a program contains something like

\[
C = \text{KEYCODE}() \\
N = \text{SEQ}(C)
\]

this becomes

\[
N = \text{KEYCODEV}() \\
C = \text{ECHAR}(N)
\]

and all use of the key value tokens defined in the KEYIN.H include record with names prefixed by K$ change to use prefix KV$. Any use of BINDKEY() in the same programs should also be modified to use the KV$ token names.

The value returned via the STATUS() function after an INPUTFIELD operation terminates on an unrecognised control character is not affected by ECS and always lies in the range 128 to 227. The INPUTFIELDV statement is identical but returns the Unicode code point value for the special keys.

**What Must Be 8-Bit Data?**

The following items must be formed only from the 8-bit character set:

- Configuration parameter data
- Pathnames and hence directory file record ids
- User names and passwords
- Account names
- Shell commands in **SH** or **OS.EXECUTE**
- QMBasic program source text with the exception of character constants.
- QMBasic subroutine call names and common block names
- Object oriented programming property and function names
- Encryption key names
- QMNet server names
- Data records in directory files unless an encoding is used. This includes system files such as $SAVEDLISTS and $COMO.
- Compiler output files

**What Can Be 16-Bit Data?**

- Dictionary item names and content
Alternate key index names
Character strings manipulated by QMBasic programs
String constants in QMBasic programs and dictionary items
Record ids and data in ECS mode hashed files
Encryption key values (though only the bottom 8 bits of each character will be used)

**Character Maps**

QM uses a **character mapping table** to define properties of the 65536 characters that can be represented internally by 16-bit data. For each character, this table identifies the upper and lower case equivalents, the sort weight, and the character attributes such as whether it is a letter. An application can use the `IS.ALNUM()`, `IS.ALPHA()`, `IS.DIGIT()`, `IS.GRAPh()`, `IS.MARK()`, `IS.SPACE()`, `IS.USER.CHAR()` and `IS.WIDE()` functions to test the attributes of a character. The character maps also control the behaviour of the MCL, MCT and MCU conversion codes and the related `UPCASE()` and `DOWNCASE()` functions.

For characters defined as being digits, the mapping table also identifies the decimal value of the digit. Clearly, the Arabic numerals 0 to 9 should be assigned the numeric values zero to nine but the ability to assign other characters as digits and to set their numeric value allows characters that represent numbers in other forms to be used. For example, the Arabic-Indic digits zero to nine are assigned Unicode code point values U+0660 to U+0669. The QMBasic character string to numeric conversions will accept as input any character defined as a digit in the mapping tables and use its associated numeric value. The QMBasic numeric to character string conversions always produce numbers formed from the Arabic numerals.

The character maps include a user definable attribute that can be used in any way that an application designer finds useful.

A range of standard character maps are available for download from the QM web site. A developer can use the `EDIT.MAP` command to modify an existing map or to create a new map. Map names are case insensitive.

A QM system can load up to eight mapping tables defined by the `ECSMAP` configuration parameter into shared memory allowing, for example, different sorting rules depending on the local conventions for users in different countries. Additional mapping tables can be loaded into the private memory of a QM process if the required table is not in shared memory. Because the tables are over half a megabyte in size, there is considerable advantage in loading the most commonly used tables in shared memory.

A QM process will use the first map loaded as the default **base map** but this can be changed by use of the `ECS.MAP` command, typically from within the `LOGIN` script, or by use of the `SET.ECS.MAP()` function in a QMBasic program.

Where a hashed file has indices, the map used to sequence the indices internally can be specified when creating the file. The sort rules in this map will be used during all index operations for all users of the file but may be different from the map in use for other activity of each user's process such as sorting displayed output. An ECS mode file with alternate key indices cannot be opened unless the relevant character map is available. Modifying a map in a manner that changes the sort order may cause files that use that map to behave incorrectly. The safest approach is to rebuild the indices after modifying the map.

The base map is included in the standard QM download. Character maps for specific regional variations can be obtained via the downloads page of the openqm.com web site. The downloaded map should be saved with an uppercase name in the ecs-maps subdirectory of the QMSYS account.
Downloaded maps should be treated as templates that should be reviewed before use to verify that they meet local requirements. Users who find a need to modify a map in a manner that might be of interest to other users are encouraged to send a brief description of the change to the OpenQM support email address.

**Double Width Characters**

Unicode defines some characters as being “wide” to indicate that they will occupy two columns on a display or printer. The character maps include a wide attribute that will be set on such characters.

The presence of wide characters has an implication on format codes, headings/footings, and on several QMBasic statements. The display width functions described below are supported in non-ECS systems but behave exactly as their standard single width character equivalents since no characters have the double width attribute in the 8-bit character set.

The \texttt{FMTDW()} and \texttt{FMTDWS()} functions apply format codes in a similar way to \texttt{FMT()} but the width is specified in terms of the display width instead of the number of characters.

The \texttt{FOLDDW()} and \texttt{FOLDDWS()} are similar to \texttt{FOLD()} but base the width of each string fragment on the display width rather than the number of characters.

The \texttt{INPUTDW} statement is similar to \texttt{INPUT} but the length limit is based on display width.

The \texttt{SUBSTRDW()} performs substring extraction based on the display width of the extracted data.

**Conversion Codes**

There are three conversion codes that relate to transformation of data between its internal ECS form and various external forms:

The \texttt{BS} conversion code used as an output conversion transforms a character string to a byte string where each ECS character becomes two characters with values in the range 0 to 255. Most applications should not need to use this conversion except as described below. Used as an input conversion, byte pairs are combined to form an ECS character. The default behaviour of the \texttt{BS} conversion code is that the character pair representation of the data is in the byte ordering used by the hardware of the computer system on which the conversion is performed. Two variations, \texttt{BSL} and \texttt{BSH}, provide a low byte first and high byte first ordering respectively.

Although primarily intended for use on ECS mode systems, the \texttt{BS} conversion is also available on non-ECS systems. In this case, use as an input conversion with data that lies outside the 8 bit character range will return the low order 8 bits of each character and set a \texttt{STATUS()} value of 3.

The \texttt{MXUC} conversion is similar to MX0C in that it translates character values to or from hexadecimal but the hexadecimal values are four digits, high byte first. The \texttt{MBUC} and \texttt{MOUC} conversions provide similar capabilities for binary (16 digits per character) and octal (6 digits per character) respectively.

Again, these codes are available in non-ECS systems. Performing an input conversion with data that would result in characters outside the 8 bit range will return a null string and a \texttt{STATUS()} value of 1.

The \texttt{Xname} conversion code applies a character encoding, translating between ECS data and an external representation such as UTF-8. The \texttt{name} may be followed by a period and one or more case insensitive character qualifiers that control the behaviour of the conversion. Used as an output
conversion, data is converted from ECS to the specified form. Used as an input conversion, data is converted from its external form to ECS. This code is available on non-ECS systems but will be limited to the 8-bit character set. Invalid input data will result in the replacement character (U+FFFD) on ECS systems or a question mark on non-ECS systems and a `STATUS()` value of 1.

Users can add their own translation codes as described with the `Xname` conversion code.

**Encryption**

The ad hoc data encryption function, `ENCRYPT()`, encrypts a series of bytes. So long as the data passed to it contains no characters outside of the 8-bit set, no special action is needed by the programmer and the resultant encrypted string is fully compatible with the non-ECS version of QM. If data that may contain ECS characters is to be encrypted, it must first be converted to a byte string using the `BS` conversion code. Conversely, the `DECRYPT()` function decrypts data to a byte string which must be converted back to characters with the `BS` conversion code if the original data was encrypted in this way. Note that an application that decrypts data must know how it was encrypted. Note also that byte ordering may become significant if this differs between the system where the data is encrypted and that on which it is decrypted. The `BS` conversion code has options to force a specific byte ordering.

Field and record level encryption within QM data files is unaffected by ECS and encrypted data in non-ECS mode files is compatible between both system types.

**Special Encodings**

The Base64 and MD5 encodings provided by the `B64` conversion code and the `MD5()` function operate on byte sequences and hence may require use of the `BS` conversion code as described above for encryption.

**Transliteration**

The ECS character maps include the ability to define a transliteration character or character pair for each codepoint. These are used by the QMBasic `TRANSLITERATE()` function to construct a representation of an ECS string using only characters from the 8 bit character set. This function maps each ECS character to one or two 8 bit characters. If it is necessary to perform more complex replacements, a QMBasic subroutine should be used, possibly as a user written conversion code.

**Hashed Files and Indices**

Files that are to store ECS data must be created with the `ECS` keyword to the `CREATE.FILE` command. Existing non-ECS files can be converted to ECS mode using `CONFIGURE.FILE`. Files that are not in this mode, including those created by earlier versions of QM can be accessed by the ECS version of QM but cannot store ECS data characters. Any attempt to write a record containing such characters to a non-ECS file will fail.

Creating a file in ECS mode does not result in a file twice as large as its non-ECS equivalent. Each record is stored in either 8-bit or ECS mode depending on the data in the record. The record ids are similarly held in whichever format is required. The only exception to this is that ECS mode files that use field level encryption always store the records in ECS format. This is necessary in order to correctly maintain the data in fields to which the user is denied access when updating a record.
Alternate key indices for ECS mode files behave similarly, storing the index data in ECS form only if it contains extended characters. The indexed values are always stored in ECS form for best performance.

**Directory Files**

Encoding rules can be specified for directory files using the same names as in the Xname conversion code outlined above. The encoding name can be specified in field 7 of the F-type VOC entry that defines the file. This can be overridden by use of the ENCODING clause to the QMBasic OPEN or OPENPATH statements and this in turn can be overridden by use of the ENCODING clause in a read or write operation. If no encoding is specified or it is overridden by use of ENCODING "NULL", the data is treated as a byte stream, writing only the low order byte from each character and returning a STATUS() value of ER_ECS_DATA if this includes characters outside the 8 bit range. Note that use of a null string as the encoding name in any QMBasic statement that supports the ENCODING clause is equivalent to not having the ENCODING clause.

When using a directory file, the application developer must determine how the mark characters are to be handled. If the data in the file uses characters 251 to 255 in their Unicode definition to represent the accented characters found in European languages, the encoding option to swap the mark characters into the private use area must be enabled. Alternatively, if these characters are to be treated as the multivalue marks, this option must not be enabled.

Encoding can be used on the non-ECS version of QM but will be restricted to the 8-bit character set.

**Sequential File I/O, including devices such as serial ports**

Encoding rules can be specified for sequential files in a similar way to directory files, via the VOC F-type item or by use of the ENCODING clause to OPENSEQ, READSEQ and WRITESEQ. Note that READBLK and WRITEBLK are byte string operations and are not affected by encoding settings.

**Select Lists**

Select lists in memory (numbered lists or Pick style select list variables) may contain ECS characters. When saving a select list to disk, the target file must either be an ECS mode hashed file or a directory file with an encoding defined.

**Terminal I/O**

When using a QMConsole session on Windows systems, the font must be set to Lucida Console if characters outside the 8-bit set are to be shown correctly. Encodings set by the PTERM command are accepted but ignored.

For direct telnet or serial port connections to QM or entry from the operating system shell other than on Windows, the terminal connection normally operates in 8-bit mode but can be switched to UTF-8 by use of the PTERM command or by entering QM with the -utf8 command line option.

Except when the connection is set to operate in binary mode, QM will relocate characters 251 to 255 in the input data to code points U+F8FB to U+F8FF. These are the accented characters common in European languages that are displaced by the conventional definition of the mark characters. The opposite transformation will occur on output to the terminal. Thus a user entering, for example, a umlaut (ii) will see this character reflected back to their screen correctly even though internally it has been transformed to be the character with code point value U+F8FC. In cases where a user needs to
be able to enter the field mark, value mark or subvalue mark from the keyboard, these can be entered using Ctrl-^, Ctrl-\ or Ctrl-\ respectively if this feature is enabled (see the `PTERM` command).

### Sockets

Socket connections opened using `OPEN.SOCKET` or `ACCEPT.SOCKET.CONNECTION` are byte string interfaces. Data written to a socket should, where necessary, be encoded into an appropriate format for transmission such as UTF-8 using, for example, `OCONV()`. Writing data with characters outside the 8-bit range will transmit the least significant 8 bits of the character value. Similarly, incoming data may need to be converted from its transmission format.

### Printing

The `SETPTR` command includes an `ENCODING` option to set the character encoding to be used for output to individual print units. If no encoding is specified, only the low order 8 bits of each character are output. The encoding can also be set with the QMBasic `SETPU` statement. Printer encoding is available on non-ECS systems but restricted to the 8 bit character set.

### Data Editing

The ^nnn notation of the `ED` editor and `MODIFY` is extended to allow ^Xnnnn to enter a four hexadecimal digit character value.

The "quote char" function of the `SED` editor is extended to allow Xnnnn to enter a four hexadecimal digit character value.

### QMBasic

QMBasic program source code must be written in the 8-bit character set. The only exception is that string constants may contain characters from the extended character set. This will require that the source file is created or encoded in a manner that supports ECS characters.

Programs compiled on the non-ECS version of QM will run unchanged on the ECS version. Programs compiled on the ECS version will run on the non-ECS version so long as they do not attempt to use any ECS specific features.

### QMNet

QMNet is unaffected by ECS except that a non-ECS mode system cannot open an ECS mode file on the remote server.

### QMClient

The QMClient C API includes a set of wide character functions that can accept or return ECS data. Connections are compatible between ECS mode and non-ECS mode systems with the exception that an ECS mode file cannot be opened by a non-ECS mode client. Data returned from an ECS mode server for subroutine calls and executed commands initiated from a non-ECS mode client must not contain characters outside the 8-bit set.
Replication

The data replication system can replicate ECS files so long as the target file on the subscriber system is either an ECS mode hashed file or a directory file with an encoding.

When replicating directory files, the state of the mark mapping mode is taken into account such that the write on the subscriber is performed with the same mark mapping state as the corresponding write on the publisher. Replicating a hashed file to a directory file target will perform the write on the subscriber with mark mapping enabled.

Writes performed to directory files on the subscriber always use any encoding set in the VOC entry for the file on the subscriber system. Encodings specified for the publisher write are independent of how the data is written on the subscriber.

Related Settings

Alternative day and month names can be set for non-English languages by use of the SET.LANGUAGE command. This requires at a minimum that messages 1500 to 1502 (month names, day names, ordinal dates) have been translated to the relevant language. See Multi-language applications for more details.

Alternative national language currency and numeric format options can be set by use of the NLS command.

Character Maps on Non-ECS Mode Systems

Non-ECS mode QM systems support minimal character mapping functionality, allowing developers to change the uppercase/lowercase pairing, sort order and character attributes. This is particularly relevant to applications that need to support the accented characters found in European languages but do not need the full capabilities of ECS mode.

When a QM process starts, the map specified by the CHARMAP configuration parameter is loaded or, if this parameter is not set, a default map is loaded. The CHAR.MAP command can be used to load an alternative map dynamically but users should beware that, because non-ECS systems have only a single active character map this might invalidate alternate key indices that were created or updated using a different map. Indices may have to be rebuilt. As a general rule, an application should use just one character map.

The default character map treats only upper and lower case A to Z as letters and sorts based on the 8-bit character value. The EDIT.MAP command can be used to create or modify a character map. The map will be stored in the ascii-maps subdirectory of the QMSYS account.

As an example, an application using Windows code page 1252 and needing to use the Spanish Ñ and ñ characters could use create a map in which these characters are given the correct uppercase/lowercase pairing, marked as being letters and placed in the correct sort order.
1.11 Character Encodings

Many of the external interfaces to QM support encoding to allow characters outside the 8 bit character set to be transmitted or stored as a sequence of 8 bit characters. QM has built in support for encodings such as UTF-8 but also allows users to add their own as described with the X (encoding) conversion code.

The QMBasic file operations that read or write directory files include an ENCODING clause to specify an encoding that will override any set by the VOC F-type entry for the file or when the file was opened.

An encoding name consists of two parts; a name related to the encoding style, and an optional series of modes, each represented by a single case insensitive character, that determine how the encoding will handle certain situations. The modes are separated from the name by a period (e.g. "UTF8.B"). Some mode settings affect only input conversion or output conversion. In such situations, the mode will be ignored by the conversion that does not use it.

Use of the "A" mode setting described below for UTF-8, UTF-16 and UCS-2 encoding allows an application to read data encoded using any of these methods by recognition of a leading byte order mark. This applies only to input conversion. If there is no leading byte order mark or when performing output conversion, the "A" mode is ignored, applying whatever encoding name is specified.

Null Encoding

Specifying the encoding name as "NULL" disables encoding. This is the default behaviour of QM in the absence of an encoding name and hence should only required when overriding an encoding name set earlier. For example, an application might use the NULL encoding name in an OPEN where the VOC F-type record specifies an unwanted encoding. Note that a statement such as

```
READ REC ENCODING " " FROM FVAR
```

does not disable an encoding set via the VOC entry or in the OPEN statement. A null string as an encoding name in this context is equivalent to not having an encoding clause at all.

The NULL encoding supports just one mode qualifier:

```
M Exchange the five accented characters displaced from the positions occupied by the mark characters (U+00FB to U+00FF) with their alternative location (U+F8FB to U+F8FF).
```

JS Encoding

The JS encoding is for JavaScript or JSON. Encoding data in this way replaces certain characters with escape sequences:

```
\" Double quote
\ Backslash
\b Backspace
\f Formfeed
\n Newline
\r Carriage return
\t Tab
\unnnn Character with hexadecimal value mnnn
```
The JS encoding supports just one mode qualifier:

M Exchange the five accented characters displaced from the positions occupied by the mark characters (U+00FB to U+00FF) with their alternative location (U+F8FB to U+F8FF).

UTF-8 Encoding

UTF-8 is one of the most widely used encodings. Each character is transformed into an encoded form that is made up from one, two or three bytes.

The QM encoding name is UTF8 (without the hyphen). The optional modes are:

A Applies only to input conversion. Automatically selects UTF-8 or USC-2 encoding based on a leading byte order mark character. Because UCS-2 in a 16-bit character environment is effectively a superset of UTF-16, UTF-16 encoded data will also be recognised by this mode setting.

B Applies only to output conversion. Adds a leading byte order mark code (internal character U+FEFF, encoded to three bytes as hexadecimal EF BB BF) if not already present. Although byte ordering is irrelevant in UTF-8, the byte order mark is frequently inserted by software, perhaps simply as a way to recognise that the data is encoded in UTF-8 format.

C Applies only to input conversion. Performs a "carry" from successive uses of the conversion such that an incomplete UTF-8 sequence at the end of one conversion can be continued in the next conversion. This is of use where data may arrive in incomplete fragments such as when reading from a socket.

D Applies only to input conversion. Discards a leading byte order mark, if present.

M Exchange the five accented characters displaced from the positions occupied by the mark characters (U+00FB to U+00FF) with their alternative location (U+F8FB to U+F8FF).

P Preserves characters 251 to 255 in the input data (input conversion only). Although strictly UTF-8 data cannot contain these characters, some other multivalue products allow their use as unencoded mark characters. This mode, intended only for use when importing data into QM, copies these characters unchanged so that the dynamic array structure is maintained.

W Applies only to input conversion. Replaces embedded byte order marks not at the start of the data with the word joiner character (U+2060).

On an input conversion, characters that lie outside the supported character set (16 bit on ECS systems, 8-bit on non-ECS systems) are replaced with a substitute character. Unicode defines this as character U+FFFD. On non-ECS systems, invalid characters are replaced by a question mark.

UTF-16 Encoding

UTF-16 encoding records Unicode data as byte pairs. The UTF-16 standard specifies that the default behaviour should be high byte first but QM allows either ordering. The byte order mark (codepoint U+FEFF) can be included as the first byte pair to allow programs reading the data to automatically detect the byte order.
UTF-16 cannot encode Unicode characters U+D800 to U+DFFF as these are reserved for use in "surrogate pairs" for characters outside the BMP. Data including these characters will result in an error status from the conversion but should never occur due to their reserved role.

The QM encoding name is UTF16 (without the hyphen). The optional modes are:

A Applies only to input conversion. Automatically selects UTF-8 or USC-2 encoding based on a leading byte order mark character. Because UCS-2 in a 16-bit character environment is effectively a superset of UTF-16, UTF-16 encoded data will also be recognised by this mode setting.

B Applies only to output conversion. Adds a leading byte order mark code if not already present.

C Applies only to input conversion. Performs a "carry" from successive uses of the conversion such that an incomplete UTF-16 sequence at the end of one conversion can be continued in the next conversion. This is of use where data may arrive in incomplete fragments such as when reading from a socket.

L Specifies that the encoded data is low byte first (little endian). On an input conversion, a leading byte order mark may override this setting.

M Exchange the five accented characters displaced from the positions occupied by the mark characters (U+00FB to U+00FF) with their alternative location (U+F8FB to U+F8FF).

UCS-2 Encoding

UCS-2 encoding records Unicode data as byte pairs in a similar manner to UTF-16 but does not reserve characters U+D800 to U+DFFF for special use. The UCS-2 standard specifies that the default behaviour should be high byte first but QM allows either ordering. The byte order mark (codepoint U+FEFF) can be included as the first byte pair to allow programs reading the data to automatically detect the byte order.

The QM encoding name is UCS2 (without the hyphen). The optional modes are:

A Applies only to input conversion. Automatically selects UTF-8 or USC-2 encoding based on a leading byte order mark character. Because UCS-2 in a 16-bit character environment is effectively a superset of UTF-16, UTF-16 encoded data will also be recognised by this mode setting.

B Applies only to output conversion. Adds a leading byte order mark code if not already present.

C Applies only to input conversion. Performs a "carry" from successive uses of the conversion such that an incomplete UTF-16 sequence at the end of one conversion can be continued in the next conversion. This is of use where data may arrive in incomplete fragments such as when reading from a socket.

L Specifies that the encoded data is low byte first (little endian). On an input conversion, a leading byte order mark may override this setting.

M Exchange the five accented characters displaced from the positions occupied by the mark characters (U+00FB to U+00FF) with their alternative location (U+F8FB to U+F8FF).
1.12 Sorting

QM provides sorting options in the query processor and in the QMBasic programming language.

There are four styles of sorting available:

- **Left aligned**: A simple left justified comparison, examining corresponding characters from the start of each string until either a difference is found or the end of both strings has been reached.

- **Right aligned**: A simple right justified comparison in which the shorter item is effectively padded with leading spaces and the resultant strings are compared character by character from the start until either a difference is found or the end of both strings has been reached. If both strings can be treated as integer values, possibly with a leading sign character, a numeric comparison is performed.

- **Right aligned float**: The same as the right aligned sort above except that the test for numeric items allows non-integer values.

- **Compound**: A compound sort in which the two strings are considered to be formed from a series of alternating integer numeric and non-numeric elements. Numeric elements are sorted into numerical value, non-numeric elements are sorted into collating sequence order. If the first element is numeric, it may have an optional leading sign character. Sign characters appearing later in the strings are treated as being non-numeric characters. Compound sorts are implemented inconsistently across different multivalue database products, particularly with regard to the relative placement of items such as "0" and "+0". In most situations, these differences are irrelevant.

---

**Sorting in the Query Processor**

The query processor provides four sort clause options: **BY**, **BY.DSND**, **BY.EXP** and **BY.EXP.DSND**. The first two sort simple data into ascending or descending order respectively. The second two work on exploded data from multivalued data items. The sort operation will be performed in left aligned style unless the field being sorted is defined as right aligned in which case a compound sort is used.

---

**Sorting in Alternate Key Indices**

Alternate key indices sort the indexed values using a compound sort. Within any index entry, record ids are sorted using either left or right aligned sort mode depending on the justification specified in the dictionary.

---

**Sorting in QMBasic**

QMBasic provides two methods of sorting. The **SORT** subroutine sorts the items in a dynamic array based on a sort style and other options specified using a sort rule parameter. This also allows
removal of duplicate items, case insensitivity, choice of ascending or descending sequence and omission of null items.

The second sort method uses the concept of a sort session in which a sort is initialised using the `SORT()` function which allows up to up to 32 keys for nested sorting criteria plus an optional data element. Items are added to the sort using the `SORTADD` statement. Once all the data has been added, items may be extracted individually with the `SORTNEXT()` function or as a field mark delimited dynamic array using the `SORTDATA()` function.

It is valid for a single QM process to have multiple sorts active simultaneously though this can lead to high memory and disk space usage. It is recommended that sort sessions that are no longer required are terminated using `SORTCLEAR`. A sort session is automatically terminated if the sort variable returned by `SORT()` is overwritten or discarded.

**Sorting in ECS Mode Systems**

In a non-ECS (8-bit character) system, the collating sequence is simply the numeric value of each character. This does not allow, for example, the French é (character 233) to appear next to its unaccented equivalent.

In an ECS system, the collating sequence is defined by the character map. It is valid for different users of the same application to have different maps and hence see sorted data in a different order. Character maps for common languages can be downloaded from the openqm.com web site. Changes can be applied to a map using the `EDIT.MAP` command.
1.13 Simple Web Services

The Simple Web Services components of QM allow easy creation of web services applications. At this release, there are two components:

- **WEBSVC** Processes incoming web service requests
- **CALLHTTP** Sends a request to a web server and returns the result

The **WEBSVC** command listens for an incoming web service request and, when detected, starts a phantom process to receive the request, parsing it into the separate header and body elements, and then calls a user supplied subroutine to perform the action required to service the request. The response data is then sent to the web client. Use of a separate phantom process for each incoming request allows multiple web services requests to be handled simultaneously.

By combining the **WEBSVC** command with the connection pooling capabilities of QM, high performance can be achieved by minimising the initialisation tasks that must be performed for each received request.

The **CALLHTTP** class module allows an application to open a connection to a web server, send an HTTP style request, and receive the response. It supports the standard GET, POST, HEAD, PUT and DELETE request types.

**Secure Web Services**

QM does not currently have SSL support internally. Where HTTPS style secure web services are required, an external third party tunnel program such as **stunnel** must be used.

**MVConnect**

The **MVConnect** API implements a layer of additional functionality to aid processing of web services applications.

**See also:**
[Building a CGI Web Server Application](#), **MVConnect**
1.14 MVConnect

MVConnect is a cross-platform Web/Rest API development package. The MVConnect API for QM is a set of QMBasic subroutines that can be used to build a web/rest application that works with QM's internal WEBSVC webserver to expand its underlying functionality. These are equivalent functions to those in other languages such as working with headers, the body, form vars, return codes, etc. Additional information regarding these MVConnect APIs can be found here.

The MVConnect API library is provided as a separate package for download and installation as needed. It can be installed in the private catalogue of an individual account or in the global catalogue to make it available to all accounts.

To install from the QM command prompt, type

    QMPKG INSTALL MVCONNECT

For a private catalogue installation, this should be done in the relevant account.

See also:
Simple Web Services, WEBSVC
1.15 Application Profiling

QM includes a number of tools for gathering data relating to application performance. Collectively these allow detailed analysis of an application’s performance, a process referred to as Application Profiling. This section summarises the profiling tools.

**FSTAT**

The FSTAT command gathers data on file activity (opens, reads, writes, etc) either on a global basis or on selected files. When used in the latter manner, data collected is stored in the file and will persist across application shutdowns until cleared.

As a profiling tool, FSTAT gives an overall perception of file activity but does not identify where that activity is occurring.

```
GLOBAL FILE STATISTICS

----------System----------Total----------this run----------Per sec----------per sec
Period 94:05:27 00:00:04
Opens 4239 552 482 138.0
Reads 1856177 18505 16919 4626.3
Writes 1488262 939 145 234.8
Deletes 95775 231 184 57.8
Clears 80 8 6 2.0
Selects 1220 190 171 47.5
Splits 1304 11 0 2.8
Merges 1 0 0 0.0
AK Reads 3606 730 54 182.5
AK Writes 4801 758 113 189.5
AK Deletes 72 24 12 6.0
Cache Hits 0 0 0 0.0
```

**HSM**

The Hot Spot Monitor (HSM) can be enabled on individual processes to accumulate data that helps in identifying where an application spends most of its time and hence the places worth attention when seeking to improve performance.

```
Calls  CP time  Opens  Reads  Writes  Deletes  Program
      0.000     0       0      0       0      0 $INLINE
      0.031     1       6      2       0       2 CONTROL
      0.000     0       2      0       0       2 $DELCAT
      0.000     1       6      2       2       0 VFS
     152  0.953     1      536      0      0 $BCOMP
     129  0.000     0      0       0      0 !TRIMS
     241  0.031     4      129      4      3 *UVSUBS
     223  0.078     5      233     109   218 $BASIC
     3791 0.015     0       9       0      0 $QPROC
     128  0.000     0       0       0      0 !SMUL
     220  0.000     0       0       0      0 !SSUB
     120  0.015     0       9       0      0 $QDISP
```

**Command Logger**
The **Command Logger** generates a log of every command executed, including those from within Proc's, paragraphs, menus, programs, etc. It can be applied on a system wide basis or within specific QM processes.

The columns of the example report extract above are:
- Timestamp (as an epoch value)
- User number
- User name
- Command origin
- Command

Note that in this example, the IF command near the top of the report generates three log lines; firstly as it appears in the paragraph with the inline prompt in place, then again with the prompt expanded and finally, because the condition was true, as the actual command.

**PSTAT**

Not strictly a profiling tool, the **PSTAT** (Process Status) command shows what a QM process is currently doing. It can be used to produce a report of all logged in QM users or just one specific user. Different reporting levels allow selection of what data will be shown.

In this example, user 6 is in the SED editor at line 5407. The hexadecimal program offset and internal opcode name are of little use to an application developer but are included for diagnostics. Where a process is waiting for external activity such as keyboard input, the **PSTAT** command will show this.

**Lock Tracing**

The **Lock Tracer** records every lock action across all users of the system. The gathered data is then processed by an interactive reporting tool that produces a (lengthy) report similar to the brief extract below.
Event Profiling

The **PROFILER** command controls recording and display of application events that are often significant contributors to performance issues. The profiler report is an interactive display that allows the user to drill down starting with a list of logged event types and going down to subsequent screens that show qualifying information for a selected event type (e.g. file name in an OPEN), the program that generated the event and finally, so long as the program is compiled in debug mode, the line numbers within the programs. At each level, the count of events is shown alongside the other information. The brief extract below shows the names of subroutines called by the application and the number of times each was called.

<table>
<thead>
<tr>
<th>Subroutine call counts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Counts</td>
</tr>
<tr>
<td>---</td>
</tr>
<tr>
<td>&gt;1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>150</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>4</td>
</tr>
</tbody>
</table>
Part 2

The Command Environment
2 The Command Environment

Although applications commonly use the graphical interface capabilities of QM via Visual Basic, AccuTerm or web browsers, developers normally work from the character mode interface command prompt using a terminal emulator or directly from the system console. Commands can also originate from within application programs.

 Commands entered at the terminal or generated from within a QM application are processed by the command processor. This uses the vocabulary file (VOC) to determine the meaning of each word or symbol within the command.

The default command prompt is the colon character. Whenever this is displayed at the start of a line, QM is ready to accept a new command. The command prompt changes to a double colon if the default select list is active. This serves as a warning that the select list may impact execution of the next command. The prompt characters may be modified using the PTERM command.

By default, QM operates with "case inversion" - that is, characters entered in lowercase letters are displayed in uppercase and vice versa. The original multi-value systems date back to a time when many terminals did not have lowercase letters and hence the command language was written to work in uppercase. To make it easy to operate in situations where a user may be switching between a QM session and, for example, a Word document, QM normally applies case inversion so that the user does not need to keep hitting the caps lock key.

Actually, QM is largely case insensitive but this feature is retained for compatibility with other systems. It can be disabled by typing

```
PTERM CASE NOINVERT
```

and this would usually be done as an automated part of the login process for an end user of the application. Alternatively, the default state may be changed by use of the INVCASE configuration parameter.

A command consists of a number of tokens separated by spaces. Where a token includes spaces, it must be enclosed in quotes. QM allows use of single quotes, double quotes or backslashes interchangeably though recognition of backslashes as quotes in the command processor can be disabled with the BACKSLASH.NOT.QUOTE mode of the OPTION command.

The first token of a command normally corresponds to the name of an executable item within the VOC. This will be the name of a verb, sentence, paragraph, menu or Proc (terms that will be defined in the following sections). It is also possible to run a program from the system catalogue by typing its name as a command. Other valid actions at the command prompt are:

- **Command stack operations**, prefixed by a dot character
- **Command editor keystrokes**
- Save the command without execution by appending a question mark

The command processor performs the VOC look-up for a command in three stages; firstly by looking for a record with a name matching the first token in the command exactly as entered. If this fails, it then tries again with the name mapped to uppercase. All system verbs have uppercase names and can therefore be entered in lowercase, uppercase or a mix. For compatibility with Pick databases, a third attempt is made with any hyphens in the uppercase version of the name replaced by dots. Thus a command such as `CREATE.FILE` can be entered as `create.file` or `CREATE-FILE`.

If the command is not found in the VOC, a check is made in the private and global catalogues. If the name exists here, the catalogued program is executed. The names of catalogued programs executed in this way must commence with a letter or an asterisk. When executing a catalogued program in this
manner, the command processor will look for and handle the **LPTR** and **NO.PAGE** keywords in the same way as the **RUN** command, leaving them in place so that the executed program will also see them if it performs its own command line scan.

Finally, if the command has still not been located, the command processor looks in the **private vocabulary**, if this exists.

Many commands perform the first two phases of this look-up for the remaining tokens on the command line (file names, keywords, etc), however, commands that might have a detrimental effect if used in error (**DELETE.FILE**, for example) either insist on the file name being entered exactly as it appears in the VOC or prompt for confirmation if the name is not an exact match.

Command lines commencing with an asterisk followed by at least one space are treated as comments and are ignored except that **inline prompts** are still processed. Although comments are primarily of use within paragraphs, they can be entered directly at the keyboard when they will appear in any active **como file**.

Many QM commands return status values via the **@SYSTEM.RETURN.CODE** variable. In general, a positive or zero value indicates success. A negative value is an error code and the actual value is the negative of the codes listed in the **ERR.H** include record in **SYSCOM**.
2.1 The Command Stack

Commands entered at the terminal are stored in a command stack (to be technically correct, it is a queue but historically users have called it a stack). Commands may subsequently be recalled for re-execution by a simple short form command. By default, the stack holds the last 99 commands but this value can be changed by use of the CMDSTACK configuration parameter. The list is indexed by number such that the most recent command is numbered as 1, the oldest as 99. Conventionally, command 1 is referred to as the top of the stack.

The stack can be manipulated by commands prefixed by a dot character (period) entered at the command prompt. These allow commands on the stack to be edited and also provide facilities to save and restore sequences of commands to and from VOC paragraphs.

The stack manipulation commands are

- `.A n text` Append text to command stack entry n. There must be a space before text. Any additional spaces will be included in the appended data. If n is omitted, the top entry on the stack (position 1) is updated. The text is displayed after modification.

- `.C n /old/new/G` Change string old to new in stack entry n. If n is omitted, it defaults to one. The delimiters around old and new may be any non-space character. The space before the first delimiter may be omitted if the delimiter is not a digit. The optional G causes a global replacement, that is, all occurrences of old are replaced by new. If G is not specified, only the first occurrence of old is changed. The text is displayed after modification.

- `.C-` Used immediately after .C this undoes the change. An intervening use of .L or .? is permitted. Any other action removes the ability to undo the change.

- `.D n` Delete stack entry n. If n is omitted, the top stack entry is deleted.

- `.D name` Delete VOC entry name if it is a sentence or paragraph record. A confirmation prompt is issued prior to deletion.

- `.I n text` Insert text as stack entry n. If n is not specified, text is inserted at the top of the stack. There must be a space before text. Any additional spaces will form part of the inserted entry.

- `.L n` List the most recent n commands. The value of n defaults to 20. See Command Editor Options.

- `.L name` List VOC entry name.

- `.R n` Recall stack entry n to the top of the stack without deleting the original copy. If n is omitted, the top entry is duplicated.

- `.R name` Read VOC entry name to the top of the stack if it is a sentence or paragraph. Field one of the VOC entry is discarded, any continuation lines are merged and blank lines in a paragraph are omitted.

- `.S name n m` Save stack lines m to n as VOC entry name. The value of m and n may be entered in either order. If m is omitted it defaults to the same value as n. If n is also omitted, the top line of the stack is saved. The VOC entry will be a sentence if only a single line is saved, otherwise it will be a paragraph.
.Un

Convert stack entry \( n \) to upper case. \( n \) defaults to one if omitted.

.Xn

Execute command \( n \). If \( n \) is omitted, the last command is executed.

The repeated command is copied to the top of the stack except when executing the current topmost command.

.X file record

Execute command stored in the named file and record. This record must have the same format as a VOC record.

.? 

Display a help message regarding the stack manipulation commands.

For compatibility with other environments, a command can also be saved on the stack without execution by entering it at the command prompt with a question mark as the last character. The question mark is removed.

The command stack is saved between sessions if the VOC contains a record named $COMMAND.STACK with field 1 set to X. This record is inserted automatically when a new account is created but can be deleted if the stack is not to be saved. Presence of this record causes the command stack to be saved to, or restored from, a file named as the user's login name in the stacks subdirectory of the account in which QM was entered. On ECS mode systems, the stack is saved in UTF-8 format.

QM supports a secondary, private vocabulary file that is represented by a multi-file named PVOC with subfile names that correspond to the uppercase form of the user's login name. The .D, .L, .R and .S commands have an extended form to access the private vocabulary file:

.DP name

Delete private vocabulary entry \( name \) if it is a sentence or paragraph record. A confirmation prompt is issued prior to deletion.

.LP name

List private vocabulary entry \( name \).

.RP name

Read private vocabulary entry \( name \) to the top of the stack if it is a sentence or paragraph. Field one of the entry is discarded, any continuation lines are merged and blank lines in a paragraph are omitted.

.SP name n m

Save stack lines \( m \) to \( n \) as private vocabulary entry \( name \). The value of \( m \) and \( n \) may be entered in either order. If \( m \) is omitted it defaults to the same value as \( n \). If \( n \) is also omitted, the top line of the stack is saved. The entry will be a sentence if only a single line is saved, otherwise it will be a paragraph.

See also The Command Editor
2.2 The Command Editor

The command line editor allows editing of a command line. It is of use in correcting typing errors or repeating saved commands, possibly after modification.

The command line editor handles the following keystrokes:

- **Ctrl-A or HOME**: Move cursor to start of command.
- **Ctrl-B or Cursor Left**: Move cursor left one place.
- **Ctrl-D or DELETE**: Delete character under cursor.
- **Ctrl-E or END**: Move cursor to end of command.
- **Ctrl-F or Cursor Right**: Move cursor right one place.
- **Ctrl-G**: Exit from the command stack and return to a clear command line.
- **Ctrl-K**: Delete all to the right of the cursor.
- **Ctrl-L**: Convert command to lowercase.
- **Ctrl-N or Cursor Down**: Display "next" command from command stack.
- **Ctrl-O or Insert**: Toggle insert/overlay mode.
- **Ctrl-P**: Display "previous" command from command stack.
- **Ctrl-R**: Search back up the command stack for a given string. The string may be entered either before or after the Ctrl-R. Using Ctrl-R again, finds the next item matching the supplied string. The search is case insensitive.
- **Ctrl-T**: Interchange characters before cursor.
- **Ctrl-U**: Convert command to uppercase.
- **Ctrl-W**: Amend casing of the "word" at or preceding the cursor position, cycling through uppercase, lowercase and title case.
- **Ctrl-Z or Cursor Up**: Display "previous" command from command stack.
- **Backspace**: Backspace one place.

Entering a command line containing only a question mark shows a summary of the command editor keys.

Command Editor Options

The command stack editor operations are controlled by option codes which may be entered in field 3 of the $RELEASE VOC entry. These are:

- **E**: Position the cursor at the end of a recalled command rather than the start.
- **L**: Use the screen size to determine the default number of items shown by .L
- **O**: Start in overlay mode.
- **S**: Show the stack commands when moving back through the stack.
- **X**: Clear the recalled command if the first character typed is not a control code. This mode cannot be used with E.

See also The Command Stack
2.3 Interrupting Commands

It may be necessary to terminate a command because, perhaps, it is producing more output than expected or it is not functioning as required. The break key (usually ctrl-C) can be used to terminate processing and return to the command prompt.

To protect against accidental use of the break key, QM will display a prompt asking for confirmation that processing is to be terminated. Valid responses to this prompt are:

- **A** Abort. Returns to the command prompt in exactly the same way as an abort generated by an `ABORT` statement in a QMBasic program or an `ABORT` command in a paragraph. The `ON.ABORT` paragraph is executed, if present. The `@ABORT.CODE` variable will be set to 1. The default select list (list 0) will be cleared if it was active.
- **D** Only offered when appropriate, this option enters the QMBasic debugger.
- **G** Go. Continues processing from where it was interrupted. If the terminal supports the necessary operations, QM will restore the display image to remove the prompt.
- **P** Creates a process dump file and continues execution.
- **Q** Quit. Returns from the current command to the paragraph, menu, program or command prompt that initiated the command. The `ON.ABORT` paragraph is not executed. The `@ABORT.CODE` variable will be set to 2. The default select list (list 0) is not cleared.
- **S** Stack. Displays the call stack showing the program name and location for each entry.
- **W** Where. Displays the current program name and location.
- **X** Exit. Aborts totally from QM without executing the `ON.EXIT` paragraph. This option should only be used if QM appears to be behaving incorrectly.
- **?** Help. Displays a brief explanatory help text for each option.

The break key is initially disabled on entry to QM but it is enabled after execution of the optional QMSYS MASTER.LOGIN paragraph and LOGIN VOC item. It is often left disabled in application software so that users cannot gain access to the command environment. Developers usually need the break key enabled and this can be done using the BREAK command or the corresponding QMBasic BREAK statement.

QMBasic programs can establish a break handler to catch use of the break key and take special action. See the SET.BREAK.HANDLER statement for details.
2.4 Output Pagination

Output to the display is automatically paginated, where appropriate, by inserting a prompt at the end of each page of output. The options available at this prompt are:

- **A** Abort. Returns to the command prompt in exactly the same way as an abort generated by an `ABORT` statement in a QMBasic program or an `ABORT` command in a paragraph. The `ON.ABORT` paragraph is executed, if present. The `@ABORT.CODE` variable will be set to 1. The default select list (list 0) will be cleared if it was active.

- **Q** Quit. Returns from the current command to the paragraph, menu, program or command prompt that initiated the command. The `ON.ABORT` paragraph is not executed. The `@ABORT.CODE` variable will be set to 2. The default select list (list 0) is not cleared.

- **S** Suppress pagination. Continues execution with no further pagination prompts.

- **Other** Any other key continues execution until a further pagination prompt is displayed.

The number of lines per page can be adjusted from its initial value by use of the `TERM` command.

Pagination can be disabled by application software or by use of the `NO.PAGE` option to some commands.
### 2.5 The VOC File

The VOC file is central to everything that QM does. This file is the vocabulary of words and symbols that may appear in commands and holds many other things as well. The initial VOC file is a copy of NEWVOC from the QMSYS directory. By modifying the VOC it is possible to change the names of commands to meet particular needs of an application or user. It would be possible, for example, to include French translations of all the command names. More often, changes are made simply to use wording that is more appropriate to the manner in which the product is used.

Records in the VOC are of differing types, the type of the record being determined by the first one or two characters of field 1 of the record. The remainder of field 1 after the identifying characters may contain any value and is typically used to comment the role of the VOC entry.

The VOC record types are

- **D** Data item
  - Defines a field within a data file. D type entries may appear in the VOC but are more commonly found in dictionaries.

- **F** File
  - Defines a file, relating its application level name for use within QM to its operating system pathname.

- **K** Keyword
  - Many commands have keywords which affect the behaviour of the command or introduce optional clauses in the command syntax.

- **M** Menu
  - A menu record defines a menu that can be displayed by executing the VOC entry.

- **PA** Paragraph
  - A paragraph is a sequence of commands that can be executed by entering the name of the VOC entry.

- **PH** Phrase
  - A phrase is a short form for a sequence of items to be substituted into query processor commands.

- **PQ** PROC
  - A PROC is the predecessor of paragraphs. QM supports PQN style PROCs for use when migrating applications. It is recommended that new developments should use paragraphs or QMBasic programs instead.

- **Q** Remote File
  - A remote file pointer refers to a file in another QM account, perhaps on a different server.

- **R** Remote
  - An R type VOC entry points to a record in another file which is constructed in the same way as an executable (M, PA, R, S or V type) VOC entry.

- **S** Sentence
  - A sentence is a single command.

- **V** Verb
  - A verb is the portion of a command which identifies the part of QM which will process it.
X Other
X type records may be used to store miscellaneous information in the VOC.

VOC record types Y and Z will never be defined as part of the standard QM product and are available for whatever purpose a developer may find useful. Other type codes not listed above may be defined in future releases.

Users may add handlers for other VOC record types that are to be usable as commands. This is done by creating a VOC record named $VOC.PARSER:

- Field 1 X
- Field 2 A multivalued list of VOC record type codes.
- Field 3 A corresponding multivalued list of catalogued handler subroutine names.

The handler is a QMBasic subroutine taking two arguments; the verb name and the VOC record. It is necessary to use LOGTO or exit and re-enter QM after adding or modifying the $VOC.PARSER record.

The VOC includes an F-type entry referencing itself so that commands that access the VOC do not have to treat it as a special case. Users must not modify this VOC entry as any change is likely to cause QM to malfunction because internal components reference the VOC by pathname.

The VOC also includes a Q-type entry named MD as a synonym for VOC for compatibility with other systems.

Making the VOC Case Insensitive

The CREATE.ACCOUNT command has an option to create the account with a case insensitive VOC file. Converting the VOC of an existing account is a little more complex as the reconfiguration of the VOC file requires that no QM process has it open at the time. The easiest way to achieve this is to set up an F-type VOC entry from some other account, referencing the pathname of the VOC that is to be modified. Although a Q-pointer may seem more logical, the CONFIGURE.FILE command requires that the file is accessed via an F-type item.

Once the F-type item is in place and all users of the account have logged out or moved to another account, use CONFIGURE.FILE with the NO.CASE option to reconfigure the required VOC file.

It is strongly recommended that the original VOC file is backed up before making this modification.

The Private VOC

The vocabulary file is shared by all users of the account to which it belongs. To allow users to store private commands (verbs, sentences, paragraphs, menus, Procs), QM also supports use of a private vocabulary file that is separate for each operating system level user name. This is a multi-file named PVOCS with subfiles names that are the uppercase version of the user’s login name. The private vocabulary is checked only after failing to find the item in the VOC and the catalogue. It is only used to locate command names, not other VOC record types such as phrases, keywords or files.

The private vocabulary file may be created and modified directly by of CREATE.FILE, ED, SED, etc. Alternatively, the “dot commands” of the command stack editor include options in .D, .L, .R and .S to access the private vocabulary. First use of .SP in this mode will create the private vocabulary file if it does not already exist.
Unlike the VOC file, which the command processor opens by pathname, the private vocabulary is opened using the normal indirection via a VOC F-type record. It is therefore possible to set up the VOC record such that groups of users share a common private vocabulary or so that one user sees the same private vocabulary from multiple accounts.

The optional LOGIN, ON.EXIT, ON.LOGTO and ON.ABORT command scripts must reside in the VOC file, not the private VOC.
VOC D-type records - Data items

A D-type record defines a field stored in a data file.

Although D-type records may be stored in the VOC file, they are more usually found in dictionaries. A D-type entry in the VOC can be used to reference a field in any file whereas a D-type entry in a dictionary can only be used in queries against the associated file.

A D-type record has up to 8 fields:

1:  D { descriptive text }
2:  Field number. This is the position in the data record at which the field described by this dictionary entry can be found. A value of zero denotes the record id.
3:  { Conversion code }
4:  { Display name. This will be used as the default column heading by the query processor. }
5:  Format specification
6:  Single/multivalue flag. Set as S if the field is always single valued or M if it can be multivalued.
7:  { Association name. Where a multivalued field has a value by value relationship with some other multivalued field defined in the same dictionary, this name links the fields together. }
8:  { Available for user use in any way. Not referenced by QM. }

Fields 9 onwards are reserved for internal use and users should not assume anything about their content.

See Dictionaries for more information.
VOC F-type records - File definitions

Every file referenced by an application is accessed via an F-type VOC record. This record maps the QM name of the file to the pathnames of the data and dictionary components.

1:  F  { descriptive text }
2:  The pathname(s) of the directory that represents the data portion of the file. In a multi-file, the pathname of each subfile appears as a separate value in this field.
3:  Dictionary pathname. This field is empty if the file has no dictionary.
4:  Subfile names for a multifile. This field is empty for a simple file.
5:  File inclusion flags for ACCOUNT.SAVE and FILE.SAVE.
   There are three possible values:
   D  Include only the dictionary of this file in the save
   E  Exclude this file from the save
   I  Include this file in the save
   Leaving the field empty causes ACCOUNT.SAVE and FILE.SAVE to fall back on alternative file selection methods.
6:  Special file open modes for QMBasic OPEN and OPENSEQ operations. Value 1 of this field applies to the data part of the file. For a multifile, these modes apply to all elements. Value 2 applies to the dictionary part. Both may be any compatible combination of the codes listed below.
   B  Sets the line terminator for a directory file on read to be either a CR/LF pair or a lone LF, treating lone CR characters as part of the data. On write, the operating system line terminator is used. Ignored for other file types.
   C  Sets the line terminator for a directory file to be a CR/LF pair. Ignored for other file types.
   L  Sets the line terminator for a directory file to be a LF character. Ignored for other file types.
   M  For a directory file, causes the file to be opened with mark mapping disabled. This is useful for files containing binary data such as scanned images.
   N  Specifies that the file will always be opened non-transactionally when using this VOC item.
   R  Specifies that the file is to be read-only when opened using this VOC item.
   S  Sets synchronous (forced write) mode on the file (hashed files only).
   T  Suppress translation of restricted characters in directory file record ids. This affects only the data portion of the file. See also the NO.MAP option to the QMBasic OPEN statement. This flag can be set using the NO.MAP option to the CREATE.FILE or CONFIGURE.FILE commands.
   X  Sets the line terminator for a directory file to be either a CR/LF pair or a lone LF, stripping lone CR characters. Ignored for other file types.
   Applications that make direct access to this field should allow for the presence of codes not in the above list. For a discussion of the line terminator modes see directory files.
7:  Default character encoding for this file. This is only relevant to directory files or sequential files. It is valid for a file to be referenced from more than one F-type record with different default encodings. Value 1 of this field applies to the data part of the file. For a multifile, the encoding applies to all elements. Value 2 applies to the dictionary part.

Either pathname field may be blank to indicate that the file portion does not exist.

Three special pathname prefixes are allowed:
• @QMSYS will be replaced by the QMSYS account directory pathname, ensuring that references to items in the QMSYS account will still function if a new release is installed at a different location.

• @TMP will be replaced by the pathname in the TEMPDIR configuration parameter.

• @HOME will be replaced by the value of the HOME environment variable on Linux or the HOMEPATH environment variable on Windows. Use of this prefix allows different users to see a different personal version of the same file.

By using F-type VOC items to locate files indirectly rather than embedding file pathnames in the application, the VOC entry becomes the only place where the pathname is recorded. If a file is moved, perhaps to balance loading across multiple disks, only the VOC entry needs to be amended; the application itself is not affected.

Where two or more accounts share a file, the VOC files in each account could have F-type records mapping the QM name to the pathnames. This is not recommended. Instead, the account that owns the file should have an F-type record and all other accounts should have Q-type records to access the file indirectly.

The pathname of either or data or dictionary portion of a file may be specified as VFS:server:detail to make use of the Virtual File System.

A summary of F-type VOC records may be displayed or printed using

**LISTF**  Show all F-type entries

**LISTFL**  Show only local files (in the account directory)

**LISTFR**  Show only remote files (not in the account directory)
VOC K-type records - Keywords

Keywords affect the behaviour of commands or introduce optional components in the command syntax. Keywords are defined by K-type VOC records.

1: K  { descriptive text }
2: Keyword number
3: { alternative expansion }

Each keyword is assigned a number which appears in field 2 of a keyword VOC entry.

Keywords with internal number 0 in field 2 are ignored by the query processor and some other parts of QM. They are provided to allow construction of more natural English sentences. For example, the THAN keyword can be used with other elements such as GREATER and LESS to allow a query such as

```
LIST STOCK WITH PRICE GREATER THAN 100
```

instead of

```
LIST STOCK WITH PRICE GREATER 100
```

Users can freely add new keywords with internal number 0 as required.

In some cases, a keyword is also needed as a command name (e.g. OFF which is a synonym for QUIT but also a modifier in several other commands). A keyword can never be the first token in a command. If the command processor finds a K-type VOC item used as the first token in a command, it looks for an alternative VOC record structure starting at field 3.

Thus, as an example, the OFF VOC entry reads

```
1: K
2: 20
3: V
4: IN
5: 1
```

where fields 3 onwards contain an alternative V-type (verb) definition.

A summary of K-type VOC records may be displayed or printed using LISTK.
VOC M-type records - Menu definitions

A VOC menu record defines a menu of numbered options to be displayed to the user when the menu entry is executed. Because menu records may be very large, they are often stored in some other file with a VOC R-type record as a remote pointer to the actual menu definition.

A menu record has 11 fields:

1: M { descriptive text }
2: Menu title line to appear at the top of the screen.
3: Item text. This field is multivalued with one value for each menu entry. Blank entries are allowed to insert spacing in the menu. Each menu entry is numbered except as described under field 4 below.

The descriptions are normally displayed starting on the third line of the screen, left justified. If the menu has more items than will fit in a single column on the screen and the items are all sufficiently short, the menu will be displayed as two columns. Any menu items that will not fit on the screen are lost.

4: Action. This field is multivalued with entries corresponding to the text in field 3. If the action is terminated by a semicolon, the menu processor issues a "Press return to continue" prompt when the command is completed. Blank entries cause the field 3 text to be treated as a sub-title and not numbered on the displayed menu.

5: Help text. A multivalued set of one line help texts corresponding to each menu option in the previous fields.

6: Access key. An optional multivalued set of access control keys corresponding to the menu items. The access key value is passed to the access control subroutine if this is used.

7: Hide inaccessible entries. This field may be single valued in which case it applies to all menu entries or it may have one value for each menu item. Each value present is a numeric representation of a Boolean (1 or 0 corresponding to True or False) flag indicating whether inaccessible menu entries should be hidden (not displayed) or shown as unavailable.

8: Access subroutine. This optional field contains the name of an access control subroutine (see below).

9: Prompt text. If present, this text replaces the default option prompt.

10: Exit codes. An optional multivalued list of codes which when entered at the option prompt will exit from the menu. If this field is blank, entering a null response to the menu prompt will exit from the menu. Because exit codes are processed before option numbers, it is possible to include an option that causes an exit by specifying the option number as an exit code.

11: Stop codes. An optional multivalued list of codes which when entered at the option prompt will generate an abort event, terminating all active processing and returning to the command prompt. If this field is blank, it defaults to Q. Because stop codes are processed before option numbers, it is possible to include an option that causes a stop by specifying the option number as a stop code.

Menus may be constructed and maintained using the menu editor MED.

A summary of M-type VOC records may be displayed or printed using LISTM.

Access Control
A menu may include run time selection of which options are to be offered to the user. This is managed by a user written access control subroutine named in field 8 of the menu definition. When the menu is displayed, this subroutine is called for all entries that have an access key defined in field 6 to determine whether the option is to be offered. The subroutine takes three arguments; the returned True/False accessibility flag, the menu name and the access key. Typically, the access key would be a name that refers to a group of users such as ADMIN or SALES. Access control subroutines may be used by multiple menus and, where necessary, can use the menu name in the decision process that controls display. The subroutine should return the first argument as True (or numeric value 1) if the menu item is to be allowed, False (or numeric value 0) if it is to be disabled.

Field 7 of the menu definition contains the "hide" flag which controls the action taken when a menu option is disabled for a user. Setting this to 0 or leaving it blank, displays the option with its option number enclosed in parentheses. Entry of this option number at the menu prompt will be ignored. Setting the hide flag to 1, completely hides the disabled option.

An example of a simple access control subroutine is shown below:

```vbnet
SUBROUTINE ACCESS(OK, MENU.NAME, KEY)   BEGIN CASE      CASE KEY = 'ADMIN'         OK = (@LOGNAME = 'ADMINISTRATOR')      END CASE          CASE 1      OK = @FALSE   OPEN 'USERS' TO USER.F THEN        READ USER.REC FROM USER.F, @LOGNAME THEN          OK = (USER.REC<U.DEPT> = KEY)        END      END CASE   END RETURN
```

For an access key of ADMIN, this subroutine enables the menu option only if the user is logged in as an administrator. For all other access key values, it reads a user record and checks that the user is in the relevant department. The U.DEPT token used above would be defined in an include record.
VOC PA-type records - Paragraphs

A paragraph is a sequence of stored commands or sentences which will be executed in turn by entering the paragraph name in response to the command prompt.

1: PA { descriptive text }
2: First sentence
3: Second sentence
4: etc...

Field 1 of the VOC paragraph record commences with PA, fields 2 onwards are the commands to execute. Any sentence within the paragraph may be broken into shorter parts by using the underscore character to indicate that the command continues on the next line.

The Cn and ln control codes of inline prompts may be used to substitute additional text from the sentence that started the paragraph into the commands within the paragraph.

Paragraphs may contain a number of special commands and constructs that are not allowed in sentences. These are:

- **DATA** Embedded data for an application
- **IF** Conditional execution
- **GO** Jumps to labels
- **LOOP** Repeated execution of a loop
- **$ECHO** Trace execution of the paragraph

Comment lines may be included in a paragraph. These have an asterisk as the first non-space character. There must be at least one space between the asterisk and the actual comment text. Note that expansion of inline prompts is the first action of the command processor and hence inline prompts are evaluated in comment lines. This slightly odd behaviour is useful in controlling the timing and sequence of inline prompts independently from the use of their substitution text.

Paragraphs may invoke other paragraphs. Beware of accidental recursive invocation of the same paragraph.

There are four reserved paragraph names for special functions. These are:

- **LOGIN** Executed on entry to QM and also when the LOGTO command is used to switch to a new account.
- **ON.LOGTO** Executed on use of the LOGTO command before switching to the new account.
- **ON.EXIT** Executed on leaving QM by use of the QUIT command.
- **ON.ABORT** Executed when QM aborts a program.

The QMSYS account may contain a paragraph with a further reserved name:

- **MASTER.LOGIN** Executed on initial entry to QM in any account before the LOGIN paragraph.
For more details of the above, see Command Scripts.

A summary of PA-type VOC records may be displayed or printed using LISTPA.
VOC PH-type records - Phrases

A phrase can be used in query processor sentences. When the sentence is executed, the phrase name is replaced by the phrase expansion. Typically, phrases are used to give names to groups of fields to be displayed or selection criteria.

1: PH { descriptive text }
2: Phrase expansion

Phrases may be included in the VOC but are more commonly found in dictionaries. A phrase in the VOC can be used in queries against any file whereas a phrase in a dictionary can only be used in queries against the associated file.

Where a phrase is very long it may be broken into multiple lines within the VOC record by terminating all but the final line with an underscore character. When the phrase is substituted into a command, the lines are merged, replacing the underscore with a single space.

A summary of PH-type VOC records may be displayed or printed using LISTPH.
VOC PQ-type records - PROCs

PROCs are the predecessor of paragraphs. They are generally thought to be much harder to understand and maintain but are supported in QM for compatibility with other systems. New applications should use paragraphs or QMBasic programs in place of PROCs.

```
1:   PQ{N}  { descriptive text }
2+:  PROC statements
```

PROCs come in two styles identified by the VOC record type; standard PROCs (PQ) and new style PROCs (PQN). QM supports the major features of PROCs but is not a full implementation of the various PROC environments found in other multivalue products.

Because development of new PROCs is discouraged, only an overview of what elements of PROCs are supported by QM is given here. It is not intended as a detailed reference document or a learning aid.

Proc Buffers

A PROC works by manipulating data in a set of buffers, each stored internally as a field mark delimited dynamic array (PQN) or a space delimited string (PQ). These are:

**The Primary Input Buffer (PIB)**

The PIB initially holds the command that started the PROC and any command line options. A PROC can use the PIB to store other data during its operation.

**The Secondary Input Buffer (SIB)**

The SIB is typically used to store user input entered in response to the IN statement.

**The Primary Output Buffer (POB)**

The POB is used to construct a command to be executed. Execution of the assembled command is triggered by use of the P statement or by termination of the PROC.

**The Secondary Output Buffer (SOB)**

The SOB, often called the stack, is used to hold data to be processed by the command in the POB. It can also hold supplementary commands to be executed after the POB has been executed.

At any moment, one input and one output buffer is considered as being active. The SP and SS statements can be used to make the primary or secondary input buffer active respectively. Similarly the STOFF and STON statements can be used to select the primary or secondary output buffers as active.

The input buffer pointer is used to identify a position within the active input buffer.

When a PROC starts, the primary input and output buffers are active and the input buffer pointer points to the start of the PIB.
The File Buffers

There are ten file buffers, numbered from 0 to 9. File buffers 1 to 9 are the standard file buffers. File buffer 0 is the fast file buffer and can be accessed with a special buffer reference syntax.

Select List Buffers

The eleven numbered select lists can be accessed using the select list buffers.

Buffer References

Many statements can reference buffers using the tokens shown below:

<table>
<thead>
<tr>
<th>Token</th>
<th>Buffer</th>
<th>Direct</th>
<th>Indirect</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>Primary input buffer</td>
<td>%1</td>
<td>PIB field 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>%#2</td>
</tr>
<tr>
<td>#</td>
<td>Active output buffer</td>
<td>#1</td>
<td>AOB field 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>#%1</td>
</tr>
<tr>
<td>&amp;</td>
<td>File buffer</td>
<td>&amp;4.2</td>
<td>File 4, field 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&amp;%1.%2</td>
</tr>
<tr>
<td>&amp;</td>
<td>Fast file buffer</td>
<td>&amp;1</td>
<td>Field 1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>&amp;%2</td>
</tr>
<tr>
<td>!</td>
<td>Select list</td>
<td>!5</td>
<td>List 5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>!%1</td>
</tr>
</tbody>
</table>

An indirect reference uses the content of one buffer to index into another.

In a file buffer, field 0 references the record id associated with the buffer.

A-References

An A-reference is a reference to data in the active input buffer using the syntax of the A statement described in the following section. When used in this form, an A-reference does not move the input pointer or change the content of the buffers.

PROC Statement Summary

A Move a field from the active input buffer to the end of the active output buffer.

\[A\{c\}\{p\}\{,m\}\]
Move up to \(m\) characters of field \(p\) to the output buffer, enclosing the text in character \(c\). \(c\) may be any character except a digit, left bracket or comma and defaults to a space. Specifying \(c\) as a backslash suppresses the surround character. The surround character is ignored if the data is copied to the secondary output buffer. If \(p\) is omitted, data is copied from the field addressed by the current position of the input pointer. If \(m\) is omitted, data is copied until the end of the field is reached. The input pointer is positioned following the last character moved.

\[A\{\{n\}\{,m\}\}\]
Move up to \(m\) characters, starting at character \(n\), to the output buffer. If \(n\) is omitted, data is copied from the current position of the input pointer. If \(m\) is omitted, data is copied until the end of the field is reached. The input pointer is positioned following the last character moved.
Data copying normally terminates at the end of the field. Use the PROC.A or PICK.PROC modes of the **OPTION** command to enable compatibility with D3 where the copy continues past the end of the field.

**B**  
Move the input pointer back to the previous field.

If the input pointer is at the start of a field, it is moved back to the start of the previous field. Otherwise it is moved back to the start of the current field.

**BO**  
Move the output pointer back to the previous field

The output buffer pointer is moved back to the previous field, truncating the data at its new position.

**C**  
Comment

*tex*

All text following the C is ignored.

**D**  
Display fields from the active input buffer

D{ref|p}{,m}{+}

*ref* is a direct or indirect reference to a buffer containing the field number of the active input buffer that is to be displayed.

*p* is the field number of the active input buffer that is to be displayed. If *p* is zero, the entire input buffer is displayed.

*m* is the maximum number of characters to be displayed.

+ suppresses the normal newline after display

**DB**  
Display all input and output buffers

The content of the primary and secondary input and output buffers is displayed.

**DEBUG**  
Enters the Proc debugger.

This command is supported only in QMConsole sessions on Windows or when using AccuTerm with a terminal type with a -at suffix (e.g. vt100-at). The debugger shows a portion of the Proc around the current line and the content of the PIB, POB, SIB and SOB buffers. The currently active buffers are highlighted. If the buffer content exceeds the space available to display it, the final characters shown are an ellipsis.

The debugger supports the following command keystrokes:

- F1 or ? Show help screen
- F4 or V View application screen
- F5 or R Run (exit debug mode)
- F8 or S Step
- Q Quit

**DF**  
Display file buffer

DF{n}
The content of the specified file buffer is displayed. If \( n \) is omitted or specified as zero, the fast file buffer is displayed.

**DS** Display select buffer

\[ \text{DS} \]

The content of the specified select buffer is displayed. If \( n \) is omitted, it defaults to zero.

**F** Moves the input buffer pointer forward

The input buffer pointer is moved forward to the start of the next field. If the pointer was in the last field, it is moved to the end of the buffer.

**F;** Perform stack based arithmetic

\[ \text{F;} \text{element\{element...\}} \]

The **F;** statement performs integer arithmetic using a stack. The **element** list contains values to be added to the stack and operators to be performed against the stack values.

- **ref** A direct or indirect reference to a buffer element to be placed on the stack.
- **n** A numeric constant to be placed on the stack. The value may be preceded by **C**.
- **+** Adds the top two stack items, replacing them by the result.
- **-** Subtracts the top stack item from the next item, replacing them by the result.
- **\( \ast \)** Multiplies the top two stack items, replacing them by the result.
- **/** Divides the second item on the stack by the first item, replacing them by the truncated integer result.
- **R** Divides the second item on the stack by the first item, replacing them by the remainder value.
- **\{** Interchanges the top two items on the stack.
- **\_** Interchanges the top two items on the stack.
- **?P** Moves the top item from the stack into the primary input buffer at the input pointer position.
- **?ref** Moves the top item from the stack into the specified register location.

**F-CLEAR** Clear a file buffer

\[ \text{F-C\{} \text{LEAR} \text{\} \, n} \]

The file buffer for file \( n \) is cleared.

**F-DELETE** Delete a record from an open file

\[ \text{F-D\{} \text{DELETE} \text{\} \, n} \]

The record identified by the file and id associated with file buffer \( n \) is deleted. An error will be reported if there is no open file associated with the file buffer.

**F-FREE** Release a record lock in an open file

\[ \text{F-F\{} \text{REE} \text{\} \, n \{id\ref\}} \]

The record lock identified by the file and id associated with file buffer \( n \) is released. The record id may be specified using a buffer reference. If no id is specified or it is a null string, all locks in that file are released. An error will be reported if there is no open file associated with the file buffer.

If no file number or id are specified, all locks associated with files opened by the PROC are released.
The Command Environment

F-OPEN  Open a file

\[
\text{F-OPEN} \ n \ \{\text{DICT}\} \ \{\text{filename|ref}\}
\]
Opens the file specified by filename or by the buffer addressed by ref, associating it with file buffer n. The DICT qualifier specifies that the dictionary portion of the file is to be opened. If the file cannot be opened, the PROC continues at the next statement, otherwise this statement is skipped. All files are closed on return to the command processor.

F-READ  Read a record from an open file

\[
\text{F-READ} \ n \ \{\text{id|ref}\}
\]
The record with id specified by id or by the direct or indirect ref is read into file buffer n. If the record cannot be found, the PROC continues at the next statement, otherwise this statement is skipped. In either case, the record id will be stored as field zero of the file buffer.

F-UREAD  Read a record from an open file with an update lock

\[
\text{F-UREAD} \ n \ \{\text{id|ref}\}
\]
The record with id specified by id or by the direct or indirect ref is read into file buffer n, locking it for update. If the record cannot be found, the PROC continues at the next statement, otherwise this statement is skipped. In either case, the record id will be stored as field zero of the file buffer and the process will own the lock.

F-WRITE  Write a record to an open file

\[
\text{F-WRITE} \ n
\]
The record stored in file buffer n is written using the id stored in field zero of the file buffer.

FB  Read a record into the fast file buffer

\[
\text{FB} \ \{\text{U} \ \{\text{DICT}\} \ \text{filename|ref1|id|ref2}\}
\]
The file identified by filename or ref1 is opened to the fast file buffer and the record identified by id or ref2 is read into the buffer. The U option specifies that an update lock is required. If the file cannot be opened or the record cannot be found, the PROC continues at the next statement, otherwise this statement is skipped. Where the action fails because the file was opened but the record could not be found, the id will be stored in field zero of the file buffer and the process will own the update lock if the U option was specified.

GO  Jump to a label or a mark (Synonyms G and GOTO)

\[
\text{GO} \ \text{label|A-ref|ref|F|B}
\]
The PROC continues execution at the given position. label specifies a numeric label attached to the destination. A-ref is an A-reference used to determine the destination label. ref is a direct or indirect buffer reference to a location containing the label. F jumps forward to the next M statement in the PROC. B jumps to the location of the last M statement executed within the PROC.
GOSUB Enter a labelled subroutine

GOSUB label
Label specifies a numeric label at the start of the subroutine.
Execution continues at the given location. The subroutine may return to the statement following the GOSUB by use of RSUB.

H Add text to the active output buffer

H{text|ref}
The literal text or the content of the buffer location identified by the direct or indirect ref is added to the active output buffer.

IF Conditional execution

IF {N} condition statement
N specifies that a numeric comparison is to be performed where only the leading numeric part of the data to be tested is used.
The condition may take several alternative forms referencing an item which may be:
A-ref Data obtained using an A-reference
ref A direct or indirect buffer reference
E The value of @SYSTEM.RETURN.CODE
S n Tests whether select list is active. n defaults to 0 if omitted.
The conditions are:

item Tests that item is not blank. Used with E, this tests whether the value is negative.
#item Tests that item is blank. Used with E, this tests whether the value is not negative.
item op text|ref Compares item with unquoted literal text or a value obtained from a direct or indirect buffer reference. The operator op may be

= Equality
# Inequality
> Greater than
< Less than
] Greater than or equal to
[ Less than or equal to

If text or ref is enclosed in round brackets and the operator is = or #, it is treated as a pattern match. The text item should be enclosed in single or double quotes if it contains spaces.

If the data identified by text or ref is multivalued and the operator is = or #, the operator tests whether item appears in the multivalued data. There are two extended syntaxes available with this style of test:

IF item = AVM BVM C GO 10 VM 20 VM 30
and

IF item = AVM BVM C GOSUB 10 VM GO 20 VM XDone

The first form, applicable to GO only, jumps to one of a list of labels dependant on the value of the item. The second form takes a multivalued list of statements to be executed dependant on the value of item.
The default behaviour is that the IF command compares the two items in a case sensitive manner. The PROC.IF.NO.CASE setting of the OPTION command can be used to make the comparison case insensitive.
IH  Insert test in the active input buffer

$I\{B\}Htext[ref]\{\}\$

Copies the unquoted literal text or the data addressed by the direct or indirect buffer ref to the active input buffer at the position given by the input buffer pointer. If this is positioned at the start of a field, the entire field is replaced. If it is positioned part way through the field, the new data is appended to the portion before the input pointer position. The input buffer pointer is not moved by this operation.

For compatibility with other systems, the following additional syntax elements are recognised in PQN style Procs or in PQ style Procs executed without the PICK.PROC mode of the **OPTION** command enabled:

The \ token with no preceding space, clears the field addressed by the input buffer pointer. If the pointer is positioned part way through a field, characters before the pointer position are retained.

The \ token with a preceding space, inserts an empty field. This syntax is not recognised as a special case if the PICK.PROC mode of the **OPTION** command is enabled. Leading and trailing spaces are removed and multiple embedded are compressed to single spaces. If the B option is not present, the spaces are then converted to field marks.

IN  Input data from the terminal to the secondary input buffer

$I\{B\}N\{c\}$

The secondary input buffer is activated and the user input overwrites any existing content. All leading and trailing spaces in the input data are removed and multiple embedded spaces are compressed to a single space. If the B option is not present, all remaining spaces are then replaced by field marks. The optional prompt character c specifies an alternative to the default of a question mark and remains in effect for subsequent input until another prompt character is set.

IP  Input data from the terminal to any buffer

$I\{B\}P\{B\}\{c\}\{ref\}$

User input overwrites the location specified by the direct or indirect ref. If ref is omitted, the field addressed by the input buffer pointer in the primary input buffer is overwritten. All leading and trailing spaces in the input data are removed and multiple embedded spaces are compressed to a single space. If the B option is not present, all remaining spaces are then replaced by field marks. The optional prompt character c specifies an alternative to the default of a question mark and remains in effect for subsequent input until another prompt character is set. The prompt character must be present if ref is used.

In a PQ style Proc, entering a blank response retains the existing content of the input buffer. A PQN style Proc would clear the buffer.

IS  Input data from the terminal to the secondary input buffer

This is a synonym for IN described above.

L  Send output to a printer

$L\{text[ref](col),...\}\{+\}$
Outputs the items specified in the comma separated list. These may be quoted literal text or the data addressed by the direct or indirect buffer ref. Use of ref may be followed by an input conversion code enclosed in semicolons or an output conversion code enclosed in colons. The (col) element can be used to move to a specific column number where the leftmost column is column one. The + element suppresses the normal newline at the end of the output. The list may span multiple lines by breaking it after a comma.

LC Close printer

The printer is closed and the output is passed to the underlying print management system for printing.

LE Page eject

Starts a new page

LHDR Set page header

LHDR[<text>|ref(col)|P|T|Z|n|...]

Sets the page header using the items specified in the comma separated list. These may be quoted literal text or the data addressed by the direct or indirect buffer ref. Use of ref may be followed by an input conversion code enclosed in semicolons or an output conversion code enclosed in colons. The (col) element can be used to move to a specific column number where the leftmost column is column one. The P element inserts the page number. The T element inserts the date and time. The Z element restarts page numbering. The n element specifies a number of newlines. The list may span multiple lines by breaking it after a comma.

LN Redirect printer output to the terminal

Specifies that output from the L statement is to be directed to the terminal. This is mainly useful for debugging purposes.

M Mark

The M statement marks a location in a PROC for use by the GO F and GO B operations.

MV Move data from one location to another

MV destination source

destination is a direct or indirect reference to the buffer location to which data is to be copied. source is a list of one or more items to be copied. Each item may be direct or indirect buffer reference or a quoted literal string. A comma separating two items inserts the items as separate fields. Use of two or more consecutive commas with no source item between them skips fields in the destination. An asterisk between two items concatenates them. An asterisk after a file buffer reference as the last item in the source list copies all remaining fields from the file buffer.
An asterisk followed by a number after a file buffer reference in the *source* list copies the given number of fields from the file buffer. An underscore as the last item in the list truncates the destination by removing all fields after the last one copied.

**MVA**  Move data from one location to another as a sorted multivalued field

*MVA destination source*
*destination* is a direct or indirect reference to the buffer location to which data is to be copied. *source* is a direct or indirect buffer reference or an unquoted literal string. The source data is inserted as a new value in the multivalued destination using a left aligned ascending sort order to determine its position. The item will not be inserted if it would duplicate an existing entry in the list.

**MVD**  Delete an entry from a multivalued field

*MVD destination item*
*destination* is a direct or indirect reference to the buffer location from which the data is to be deleted. *item* is a direct or indirect buffer reference or an unquoted literal string. The multivalued destination is searched for the first occurrence of *item*, removing this entry from the list.

**O**  Output text to the terminal

*Otext{+}*
The unquoted literal *text* is displayed on the user’s terminal. The optional + token suppresses the normal newline after output.

**P**  Process the command in the primary output buffer

*P*[P] {H} {X} {W} {Ln}*
The command in the primary output buffer is passed to the command processor for execution. Any data in the secondary output buffer is queued up as data for use by the executed command. If there is any unprocessed data remaining after the command has been executed, the first field of this data is passed to the command processor for execution, using the remaining fields as data. This cycle continues until all the data has been processed. The P option displays the content of the output buffer before execution of the command. The H option suppresses terminal output by the executed command. The X option terminates the PROC after the command has been executed. The W option displays the command and prompts the user to confirm whether it should be executed. Valid replies are Y to execute the command, N to terminate the PROC without executing the command and S to skip the command but continue execution of the PROC. The Ln option sets process task lock *n* for the duration of the command. After the command has been executed, the output buffers are cleared and the primary output buffer is activated. There is an implied P command at the end of a PROC.

**Q**  Quit

*Qtext*
The PROC and all other underlying programs, paragraphs, menus, etc are terminated, displaying the optional unquoted text on the user’s terminal. The user is returned to the command prompt, executing any ON.ABORT VOC entry on the way.

RI Reset input buffers

RI[f(col)]
Used with no options, this statement clears both input buffers, resets the pointer to the start of the primary input buffer and makes this the active buffer.
The f option specifies that the primary input buffer is to be cleared from field f onwards, leaving the input buffer pointer positioned at the end of the remaining data.
The (col) option specifies that the primary input buffer is to be cleared from the given character position, leaving the input buffer pointer positioned at the end of the remaining data.

RO Reset output buffers

Both output buffers are cleared and the primary output buffer is activated.

RSUB Return from a GOSUB

RSUB{n}
Without the n option, the PROC continues execution at the statement following the last GOSUB executed.
n specifies that execution is to continue starting n lines following the GOSUB.
The RSUB statement is ignored if the PROC is not in a subroutine.

RTN Return to a calling PROC

RTN{n}
The PROC returns to the PROC from which it was called, continuing execution n lines after the [] statement that called the current PROC. If n is omitted, it defaults to 1.

S Set the input buffer pointer

S[f|ref|(col)]
Moves the input buffer pointer of the active input buffer at the specified position.
f specifies that the pointer is to be positioned at field f.
ref is a direct or indirect buffer reference used to obtain the field number.
The (col) option sets the pointer to the given character position.

SP Active the primary input buffer

The primary input buffer is activated.

SS Active the secondary input buffer

The secondary input buffer is activated.

STOFF Active the primary output buffer
The primary output buffer is activated. This statement can also be written as **STOF** or **ST OFF**.

**STON** Active the secondary output buffer

The secondary output buffer is activated. This statement can also be written as **ST ON**.

**T** Terminal output

\[
T \{ \text{element} \ldots \}
\]

Outputs each element of a comma separated list to the terminal. The elements may be:

- **text** Quoted literal text
- **ref** A direct or indirect buffer reference identifying the data to be displayed. This may be followed by an input conversion code enclosed in semicolons or an output conversion code enclosed in colons.
- \((col)\) Position the cursor to the specified column of the current line. The value of \(col\) may be given as a number or as a direct or indirect buffer reference.
- \((col,\text{row})\) Position the cursor to the specified row and column. The value of \(row\) and \(col\) may be given as a number or as a direct or indirect buffer reference.
- **B** Sounds the terminal "bell".
- **C** Clears the screen.
- **D** Pauses for one second.
- **In** Displays character \(n\) where \(n\) may be given as a number or a buffer reference.
- **L** Terminates a \(T\ldots L\) loop.
- **Sn** Emits \(n\) spaces where \(n\) may be given as a number or a buffer reference.
- **T** Starts a \(T\ldots L\) loop where the elements enclosed in the loop will be executed three times.
- **U** Moves the cursor up by one line.
- **Xn** Displays character \(n\) where \(n\) may be given as a number or a buffer reference to a two digit hexadecimal value.
- + Suppresses the normal newline after display.

The (\(col\)) and (\(col,\text{row}\)) elements can also be used to access the terminal control codes that use negative \(col\) values.

The list of elements for display can span multiple lines by breaking it after a comma.

**TR** Enable or disable tracing

**TR {ON|OFF}**

The **ON** option causes the PROC processor to display each statement before it is executed. The **OFF** option terminates trace mode. The space before the mode keyword can be omitted. If no mode is specified, tracing is enabled.

**U** Call a QMBasic program

**Uname**

The catalogued program identified by \(name\) is called. This program should take no arguments and can access the PROC buffers using @-variables. The \(U\) command is frequently used to reference Pick style "user exits". Many of these are provided in source form in the BP file of the QMSYS account and can be compiled as needed. The user exits described below are integrated into the PROC processor and always available.

**U01A6** Screen painting
**U01A6**

*action, action...*

This special user exit emulation processes successive lines of the PROC as being a comma separated list of codes that control updates to the screen. Processing continues on the next line of the PROC if the final character of the current line is a comma. The codes available are:

- *(col, row)* position cursor to specified location.
- B sound the terminal "bell".
- C clear screen.
- In display character with ASCII decimal value *n*.
- Xn display character with ASCII hexadecimal value *n*.
- "text" display *text*. The cursor is left positioned immediately following the *text*.

The *col, row and n* values may be buffer references.

**U01AD** Fetch data from a file

**U01AD**

*filename id fno action*

This special user exit emulation reads four parameters from the next line of the PROC. These may be literal values or buffer references. The record identified by *filename* and *id* is read and field *fno* is extracted from it. The case insensitive value of *action* determines the destination for this data:

- a append to alternative (non-active) output buffer.
- p append to primary input buffer.
- s append to active output buffer.
- t display on user's terminal.
- v verifies that record exists.
- va verifies that attribute *fno* exists.

If the successful, the next line of the PROC is skipped, otherwise it is executed.

**U31AD** Get QM user number

**U31AD**

*code*

This user exit gets the QM user number (equivalent to a Pick port number) and places it in a destination determined by the *code* value on the next line of the PROC:

- a alternative (non-active) output buffer.
- p primary input buffer.
- s active output buffer.
- t display on user's terminal.

**X** Exit from the PROC

**Xtext**

Displays the optional unquoted *text* and terminates the PROC, returning to the calling PROC, program, menu, etc.

**+** Add an integer value to a numeric field

**+n**
The specified numeric value is added to the field of the active input buffer identified by the input buffer pointer. Non-numeric data is treated as zero.

- Subtract an integer value from a numeric field

\[-n\]
The specified numeric value is subtracted from the field of the active input buffer identified by the input buffer pointer. Non-numeric data is treated as zero.

( ) Transfer control to another PROC

\[\{\text{DICT} \ filename \ {id} \} \{label\}\]
The PROC identified by the given \textit{filename} and \textit{id} is executed, starting at \textit{label}, or the first statement if no \textit{label} is specified. If \textit{id} is omitted, the record id is obtained from the field of the active input buffer addressed by the input buffer pointer. The buffers and pointers are not changed by this statement. Control does not return to the current PROC when the called PROC terminates.

[ ] Transfer control to another PROC

\[\{\{\text{DICT} \ filename \ {id}\}\} \{label\}\]
The PROC identified by the given \textit{filename} and \textit{id} is executed, starting at \textit{label}, or the first statement if no \textit{label} is specified. If \textit{id} is omitted, the record id is obtained from the field of the active input buffer addressed by the input buffer pointer. If no \textit{filename} or \textit{id} are specified, transfer is controlled to the same Proc at \textit{label}. The buffers and pointers are not changed by this statement. Control returns to the current PROC when the called PROC executes a RTN or X statement.
VOC Q-type records - Remote file pointers

A Q-type VOC record points to a file defined in the VOC of another account.

1:  Q  { descriptive text }
2:  Account name or pathname. Leave blank for the same account.
3:  VOC record name in target account. If blank, VOC is assumed.
4:  Server name for files accessed using QMNet

Field 2 contains the account name. If field 2 is blank, the target record is assumed to be in the same VOC file as the Q-pointer. If field 4 is blank and the user does not have account barring restrictions (see Application Level Security), the account location may be specified as a pathname.

Field 3 holds name of a VOC record in the target account. This VOC item must be either an F-type (file) or a further Q-type record. A chain of Q pointers is inefficient and should be avoided. To trap closed loops of Q-pointers, a chain with more than ten Q-pointers will cause an error when attempting to open the file.

If the remote file is on a different QM server, this is specified by putting the server name in field 4 of the VOC entry. The network address and user authentication information is defined using the SET.SERVER command.

The SET.FILE command provides an easy way to create Q-pointers.

Access to a file via a Q-pointer may fail if account barring restrictions are active for the user attempting to open the file.

A summary of Q-type VOC records may be displayed or printed using LISTQ.
VOC R-type records - Remote pointers

An R type VOC entry points to a record in another file which is constructed in the same way as an executable (M, PA, R, S or V type) VOC entry.

1: R { descriptive text }
2: File name
3: Record name
4: { Security subroutine name }

R-type VOC entries are used to:

- Move large paragraphs and menus out of the VOC as large records degrade the performance of the hashing process.
- Reference a common version of a VOC item to be used from multiple accounts.
- Add security checks prior to command execution.

The file name in field 2 must correspond to an F-type or Q-type entry in the same VOC.

The record name in field 3 is the record in the target file that holds the item to be executed.

An R-type VOC record can optionally hold the name of a catalogued security subroutine in field 4. This subroutine can be used to determine whether the user is to be allowed to execute the command pointed to by the R-type record. If the validation fails or the subroutine cannot be found in the catalogue a message is displayed:

This command is restricted (verb)

A summary of R-type VOC records may be displayed or printed using LISTR.
VOC S-type records - Sentences

Where a particular command is executed frequently, it may be useful to store it as a sentence.

1: S { descriptive text }
2: Sentence text

A sentence is a command containing a verb and, optionally, its arguments. Sentence names may be entered in response to the command prompt in the same way as a verb. Any arguments following the sentence name on the command line entered at the keyboard will be appended to the sentence retrieved from the VOC.

Field 1 of a VOC sentence record must commence with a letter S. Field 2 holds the text of the sentence. This text replaces the sentence name in the current command and parsing continues with the first word of the substituted sentence.

Where a sentence is very long it may be broken into multiple lines within the VOC record by terminating all but the final line with an underscore character. When the sentence is executed, the lines are merged, replacing the underscore with a single space.

Any additional text following the sentence name in a command that starts the sentence will be appended to the sentence expansion retrieved from the VOC. For example, the EDIT.LIST command is actually a sentence stored as

1: S
2: ED $SAVEDLISTS

Typing

EDIT.LIST MYLIST

actually executes the command

ED $SAVEDLISTS MYLIST

This automatic appending of additional text in the command makes stored sentences very useful as the start of commands but prevents effective use of some inline prompt control codes in the sentence expansion.

A summary of S-type VOC records may be displayed or printed using LISTS.
VOC V-type records - Verbs

A V type (verb) record defines a command name and determines the QM component that will be used to process the command.

1: V { descriptive text }
2: Dispatch code
3: Processor
4: { Qualifying information }
5: { Security subroutine }
6: { Parenthetical option letters }
7: { Parenthetical option keywords }

The dispatch code identifies the type of processor referenced by field 3. It may be:

- **CA**: A catalogued verb. Field 3 holds the catalogue name of the function to be executed. For system supplied verbs, field 4 may also be significant and should not be altered.
- **CS**: A locally catalogued function program. This format allows a QMBasic program to CALL a function that is in the compiler output file rather than in the catalogue.
- **IN**: An internal verb. Field 3 holds an identifying number which determines the action of the verb.
- **OS**: An operating system command. QM will execute an operating system command made up from the contents of field 3 of the VOC record (which may be null) followed by the remainder of the current sentence after the verb.

Users may add their own V-type records for catalogued programs (usually by use of the CATALOGUE command) or make copies of standard records to provide synonyms for other verbs.

The optional qualifying information in field 4 will be copied to the @OPTION variable when executing the verb.

A V-type VOC record can optionally hold the name of a catalogued security subroutine in field 5. This subroutine can be used to determine whether the user is to be allowed to execute the command. If the validation fails or the subroutine cannot be found in the catalogue a message is displayed:

```
This command is restricted (verb)
```

Fields 6 and 7 allow implementation of Pick style parenthetical options. Field 6 is a multivalued list of option letters and field 7 is a corresponding multivalued list of the equivalent QM keywords. When the verb is encountered by the command processor, the parenthetical options are replaced by the corresponding keywords.

A summary of V-type VOC records may be displayed or printed using LISTV.
VOC X-type records - Miscellaneous storage

X-type VOC items are miscellaneous data storage records which may be used in any way the application designer wishes.

1: X { descriptive text }
2: user data

Fields 2 onwards are available for data storage. Users may freely create X type records for their own purposes but should avoid names containing $ signs as these may clash with system defined records.

The following X-type VOC records have special meaning in QM:

- **$ACCOUNT.ROOT.DIR**: Default location for new accounts in `CREATE.ACCOUNT`. Also used by `RESTORE.ACCOUNTS`.
- **$BASIC.IGNORE**: Contains pattern match templates for records to be ignored by the `BASIC` command.
- **$BASIC.OPTIONS**: Sets default options for the `BASIC` command.
- **$BIGLTRS**: Only relevant in the QMSYS account, this record configures how `BLOCK.TERM`, `BLOCK.PRINT` and print job banner pages form the large character outlines.
- **$COMMAND.STACK**: Controls saving of the command stack on leaving QM.
- **$DEBUG.OPTIONS**: Specifies default options for the QMBasic debugger.
- **$ED.OPTIONS**: Sets default modes for the `ED` editor.
- **$ENCR**: Controls automatic encryption in `CREATE.FILE`.
- **$INDEX.PATH**: Specifies a default location for index subfiles in `CREATE.INDEX` and `MAKE.INDEX`.
- **$PCL**: Sets default parameters for PCL printers in `SETPTR`.
- **$PRIVATE.CATALOGUE**: Specifies the location of the private catalogue. The American spelling may be used.
- **$QUERY.DEFAULTS**: Sets default options for the query processor. Also valid in dictionaries.
- **$RELEASE**: Contains the release level of QM. Also holds configuration options for the command stack editor.
- **$REPLICATE**: Controls account replication.
- **$SCRB.FILE**: Sets the default name of the screen definitions file for the `SCRB` screen builder.
- **$SETPTR.DEFAULTS**: Inserts default options into a `SETPTR` command.
- **$SPOOLERS**: Only relevant in the QMSYS account, this record contains configuration data for the print spooler on Linux/Unix systems.
- **$VOC.PARSER**: Specifies processors associated with user defined VOC record types.
&SED.OPTIONS&

Sets options for the SED editor.
Security subroutines

An R-type or V-type VOC entry can optionally include the name of a catalogued security subroutine in field 4 (R-type) or field 5 (V-type). This subroutine can be used to determine whether the user is to be allowed to execute the command.

The security subroutine is written using QMBasic. A simple subroutine that prompts for a password is shown below.

```
SUBROUTINE SECURITY(OK, VERB, REMOTE.FILE, REMOTE.ID)
  PROMPT ''
  FOR I = 1 TO 3
    DISPLAY 'Enter security password: ' :
    INPUT PASSWORD HIDDEN
    IF PASSWORD = 'FSKJJ' THEN RETURN @TRUE
    DISPLAY 'Incorrect password'
  NEXT I
  RETURN @FALSE
END
```

The arguments to this subroutine are:

- **OK**
  - Used to return the result of the validation. This should be set to True if the command is to be allowed, False if it is to be rejected.

- **VERB**
  - The name of the R/V-type VOC entry being processed.

- **REMOTE.FILE**
  - The name of the file containing the remote item to be executed. This is a null string for a security subroutine referenced from a V-type VOC record.

- **REMOTE.ID**
  - The record of the remote item to be executed. This is a null string for a security subroutine referenced from a V-type VOC record.

If the validation fails or the subroutine cannot be found in the catalogue a message is displayed:

```
This command is restricted (verb)
```

The security subroutine mechanism can also be used for other purposes such as auditing use of specific commands, diagnostic traps, etc.
2.6 Inline Prompts

Inline prompts provide a means to prompt for data needed by a sentence or paragraph when it is executed. Except as shown for the Q control element below, the DATA command does not affect inline prompting which always takes its response from the keyboard. There are also variants on inline prompts that retrieve data from other sources, providing a generalised way to substitute variable items into a command.

An inline prompt has the general form

$$\llbrace \{control\}, text \{, check\}\rrbrace$$

where

- **control** determines the way in which the prompt is displayed and how it is actioned on subsequent execution of the same statement or another with the same **text**. Control codes are generally case insensitive. A single inline prompt may have multiple comma separated control elements or none.

- **text** is the prompt text to be displayed. An equals sign is automatically added to the end of the prompt text. The text should be enclosed in single quotes, double quotes or backslashes if it contains characters that might otherwise lead to mishandling of the prompt such as a comma.

- **check** is used to check whether the response to the prompt is valid. If omitted, no checking is performed.

The **control** options are of two types; display controls and response controls. Both are optional and may be intermixed.

A display control may contain any of the following items. Multiple items may be concatenated, separated by commas, and are performed in the order in which they appear.

- **@((col, row))** specifies the display position for the prompt text. The QMBasic @() function variants with a negative value for the first (or only) argument are also supported.

- **@BELL** sounds the audible warning. The **BELL OFF** command will suppress this action.

- **@CLR** Clears the display.

- **@TOF** Positions to the top left of the display. This is equivalent to use of @((0,0)).

A response control consists of one of the items listed below. If omitted, the prompt is actioned only for the first occurrence of a prompt with the given **text**. Subsequent execution of the same inline prompt or another with the exactly same **text** will not cause a prompt to be displayed but will use the response to the previous prompt. All prompt responses are discarded on return to the command prompt. The **CLEAR.PROMPTS** command can be used to discard all inline prompt responses from within a paragraph.

- **A** Always prompt. This is usually needed on the first occurrence of each prompt in a loop so that each iteration of the loop prompts again.
<table>
<thead>
<tr>
<th>Control Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cn</td>
<td>Used in paragraphs to take the $n$'th token of the sentence that started the paragraph as the response to the prompt. No prompt text will appear unless the implicit response fails to meet the check conditions. The C control code has several extended forms:</td>
</tr>
<tr>
<td>Cmn</td>
<td>Returns tokens $m$ to $n$.</td>
</tr>
<tr>
<td>Cn+</td>
<td>Returns tokens $n$ onwards.</td>
</tr>
<tr>
<td>C#</td>
<td>Returns the number of tokens in the command line.</td>
</tr>
</tbody>
</table>

All formats of the C control code except C# may include a default value. For example, `<C4:SALES>`

The default value will be applied if the prompt would otherwise return a null string.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHANGE(str, old, new)</td>
<td>Change all occurrences of substring <code>old</code> in <code>str</code> with <code>new</code>. Argument values may be enclosed in quotes if necessary to avoid any syntactic ambiguity.</td>
</tr>
<tr>
<td>CONFIG(x)</td>
<td>Returns the value of the named configuration parameter.</td>
</tr>
<tr>
<td>F(file, key {, field {, value {, subvalue}}} })</td>
<td>Use record <code>key</code> of the data portion of the named <code>file</code> as the response to the prompt. The optional <code>field</code>, <code>value</code> and <code>subvalue</code> allow extraction of a particular part of the record.</td>
</tr>
<tr>
<td>ICONV(str, code)</td>
<td>Apply input conversion <code>code</code> to <code>str</code>. Argument values may be enclosed in quotes if necessary to avoid any syntactic ambiguity.</td>
</tr>
<tr>
<td>In</td>
<td>Used in paragraphs to take the $n$'th token of the sentence that started the paragraph as the response to the prompt. If this is a null string or the implicit response fails to meet the check conditions, an input prompt appears.</td>
</tr>
<tr>
<td>Ln</td>
<td>Extracts the next item from select list $n$. If $n$ is omitted, it defaults to zero. When the select list is exhausted, a null string is returned. This control code does not need to be used with the A control code in a loop as it always reads the next list entry.</td>
</tr>
<tr>
<td>OCONV(str, code)</td>
<td>Apply output conversion <code>code</code> to <code>str</code>. Argument values may be enclosed in quotes if necessary to avoid any syntactic ambiguity.</td>
</tr>
<tr>
<td>Q</td>
<td>Allows an inline prompt response to be taken from the DATA queue.</td>
</tr>
<tr>
<td>R</td>
<td>Prompt repeatedly for input, concatenating data with an intervening space until a blank line is entered.</td>
</tr>
<tr>
<td>R(string)</td>
<td>Prompt repeatedly for input, concatenating the responses with <code>string</code> between responses until a blank line is entered. The <code>string</code> may not include field mark characters. An abort will occur if a field mark is specified as part of <code>string</code>.</td>
</tr>
</tbody>
</table>
$n$
Take the $n$'th token of the sentence entered at the command prompt as the response to the prompt. If this is a null string or the implicit response fails to meet the check conditions, an input prompt appears.

**SUBR**(name)
Execute catalogued QMBasic function *name*, returning the result of this function as the value of the inline prompt.

**SUBR**(name, arg1, arg2)
Execute catalogued QMBasic function *name*, passing in the given arguments and returning the result of this function as the value of the inline prompt. Up to 254 arguments may be specified. These may be enclosed in quotes if necessary to avoid any syntactic ambiguity.

**SYSTEM**(n)
Returns the value of the QMBasic **SYSTEM**(n) function.

**U**
Converts the data entered in the response to the prompt to uppercase. Note that this control must appear before any other control options to which it is to apply.

[@var]
The name of an @-variable, including user defined names (see the **SET** command), may be used to retrieve the value of the given variable. A default value may be applied by use of a prompt of the form:

<<[@name:value]>>
The default value will be applied if the prompt would otherwise return a null string.

[@$var]
The name of an operating system environment variable may be used to retrieve the value of the given variable. A default value may be applied by use of a prompt of the form:

<<[$name:value]>>
The default value will be applied if the prompt would otherwise return a null string.

**Note:** The C and S control may appear to be backwards. The C control bases its result on the content of the @SENTENCE variable and the S control bases its result on the content of the @COMMAND variable. These control elements were originally defined in another multivalue product and QM preserves this apparently incorrect implementation for compatibility.

The **Cn** and **In** control codes are only useful in paragraphs. A command that references a paragraph (PA-type VOC entry) may include additional text that can be accessed using these control codes. In a command that references a stored sentence (S-type VOC entry), any additional text following the sentence name is added to the end of the sentence expansion, making effective use of these control codes impossible. There is no reason why a paragraph cannot contain only a single sentence if these control codes are to be used.

4.0-0
The \texttt{<<@var>>} construct allows \texttt{@} variables to be inserted into any command, simplifying paragraph structure. For example:

\begin{verbatim}
SAVE.LIST LIST.<<@USERNO>>
\end{verbatim}

could be used to save a select list with a name that includes the user number to ensure uniqueness for the duration of the user's QM session.

The \texttt{check} code performs simple data validation and is either a \texttt{pattern match template} or an input conversion code. Multiple patterns can be specified separated by the word \texttt{OR} with a single space either side. For example,

\begin{verbatim}
3N'-4N OR 4N'-3N
\end{verbatim}

would accept numbers of the form 123-4567 or 1234-567.

Input conversion codes must be enclosed in brackets. Any of the codes that can be used with the \texttt{ICONV()} function may be used. The check is considered successful if no conversion error occurs. The value returned by the prompt is the actual text entered, not the result of the conversion.

Entry of QUIT at the keyboard in response to an inline prompt will abort and return to the command prompt. The \texttt{@ABORT.CODE} variable will be set to 2 as for QUIT entered at other prompts.

Inline prompts are expanded as the first stage of processing a command. This has two important effects:

- An inline prompt in a \texttt{comment} line will be evaluated. This slightly strange effect can be useful as a means to perform all prompts at the start of the paragraph rather than as they are needed, perhaps much later.

- An \texttt{IF} command in a paragraph where the conditioned statement includes an inline prompt will display the prompt before determining whether the condition is True. The paragraph can always be structured in a way to avoid this behaviour.

A single command may contain multiple inline prompts. These will be evaluated left to right. Nested inline prompts will be evaluated from the inside outwards.

The response to an inline prompt may not include the left or right double angle bracket symbols (<< and >>) or field marks. Entering a response containing one of these will cause the prompt to be repeated.

When used as an item to test in an IF command, the prompt will usually need to be enclosed in quotes so that a response that contains spaces or reserved keywords does not cause the command parsing to fail. For example

\begin{verbatim}
IF <<Customer number>> = "" THEN STOP
\end{verbatim}

should be written as

\begin{verbatim}
IF "<<Customer number>>" = "" THEN STOP
\end{verbatim}

\textbf{Examples}

\begin{verbatim}
PA
SELECT ORDERS SAVING UNIQUE CUST.NO_ WITH DATE AFTER <<Start date>>
LIST CUSTOMERS NAME_
   HEADING "Clients ordering after <<Start date>>"
\end{verbatim}
The above paragraph uses two identical inline prompts. The first asks the user to enter a start date for a SELECT operation. The second prompt will not repeat the request as the answer is already known. Note that the prompt text can contain spaces and that the inline prompt substitution occurs even though it is part of a quoted string.

```
PA
* <<<Target file>>>
RUN OVN.PROCESS
OVN.RPT <<<Target file>>>
```

In this example, the inline prompt is in a comment. Although the prompt will be issued at the start of the paragraph, the result is not used until the final command. If the OVN.PROCESS program takes a considerable length of time to execute, this technique allows the prompt to be answered early, without having to wait for the program to complete.

```
PA
* <<<I2,Target file>>>
RUN OVN.PROCESS
OVN.RPT <<<Target file>>>
```

By adding the I2 control option to the previous example, the paragraph will take the target file name from the second word on the command line if present, prompting if it is not given.

```
PA
LOOP
  IF <<<A, Customer>> = "" THEN STOP
  LIST ORDERS WITH CUST.NO = <<<Customer>>>
REPEAT
```

This paragraph uses a loop in which the user is prompted to enter a customer number. If this is a null string, the paragraph terminates. Otherwise it lists the ORDERS file records for this customer. Note the use of the A option on the first prompt in the loop so that it is repeated on each cycle of the loop. Without this, the paragraph would list the orders for the same customer continuously until stopped by user action.

```
PA
LOOP
  IF <<<L,ID>> = "" THEN STOP
  MYPROG <<<ID>>
REPEAT
```

This paragraph uses a loop to walk through the default select list, executing MYPROG for each entry taken from the list.

```
SAVE.STACK "<@LOGNAME>>"
DISPLAY Domain is "<USERDOMAIN>>
```

These two commands show use of the inline prompt mechanism to insert the value of system variables into commands. The first uses an internal QM @variable to retrieve the user’s login name, the second accesses the operating system USERDOMAIN environment variable. Note the use of quotes in the first command to allow for the possibility of the user name containing a space.
DISPLAY Licence number "SYSTEM(31)"

This example uses the inline prompt mechanism to access the QM licence number via the QMBasic `SYSTEM()` function.

See also:
CLEAR.PROMPTS
2.7 Pattern Matching

Pattern matching is a way to test whether data has a particular structure. It can be used for data validation and as a means to extract the part of a data item that matches against a specified element of the pattern. The pattern matching operations are the query processor LIKE and UNLIKE keywords, the QMBasic MATCHES operator, MATCHFIELD() and MATCHES() functions, and the M search of ED. All of these compare a character string with a pattern template.

Pattern matching breaks the character set into three classes of character, each represented by a character type code:

- **A** Alphabetic, upper and lowercase A - Z
- **N** Numeric, digits 0 - 9
- **X** Any character, including alphanumerics

On ECS mode systems, determination of character class is based on the character map in use.

There are also three ways to specify how many characters are present:

- **4** Exactly 4 characters
- **2-7** Between 2 and 7 characters
- **0** Any number of characters, including none

The template consists of up to a maximum of 30 concatenated elements formed from pairs of lengths and character type:

- **0X** Zero or more characters of any type
- **nX** Exactly \( n \) characters of any type
- **n-mX** Between \( n \) and \( m \) characters of any type
- **0A** Zero or more alphabetic characters
- **nA** Exactly \( n \) alphabetic characters
- **n-mA** Between \( n \) and \( m \) alphabetic characters
- **0N** Zero or more numeric characters
- **nN** Exactly \( n \) numeric characters
- **n-mN** Between \( n \) and \( m \) numeric characters
- **"string"** A literal string which must match exactly. Either single or double quotation marks may be used. Use of the $MODE NOCASE.STRINGS compiler directive makes the comparison case insensitive.

The values \( n \) and \( m \) are integers with any number of digits. \( m \) must be greater than or equal to \( n \).

The 0X code is a wildcard that matches against anything. It has a commonly used synonym:

- **...** Zero or more characters of any type

The 0A, nA, 0N, nN and "string" patterns may be preceded by a tilde (~) to invert the match condition. For example, ~4N matches four non-numeric characters such as ABCD (not a string which is not four numeric characters such as 12C4).

A null string matches patterns ..., 0A, 0X, 0N, their inverses (~0A, etc) and ".".
The 0X and \( n-mX \) patterns match against as few characters as necessary before control passes to the next pattern. For example, the string ABC123DEF matched against the pattern 0X2N0X matches the pattern components as ABC, 12 and 3DEF.

The 0N, \( n-mN \), 0A, and \( n-mA \) patterns match against as many characters as possible. For example, the string ABC123DEF matched against the pattern 0X2-3N0X matches the pattern components as ABC, 123 and DEF.

A pattern may contain unquoted literal elements so long as they do not cause ambiguity. Note that each character will be treated as a separate literal element such that a pattern

\[ 3AXYZ3A \]

has five elements and will match a string that is formed from three letters followed by the three literal characters X, Y, Z and a further three letters. The significance of the literal characters being treated as separate elements comes with \texttt{MATCHFIELD()} and \texttt{PARSE()}.

The \textit{template} string may contain alternative patterns separated by value marks. The source data will match the overall pattern if any of the pattern values match. If a match is found, the \texttt{INMAT()} function can be used to retrieve the value position within the pattern that matched.

The \texttt{MATCHESS()} function can be used to compare each element of a dynamic array with a pattern, returning a equivalently structured dynamic array of True/False values. Note the spelling of this function with the trailing S to “pluralise” the name in the same way as other multivalue function names.

**Examples**

"A123BCD" would match successfully against patterns of

- 1A3N3A
- 1A1-3N3A
- 'A'1-3N3A
- 0A0N0A
- 1A...3A
- 1A-3A3A

and many more

It is often acceptable to omit the quotes around literal components. The above example would also match

\[ A1-3N3A \]

There is no confusion between the leading A as a literal or as a character type as it is not preceded by a length value. It is, however, recommended that the quotes should be included. Omitting the quotes in a pattern used in the \texttt{MATCHFIELD()} function may affect the function's behaviour as each character of the literal will be counted as a separate component of the pattern.

A program might need to test whether data entered by a user is a non-negative integer (whole number) value. The QMBasic \texttt{NUM()} function can be used to test for numeric data but this would allow fractional or negative values. Testing against a pattern of "1-4N" would allow only integer values in the range 0 to 9999. To remove the upper limit, a pattern of 1N0N tests for one digit followed by any number of further digits, including none.
2.8 Printing

QM applications do not drive printing devices directly. Instead they reference numbered print units with no knowledge of where the output will actually go. This leads to a very flexible printing system where the output can be sent to a printer, a file or the user’s screen. QM uses the underlying Print Manager on Windows or the operating system spooler on other platforms to perform output to printer devices.

Each QM session has its own pool of print units, numbered from 0 to 255. In most cases, if a print unit is not specified in a command, printer 0, the default printer, is used. Application developers are free to use these print units in any way that meets their needs. They might correspond to different printers, different paper types on the one printer, selection of portrait or landscape mode, etc. Although it is unlikely, all 256 print units can be used simultaneously.

The QMBasic PRINT statement directs its output to printer 0 unless the ON clause is used to specify some other printer:

```
PRINT ON 4 "Invoice Summary"
```

Within QMBasic programs printer 0 is treated as a special case. If the program has not used the PRINTER ON statement (or the LPTR qualifier to the RUN command), output to printer 0 is actually sent to the user’s screen rather than the printer. This allows an application to use either the screen or the default printer simply by choosing whether to execute the PRINTER ON statement rather than having to implement two alternative paths for every place that performs output.

QMBasic programs can also reference print unit -1 as a synonym for the user's screen.

Two Printing Models

There are two related methods by which the physical destination of printed output can be set. For compatibility with Information style multivalue products, QM supports the SETPTR command to set the characteristics of a print unit, including its destination.

Alternatively, for compatibility with Pick style products, a set of printer characteristics can be associated with a numbered form queue by use of the SET.QUEUE command which saves the characteristics in the file system for future reference by the SP.ASSIGN command.

The fundamental difference between the two approaches is that SETPTR directly specifies printer characteristics that take immediate effect whereas SET.QUEUE defines characteristics that can be set later by referencing the form queue number in an SP.ASSIGN command. Internally, SP.ASSIGN is simply executing a SETPTR command based on the characteristics previously defined using SET.QUEUE. The two printing models can be mixed in any way the application designer wishes.

Setting Print Unit Characteristics with SETPTR

Unless otherwise defined, print unit 0 is directed to the system's default printer and all other print units are directed to the $HOLD file. Almost all applications will need to modify this default behaviour by using the SETPTR command. This command is frequently executed from the MASTER.LOGIN paragraph in the QMSYS account (to affect all users), from the LOGIN paragraph of a specific account (to affect only users of that account), or from within the application.

The SETPTR command defines the shape of the printed page (width, depth, margins), the destination and various options relating to the treatment of the output:

```
SETPTR unit {, width, depth, top.margin, bottom.margin, mode {, options } }
```
A print unit can operate in several modes:

**Mode 1** directs the output to the underlying operating system print processor, usually to send it to a physical printer.

**Mode 3** formats the data ready for printing but directs it to a record in the $HOLD file from where it may subsequently be viewed on the screen with a suitable editor or sent on to a printer when required. The hold file is most commonly used to defer printing until a process has completed, to gather diagnostic output, or for testing. The name of the file used by mode 3 can be set using the AS option of the SETPTR command which includes the ability to add a rotating sequence number for generate a different name for each output job. This sequence number can be determined using the QMBasic GETPU() function or the @SEQNO variable.

**Mode 4** directs the output to the stdout (standard output) file unit.

**Mode 5** buffers the data in the $HOLD file and then sends it to the terminal when the printer is closed, prefixing it with the control code to enable the terminal auxiliary printer port and disabling this port on completion of the print. This feature relies on the mc5 (aux on) and mc4 (aux off) terminfo items being set correctly.

**Mode 6** combines the actions of modes 1 and 3, creating a file and also printing the data.

A print job commences when the first line of output is sent to the printer and normally terminates when the program closes the print unit either explicitly or implicitly by returning to the command processor. It is possible to merge output from several successive print programs into one job by use of the PRINTER command. The KEEP.OPEN option used before output commences followed by the CLOSE option after the final program completes treats the entire sequence as a single print job.

**Pick Style Form Queues**

Although Pick style systems support the ON clause of the Basic PRINT statement, applications that need only one printer at a time more usually determine the output destination by selecting a numbered form queue. As an aid to migration, QM provides limited support for Pick style form queues by use of the SP.ASSIGN command. Internally, QM needs to relate form queue numbers to the equivalent SETPTR options and this is managed by a mapping file, $FORMS, using the SET.QUEUE command.

The SET.QUEUE command is very similar to SETPTR but instead of assigning the specified characteristics to a numbered print unit, they are stored in the $FORMS file with a numeric id that corresponds to the form queue number. Subsequent use of the SP.ASSIGN command picks up the form queue details from the $FORMS file and applies these to printer 0 by internal use of SETPTR. The R option of SP.ASSIGN allows the form characteristics to be applied to a different print unit.

Whereas each QM process has its own set of 256 numbered print units, form queue numbers and their settings are normally shared across all QM sessions because the same $FORMS file is visible to all accounts. By creating alternative $FORMS files and modifying the VOC F-pointer, it is possible to maintain separate form definitions for specific accounts or groups of accounts.

For more information, see the SP.ASSIGN and SET.QUEUE command descriptions.

**Printing on Windows**

Windows defines two alternative printing interfaces. The graphical device interface (GDI) allows a Windows application to construct complex text and graphics images whereas the non-GDI mode (known in QM as raw mode) is a much simpler interface that permits only simple text output. QM uses the raw mode by default though, for compatibility with older releases, the GDI configuration parameter can be used to make GDI the default. QM does not provide any functions to generate
GDI graphics. When using GDI mode, the printer's default font can be replaced by a fixed pitch font of a given size by use of the FONT.SIZE option to `SETPTR`.

Some options of the `SETPTR` command are applicable only to one or other of the two modes. Also, some options may not be supported by all Windows print drivers. In most cases, inapplicable options are simply ignored.

### Printing on Other Platforms

QM normally uses the underlying `lp` command to print data on these platforms though this can be modified by use of the `SPOOLER` configuration parameter or the SPOOLER option of the `SETPTR` command. `SETPTR` options that are not applicable are ignored.

Use of print spoolers other than `lp` may require QM to use different options to the command to pass the printer name, copy count, etc. To support this, there is an X-type record in the VOC file of the QMSYS account named `$SPOOLERS`. This has six associated multivalued fields such that:

- **Field 2**: Spooler name
- **Field 3**: Option to specify printer name
- **Field 4**: Option to specify copy count
- **Field 5**: Option to specify banner text
- **Field 6**: Option to set spooler options
- **Field 7**: Option to set landscape mode

In each case, the variable data to be inserted into the option is represented by a percent sign (`%`). The `$SPOOLERS` record released with QM contains a definition for use of `lp`. This has

- **Field 2**: `lpr`
- **Field 3**: `-P %`
- **Field 4**: `-K %`
- **Field 5**: `-T "%"`
- **Field 6**: `-Z "%"`
- **Field 7**: `-Z "landscape"`

### Printing to a File

The `SETPTR` command can be used to direct output to a record in the `$HOLD` file or to any specific pathname on the server system. Hold file entries have a default name of `Pn` where `n` is the print unit number but this can be modified in `SETPTR` to use a different name and/or to add a rolling sequence number to the name.

When a print unit is directed to a file, a check is made to see if there is a catalogued subroutine named `HOLD.FILE.LOGGER` and, if there is, this is called when the file is opened and again when it is closed. This subroutine can be used, for example, to build a log of hold file entries or to take some action after the file has been closed. The subroutine takes three arguments: the print unit number, a flag indicating if this is an open (1) or a close (0), and the pathname of the file being created.

### Print Prefix Files

The `PREFIX` option of the `SETPTR` command can be used to specify the pathname of a file containing printer initialisation commands. The content of this file is sent to the printer before the first output from the application. A typical use of a prefix file might be to select a paper tray.

### PCL Printer Support
The printing system of QM includes features for greater control of PCL printers. These include font selection, basic graphics and enhanced report formats. The PCL features are enabled by including the PCL option in the SETPTR command when defining the printer characteristics.

By default, a PCL printer will print in Courier font at 10 characters per inch and 6 lines per inch. The SETPTR command includes options to specify alternative values for the character and line spacing. The paper size defaults to A4 but can also be amended using SETPTR. The LANDSCAPE option will rotate the page through 90 degrees. Default values for the PCL parameters can also be set using a $PCL VOC record. For more details, see the SETPTR command.

The query processor also has special support for PCL printers in report generation commands (e.g. LIST). Page headings, footings and breakpoint values are printed in bold face. The BOXED option draws a box around the page and separates the heading and footing from the text with horizontal lines.

**Graphical Overlays**

The OVERLAY option of the SETPTR command can be used to specify the name of a catalogued subroutine that will be called at the start of each page of output and can be used to emit printer specific control codes to draw a graphical overlay on the page. The OVERLAY option of the query processor reporting commands performs the same action but applies only to the report in which it is used.

In either usage, the catalogued subroutine takes a single argument which is the print unit number. Any control strings output by this subroutine should be emitted using the PRINT statement, normally with the trailing carriage return/line feed suppressed. For PCL printers, it is recommended that the standard QMBasic PCL control string functions should be used.

**Example**

```qmbasic
subroutine overlay(pu)
$catalog overlay
$include pcl.h

s = pcl.save.csr()  ;* Save cursor position
s := pcl.box(0,0,2320,3300,2,10) ;* Draw box
s := pcl.restore.csr()  ;* Restore cursor position
s := pcl.left.margin(1)  ;* Left margin column 1
print on pu s :
return
end
```

The above subroutine draws a box around an A4 sized page on a PCL printer. Note how it saves the cursor position to ensure that subsequent application output appears at the correct place.

Because the subroutine is called before any other output to the page, it is possible for the subroutine to make other changes to the page settings. Note in the above example how the left margin is indented to bring the application output away from the left edge of the box.

**Commands Relating to Printing**

- **CLEAN.ACCOUNT**: Clears $HOLD and other system temporary files.
- **PRINTER**: Various printer control operations
- **SET.QUEUE**: Define a form queue
- **SETPTR**: Display or set print unit characteristics
SP ASSIGN
Set printer parameters using form queue emulation

SPOOL
Sends record(s) to the printer

**Query Processor Keywords Relating to Printing**

FOOTING
Set page footing

HEADING
Set page heading

LPTR
Direct output to a printer (applies to many commands)

**QMBasic Statements and Functions Relating to Printing**

FOOTING
Set page footing

HEADING
Set page heading

GETPU()
Retrieve print unit characteristics

PAGE
Force a new page

PRINT
Emit data to a print unit

PRINTER CLOSE
Close a print unit

PRINTER OFF
Direct print unit 0 to the screen

PRINTER ON
Direct print unit 0 to the printer

PRINTER RESET
Resets the default print unit

PRINTER DISPLAY
Direct printer output to the screen

PRINTER FILE
Direct printer output to a file

PRINTER NAME
Direct printer output to a named printer

SETPU
Set print unit characteristics
2.9 Dates, Times and Epoch Values

Dates

Dates are usually stored as a number of days since 31 December 1967 (day zero). All dates after that point are represented by positive numbers. All dates before that point are represented by negative numbers. This form of date is used by all multivalue databases and means that there was no issue with the millennium (day 11689). The multivalue world had its own date crisis on 18 May 1995 (day 10000) when developers discovered that they had stored the date as four characters of a composite record id or were sorting dates as character strings rather than numbers such that 17 May 1985 (day 9999) came after 18 May 1995. This potential problem still applies to any application that handles historic dates but the advantages of working with a simple day number internally far outweigh any disadvantages.

A user can see the current date in its internal or external form by using the DATE command. This command can also translate an arbitrary internal or external form date to its counterpart.

A programmer can access the current date using the DATE() function or the @DATE variable. The difference between the two is that DATE() returns the internal form of the date when the function is executed whereas @DATE is the internal form of the date when the currently executing command began.

The D conversion code can be used by applications to transform dates between their internal and external form. This conversion code can decode many elements from an internal date including day of the month, day of the week as a number (Monday = 1, Sunday = 7), day of the week as a word, Julian date (days into year), month as a word or a number, ISO week number, quarter of the year, and year. The day of the week as a number can also be calculated simply by taking the remainder from dividing the day number by seven.

Times

Times are usually stored as a number of seconds since midnight (0 to 86399).

A user can see the current time in its internal or external form by using the TIME command. This command can also translate an arbitrary internal or external form time to its counterpart.

A programmer can access the current time using the TIME() function or the @TIME variable. As described above for dates, the difference between the two is that TIME() returns the internal form of the time when the function is executed whereas @TIME is the internal form of the time when the currently executing command began.

The MT conversion code can be used by applications to transform times between their internal and external form.

Composite Dates and Times

Sometimes, an application developer may find it useful to have a composite item that represents both the date and time. One way to do this is to store it as the day number * 86400 plus the time, a value that equates to seconds since midnight between 30 and 31 December 1967. QM provides a function, SYSTEM(1005), to return the current time in this form.

The QMBasic TIMEDATE() function returns a combination of the current date and time in text format.
Epoch Values

Today’s business applications are frequently used by networked users who may span time zones. The date and time functions mentioned above all return data appropriate to the time zone of the QM process for the user running the application. This time zone is initially determined by the setting of the TZ environment variable at the operating system level on entry to QM but can be modified using the TIMEZONE private configuration parameter or the QMBasic SET.TIMEZONE statement. The time zone can differ from one user to another, thus, if two users in different time zones evaluate the TIME() function simultaneously, the returned value will be different. In some cases, storing a date or time that is appropriate only to the user who saved it may be meaningful but it is more likely that we need a way to store dates and times in a time zone independent manner that represents a moment in time. A value stored in this form can then be converted to the local time zone of the user later, giving a local representation of that moment in time such that a single stored time value may be represented differently on the screen for different users.

The underlying principle of this extension to the multivalue date and time handling is that we store the date/time as an epoch value. This is the number of seconds since the start of January 1970 UTC (closely related to GMT) and is a concept that is widely used in other computer systems. Because epoch values are independent of the user’s time zone, the same data can be used unambiguously by all users of the system, regardless of where they are in the world.

In some 32-bit operating systems, epoch values are defined to be 32 bit signed integers. This leads to a limitation that they can only represent times between 13 December 1901 and 19 January 2038. This is a widely recognised issue, potentially more widespread than the “millennium bug”, that will require changes to operating systems and run time libraries.

More seriously, Windows support for time zones at the application level is very poor. Although QM’s epoch handling functions will operate to some extent, the time zone name must include the offset from GMT (which implies that it cannot automatically resolve daylight saving time for an arbitrary date) and Windows is incapable of processing dates before 1 January 1970 as epoch values.

QMBasic provides several functions to work with epoch values:

epoch = EPOCH()
Returns the current epoch value.

date = MVDATE(epoch)
Returns the date part of an epoch value as a day number relative to 31 December 1967 in the user’s time zone.

time = MVTIME(epoch)
Returns the time part of an epoch value as a number of seconds since midnight in the user’s time zone.

time.string = MVDATE.TIME(epoch)
Returns the date and time values as a string with an underscore between the two elements.

epoch = MVEPOCH(time.string)
Returns the epoch value for the given time string in the form returned by MVDATE.TIME().

epoch = MVEPOCH(date, time)
Returns the epoch value for the given date and time values.

The F conversion code provides the combined facilities of the D and MT conversion codes plus several features that only apply to epoch values.

Linux/Mac Time Zones

In Linux and Mac systems, time zone names are taken from the Olson Database and are constructed from an area and location pair (e.g. America/New_York). In some cases, the location element is further divided (e.g. America/Indiana/Indianapolis). Names such as GMT, EST, CET, etc are also recognised.

The Olson Database is part of the operating system, not QM. Updating this database is automatic on some platforms (e.g. Mac) but may require actions by the system administrator on some other platforms.

**AIX Time Zones**

AIX provides partial support for time zones. The default time zone is taken from the TZ environment variable. Any time zone set using the `CONFIG` command or the QMBasic `SET.TIMEZONE` statement must be in the format required by AIX (e.g. EST5 or EST5EDT). The $Z$ element of the `E` conversion code (time zone name) will return a null string.

**Windows Time Zones**

Windows support for time zones is very limited. The time zone name is of the form

$$z\{+\ |\ -\}\{d\}\{s\}$$

where

- $z$ is a three character string representing the name of the current time zone. For example, the string “CET” could be used to represent Central European time but Windows make no use of the actual name.

- $d$ is an optionally signed item with one or two digits specifying the local time zone’s difference from GMT in hours. Positive numbers adjust westward from GMT and negative numbers adjust eastward from GMT. (e.g. EST5, PST+8, CET-1).

- $s$ is an optional three character field that represents a name for the local time zone when daylight saving time is in effect.
2.10 Error Handling

QM provides a number of ways to trap errors, some applicable to the command environment, some to QMBasic programs. This section summarises the features of QM error handling.

Error Numbers

QM uses a simple system of error numbers to identify the cause of an error. The formal definition of these numbers is in a QMBasic include record named ERR.H in the SYSCOM file. The list can also be found in the Error Numbers section of this manual.

The @SYSTEM.RETURN.CODE Variable

Many commands leave status information in the @SYSTEM.RETURN.CODE variable. The value of this variable can be tested within a paragraph or a QMBasic program to determine whether the command was successful. In general, success is indicated by the value of this variable being zero. Because the query processor uses @SYSTEM.RETURN.CODE to return the number of records processed, error values are usually returned as the negative of the error number.

The REPORT.SRC command can be used to enable automatic display of the value of @SYSTEM.RETURN.CODE after every command. This is useful as a debugging aid during application development.

The @USER.RETURN.CODE Variable

User written programs cannot update @SYSTEM.RETURN.CODE. QM provides a similar variable, @USER.RETURN.CODE, for use in user written programs. This variable starts zero and is never updated by QM itself. It can be set from programs or by use of the SET command in a paragraph.

The QMBasic STATUS() Function

Many QMBasic program operations leave status information that can be accessed using the STATUS() function. Although often used for error numbers, the data returned by the STATUS() function may have other roles. For example, some operations that attempt to get locks return the user number of the user who owns the lock if the action cannot be completed. Use of the STATUS() function is documented with the relevant QMBasic statements and functions.

The QMBasic OS.ERROR() Function

Where an error reported by QM is related to some underlying error reported by the operating system, the original operating system error code can often be obtained using the OS.ERROR() function. The QM errors for which this extended error information is available are documented with the relevant QMBasic statements and functions. They are also marked in the SYSCOM ERR.H record.

Operating system error numbers vary between platforms and are documented by the operating system vendor.
The QMBasic ON ERROR Clause

Many QMBasic statements that could fail due to external influence have an optional **ON ERROR** clause. This would be executed, for example, when attempting to write to a file for which the user does not have write access.

The default action of QM to this type of error if no **ON ERROR** clause is present is to abort the program with a suitable diagnostic message. The **ON ERROR** clause allows the program to catch this error and handle it internally instead of aborting.

Very few programs ever need to use the **ON ERROR** clause. It is only applicable in situations where the program can take some meaningful action to recover from the error. The default action of QM is usually appropriate and informative. A program that simply uses the **ON ERROR** clause to abort is likely to produce a less helpful diagnostic message than would have been displayed without this clause.
Part 3

The QM File System
3 The QM File System

File Types

The files (or tables) used by QM are of two types; directory files and dynamic hashed files. Directory files do not give high performance but allow data to be accessed from outside of the QM environment. They are therefore frequently used for data interchange with other software. Dynamic hashed files offer very high performance and are typically used for the bulk of the data stored by an application. In addition, QM supports distributed files which are a way of viewing a collection of separate hashed files as though they are one file.

Facilities are provided to create data files, enter, modify and retrieve data, produce reports and, where the data processing operation required cannot be achieved using the standard commands, to construct powerful programs with the minimum of effort.

Most files consist of two parts; a data part holding the actual data and a dictionary part holding a description of the structure of data records. Files do not have to have both parts. Files with no dictionary portion are fairly common. Dictionaries with no data part usually exist only to provide a single dictionary common to the data portion of many files. Multiple files that have identical data structure may share a single dictionary.

For compatibility with other multivalue database products, QM also supports multifiles. These are a collection of data files that share a common dictionary where the component files are referenced by a two part name consisting of a file name and a subfile name separated by a comma. The subfile name will be treated as case insensitive if the VOC (and hence the file name) is case insensitive. See the CREATE.FILE command for further details.

Files contain data stored as records which are the basic unit of file access. Each record has a record id (or key) which must be unique within the file. A record id may contain any character except the mark characters (chars 251 to 255) or the ASCII NUL character (char 0). Null record ids (zero length) are not allowed in QM and the key length must not exceed the setting of the MAXIDLEN configuration parameter. If the system is in ECS mode, characters outside the 8-bit character set are only valid for files configured in ECS mode. The FCONTROL() function with action key FCSVALID.ID can be used in an application to validate a record id.

Special Filename Syntaxes

Normally, QM commands that reference files use a file name that corresponds to an F or Q-type VOC entry which, in turn, references the actual operating system file to be accessed. There are four special extended syntaxes for filenames that allow access to files without needing a VOC entry. Use of these is controlled by the FILERULE configuration option. Users should consider any impact on the security of their system before enabling these.

The four extended syntaxes are:

- Implicit Q-pointer: account:file
- Implicit QMNet pointer: server:account:file
- Pathname: PATH:pathname
- Virtual file system: VFS:server:qualifier

Note that in the pathname form, depending on context, Windows users may need to use forward slash characters (/) as directory delimiters because the backslash (\) is reserved as a string quote. Alternatively, the entire "PATH:pathname" construct can be quoted.

Because some VFS handlers may need to process filenames that include spaces (e.g. DICT name), any tilde characters (~) in the qualifier of a VFS reference will be replaced by spaces.
These special syntaxes cannot be used with a multifile component name.

For details of file processing from QMBasic programs, see File Processing.
Creating and Deleting Files

As a general rule, files that may be accessed from the operating system level must be directory files, and all other files should be dynamic files. Dynamic files give best performance but records cannot be accessed from outside of QM.

QMBasic source programs are normally stored in directory files. Dictionaries are automatically created as dynamic files regardless of the type of the associated data file as a directory file dictionary could cause severe performance degradation in query processor commands.

Files are created using the CREATE.FILE command which normally creates both a data and dictionary portion for the file. Either may be created individually by use of the DATA or DICT keywords. Dictionary portions are not required for files used to hold QMBasic source programs or include records.

The CREATE.FILE command creates the file and also writes an F-type record to the VOC. If the file is to be accessed from more than one account, this F-type record may be duplicated in the other accounts, changing the pathnames in fields 2 and 3 to include the drive and directory components as necessary. A better method is to use Q-type VOC entries for remote file pointers.

The data portion of a file is created at the specified or default minimum modulus size and with no records. The dictionary portion has a single record, @ID, added to it to represent the record key.

Files may be deleted using the DELETE.FILE command. The DATA or DICT keywords allow deletion of just the appropriate portion of the file.

Where the file's pathname in the VOC includes drive or directory components, the DELETE.FILE command assumes the file to be in some other account and prompts for confirmation that it should be deleted. If the file is to be retained but the reference to it from the current account is to be deleted, use the DELETE command to delete the VOC record.

Rules and Restrictions

The QM file system uses the underlying operating system file structures to store its data. This imposes some rules on how the operating system level files should be managed.

A QM filename may contain any characters except for control characters (chars 0 to 31), spaces, or mark characters (chars 251 to 255). On an ECS mode system, the name must contain only characters from the 8-bit set. If the QM filename contains characters that are not valid in operating system file names, its pathname is modified to use substitute characters. This translation is automatic and usually not visible to the application. A full list of the substitutions is shown in the directory files section as this is where the issue most commonly arises.

On Windows XP and later systems, use of mapped drives can assign different physical locations to the same drive letter for different users or conversely lead to a file having different pathnames in different processes. This will cause serious problems to QM as it is impossible to identify a file uniquely. In particular, the locking system is likely to become unreliable, including the low level internal locks that control structural integrity of files.

All QM users should use the same mappings. For users entering QM via a network connection, including connections looped back to the same machine, the NET USE command may need to be used to create the drive mapping. This can be included in the LOGIN or MASTER.LOGIN paragraphs in the form
SH NET USE X: \SERVER\DIR password /USER:username

where X: is the mapped drive letter and \SERVER\DIR is the target directory to which X: is to be mapped.

Similarly, use of chroot on non-Windows systems destroys the uniqueness of file names. Although this may work in some cases, it can lead to ambiguities that will cause QM to fail in unpredictable ways. Use of this command is at the user's own risk.

Where the mapped drive would reference another system running QM, the QMNet network file access mechanism should be used instead. This guarantees correct concurrency control between the two servers.

Systems other than Windows allow a file to be renamed or deleted while it is in use. This action is likely to cause QM to fail and should not be used.

Mark Mapping

Database records are often entered, modified or retrieved by the ED, SED and MODIFY commands.

In directory files, the internal field mark character is replaced by the ASCII newline character when a record is written to disk so that fields appear as lines if the record is viewed, edited or printed from outside QM. Conversely, ASCII newlines are converted to field marks on reading a record. Mechanisms are provided in QMBasic (see the MARK.MAPPING statement) to turn off this translation when handling binary data.
3.2 Directory Files

A directory file is represented by an operating system directory and the records within it by operating system files. The record key is the name of the operating system file holding the data for the record except where this would be an invalid name in which case QM performs automatic name mapping as described below.

Directory files do not give the high performance of the hashed file system but are mainly used for data that is to be processed from outside of QM or for very large records (hundreds of kilobytes). Typical uses include storage of QMBasic programs, COMO (command output) files, and saved select lists.

Records in directory files may be read, written or deleted by applications in exactly the same way as records in hashed files. The QMBasic programming language provides some additional operations for directory file access. A record may be opened using the OPENSEQ statement and then processed as a sequential file on a line by line basis (READSEQ, WRITESEQ, etc) or as a simple binary item (READBLK, WRITEBLK, etc). In addition, programming statements are provided to simplify processing of comma separated variable format data (READCSV, WRITECSV).

On all systems except Windows, the DIR.DTM mode of the OPTION command can be used to cause writes to directory files or closing a sequential file that has been written to update the date/time modified of the directory.

The F-type VOC entry for a directory file has the pathname of the directory that represents the file in field 2.

Applications frequently open the same file multiple times with different file variables in the same process. The number of simultaneous opens of the same directory file in a single process cannot exceed 255. Copying a file variable for an open file does not count as a separate open.

Record Ids

Where a record id contains characters that are not valid in operating system file names, QM automatically replaces them with an alternative representation. This is totally invisible from inside QM but other software that accesses directory file records must allow for these translations. Rather than have a different set of translations for each platform, QM adopts a single set based on the most restrictive platform (Windows) so that data may be moved between environments without modification of record names. The translations performed are:

<table>
<thead>
<tr>
<th>Character</th>
<th>Replacement 1</th>
<th>Replacement 2</th>
<th>Replacement 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>%A</td>
<td>%</td>
<td>%P</td>
</tr>
<tr>
<td>\</td>
<td>%B</td>
<td>&quot;</td>
<td>%Q</td>
</tr>
<tr>
<td>,</td>
<td>%C</td>
<td>/</td>
<td>%S</td>
</tr>
<tr>
<td>=</td>
<td>%E</td>
<td>+</td>
<td>%V</td>
</tr>
<tr>
<td>&gt;</td>
<td>%G</td>
<td>:</td>
<td>%X</td>
</tr>
<tr>
<td></td>
<td></td>
<td>%I</td>
<td>;</td>
</tr>
<tr>
<td>&lt;</td>
<td>%L</td>
<td>?</td>
<td>%Z</td>
</tr>
</tbody>
</table>

In addition, use of a period or tilde character as the first character in the name will translate this character to %D or %T respectively. QM uses names with an untranslated leading percent character to represent the subfiles of a hashed file. External applications should avoid writing such items into directory files as they may cause confusion.

Character mapping in record ids can be suppressed by use of a flag in field 6 of the F-type VOC entry that defines the file or by use of the NO.MAP option to the QMBasic OPEN, OPENPATH, OPENSEQ and DELETESEQ statements.
The length of a record id after translation of invalid characters as above must not exceed 255 bytes.

Depending on the operating system in use, record ids in directory files may be case insensitive.

Note that the Windows file system does not allow file names that clash with Windows device names such as AUX, COM1-COM9, CON, LPT1-LPT9, NUL or PRN. Also, Windows ignores a trailing period in a file name. Thus files named "ABC," and "ABC" are the same item. Use of any of these as a record id in a directory file will cause an illegal record id error.

Mark Mapping

When a record is written to a directory file, any field mark characters are converted to the operating system dependent representation of a newline (See "line terminator selection" below). Thus, each field becomes a line of text which allows the data to be processed by external software that does not understand the concept of field marks. Conversely, when data is read from a directory file, the newlines are translated to field marks. Where the data contains value marks or subvalue marks, these are not translated as it is assumed that whatever software will process this data must understand multivalued data.

One common use of directory files is to store scanned documents, digital photographs, etc. In this case, the data is not text divided into fields using the field mark character but is simple binary data that may contain any sequence of bytes. The data will nearly always contain bytes that appear to be field marks and other bytes that are the ASCII linefeed character. On writing the data to disk, the field marks will be converted to newlines. On reading the record back again, all of the newlines get converted to field marks such that the record does not match the original data written. This is clearly unacceptable. Application developers using directory files to store binary data must suppress the translation of field marks by use of the QMBasic **MARK.MAPPING** statement.

Line Terminator Selection

To allow a directory file opened with **OPEN** or **OPENSEQ** to be processed on a different system type from where it was created, QM provides four alternative ways to handle the line terminator. The default mode is appropriate to the operating system in use but can be changed using the **FCONTROL()** function or by use of a mode flag in field 6 of the F-type **VOC** record that defines the file.

Mode 0 (LF) When reading data, the line terminator is a line feed (LF) character. Carriage return (CR) characters are treated as part of the data. The **WRITE** statement converts field marks to LF characters. The **WRITESEQ** statement adds an LF after each line of text. This mode is the default on Linux/Unix systems. VOC mode flag L selects this mode.

Mode 1 (CRLF) When reading data, the line terminator is a CR/LF pair. CR or LF characters that are not paired are treated as part of the data. The **WRITE** statement converts field marks to a CR/LF pair. The **WRITESEQ** statement adds a CR/LF pair after each line of text. VOC mode flag C selects this mode.

Mode 2 (BOTH) When reading data, both a CR/LF pair and a lone LF are recognised as the line terminator. Lone CR characters that are not paired are treated as part of the data. The **WRITE** statement converts field marks to the standard operating system representation of a new line (a CR/LF pair on Windows, LF on Linux/Unix). The **WRITESEQ** statement adds the operating system line terminator after each line of text. This is the default for sequential files opened with **OPENSEQ** on Windows. VOC mode flag B selects this mode.

Mode 3 (NOCR) As for mode 2, when reading data, both a CR/LF pair and a lone LF are recognised as the line terminator but lone CR characters are discarded. The **WRITE** statement converts field marks to the standard operating system representation of a new line (a CR/LF pair on
Windows, LF on Linux/Unix). The **WRITESEQ** statement adds the operating system line terminator after each line of text. For compatibility with earlier releases, this is the default for directory files opened with **OPEN** on Windows. VOC mode flag X selects this mode.

Tokens for the mode names are provided in the SYSCOM KEYS.H record as FC$CRLF.LF, FC$CRLF.CRLF, FC$CRLF.BOTH and FC$CRLF.NO.CR.

**Character Encoding**

A character encoding can be applied to directory file data. This can be specified in three ways:

- Field 7 of the **VOC F-type** entry sets the default encoding to be used.
- The QMBasic **OPEN** or **OPENPATH** statement can specify an encoding that will override any encoding set in the VOC record.
- QMBasic read and write operations can specify an encoding that will override any encoding set in the VOC record or on opening the file.

Note that specifying the encoding as a null string in a file open operation or a read/write operation is equivalent to omitting the ENCODING clause and will, therefore, retain any default set earlier. If it is necessary to disable an encoding, the encoding name "NULL" should be used.

On ECS mode systems, if no encoding is set, only the low order byte of each character will be written to the file.

**System Related Items in Directory Files**

The directory that represents a QM directory file may contain a file named %header%. If this is present, it contains control information used by QM and must not be modified or deleted. This item will not appear in the results of a select operation against the directory file and will not be deleted by **CLEAR.FILE**. If a directory file is copied to an alternative location using operating system level commands (e.g. as a backup), it may be necessary to delete the %header% file in the copy if the copy might be updated from within QM. This is necessary as the %header% file controls replication and triggers.

**Careful Update**

When writing a record to a directory file, QM normally opens the operating system file that will represent this record and writes to it, overwriting any existing data. There is a possibility of data loss when updating an existing record if the system fails during this write (e.g. a power outage) or if there is insufficient disk space. To prevent this, the **SAFEDIR** configuration parameter can be set to adopt a "safe update" technique where the data is written to a temporary file, the original is deleted and the temporary item is renamed to replace the original. This removes nearly all possibility of losing the record but degrades performance of the write.

**Directory File Dictionaries**

Just like any other QM file, a directory file can have a dictionary containing definitions that are specific to this file, however, directory files frequently contain simple text records that do not need field definitions. There is a generic directory file dictionary named **DIR_DICT** available from the QMSYS account. To use this, either create the directory file without a dictionary or use **DELETE.FILE** to delete the dictionary portion of an existing file. Then edit the VOC entry to set field 3 to be
@QMSYS\DIR_DICT

using the directory separator appropriate to your platform.

This dictionary contains some useful items:

- **DAYS**: The number of days since this record was modified.
- **DTM**: The date and time of the last modification to the record.
- **OWNER**: The user name of the owner of the record (not Windows).
3.3 Dynamic Hashed Files

A dynamic hashed file is represented by an operating system directory, the records within it stored in a fast access file format in the directory. Users should not place any other files in the directory or make any modifications to the files placed there by QM. Dynamic files are so called because of the dynamic reconfiguration of the file which takes place automatically to compensate for changes in the file’s size and record distribution.

By default, record keys may have between 1 and 63 characters but these may not include mark characters or the ASCII null character. This length limit can be increased to a maximum of 255 by changing the value of the MAXIDLEN configuration parameter but this can lead to compatibility problems when transferring data to other systems and increases the size of QM’s internal locking tables.

A dynamic file has two parts; a primary subfile which is examined first when looking for data and an overflow subfile which contains data which does not fit into its correct location in the primary subfile. The primary and overflow subfiles are represented by operating system files named %0 and %1. Prior to release 2.8-0, the names were ~0 and ~1, but this caused problems with some poorly designed system cleanup utilities that assumed names commencing with a tilde represented temporary items that could be deleted. There may be additional items named %2, %3, etc that store data for alternate key indices.

Users do not need to understand the mechanisms that are involved in accessing dynamic files though the following information will help in determining settings for the parameters which control file configuration and hence performance. In most cases these can be left at their default values.

Data within a dynamic file is stored in record groups. The number of groups in the files is known as the modulus. The group in which a record is located is determined mathematically by using the hashing algorithm associated with the file.

A group consists of a fixed sized area in the primary subfile and, if the data assigned to the group does not all fit into this area, as many additional overflow subfile blocks as are needed will be created. A dynamic file performs best when the data is distributed evenly across each group and no group extends into the overflow area. In reality, this is almost impossible to achieve whilst still keeping each group reasonably full. A well tuned dynamic file typically has less than 20 percent of its data in overflow.

The group size parameter determines the size of the primary subfile groups as a multiple of 1024 bytes. This parameter may have a value in the range 1 to 8 and defaults to 1 though this default can be changed using the GRPSIZE configuration parameter. It should be set to a multiple of the disk block size if this value is known. As a general rule, use values of 1, 2, 4 or 8, avoiding numbers that are not powers of two as these can lead to data alignment related performance issues.

Where a file contains very large records, performance can be improved by placing these in disk blocks of their own with just the record key and a reference to their location stored in the primary subfile. Such records are known as large records and the size above which data is handled in this way is configurable. The default value of 80% of the group size is good for most purposes. Because a large record has only its key stored in the primary subfile, a SELECT operation will be faster if the group is mainly large records but reading the record’s data will require at least two disk accesses. Also, since large records are held in their own disk block(s) rather than sharing with other records, surplus space at the end of the final block is wasted resulting in higher disk space usage. If the file will be used frequently in SELECT operations where selection is based only on the record id, a lower large record size may be beneficial. If data records are frequently read from the file, a higher large record size may help. In general it is best only to change the large record size if performance problems are seen.
The number of groups in a dynamic file changes with time. QM uses two parameters to determine when the number of groups should change. At any time, the file’s load value is the total size of the data records (excluding large records) as a percentage of the primary subfile size. This value changes as records are added, modified or deleted. It may have a value in excess of 100%, indicating that there is very high usage of overflow space. The split load value (default 80%) determines the load percentage at which an additional group will be added to the file by splitting the records in one group into two. The merge load value (default 50%) determines the point at which two groups are merged back into one. A split may result in the load falling below the merge load or, conversely, a merge may result in a new load value above the split load. In neither case will the file be immediately reconfigured back again.

The split and merge loads determine the way in which the file’s modulus and hence actual load vary. A low load results in reduced overflow at the expense of increased disk space. Conversely, a high load increases overflow but reduces disk usage. High overflow in turn results in poor performance as more disk blocks must be read to find a record. The split load value determines the load at which a group will be split into two, the merge load determines the load at which groups will be merged. The difference between the two values needs to be reasonably large to avoid continual splitting and merging of groups.

The minimum modulus value determines the size below which the file will not merge groups. The default setting of this parameter is one, resulting in full dynamic reconfiguration. If the file is subject to frequent addition or deletion of large numbers of records so that its modulus varies widely, it may be worth setting the minimum modulus to a typical average size or higher, however, a file with a higher modulus than is necessary is relatively slow in SELECT operations that must read the entire file. The minimum modulus parameter can also be used to pre-allocate primary subfile disk space when creating a new file, minimising fragmentation.

Record ids in dynamic files are normally case sensitive. Case insensitive ids can be selected when the file is created or a file can be converted at a later date using the CONFIGURE.FILE command.

The total size of a dynamic file is limited to 2Gb for file versions 0 and 1, and 2147483647 groups (up to 16384Gb) for version 2 upwards.

The F-type VOC entry for a dynamic file has the pathname of the directory that represents the file in field 2.

Synchronous (Forced Write) Mode

The QM file system is highly reliable, however, it is possible for power failures or similar events to cause the system to shutdown without committing to disk data that is in the operating system cache. For critical files, it may be useful to enable synchronous (forced write) mode where every write is flushed to disk immediately. This significantly reduces the risk of file corruption at system failure but will have a severe impact on performance if the file is updated frequently.

Synchronous mode can be enabled in a number of ways:

- The FSYNC configuration parameter. This is an additive value that has three modes of operation to control when forced writes occur (see Configuration Parameters).
- Using the SYNC qualifier in a QMBasic OPEN or OPENPATH statement.
- Using the QMBasic FCONTROL() function to set synchronous mode.
- Setting the S flag in field 6 of the F-type VOC entry that describes the file.

Disabling File Resizing
Although dynamic files are very reliable, the split/merge mechanism that maintains optimum file performance introduces the possibility of file corruption in the event of a power failure or other situation that causes outstanding write operations not to be completed. QM offers a mode of operation that forms a hybrid between the dynamic file system and the static files found in many other database products.

The NO.RESIZE option of the `CONFIGURE.FILE` command can be used to disable splits and merges, locking the file at its configuration when the command is issued. As new data is added, the file will extend into overflow, reducing performance. Conversely, if large volumes of data are deleted, the groups will become less tightly packed, again resulting in reduced performance. Files can be created with this mode set by use of the NO.RESIZE option to the `CREATE.FILE` command.

The file can be reconfigured using the IMMEDIATE mode of the `CONFIGURE.FILE` command. This performs the outstanding splits or merges, bringing the file back to the configuration that it would have had if resizing had not been disabled. For typical file update patterns and reasonably frequent use, this should be considerably faster than the equivalent resizing of a static file system.

One scenario for use of this mechanism would be to operate the file(s) with resizing disabled during normal day time activity, perform backups at the start of an overnight downtime period and then use the `CONFIGURE.FILE` to reconfigure the files ready for the next day. In the unlikely event of a system failure during the reconfiguration process, the backup provides an up to date copy of the data. This resizing operation is fully interruptible and can be performed while the file is in use.

**Automatic Sequential Record Key Generation**

Although most applications use their own mechanism for allocating unique record ids, QM includes the ability to do this automatically as a simple sequential number. This is used by the CREATING.SEQKEY qualifier of the QMBasic WRITE statement or by use of the CREATING.SEQKEY command line option of the ED, MODIFY and SED editors.

The initial value of this sequential counter can be set by use of the NEXT option of the `CONFIGURE.FILE` command and defaults to 1 if not explicitly initialised. The same command allows display of the next sequential key value.

Application software can use the QMBasic FCONTROL() function to set or get this value or FILEINFO() to query it.

**The Dynamic Hashed File Cache**

To improve performance of applications that repeatedly open and close the same file (e.g. a loop that calls a subroutine that opens a file locally), QM maintains a cache of files that have been recently closed at the application level, actually keeping them open at the operating system level. This mechanism, the DH file cache, means that if the application reopens a file that is in the cache, there is very little work to be done inside QM.

The size of the DH file cache is controlled by a private configuration parameter, DHCACHE, that defaults to 10 and may take any value between 0 and 50. For most applications, the default value will work well.

The cache is automatically flushed at any action within QM itself that may require a cached file to be closed (e.g. deleting the file), including situations where the action of one QM user may require the cache to be flushed in some other user’s process. The cache is always flushed on return to the command prompt. A QM process can force the cache to be flushed in all processes using the QMBasic FLUSH.DH.CACHE statement.
Operating System Related Performance Issues

A dynamic hashed file will usually perform best if the group size is a multiple of the operating system page size. In most systems this is 4kb.
3.4 Data Collection Files

A data collection file is a variant of a dynamic file in which the records are data collections, sets of items structured as arbitrarily multi-dimensional name/value pairs. A data collection file is created using CREATE.FILE with the COLLECTION option.

The dictionary part of a data collection file is a standard dynamic file. The only valid use of A/D/S type items in the dictionary is to reference the record id (field 0) because data collections do not have the concept of fields, values or subvalues (though a character string in a collection could be structured in this way). Instead, dictionaries use E-type (element path) items to identify elements of the collection by name. These have the same form as a D-type record except that field 2 holds the element path instead of a field number.

The data part stores records as data collections. Reading a record in a QMBasic program yields a data collection variable. Conversely, only a data collection can be written to these files.

Elements of the collection that have types that cannot be stored in a file (e.g. file variables) will be stored as SQL style null values.

A data collection file may have indices built on data defined by an E-type dictionary record in exactly the same way as those based on D-type items in other files. It is also possible to define a trigger function in a collection file in which case the record data passed in to the trigger function will be a collection instead of a string. Only record level encryption is supported for collection files.

The query processor can access collection files by using E-type dictionary items instead of D-type items. The ELEMENT "name" construct can be used to reference a collection element on the command line.

Collection files are fully supported by QMNet, data replication, transactions and record level encryption.

Records in a data collection file can be viewed and edited using the collection editor, CED.
### 3.5 Distributed Files

A distributed file holds no data. Instead it is simply a reference to a set of separate dynamic files that may be treated by an application as though they were a single file.

Distributed files allow an application to break a large data set into smaller pieces and reconstruct it in various ways to optimise performance. For example, a sales processing system might store orders for each month as a separate file (ORDERS-JAN09, ORDERS-FEB09, ORDERS-MAR09, etc). Reports of orders in the current month now only require the query to be run using the file that holds this month’s orders. A report based on all orders ever received would require all the separate order files to be processed. Rather than running multiple separate queries and merging the results in some way, this is achieved by creating a distributed file that references all of the monthly order files.

A distributed file allows access to data in all of its component part files without any special logic in the application itself. When a record is to be accessed, QM will work out which part file would contain the record by applying the partitioning algorithm that defines the distributed file structure. This algorithm is supplied by the user and is effectively a higher level of hashing that translates a record id to a file part number.

There is no reason why multiple distributed files cannot reference the same part files in different combinations. As well as having a distributed file that combines all of the monthly orders files in the example above, we could also have distributed files to bring together all orders for a quarter or for a year.

The underlying requirements for a distributed file are:

1. The record ids must be unique across the entire set of part files. In the example above, the ids would probably be order numbers.

2. Given a record id, it must be possible for the partitioning algorithm to determine which part file the record would be in. This transformation may be as simple or as complex as the application design requires. In our order processing example, having the date as part of the record id would make the process very simple. On the other hand, the partitioning algorithm could use a list of order number ranges for each month.

3. The part files would normally hold records with the same structure and a single dictionary would be used for all parts as well as for the distributed file itself.

4. The part files may not be data collection files.

#### The Partitioning Algorithm

The partitioning algorithm is written as an I-type expression in the dictionary that defines the part files. This calculation must be based only on examination of the record id, not the data in the record, because the data is not available when identifying the part that will contain the record for a read operation. It is possible for the expression to use TRANS() functions to reference other files but the performance impact of this could be very severe as the expression will be evaluated for every read, write or delete.
The partitioning algorithm must return an integer part number. This must be in the range 0 to 2147483647 and does not need to be a simple sequential number. Our order file example might use the year and month as a four digit number (0811, 0812, 0901, 0902, etc).

Where a file uses case insensitive record ids, the result of the partitioning algorithm must not be affected by the casing of the supplied id. This is because case insensitivity is a property of the part file and is therefore unknown when the expression is evaluated.

Because an application can update the individual part files independently of the distributed file, it is essential that records written in this way are placed into the correct part file. Failure to ensure this may lead to all manner of strange results such as a record appearing in a select list but not being found by a read operation.

Attempting to access a distributed file where the partitioning algorithm returns a value that does not correspond to any of the file parts will cause the operation to take the ON ERROR clause. If this clause is omitted, the program will abort. The last part number returned by the partitioning algorithm, including errors, can be determined using key FL$LAST.PART in the FILEINFO() function.

**Distributed File Related Commands**

A distributed file is created using the **ADD.DF** command:

```
ADD.DF dist.file part.file part.no algorithm [RELATIVE]
```

where

- `dist.file` is the name of the distributed file.
- `part.file` is the name of the part file to be added.
- `part.no` is the part number associated with this part file.
- `algorithm` is the name of the partitioning algorithm in the dictionary of `part.file`.

First use of the command will create the distributed file, compiling the partitioning algorithm and adding the named part file. The newly created distributed file will share the dictionary of the first part file. Subsequent use of the ADD.DF command will add further part files. The `algorithm` need not be specified for the second and subsequent parts and will be ignored if present.

Note that the partitioning algorithm is copied into the distributed file. Changing the expression in the dictionary will not have any effect. It will be necessary to reconstruct the distributed file.

A new part may be added to a distributed file at any time though QM processes that already have the file open will not see the new part until the file is closed and reopened.

A part file can be removed from a distributed file using the **REMOVE.DF** command:

```
REMOVE.DF dist.file part.file
```

where

- `dist.file` is the name of the distributed file.
- `part.file` is the name of the part file to be removed. The part number can be used instead of the name. Use of the keyword **ALL** in place of `part.file` deletes the entire distributed file.

Removing the final part also deletes the distributed file. A part may be removed at any time though QM processes that already have the file open will continue to use the part until the file is closed and reopened.
The components of a distributed file can be listed using the **LIST.DF** command:

```
LIST.DF dist.file
```

where

```
dist.file
```

is the name of the distributed file.

This command shows a list of the part file pathnames and their associated part numbers.

### Alternate Key Indices in Distributed Files

Alternate key indices can be used with distributed files but are defined on the part files in the usual way, not on the distributed file. When a distributed file is opened, QM determines which index names are defined in all of the part files and only these indices are available when referencing the distributed file. Note that this process is based only on the index name. If two part files have indices of the same name that are defined differently, the effects are undefined.

### QMBasic Programming with Distributed Files

A QMBasic program that opens a distributed file can access the data in this file in exactly the same way as for any other file. The partitioning algorithm will be applied internally by QM for all record level operations to determine which part file should be accessed.

Locks are maintained at the part file level. Locking a record via an operation that references the distributed file will apply the lock to the part file in which that record would reside based on the partitioning algorithm.

Obtaining a file lock on a distributed file will acquire the file lock on all of the component part files. Because a process is never blocked by its own locks, a program can obtain the file lock on an individual part file that has been opened separately and then go on to obtain the file lock on the distributed file. Releasing the file lock on the distributed file would effectively also release the file lock on the individual part file. Conversely, releasing the file lock on the individual part file would mean that the file lock on the distributed file was no longer complete. Given the probable inappropriateness of file locks in distributed files, it is unlikely that this situation is of concern to application developers.

The **FILEINFO()** function will return a file type code of FLSTYPE.DIST (7). The **FL$PARTS** key value of **FILEINFO()** will return a field mark delimited list of the part numbers of each part file. Many of the other key values of this function are inappropriate to distributed files but can be applied to an individual part by using the **DFPART()** function to reference the part file.

The **INDICES()** function will return data for indices that are common to all part files.

The **SELECTINDEX** statement will return composite data constructed from the indices of all part files. The index scan position pointer normally set by this operation is irrelevant to distributed files.

The index scanning functions provided by **SETLEFT**, **SETRIGHT**, **SELECTLEFT** and **SELECTRIGHT** are not available with distributed files.

### Other Uses of Distributed Files
Distributed files can also be used to:
1. Spread a large file over multiple disks for load balancing.
2. Overcome operating system file size limits.
3. Store a file that is larger than a single available disk drive.
4. Physically split a file over multiple servers using QMNet.

System Administration Issues

Use of distributed files with a large number of part files requires the system administrator to think carefully about the setting of the NUMFILES configuration parameter as each part file will be opened internally as a separate file.
QMNet Network File Access

QMNet uses the **QMClient** interface to provide an extension to the QM file system allowing network access to files on another QM system. Unlike use of NFS or mapped network drives, QMNet provides locking of remote records, ensuring that data integrity can be maintained on distributed data. Where both ends of the connection support it, the network traffic is encrypted.

Use of QMNet creates a server process on the remote system for each separate QM session that has one or more files open through QMNet. The **NETWAIT** configuration parameter can be used to amend the default 10 second timeout that is applied when opening the connection.

The server process will consume a licence except where device licensing allows this to be shared with another connection from the same source. The server process will terminate when the local system closes the final file on that server but the **NETDELAY** configuration parameter can be used to keep the server process alive for a short period to improve performance of applications that repeatedly open files on the same server.

If QM's security system is enabled on the remote system, the user name of the server process as defined using **SET.SERVER** must be registered for access to QM (see Application Level Security).

Two steps are necessary to use QMNet. Firstly, the server must be defined, mapping the server name to a network address, user name and password. Secondly, the remote file must either be defined using a Q-type VOC record or it may be accessed using the extended filename syntax `server:account:file`.

**Defining the Server**

QMNet supports two styles of server definition; **public servers** and **private servers**.

Public server definitions are available to all users of the system. They are created by a user with administrative rights in the QMSYS account using the **SET.SERVER** command and persist until they are explicitly deleted.

Private server definitions are local to the QM session in which they are defined. They are created using the **SET.PRIVATE.SERVER** command and take priority over a public server definition of the same name. The **SECURITY** configuration parameter can be used to disallow creation of private server names that are also defined as public servers, preventing a malicious user rediecting a QMNet connection. If QM's security system is enabled, access to this command may be restricted by the system administrator. A private server definition would normally be created dynamically by the application, either supplying the authentication details internally or prompting the user.

Both commands have the same form:

```
SET.SERVER name ip.address user.name password
```

and prompt for items not provided on the command line. The remote server must have remote access enabled by setting the **NETFILES** configuration parameter to 2.

A server defined with the **SET.SERVER** command can be accessed by all users. The **ADMIN.SERVER** command can be used to create or modify server definitions to apply restrictions on which users can access the server.

**Defining the Remote File**
Each remote file is defined by an extended form of the Q-type VOC entry where field 4 contains the name of the server.

Once the file has been defined, it may be accessed by programs in the same way as a local file. The following restrictions apply to access from QMBasic programs:

- The **OPENSEQ** statement and related sequential file access operations are not supported.
- Access to remote files inside transactions will be non-transactional.
- A maximum of 10 separate servers may be accessed at one time by any one QM process. There is no practical limit to the number of files that may be open on each server.

**Listing Server Definitions**

A list of all defined QMNet servers accessible by the user can be displayed using the **LIST.SERVERS** command.

```
LIST.SERVERS {ALL}
```

The **ALL** option, available only to users with administrative rights, includes servers to which the user has no access.

**Deleting a Server Definition**

The definition for a remote server may be deleted using the **DELETE.SERVER** or **DELETE.PRIVATE.SERVER** command. Deletion of public server definitions is restricted to users with administrative rights in the QMSYS account.

```
DELETE.SERVER name
```

**Diagnosing Connection Problems**

There are many reasons why a newly defined QMNet server may fail to connect. These include issues with firewalls, access rights, configuration settings, etc. The **FIND.FILE** command is a useful diagnostic aid when investigation connection issues.

**Locking QMNet Files**

An application may use record and file locks on files opened via QMNet using exactly the same methods as for local files. The lock is maintained on the remote system and will not appear in the output from a **LIST.READU** command or the QMBasic **GETLOCKS()** function on the local system.

**Server Security**

The default behaviour of the **SET.SERVER** command is to create a server definition that may be accessed by all users of the system. There is a potential security weakness here because the process started on the remote system to handle the QMNet connection runs as the user name specified in the server definition, regardless of the user name of the local user accessing the remote file. Security can be improved by arranging that the user name used for the remote process is dependent on the user name or user group of the local user. This can be achieved by use of the **ADMIN.SERVER** command or an extended form of the **SET.SERVER** command.
See also:
ADMIN.SERVER, DELETE.SERVER, LIST.SERVERS, SET.SERVER
3.7 The Virtual File System

The Virtual File System (VFS) allows application designers to provide access to data that appears to
an application as a file but may actually be something quite different. Possible uses of the VFS
include:

- Providing access to data in other database environments.
- Accessing data transparently over a network where QMNet is not appropriate.
- Implementing an alternative encryption layer on top of standard QM files.

There are two styles of VFS handler; **internal** using a QMBasic class module to perform the file
system operations, and **external** using a C program. Development of an external handler requires
skills beyond QMBasic but allows access to run time libraries provided by other database vendors.

The role of the VFS handler is largely to make the external data appear to be a QM file. It may be
possible to achieve very close compatibility but often there are some features that are difficult to link
between the two environments. For example, the index scanning operations of QM may have no
direct equivalence in the external data source. Where it is not possible to provide full compatibility,
some features of QM may not be available when using the VFS.

Before using any of the VFS handlers available via the openqm.com web site, take care to read the
comment text at the head of the source code as this may include important issues to consider before
using the VFS.

The Internal VFS Handler Class Module

A VFS handler is a globally catalogued QMBasic class module that intercepts all attempts to access
the file (read, write, select, etc). It processes requests, storing or retrieving data as appropriate.

A template class module named VFS.CLS is provided in the BP file of the QMSYS account. This
includes a brief description of each of its component functions and subroutines.

The External VFS Handler C Program

A heavily commented skeleton program is available for download from the openqm.com web site.
There are also variants adapted to interface with other products, including (where redistribution rights
permit) pre-built versions that require no additional programming.

The skeleton program contains all of the C coding necessary to interface with a parent QM process
and dummy functions corresponding to each of the file system operations handled by the VFS.
These functions need to be modified to interface with whatever file system is to be accessed by the
VFS handler. Unlike the internal VFS where there is a separate instance for each file opened, an
external handler may be shared between multiple files opened from the same QM process. The
handler communicates with its parent QM process through a pipe, ensuring that there is complete
isolation between the two processes which, in turn, protects the QM process from the effects of any
programming errors in the VFS handler.

Creating a Virtual File System

There are three steps; creating the VFS handler, defining the VFS server and creating the VOC
entries.

**Step 1:** Creation of the VFS handler involves modification and compilation as described above.
Step 2: The handler name and server connection information is stored as a VFS server definition which is created using the SET.VFS.SERVER command. Some parts of this definition such as the remote server connection details may be irrelevant for a particular use. Once a VFS server definition has been created, the ADMIN.SERVER command can be used to provide more closely controlled user authentication.

For an internal VFS handler, the handler name corresponds to a globally catalogued VFS handler class module.

For an external VFS handler, the handler name has a case insensitive prefix of EXT separated from the rest of the name by a single space. The actual name of the VFS handler program is formed by adding a prefix of vfs_ to the handler name. For example, the VFS handler available from the openqm.com web site to access U2 files would be vfs_u2 on Linux or vfs_u2.exe on Windows. The handler program is started when the first file using that VFS server is opened and remains active until the last file is closed.

Step 3: Like all files, a VFS file is identified by an F-type VOC item. It is possible for only some parts of a file to be VFS items. Thus a file might have a VFS data part but a normal dictionary part. The components of a multifile can be a mix of VFS and normal items.

A VFS item is identified by the pathname in the F-type VOC entry being specified as "VFS:server" where server identifies the VFS server definition which in turn identifies the VFS handler program and connection information. There is an optional third component to this syntax which will be passed to the handler on opening the VFS item. The full syntax of the VOC item is thus

\[ VFS:server:qualifier \]

This third component could be used, for example, when a single VFS handler is used to access many files. The qualifier might be the pathname or other reference to the actual file to be opened by the handler. It may include further colons dividing it into sections in any way that the developer wishes.

Dictionaries

It is important to note that using the VFS to access dictionaries on remote systems may have unexpected effects. For example, TRANSD() functions or correlatives using T-conversions will be compiled in the context of the local QM system and hence file names on items within the system accessed via the VFS may be meaningless.

As a general rule it is best to create local copies of the dictionaries with appropriate VFS file name references to avoid this problem.

Partial Select Lists

The QM file system optimises select list generation by arranging that the QMBasic SELECT statement (not the query processor equivalent) used against a hashed file actually performs the select group by group as the READNEXT statement is used to walk through the list. Anything that requires the list to be completed (e.g. using SELECTINFO() to determine the number of items in the list) will cause the remainder of the list to be constructed immediately.

A VFS handler can work in much the same way. The VSSELECT function can return the entire list or just the initial part of the list. When a program processing this list reaches the end of the list returned by VSSELECT, the VSCONTINUE.SELECT function is called to return the next part of the list. The VSCOMplete.SELECT function will be called if the remainder of the list should be returned as a single item and the VSEND.SELECT subroutine is called to terminate generation of a
partial list. In an external handler, the function names referenced above have an underscore in place of the dollar sign.

Internal VFS handlers that do not use partial list construction can omit the V$CONTINUE.SELECT, V$COMPLETE.SELECT and V$END.SELECT entry points. External VFS handlers must include these functions, returning an empty list from the first two if not implemented.

Alternate Key Indices

A VFS handler that cannot accurately provide the documented behaviour of the INDICES(), SELECTINDEX, SELECTLEFT, SELECTRIGHT, SETLEFT and SETRIGHT operations may lead to issues with, for example, using the query processor against the VFS file. It may be best simply to return a null string from the INDICES() function so that the file appears to have no indices.

Alternatively, if the VFS handler supports simple index look up operations (SELECTINDEX) but not the index scanning operations, it should be written such that use of FILEINFO() with key FLSINDEX.SCAN returns False (0). This will cause the query processor to reject an index when using relational operators that require the unsupported functions.

Extended Filename Syntax

If enabled using additive value 8 to the FILERULE configuration parameter, a VFS file may be referenced using an extended filename syntax:

```
VFS:server:qualifier
```

where `server` is the name of a VFS server defined as described above and `qualifier` is the optional qualifying data defining the file to be opened. Because some VFS handlers may need to process filenames that include spaces (e.g. DICT `name`), any tilde characters (~) in the `qualifier` will be replaced by spaces.

The VFS Cache

The VFS includes an optional caching mechanism whereby a file that is closed by an application is actually held open in a cache so that a further open for the same file will execute faster by taking the file back from the cache. Use of TRANS() or Tfile conversions in a query is likely to benefit considerably from use of this cache. The size of the cache is set by the VFSCACHE private configuration parameter in the range 0 to 50. This mechanism is very similar to the file cache used by hashed files and the two caches are cleared together on return to the command prompt or on use of the FLUSH.DH.CACHE QMBasic statement as well as automatically at some points where a cached file could cause problems.
3.8 Creating and Modifying Data

QM provides several data entry and modification tools, each designed to fit a particular mode of use.

ED  The line editor
MED  The menu editor
MODIFY  The dictionary driven editor
SED  The full screen editor
UPDATE.RECORD  The data update utility
3.9 Dictionaries

Every data file normally has an associated dictionary which describes the structure of the data records in the file. The dictionary is normally only used by the query processor and a few other commands. Application developers may find it useful to construct data structure definitions from the dictionary for use in programs using the GENERATE command.

It is possible to create a file that has no dictionary or for multiple files to share a common dictionary.

A dictionary contains the following types of record, identified by the leading characters of field 1.

A  Pick style data definitions. An A-type record describes data that is held in a field of the record or calculated from that data.

C  Calculated values. Similar in concept to an I-type, a C-type is a program that returns a calculated value via the @ANS variable. C-types are provided for compatibility with other environments and I-types should be used by preference.

D  Direct data types. A D-type record describes data that is held in a field of the record.

E  Element references in data collection files.

I  Indirect data types. An I-type record describes data that can be evaluated from data in fields of the record, perhaps with reference to other records or other files. It is essentially a small program that returns a value.

L  Links to other files. Used only in query processor commands and dictionary I-type record expressions.

PH  Phrases for use in query processor commands.

S  Pick style data definitions. An S-type record describes data that is held in a field of the record or calculated from that data. S-type records are identical to A-type records in QM.

X  Other miscellaneous data.

The names given to A, C, D, E, I and S-type dictionary records are the names used in queries to reference the field. Every field to be used in query processor sentences must have a corresponding dictionary record. There may be multiple dictionary references to a single field thus allowing synonyms for query processor commands, perhaps with different default display characteristics.

Field 11 of all data definition record types (A, C, D, E, I, S) is reserved for user use.

Dictionary record types Y and Z will never be defined as part of the standard QM product and are available for whatever purpose a developer may find useful. Other type codes not listed above may be defined in future releases and should therefore be avoided.

To view the content of a dictionary, use LIST or LISTDICT. The latter produces a Pick style dictionary report but shows only A, D, I and S type records.
Dictionary A and S-type records

A and S-type dictionary items are an alternative to the preferred D and I-type items that describe data stored in database files. QM provides a limited subset of the full A and S-type functionality found in other multivalue database environments. There is no difference between A and S-type records within QM.

The structure of an A or S-type record is:

1: Type (A or S) plus optional description
2: Location of this field (field number)
3: { Display name to be used by LIST or SORT when displaying this field. The text can commence with 'R' (including the quotes) to right justify the heading, 'X' to suppress the normal dot filler characters, or 'RX' to apply both modifications. }
4: { Association }
5: Reserved
6: Reserved
7: { Conversion code for entry and display of this field }
8: Correlative code.
9: Display justification
10: Display width
11: Available for user use. Not referenced by QM.
12: { Data integrity constraints expression. }
13+ Reserved for internal use

Field 2 of an A or S-type record holds the field number of the data record to which this dictionary entry relates. This must be a positive integer value. A value of zero may be used to refer to the record id. For compatibility with other multivalue database products a value of 9998 or 9999 will be recognised by the query processor as references to the sequentially incremented item number within the query and the length of the record respectively. Both of these special cases are better implemented using I-type records. Where field 8 contains an A or F correlative, the value in field 2 is not used.

Field 4 defines the relationship between associated multivalued fields. Within an association, one field is considered to be the controlling item and the remainder are considered as dependant. The controlling field has \text{C}pqr\text{r} in field 4 where \( p, q, r \) (etc) are the field numbers of the associated items. The dependant fields all have \text{D}n in field 4 where \( n \) is the field number of the controlling field. Internally, QM converts Pick style association definitions into an association name __n. Alternatively, this dictionary field may contain an association name as used in C/D/E/I-type records. For compatibility with some other multivalue database products, the ASSOC.UNASSOC.MV mode of the OPTION command can be used to make the query processor treat all multivalue fields that are not in defined associations as being associated together.

Field 7 contains an optional conversion code to be applied immediately before the data is displayed in a query processor report. QM has only partial support for use of A or F-correlative expressions in this field.
Field 8 contains an optional expression to be evaluated to calculate the value of the item. This may be an A or F correlative, an IF expression as an implied A-correlative, or a conversion code to be applied to the field identified in field 2 of the dictionary record. Conversion codes appearing in field 8 are applied immediately to the data extracted from the record and hence affect sorting and selection.

Field 9 contains the justification code L, R, T or U that determines the alignment of data in a query processor report. For compatibility with Pick style systems, the justification letter may be followed by an "X" qualifier. When used as a display item in a query processor report, the display width of all items with this qualifier is expanded to use any spare space across the line, forcing the inter-column spacing to be one character.

Field 10 contains the field width to be used in a query processor report.

**Important note:** In Pick systems, correlatives are processed in an interpretive manner. QM compiles A and S-type dictionary items in a similar way to I-types. This results in better performance but, if one dictionary item uses the value of another, it will be necessary to compile both if changes are made. The `COMPILE.DICT` command with no record ids will compile all A, C, I and S-type items in the specified dictionary.

QM does not support the LPV (load previous value) data item found on Pick systems as it is dependent on the exact sequence in which the query processor evaluates expressions. The query optimiser of QM could cause this to behave in an unexpected manner. It is always possible to restructure dictionary items that used LPV to work without it.
Correlatives

A correlative is an expression in field 8 of an A or S-type dictionary item that derives a value from data in a database record. Although similar in concept to the preferred I-type dictionary items, correlatives are less powerful and more difficult to maintain. They are provided in QM to aid migration of applications from other multivalue environments. New developments should use I-type items instead and it is recommended that correlatives should be converted to I-types as part of the migration process.

There are two styles of correlative:

- **A-correlatives** are algebraic expressions and are relatively easy to understand.
- **F-correlatives** are written in reverse Polish notation which makes them difficult for less experienced developers to understand.

The following additional items may be used in place of a correlative expression:

- **CALL name** Calls a catalogued QMBasic subroutine with a single argument which contains the field value (possibly multivalued) and should be updated by the subroutine. The @ACALL.FNO variable contains the field number from the dictionary record.
- **DI** Converts an external format date to an internal day number.
- **conv** An output conversion code. Applies the specified conversion code to the data.

Field 8 of the dictionary may be multi-valued. The first value must be one of the items listed above. The remaining values may be conversion codes or format codes to be applied to the result of the correlative. Format codes must be enclosed in either single or double quotes.

String operations in correlatives are normally performed in a case sensitive manner. The type code for an A or F-correlative can be followed by NOCASE; to select case insensitive string operations for this specific correlative. Alternatively, the CORRELATIVE.NOCASE mode of the **OPTION** command can be used to make all correlatives use case insensitive string operations.

On other multivalue databases, correlatives are processed interpretively and A-correlatives are translated into their equivalent F-correlative format at the start of a query for improved performance. On QM, correlatives are compiled in much the same way as I-type expressions so the potential minor performance advantage of writing an F-correlative directly is lost. The query processor will compile correlatives automatically when they are first used. The **COMPILE.DICT** (CD) command can be used to force a compilation.

**Example**

```plaintext
A:N(TEL)\n"L(3#) 3#-4#"
```

The above correlative takes the contents of the field named TEL and then applies a format mask to it.

```plaintext
A:NOCASE:N(CODE) = 'AX'
```
The above correlative tests whether the CODE field contains AX. The NOCASE prefix causes string comparisons to be case insensitive.

**A-Correlatives**

An A-correlative is an algebraic expression that applies operators to fields, constants and other data to produce a result. It is similar in concept to an I-type expression but very limited and often difficult to maintain.

The expression is prefixed by A and an optional semicolon. Where the expression consists only of an IF element, the leading A; can be omitted.

**Data items**

- **n** A field number. This field is extracted from the current record
- **N(name)** A field name. This name is looked up in the dictionary when the correlative is compiled and run time code generated to extract the field from the current record. QM extends the capabilities of this function compared to other multivalue products by allowing name to be a reference to another calculated item (I-type, C-type or correlative).
- **"string"** A constant. All constants, including numeric values, must be enclosed in single quotes, double quotes or backslashes.

Any of the above three data item types may be followed by **R** to indicate that the **REUSE()** function is to be applied to the data.

- **D** The internal date. This is the actual date at the point when the function is executed, not a reference to the @DATE variable (which does not change during a command).
- **T** The internal time. This is the actual time at the point when the function is executed, not a reference to the @TIME variable (which does not change during a command).
- **@NB** The breakpoint level. The leading @ may be omitted.
- **@ND** The detail line counter. The leading @ may be omitted.
- **@NI** The item counter. The leading @ may be omitted.
- **@NS** The subvalue counter. The leading @ may be omitted.
- **@NV** The value counter. The leading @ may be omitted.

**Functions and Operators**

- **+** Adds operands
- **-** Subtracts operands
- ***** Multiplies operands
- **/** Divides operands. Note that this is an integer division.
- **=** Relational equality test.
# or <> Relational inequality test.
> Relational greater than test.
< Relational less than test.
>= Relational greater than or equal to test.
<= Relational less than or equal to test.

\[ p[q,r] \] Returns a substring of \( p \) starting at character \( q \), \( r \) characters long.

\[ R(p,q) \] Returns the remainder from dividing \( p \) by \( q \).

\[ S(p) \] Returns the sum of all the values in multivalued item \( p \).

\[ \text{IF } p \{ \text{THEN } q \} \{ \text{ELSE } r \} \]

Returns \( q \) if \( p \) is True, \( r \) if \( p \) is False. Absent elements return a null string.

\( (\text{conv}) \) Applies the given conversion code to the expression value. Note that \( \text{conv} \) is not quoted.

**AND** Logical AND

**OR** Logical OR.

### Examples

\[ A; 3 \times 2 \]
Multplies the content of field 3 by the content of field 2.

\[ A; (N(PRICE)+N(TAX)) \text{ (MD2)} \]
Adds the PRICE and TAX fields. The result is then converted using an MD2 conversion code.

\[ A; N(PRICE) \times "1175"/"1000" \]
Adds 17.5% tax to the single valued PRICE field. Note the need to perform the calculation in 
two steps because correlatives use integer arithmetic.

\[ A; N(PRICE) \times "1175"R/"1000"R \]
Adds 17.5% tax to each value in the multivalued PRICE field. Note the use of the R qualifier 
on both constants in this expression.

\[ A; \text{DESCRIPTION} ["1", "20"] \]
Extracts the first 20 characters of the DESCRIPTION field.

\[ A; \text{IF } N(QTY) < "10" \text{ THEN } "Re-order" \text{ ELSE } "" \]
Returns "Re-order" if the QTY field is less than 10, otherwise returns a null string.

### F-Correlatives

On systems that process correlatives interpretively, A-correlatives are converted to the more efficient F-correlative format at the start of a query that uses them. F-correlatives use "reverse Polish" notation which, whilst actually very simple, is difficult for inexperienced developers to maintain.

QM compiles both A and F-correlatives to its own internal instruction set and, therefore, the potential advantage of writing these more complex expressions on other systems is irrelevant.
Reverse Polish notation defines data and operations to be performed on it as a series of steps that manipulate an internal stack. Each step is separated by a semicolon. As an example, consider a simple A-correlative expression such as

\[ A;1*"11"/"10" \]

that adds 10% to the value in field 1. As an F-correlative expression this becomes

\[ F;1;"11";*;"10";/ \]

where the steps are

1. Push the value of field 1 onto the stack
2. "11" Push the constant "11" onto the stack
3. * Multiply the top two items on the stack, replacing them with the result
4. "10" Push the constant "10" onto the stack
5. / Divide the top two items on the stack, replacing them with the result

**Data items**

- \( n \) A field number. This field is extracted from the current record
- "string" A string constant. All constants of this style, including numeric values, must be enclosed in single quotes, double quotes or backslashes.
- \( C \) value A constant value.

Any of the above three data item types may be followed by \( R \) to indicate that the \( \text{REUSE()} \) function is to be applied to the data. The field number form can be followed by a conversion code in brackets. If used with \( R \), the sequence is \( nR(\text{conv}) \).

- \( D \) The internal date. This is the actual date at the point when the function is executed, not a reference to the \( @\text{DATE} \) variable (which does not change during a command).
- \( T \) The internal time. This is the actual time at the point when the function is executed, not a reference to the \( @\text{TIME} \) variable (which does not change during a command).
- \( @\text{NB} \) The breakpoint level. The leading @ may be omitted.
- \( @\text{ND} \) The detail line counter. The leading @ may be omitted.
- \( @\text{NI} \) The item counter. The leading @ may be omitted.
- \( @\text{NS} \) The subvalue counter. The leading @ may be omitted.
- \( @\text{NV} \) The value counter. The leading @ may be omitted.

**Functions and Operators**

- \( + \) Adds the top two items on the stack, replacing then with the result
- \( - \) Subtracts the top item on the stack from the next item, replacing them with the result.
- \( * \) Multiplies the top two items on the stack, replacing then with the result.
- \( / \) Divides the second item on the stack by the top item, replacing them with the result. Note that this is an integer division.
: Concatenates the top stack item on to the end of the second stack item, replacing then with the result

= Replaces the top two items on the stack by 1 if they are equal, 0 if they are unequal.

# Replaces the top two items on the stack by 1 if they are unequal, 0 if they are equal.

> Replaces the top two items on the stack by 1 if the top item is greater than the second item, otherwise 0.

< Replaces the top two items on the stack by 1 if the top item is less than the second item, otherwise 0.

] Replaces the top two items on the stack by 1 if the top item is greater than or equal to the second item, otherwise 0.

[ Replaces the top two items on the stack by 1 if the top item is less than or equal to the second item, otherwise 0.

[[ Replaces the top three stack items with a substring of third stack item, starting at the character position given by the second stack item, with a length determined by the top stack item.

P Duplicates the top stack item.

R Replaces the top two stack items with the remainder from dividing the second item by the top item.

S Replaces the top stack item by the sum of all the values in multivalued item at the top of the stack.

_ Exchanges the top two items on the stack.

^ Deletes the top item from the stack.

(conv) Replaces the top item on the stack with the result of applying the given conversion code to the current top stack item. Note that conv is not quoted.

& Replaces the top two items of the stack with the result of a logical AND between their values.

! Replaces the top two items of the stack with the result of a logical OR between their values.

Flow Control

J label Unconditionally jumps to label. There must be a space before the label name.

JF label Jumps to label if the top stack item is False. There must be a space before the label name.

JT label Jumps to label if the top stack item is True. There must be a space before the label name.

~label Defines a label. The label name may be followed by a space or a semicolon. In QM, labels in correlatives must be constructed from a letter optionally followed by further
letters, digits, periods, percent signs or dollar signs. Alternatively a label may be wholly numeric.

Examples

\[ F;3;2;* \]
Multiplies the content of field 3 by the content of field 2.

\[ F;3;7;+;(MD2) \]
Adds the field 3 and field 7. The result is then converted using an MD2 conversion code.

\[ F;3;"1175";*;"1000";/ \]
Adds 17.5% tax to the price in field 3. Note the need to perform the calculation in two steps because correlatives use integer arithmetic.

\[ F;3;"1175"R;*;"1000"R;/ \]
Adds 17.5% tax to each value in the multivalued price in field 3. Note the use of the R qualifier on both constants in this expression.

\[ F;1;"1";"20";[] \]
Extracts the first 20 characters of field 1.

\[ F;1;D;>;JT LATE;"";J END;~LATE;"Overdue";~END \]
Tests whether the due date in field 1 has passed. If so, the returned value is "Overdue", otherwise a null string is returned.
Dictionary C-type records

A C-type record defines a calculated value and has the definition shown below.

1:   C { descriptive text }
2:   QMBasic program, multivalued.
3:   { Conversion code }
4:   { Display name. This will be used as the default column heading by the query processor. A special value of a backslash character can be used to specify that no heading is to be displayed. The text can commence with 'R' (including the quotes) to right justify the heading, 'X' to suppress the normal dot filler characters, or 'RX' to apply both modifications. If the display name is multivalued, each value appears as a separate line. }
5:   Format specification
6:   Single/multivalue flag. Set as S if the field is always single valued or M if it can be multivalued.
7:   { Association name. Where a multivalued field has a value by value relationship with some other multivalued field defined in the same dictionary, this name links the fields together. See Associations for more details. }
8:   Available for user use in any way. Not referenced by QM.
9-10: Reserved for internal use.
11:  Available for user use in any way. Not referenced by QM.

Fields 12 onwards are reserved for internal use and users should not assume anything about their content.

A C-type dictionary item is a QMBasic program written with each line of the program as a separate value in field 2. The EV (edit values) command of ED may help in editing this field. The Dive function of SED provides similar functionality.

The program must return a result via the @ANS variable. This variable is initially zero on entry to QM, is automatically updated to contain the result of I-type expressions and should be updated by C-types. Although it is possible to use @ANS to pass a value from evaluation of one C or I-type item to the next, this is not recommended as the sequence of execution may be indeterminate.

The program can reference data defined by other items in the same dictionary using the \{name\} construct where name is an A, C, D, I or S-type item.

A $BASIC_OPTIONS record in the dictionary or in the VOC may affect compilation of C-type items.

C-type programs may not use the following QMBasic components:

$CATALOGUE
$DEBUG
$QMCALL
CLASS
DEBUG
FUNCTION
PROGRAM
SUBROUTINE
Dictionary D-type records

A D-type record defines a field stored in a data file and has the definition shown below.

1:  D { descriptive text }

2:  Field number. This is the position in the data record at which the field described by this dictionary entry can be found. A value of zero denotes the record id. When used with a data collection file, a D-type item can only reference the record id as the records in the file are not dynamic arrays.

   For compatibility with other multivalue database products a value of 9998 or 9999 will be recognised by the query processor as references to the sequentially incremented item number within the query and the length of the record respectively. Both of these special cases are better implemented using I-type records.

3:  { Conversion code }

4:  { Display name. This will be used as the default column heading by the query processor. A special value of a backslash character can be used to specify that no heading is to be displayed. The text can commence with 'R' (including the quotes) to right justify the heading, 'X' to suppress the normal dot filler characters, or 'RX' to apply both modifications. If the display name is multivalued, each value appears as a separate line. }

5:  Format specification

6:  Single/multivalue flag. Set as S if the field is always single valued or M if it can be multivalued.

7:  { Association name. Where a multivalued field has a value by value relationship with some other multivalued field defined in the same dictionary, this name links the fields together. See Associations for more details. }

8:  Available for user use in any way. Not referenced by QM.

9-10: Reserved for internal use.

11:  Available for user use in any way. Not referenced by QM.

12:  { Data integrity constraints expression. }

Fields 13 onwards are reserved for internal use and users should not assume anything about their content.
Dictionary E-type records

An E-type record defines a data item stored in a data collection file and has the definition shown below.

1: E { descriptive text }
2: Element path. This may include use of an asterisk to reference all elements of an array as a multivalued item as described for the ELEMENT query processor keyword.
3: { Conversion code }
4: { Display name. This will be used as the default column heading by the query processor. A special value of a backslash character can be used to specify that no heading is to be displayed. The text can commence with 'R' (including the quotes) to right justify the heading, 'X' to suppress the normal dot filler characters, or 'RX' to apply both modifications. If the display name is multivalued, each value appears as a separate line. }
5: Format specification
6: Single/multivalue flag. Set as S if the field is always single valued or M if it can be multivalued.
7: { Association name. Where a multivalued field has a value by value relationship with some other multivalued item defined in the same dictionary, this name links the items together. See Associations for more details. }
8: Available for user use in any way. Not referenced by QM.
9-10: Reserved for internal use.
11: Available for user use in any way. Not referenced by QM.

Fields 12 onwards are reserved for internal use and users should not assume anything about their content.

See also:
Data collections, ELEMENT
Dictionary I-type records

An I-type record defines calculated data and has the definition shown below.

1: I { descriptive text }
2: Expression
3: { Conversion code }
4: { Display name. This will be used as the default column heading by the query processor. A special value of a backslash character can be used to specify that no heading is to be displayed. The text can commence with ‘R’ (including the quotes) to right justify the heading, ‘X’ to suppress the normal dot filler characters, or ‘RX’ to apply both modifications. If the display name is multivalued, each value appears as a separate line. }
5: Format specification
6: Single/multivalue flag. Set as S if the field is always single valued or M if it can be multivalued.
7: { Association name. Where a multivalued field has a value by value relationship with some other multivalued field defined in the same dictionary, this name links the fields together. See Associations for more details. }
8: Available for user use in any way. Not referenced by QM.
9-10: Reserved for internal use.
11: Available for user use in any way. Not referenced by QM.

Fields 12 onwards are reserved for internal use and users should not assume anything about their content.

To simplify editing of compound I-type expressions, the "edit values" mode of both the ED and SED editors can be used to break the expression such that each element appears on a separate line.
Dictionary L-type records

An L-type record represents a link to another file. It can be used in query processor commands to reference fields in a dependent file without the need to create an I-type \texttt{TRANS()} expression for each field. Links can also be used within I-type expressions to minimise the number of explicit \texttt{TRANS()} operations.

1: L \{ descriptive text \}
2: Id expression
3: File name
4: \{File expression\}

The expression in field 2 is constructed in exactly the same way as an I-type expression and derives the record id of the record(s) in the linked file from data in the original file.

Links can be used to reference A, C, D, I or S-type items in the remote file.

The optional file expression in field 4 allows for more complex linking where the file to be referenced is one of a number of files that share a common dictionary, for example, sales divided by region. The name in field 3 is used to identify the file whose dictionary should be referenced to look up names used in the expressions in fields 2 and 4. The file expression in field 4 derives the name of the actual file to be referenced when retrieving data. Where field 4 is empty, the file name in field 3 is used for both purposes.

Examples

If a library application has two files, \textsc{BOOKS} and \textsc{TITLES} where the record id of \textsc{BOOKS} is formed from the id of the corresponding \textsc{TITLES} record and the copy number separated by a hyphen, the following link placed in the dictionary of the \textsc{BOOKS} file could be used to access the associated \textsc{TITLES} record:

1: L
2: @ID['-', 1, 1]
3: TITLES

Queries based on the \textsc{BOOKS} file could then reference data from the \textsc{TITLES} file using field names made up from the name of the link record, a \% character and the name of the \textsc{TITLES} field to be accessed. For example, if the above link was named TTL, a query such as

\begin{verbatim}
LIST BOOKS TTL%TITLE TTL%AUTHOR
\end{verbatim}

could be used to print a list of book titles and their authors.

The same data could be referenced in an I-type item in the dictionary of the \textsc{BOOKS} file by use of simply

\begin{verbatim}
TTL%TITLE
\end{verbatim}

as the expression (field 2). This is exactly equivalent to writing

\begin{verbatim}
TRANS(TITLES, @ID['-', 1, 1], TITLE, 'X')
\end{verbatim}

Use of links in this way removes the need to include the link expression (@ID['-', 1, 1]) in every reference to items in the \textsc{TITLES} file. Because use of a link causes a compile time substitution of the actual \texttt{TRANS()} operation, changing the link record requires all I-types that use it to be recompiled.
Multiple Links

A link may reference an item that is itself a further link. Extending the above example, there might be a READERS file that has a record for each user of the library and includes a LOANS field that lists the books on loan to the reader. The link from the READERS file to the BOOKS file becomes

```
OUT
1: L
2: LOANS
3: BOOKS
```

Finding the titles of the books on loan might then require a query command of the form

```
LIST READERS OUT%TTL%TITLE
```
**Dictionary PH-type records**

A phrase can be used in query processor sentences. When the sentence is executed, the phrase name is replaced by the phrase expansion. Typically, phrases are used to give names to groups of fields to be displayed or selection criteria.

1: PH { descriptive text }
2: Phrase expansion

Phrases may also be included in the VOC but are more commonly found in dictionaries. A phrase in the VOC can be used in queries against any file whereas a phrase in a dictionary can only be used in queries against the associated file.

Where a phrase is very long it may be broken into multiple lines within the VOC record by terminating all but the final line with an underscore character. When the phrase is substituted into a command, the lines are merged, replacing the underscore with a single space.

There are a number of reserved phrase names as listed below. These only operate when stored in the dictionary of the file to which they relate.

@ A phrase record defining the default list of items to be displayed by **LIST** and **SORT** in the absence of a display clause. If this record is not present, only the record ids will be shown.

@LPRTR A phrase record defining the default list of items to be displayed by **LIST** and **SORT** in the absence of a display clause when output is directed to a printer. If this record is not present, the query processor uses the @ record instead.

@MODIFY A phrase record defining the default list of items to be processed by the **MODIFY** command.

@SHOW A phrase record defining the default list of items to be displayed by **SHOW** in the absence of any other field names.
Dictionary X-type records

X-type dictionary items are miscellaneous data storage records which may be used in any way the application designer wishes.

1: X { descriptive text }
2: User data

Fields 2 onwards are available for data storage. Users may freely create X-type records for their own purposes but should avoid names containing $ signs or starting with the @ character as these may clash with system defined records.

The following X-type dictionary records have special meaning in QM:

$INCLUDE Controls processing of the GENERATE command.

$QUERY.DEFAULTS Sets default options for the query processor. Also valid in the VOC.

$STYLE Sets default query processor report style.

$VLIST Sets the default width for vertical mode reports. See the VERTICALLY keyword for details.
Associations

An association is a set of two or more multivalued fields that are related such that the values are inter-dependent. For example, an order processing database might contain a file listing the items in each order. This would require a multivalued list of products and a corresponding multivalued list of quantities. A realistic data file may contain several associated sets of fields.

The query processor needs to know about this relationship. An association is defined by giving it a name which appears in field 7 of the C/D/E/I-type dictionary entry of each field in the association. Applications migrated to QM from some other multivalue systems may also have a phrase record with this name which contains a space separated list of the fields that make up the association though this is not required in QM.

Thus, starting from any one element of the association, its dictionary entry can be used to find the phrase record which, in turn, allows us to find all the members of the association.

<table>
<thead>
<tr>
<th>PART.NO</th>
<th>QTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>6: M</td>
<td>6: M</td>
</tr>
<tr>
<td>7: SALES</td>
<td>7: SALES</td>
</tr>
<tr>
<td>SALES</td>
<td>SALES</td>
</tr>
<tr>
<td>1: PH</td>
<td>2: PART.NO</td>
</tr>
<tr>
<td>QTY</td>
<td></td>
</tr>
</tbody>
</table>

Pick style A/S-type dictionary items use a different way to define associations. Reworking the example above based on an order processing system, one of the fields is chosen as the "controlling field" and others in the association are considered as "dependent fields". In this example, the customer came to the shop to purchase some specific part and then decided how many he wanted. He did not go to the shop with the intention of buying three of something and then deciding what product it would be. Thus, the part number controls the association.

For the controlling field, the association element of the dictionary item (field 4) contains a string such as "C;2;3;4" where the numeric components are the field numbers of the dependent data items. There is no way to use field names here. For the dependent fields, field 4 of the dictionary record contains "D;1" where the numeric component is the field number of the controlling field.

Internally, QM converts the controlling/dependent field definition into an association named __n where n is the controlling field number. This name can be used in field 7 of a C/D/E/I-type entry if the dictionary has mixed data type definition styles. Alternatively, as an extension to the Pick style A/S-type items, QM also allows field 4 to contain an association name. Thus it is possible to associate field definitions of any type.

For compatibility with some other multivalue database products, the ASSOC.UNASSOC.MV mode of the OPTION command can be used to make the query processor treat all multivalue fields that are not in defined associations as being associated together. Reliance on this instead of properly defined associations is discouraged as it can lead to unexpected effects, especially with exploded sorts, and hides the relationship when the dictionary is used as the reference model of the data structure.
I-type expressions

An I-type dictionary record defines a calculation based on data in the data file records. Once an I-type item is defined, it can be referenced in query processor sentences exactly as though it was a real data field. I-type items are sometimes known as virtual attributes, a term which emphasises the fact that their values are not physically stored in the database.

An I-type dictionary item differs from a D-type item only in that field 1 contains the type code I and field 2 contains an expression that defines the actual calculation to be performed. The remaining fields are the same as in a D-type entry.

Consider a simple stock management system in which each inventory item has two price figures; the price that we paid to buy the item into the shop and the price that we will charge the customer. If these are defined by dictionary items named COST and SELL, we can calculate the profit we make selling an item with a simple I-type expression:

\[\text{SELL} - \text{COST}\]

An I-type expression consists data item names, constants, functions and operators in exactly the same way as a QMBasic expression. Whereas a QMBasic program would refer to variable names, an I-type expression uses field names defined in the file's dictionary. These names may be any data defining type (A, C, D, E, I or S). A field may also be referenced as \(F_n\) where \(n\) is the field number.

Dictionaries migrated to QM from other environments frequently contain data defining items that have hyphens in their names such as CUST-NO. Applying strict QMBasic expression rules to this name would interpret this as calculating the value of CUST minus NO. To allow names of this form, the I-type compiler treats this as a special case and correctly references the CUST-NO item. If CUST and NO were also defined in the dictionary, they take priority in resolving the expression.

Many of the @variables used by QMBasic are also available in I-types. The following @-variables are specific to I-types though some can be used by QMBasic programs to set up the working environment for the I-type.

- @FILE.NAME: The name of the file being processed by the command.
- @FILENAME: Synonym for @FILE.NAME.
- @ID: The record id of the current record.
- @NB: Break level number.
- @NI: Item counter.
- @RECORD: The data of the record being processed by the command.

Most QMBasic functions are also available in I-type expressions. The following functions are either specific to I-type expressions or modified from their QMBasic form:

- **TOTAL()**: Accumulate totals for use with the query processor CALC keyword.
- **TRANS()**: Fetch data from another file.
- **SUBR()**: Call a QMBasic subroutine.

Just like a QMBasic program, an I-type must be compiled before it can be used. The query processor will do this automatically though I-types can be compiled explicitly using the **COMPILE.DICT** (synonym CD) command which compiles one or more I-types in a dictionary.
The **MODIFY** command also provides facilities to compile I-types when they are edited. The object code is stored in the dictionary record though **ED** and **SED** both hide it.

Note that where one I-type expression uses the result of another, this is handled by a compile time substitution rather than a run time subroutine call. The implication of this is that, if the inner expression is changed, both must be recompiled. The safest way to ensure that everything is consistent is to use the **COMPILE.DICT** command to compile the entire dictionary after editing an I-type that might be used in another expression.

**Compound I-Types**

A compound I-type has multiple elements separated by semicolons, each of which is evaluated in turn, left to right. The value of the first element is stored in an internal variable named @1, the second in @2, and so on. These variables may be referenced in later elements within the compound I-type. The value of the immediately previous element may also be referred to by the symbol @.

The overall value of the I-type is the value of the final expression. QM can nest compound I-types.

Use of compound I-types can simplify complex expressions. For example, we might want to calculate someone's age in whole years from their date of birth. This is actually more complex than it may at first seem because we need to allow for people born on February 29 or performing the calculation on February 29. It might be written as a two stage compound I-type:

```
OCONV(DATE(), "D4Y") - OCONV(DOB, "D4Y"); IF OCONV(@DATE,"DMD") > OCONV(DOB,"DMD") THEN @ + 1 ELSE @
```

To simplify editing of compound I-type expressions, the "edit values" mode of both the **ED** and **SED** editors can be used to break the expression such that each element appears on a separate line.

**Working with Dynamic Arrays**

The QMBasic language has a large number of functions that work on each element of a multivalued data item in turn. These allow developers to write very elegant solutions to apparently complex tasks without resorting to use of a subroutine with loops. For more details, see Multivalue Functions.

**Compiler Modes**

An I-type expression can be prefixed by one or more mode settings separated by semicolons with the same effect as use of these modes in a QMBasic program. The available modes are:

- $MODE CHANGE.NO.OVERLAP
- $MODE COMPATIBLE.APPEND
- $MODE COUNT.OVERLAP
- $MODE INDEX.OVERLAP
- $MODE TIME.MS
- $MODE NOCASE.STRINGS

For example:

```
$MODE INDEX.OVERLAP;INDEX(TEXT, 'AB', 3)
```

The I-type compiler does not use the $BASIC.OPTIONS record to set default modes.

An I-type expression that is referenced from another I-type cannot include mode settings.
3.10 Conversion Codes

Sometimes data is not stored in the database in the same way as we would wish to present it to a user. A conversion code determines how data is translated between its internal format and the user friendly external format.

Although there are many conversion codes, the most important are those that handle dates, times and scaled decimal values.

Conversion codes appear in field 3 of a C-type, D-type or I-type dictionary item or field 7 of an A or S-type item. They determine the way in which data is converted prior to output by the query processor or when input via MODIFY or UPDATE.RECORD. Conversion codes are also used in the QMBasic ICONV() and OCONV() functions and with the query processor CONV keyword.

Multiple conversion codes may be applied in sequence by specifying the code as a multivalued string.

The standard conversion codes are:

<table>
<thead>
<tr>
<th>Conversion Code</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>B</td>
</tr>
<tr>
<td>Base64</td>
<td>B64</td>
</tr>
<tr>
<td>Byte String</td>
<td>BS</td>
</tr>
<tr>
<td>Call subroutine</td>
<td>CALL</td>
</tr>
<tr>
<td>Concatenation</td>
<td>C</td>
</tr>
<tr>
<td>Dates</td>
<td>D</td>
</tr>
<tr>
<td>Epoch</td>
<td>E</td>
</tr>
<tr>
<td>Group</td>
<td>G</td>
</tr>
<tr>
<td>HTML data</td>
<td>HD</td>
</tr>
<tr>
<td>HTML URL</td>
<td>HU</td>
</tr>
<tr>
<td>Integer</td>
<td>IS, IL</td>
</tr>
<tr>
<td>Length</td>
<td>L</td>
</tr>
<tr>
<td>Radix</td>
<td>MB, MO, MX</td>
</tr>
<tr>
<td>Radix</td>
<td>MCDX, MCXD</td>
</tr>
<tr>
<td>Character</td>
<td>MCx</td>
</tr>
<tr>
<td>Masked Decimal</td>
<td>MD, ML, MR</td>
</tr>
<tr>
<td>Packed Decimal</td>
<td>MP</td>
</tr>
<tr>
<td>Times</td>
<td>MT</td>
</tr>
<tr>
<td>Thousands</td>
<td>MK</td>
</tr>
<tr>
<td>Roman numerals</td>
<td>NR</td>
</tr>
<tr>
<td>Pattern matching</td>
<td>P</td>
</tr>
<tr>
<td>Range checking</td>
<td>R</td>
</tr>
<tr>
<td>Soundex</td>
<td>S</td>
</tr>
<tr>
<td>Substitution</td>
<td>Sx</td>
</tr>
<tr>
<td>Text substring</td>
<td>Tm,n</td>
</tr>
<tr>
<td>---------------------</td>
<td>-------</td>
</tr>
<tr>
<td>Translation</td>
<td>Tfile</td>
</tr>
<tr>
<td>User defined</td>
<td>U</td>
</tr>
<tr>
<td>Character encodings</td>
<td>Xname.modes</td>
</tr>
<tr>
<td>Fields</td>
<td>&lt;f,v,s&gt;</td>
</tr>
</tbody>
</table>
A-Correlative Conversion (A)

The A conversion code allows expressions formed as A-correlatives to be used as conversions. These are provided to ease migration from other systems and their use is not recommended for new developments.

The full format of this conversion code is

\[ A; \text{expression} \]

where

\[ \text{expression} \] is constructed in exactly the same way as an A-correlative.

Use of A-correlatives as conversion codes can lead to poor performance as they must be transformed to the equivalent F-correlative and then processed interpretively. When used in the query processor, this transformation is performed during parsing of the query sentence and hence only occurs once. When used with the QMBasic ICONV() or OCONV() functions, the transformation will occur on every use of the conversion function.

An A-correlative used as a conversion code with ICONV() or OCONV() must not contain N() functions as these cannot be resolved without the dictionary.

When used in field 8 of a Pick style dictionary record, an A-correlative expression is compiled in the same way as an I-type for optimum performance. The compiled version of the expression is stored in the dictionary in fields 15 onwards though the ED and SED editors will hide it.
Boolean conversion (B)

The Boolean conversion code converts between the internal representation of True and False and the external representation Y or N.

The full format of this conversion code is

B

Used with `OCONV()` for output conversion, this conversion code returns N for False (including a null string) or Y for True.

Used with `ICONV()` for input conversion, this conversion code returns False for a null string or a string containing the single character N in upper or lower case. It returns True for a string containing the single character Y in upper or lower case. Any other input value returns the original data and sets an error status of 1.
Base 64 Conversion (B64)

The base64 conversion code translates data to or from a format widely used for transmission over the internet.

The full format of this conversion code is

\[ B64 \]

Used with `OCONV()` for output conversion, this operation converts data to base64 format. The data is returned as a continuous sequence of encoded characters. In common use, this data would then be divided into lines of manageable length, typically 72 characters. This can be achieved with the `FOLD()` function.

Used with `ICONV()` for input conversion, this operation converts base64 data back to its original form. Newlines and other filler characters are ignored.

Base 64 encoding is a byte level operation. On an ECS system, it would be necessary to transform data that may contain ECS characters to a byte oriented form. This might be, for example, an encoding such as UTF-8 or a simple byte string from the BS conversion code. The transformation applied will depend on how the data is being used.

Example

```
OPENPATH '$HOLD' TO FVAR ELSE STOP 'Cannot open file'
MARK.MAPPING FVAR, OFF
READ TEXT FROM FVAR, 'REPORT.PDF' ELSE STOP 'Cannot read data'
MARK.MAPPING FVAR, ON
B64 = OCONV(TEXT, 'B64')
WRITE FOLD(B64, 72) TO FVAR, 'REPORT.B64'
```

The above program opens the $HOLD file. It then suppresses the normal directory file translation of newlines to field marks because the item to be read contains binary data (a PDF document). Having read this record, mark translation is re-enabled. The B64 conversion is used to translate the PDF to its base64 encoded equivalent. Finally, the `FOLD()` function is used to wrap the encoded data into 72 character lines. Because the target file is a directory file and mark translation has been re-enabled, the field marks in the data returned by the `FOLD()` function get replaced by newlines.
Byte String Conversion (BS)

The BS conversion code transforms ECS data to a byte string.

The full format of this conversion code is

\[ \text{BS}\{L | H}\]

Used with \texttt{OCONV()}, the BS conversion code transforms ECS data from its internal 16-bit character form to a series of byte pairs, each stored as two characters with values in the range 0 to 255. Although intended for use on an ECS mode system, this conversion code is also available for use on a non-ECS system where the high order byte of each converted character will always be zero.

Used with \texttt{ICONV()}, the BS conversion code transforms a series of byte pairs into ECS characters. If the source data contains characters with values greater than 255, the \texttt{STATUS()} value after conversion will be set to 3. This conversion code is also available for use on a non-ECS system where a non-zero high order byte in the input data will also set a \texttt{STATUS()} value of 3.

The default action of the BS conversion is to adopt the byte ordering of the system on which it is executed such that a character with Unicode code point value U+1234 would correspond to a byte pair hexadecimal 12 followed by hexadecimal 34 on a high byte first (big-endian) system or the reverse on a low byte first (little endian) system. The optional L or H suffix on the conversion code forces conversion using low byte first or high byte first ordering respectively.

Example

\[
\begin{align*}
S &= \text{ECHAR}(0x03A3):'ABC' \quad \text{;} \quad \text{Greek Sigma followed by ABC} \\
X &= \text{OCONV}(S, 'BSH') \\
\text{DISPLAY} \quad \text{OCONV}(X, 'MX0C')
\end{align*}
\]

The above program fragment displays

03A3004100420043
Call Conversion (CALL)

The CALL conversion code calls a subroutine. It behaves identically for input conversion with ICONV() and output conversion with OCONV().

The format of this conversion code is

```
CALL expr
```

where

```
expr    evaluates to the name of a catalogued subroutine.
```

The CALL conversion code calls a catalogued subroutine that must be defined to have a single argument. This argument is used to pass the data value being converted into the subroutine and to return the result of the conversion.

**Example**

```
X = OCONV(DATA, 'CALL MYSUB')
```
**Concatenation Conversion (C)**

The concatenation conversion code concatenates data items, optionally inserting separators between them. It behaves identically for input conversion with **ICONV()** and output conversion with **OCONV()**.

The general form of a concatenation conversion code is

\[
C(; c \ expr \ c \ expr \ ...)
\]

where

- \(c\) is the separator. This may be any single character except for a digit, a mark character, a quote or a backslash. A semicolon specifies that no separator is to be inserted. Where \(c\) is a space or semicolon and the next character is also a space or semicolon, this is treated as a further separator, allowing insertion of multiple spaces.

- \(expr\) is the data to be inserted and may be:
  - a field number. This will be extracted from the current content of **@RECORD** or from **@ID** if zero.
  - a string enclosed in single quotes, double quotes or backslashes.
  - an asterisk to substitute the data supplied in the conversion function call.

**Examples**

If **@RECORD** contains F1FMF2FMF3 and **@ID** contain 21:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>OCONV('abc', 'C;3;&quot;xxx&quot;;1')</td>
<td>F3xxxF1</td>
</tr>
<tr>
<td>OCONV('abc', 'C;3;1')</td>
<td>F3F1</td>
</tr>
<tr>
<td>OCONV('abc', 'C;&quot;aaa&quot;1&quot;bbb&quot;')</td>
<td>aaaF1bbb</td>
</tr>
<tr>
<td>OCONV('abc', 'C;1 2')</td>
<td>F1 F2</td>
</tr>
<tr>
<td>OCONV('abc', 'C;1*2')</td>
<td>F1*F2</td>
</tr>
<tr>
<td>OCONV('abc', 'C;1**')</td>
<td>F1*abc</td>
</tr>
<tr>
<td>OCONV('abc', 'C;3')</td>
<td>F3</td>
</tr>
<tr>
<td>OCONV('abc', 'C;0=1')</td>
<td>21=F1</td>
</tr>
</tbody>
</table>
Date Conversion (D)

Inside QM, dates are stored as a number of days since 31 December 1967 (day zero). All dates after that point are represented by positive numbers. All dates before that point are represented by negative numbers.

The date conversion code converts a date from its internal day number to one of a number of external formats or vice versa.

The full format of this conversion code is

\[ D \{y\} \{c\} \{fmt\} \{[f1, f2, f3, f4, f5]\} \]

where

\[ y \]

is a digit in the range 0 to 4 specifying the number of digits to appear in the year number. This defaults to 4.

\[ c \]

is the character to be used to separate the year, month and day components of the converted date. If omitted, a space is used. Specifying \( c \) as the digit 0 suppresses insertion of a separator between the year, month and day components. The same effect can be achieved by use of a null separator text string in the format modifiers (e.g. DYMD["",""])

\[ fmt \]

specifies the components to be present in the converted date. Multiple characters may be chosen from the following list subject to restrictions shown below. If \( fmt \) is omitted it defaults to MDY if American date mode is in use or DMY if European date mode is in use.

- **D** Day of month.
- **DO** Ordinal day of month (1st, 2nd, 3rd, etc)
- **E** Toggles European date format (day, month, year). See also `DATE.FORMAT`
- **J** Julian date (days since the start of the year).
- **L** Alphabetic month and day names are to appear with only the first character in uppercase instead of entirely in uppercase.
- **M** Month in format determined by format modifiers. If no format modifiers are present, a two digit month number is used unless \( c \) is present in which case a three letter alphabetic month is used.
- **MA** Month name.
- **MB** Brief month name.
- **Q** Quarter number (1 to 4).
- **W** Day of week number. Monday is day 1, Sunday is day 7.
- **WA** Weekday name.
- **WB** Brief weekday name.
- **WI** ISO week number.
- **X** ANSI X12 format date (YYYYMMDD with no separators).
- **Y** Year.
YI

ISO year number. This is not always the same as the calendar year as a date may be in the last week of the previous ISO year or in the first week of the following ISO year.

[f1,f2,f3,f4,f5] These format modifiers affect the way in which the above formats are handled. Up to five modifiers may be specified and they are associated with the formats in the order in which they appear in \textit{fmt}. Format modifiers are

\begin{itemize}
  \item \textit{n} Use \textit{n} characters.
  \item \textit{A} Display as alphabetic (applies to month component only).
  \item \textit{An} Display as \textit{n} alphabetic characters (applies to month component only).
  \item \textit{Z} Suppress leading zeros.
  \item \textit{Zn} Display as \textit{n} digits with leading zeros replaced by spaces.
  \item \textit{"text"} Uses the supplied text as the separator after the associated component.
  \item \textit{n"text"} Use \textit{n} characters and then use the supplied text as the separator after the associated component.
\end{itemize}

The following special codes are recognised as part of the D conversion code for ISO8601 format output. These may not be used with any other conversion elements:

\begin{itemize}
  \item ISO8601W \texttt{yyyyWwwd} Four digit year number, two digit ISO week number, one digit day of week.
  \item ISO8601W- \texttt{yyyy-Www-d} Four digit year number, two digit ISO week number, one digit day of week with hyphen separators.
\end{itemize}

Output Conversion of Dates

The following examples show the result of output conversion of a value of 9649 with various conversion codes. Where affected by \texttt{DATE.FORMAT} setting, both forms are shown.

\begin{center}
\begin{tabular}{lll}
\textbf{Code} & \textbf{Result} & \textbf{European date} \\
'D' & 01 JUN 1994 & 01/06/1994 \\
'D2' & 01 JUN 94 & 01/06/94 \\
'D4' & 01 JUN 1994 & 01/06/94 \\
'D/' & 06/01/1994 & 06 01 1994 \\
'D ' & 06 01 1994 & 06 01 1994 \\
'D2/' & 06/01/94 & 06 01 94 \\
'D/E' & 01/06/1994 & 01/06/94 \\
'D2 E' & 01 06 94 & 01 06 94 \\
'D.YJ' & 1994.152 & 1994 06 01 \\
'D2JY' & 152.94 & 19940501 \\
'D YMD["","]' & 940601 & 940601 \\
D2YMD["","] & 940601 & 940601 \\
'DX' & 19940601 & 19940601
\end{tabular}
\end{center}
Input Conversion of Dates

Alphabetic month names may be supplied in the external format date. At least three letters must be present and conversion is not case sensitive. If more than three letters are present, the must correctly match the spelling of the month name.

The date conversion system allows some flexibility to the order in which the components of the date may occur in the input data. Where an alphabetic month name is used, this is processed first. Numeric components are then assumed to be in the order of the remaining elements of the conversion code. Thus with a code of DDMY, '1 Jun 94' and 'Jun 1 94' would both convert to 9649, the internal representation of 1 June 1994.

A date that is entered as a simple digit sequence with no separators (e.g. 020609) will be treated as being day, month and year components in the order defined by the fmt element of the conversion code, with a maximum of two digits in the day and month. The year component will be processed as having the number of digits in the conversion code, defaulting to 4. A trailing year with only two digits will follow the normal rules as described in the next paragraph.

For dates entered as two digits, year number values in the range 30 to 99 are assumed to be 1930 to 1999 and 0 to 29 are assumed to be 2000 to 2029. This 100 year window can be moved using the YEARBASE configuration parameter.

Where not included, the day number or month number is assumed to be one and the year number to be the current year.

For compatibility with Information style multivalue products, a day number that exceeds the number of days in the month will roll forward into the next month and return a value of 3 from the
**STATUS()** function. This feature can be suppressed by enabling the NO.DATE.WRAPPING mode of the **OPTION** command.

**Alphabetic Day and Month Names**

The strings used for the alphabetic day and month names are derived from the QM messages file and can be different based on language selection. The relevant message numbers are

<table>
<thead>
<tr>
<th>Code</th>
<th>Message</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>MA</td>
<td>1500</td>
<td>Alphabetic month</td>
</tr>
<tr>
<td>MB</td>
<td>1503</td>
<td>Brief alphabetic month</td>
</tr>
<tr>
<td>WA</td>
<td>1501</td>
<td>Alphabetic day of week</td>
</tr>
<tr>
<td>WB</td>
<td>1504</td>
<td>Brief alphabetic day of week</td>
</tr>
</tbody>
</table>

In addition, message 1502 defines the ordinal numeric strings used by the DO code.

**Calendar Differences**

The process of converting an external format date to its internal day number and vice versa is not as easy as it sounds. As well as the slightly complex rules that determine which years are leap years and hence have 29 days in February, there is a problem of calendar differences. Most of the world now uses the same calendar for business purposes but this has not always been the case. There were two significant realignments, one in 1752 and an earlier one in 1583 in different parts of the world. Prior to these changes, the date in one country could be several days different from that elsewhere.

The QM date conversion operations assume the current calendar system and make no adjustment to handle these realignments. Some multivalue products implement one or other realignment. Whichever system is used, it will be incorrect in some contexts. Users who require specific handling of these changes or need to handle dates before 1 January 0001 will need to develop their own date conversion functions.

**See also:**

Dates, Times and Epoch Values
Epoch Conversion (E)

Epoch values represent a moment in time in a time zone independent manner. They are stored as a number of seconds since 1 January 1970 GMT (more correctly UTC).

The epoch conversion code converts an epoch value from its internal day number to one of a number of external formats or vice versa.

The full format of this conversion code is an extension of the D date conversion code:

\[ \text{E} \{y\} \{c\} \{fmt\} \{[f1, f2, f3, f4, f5, f6, f7]\} \]

where

\(y\) is a digit in the range 0 to 4 specifying the number of digits to appear in the year number. This defaults to 4.

\(c\) is the character to be used to separate the year, month and day components of the converted date. If omitted, a space is used. Specifying \(c\) as the digit 0 suppresses insertion of a separator between the year, month and day components.

\(fmt\) specifies the components to be present in the converted date. Multiple characters may be chosen from the following list subject to restrictions shown below. If \(fmt\) is omitted it defaults to MDY if American date mode is in use or DMY if European date mode is in use.

- **A**: ASCII format date/time (e.g. Thu 16 Apr 15:02:21 2009). No other elements may be included in the same conversion.
- **D**: Day of month.
- **DO**: Ordinal day of month (1st, 2nd, 3rd, etc)
- **E**: Toggles European date format (day, month, year). See also [DATE.FORMAT](#).
- **J**: Julian date (days since the start of the year).
- **L**: Alphabetic month and day names are to appear with only the first character in uppercase instead of entirely in uppercase.
- **M**: Month in format determined by format modifiers. If no format modifiers are present, a two digit month number is used unless \(c\) is present in which case a three letter alphabetic month is used.
- **MA**: Month name.
- **MB**: Brief month name.
- **O[]**: Offset from GMT in the form +hh:mm or -hh:mm. The colon separator is only present if included in the code.
- **Q**: Quarter number (1 to 4).
- **T{H}{S}{s}**: Time. The H, S and s elements are as for the MT conversion code, specifying 12 hour clock format, inclusion of seconds, and an alternative separator for the components of the time value.
- **W**: Day of week number. Monday is day 1.
- **WA**: Weekday name.
WB Brief weekday name.
WI ISO week number.
X ANSI X12 format date (YYYYMMDD with no separators).
Y Year.
YI ISO year number. This is not always the same as the calendar year as a date may be in the last week of the previous ISO year or in the first week of the following ISO year.
Z Time zone code

[f1,f2,f3,f4,f5,f6,f7] These format modifiers affect the way in which the above formats are handled. Up to seven modifiers may be specified and they are associated with the formats in the order in which they appear in fmt. Format modifiers are

- n Use n characters.
- A Display as alphabetic (applies to month component only).
- An Display as n alphabetic characters (applies to month component only).
- Z Suppress leading zeros.
- Zn Display as n digits with leading zeros replaced by spaces.
- "text" Uses the supplied text as the separator after the associated component.
- n"text" Use n characters and then use the supplied text as the separator after the associated component.

The following special codes are recognised as part of the E conversion code for ISO8601 format output. These may not be used with any other conversion elements:

ISO8601W yyyyWwwd
Four digit year number, two digit ISO week number, one digit day of week.

ISO8601W- yyyy-Www-d
Four digit year number, two digit ISO week number, one digit day of week with hyphen separators.

ISO8601T yyyymmddThhmmss
Four digit year number, two digit month number, two digit day number, two digit hours, two digit minutes, two digit seconds.

ISO8601T- yyy-mm-ddThh:mm:ss
Four digit year number, two digit month number, two digit day number, two digit hours, two digit minutes, two digit seconds with separators.

Time Zones

Conversion of an epoch value to an external form date and time is dependent on the time zone for which the conversion is performed. On entry to QM, the time zone to be used is taken from the operating system TZ environment variable for the user’s process. If this has not been set, the value of the TZ variable when QM was started is used. If this is also not set, GMT is used.

The time zone name is stored in a private configuration parameter named TIMEZONE. This can be modified from, for example, the LOGIN paragraph, using the CONFIG command or from within a program using the QMBasic SET.TIMEZONE statement.
Output Conversion of Epoch Values

The following examples show the result of output conversion of a value of 1234567890 in time zone EST with various conversion codes. Where affected by `DATE.FORMAT` setting, both forms are shown.

<table>
<thead>
<tr>
<th>Code</th>
<th>Result</th>
<th>European date</th>
</tr>
</thead>
<tbody>
<tr>
<td>'E'</td>
<td>13 FEB 2009</td>
<td></td>
</tr>
<tr>
<td>'E2'</td>
<td>13 FEB 09</td>
<td></td>
</tr>
<tr>
<td>'E4'</td>
<td>13 FEB 2009</td>
<td></td>
</tr>
<tr>
<td>'ET'</td>
<td>18:31</td>
<td></td>
</tr>
<tr>
<td>'E/'</td>
<td>02/13/2009</td>
<td>13/02/2009</td>
</tr>
<tr>
<td>'E '</td>
<td>02 13 2009</td>
<td>13 02 2009</td>
</tr>
<tr>
<td>'E2/'</td>
<td>02/13/09</td>
<td>13/02/09</td>
</tr>
<tr>
<td>'E/E'</td>
<td>13/02/2009</td>
<td>02/13/2009</td>
</tr>
<tr>
<td>'E E'</td>
<td>13 02 09</td>
<td>02 13 09</td>
</tr>
<tr>
<td>'E.YJ'</td>
<td>2009.44</td>
<td></td>
</tr>
<tr>
<td>'E2JY'</td>
<td>44:09</td>
<td></td>
</tr>
<tr>
<td>'E YMD'</td>
<td>2009 02 13</td>
<td></td>
</tr>
<tr>
<td>'E YMDTS'</td>
<td>2009 02 13 18:31:30</td>
<td></td>
</tr>
<tr>
<td>'EX'</td>
<td>20090213</td>
<td></td>
</tr>
<tr>
<td>'E MY[A,2]'</td>
<td>FEBRUARY 09</td>
<td></td>
</tr>
<tr>
<td>'E4DOMAYL'</td>
<td>13th February 2009</td>
<td></td>
</tr>
<tr>
<td>'E4DOMAYLTS[A3]'</td>
<td>13th Feb 2009 18:31:30</td>
<td></td>
</tr>
<tr>
<td>'E DMY[,A3,2]'</td>
<td>13 FEB 09</td>
<td></td>
</tr>
<tr>
<td>'E DMY[,A9,2]'</td>
<td>13 FEBRUARY 09</td>
<td></td>
</tr>
<tr>
<td>'E/MDY[Z,Z,2]'</td>
<td>2/13/09</td>
<td></td>
</tr>
<tr>
<td>'E DMYL,[A,]'</td>
<td>13 February 2009</td>
<td></td>
</tr>
<tr>
<td>'EDMYL[Z,A,2]'</td>
<td>13 February 09</td>
<td></td>
</tr>
<tr>
<td>'EYMD[2,2,2]'</td>
<td>09 02 13</td>
<td></td>
</tr>
<tr>
<td>'EW'</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>'EWA'</td>
<td>FRIDAY</td>
<td></td>
</tr>
<tr>
<td>'EWAL'</td>
<td>Friday</td>
<td></td>
</tr>
<tr>
<td>'EMA'</td>
<td>FEBRUARY</td>
<td></td>
</tr>
<tr>
<td>'EMAL'</td>
<td>February</td>
<td></td>
</tr>
<tr>
<td>'EQ'</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>'E-YIWI'</td>
<td>2009-07</td>
<td></td>
</tr>
<tr>
<td>'EA'</td>
<td>Fri Feb 13 18:31:30 2009</td>
<td></td>
</tr>
<tr>
<td>'EWALMALDTSZY[A3,A3]'</td>
<td>Fri Feb 13 18:31:30 EST 2009</td>
<td></td>
</tr>
</tbody>
</table>
Input Conversion of Epoch Values

Input conversion with the E code is much more restricted than the range of formats possible on output conversion. In practical use of the epoch date system, it is likely that the date and time are entered as separate items. In this case, it may be more flexible to use the D and MT conversions on the two components separately and then use the \texttt{MVEPOCH()} function to convert these to an epoch value.

The input data must always consist of a date followed by a time. The date must contain all three elements (day, month, year).

The simplest E conversion is just the letter E. This may be followed by the D, M and Y elements to override the default sequence of the date components. Use of a trailing T to represent the time is valid for consistency but is otherwise ignored as it must be the last element in the data to be converted. Use of any other conversion code elements may have unspecified effects.

There is some flexibility to the order in which the components of the date may occur in the input data. Where an alphabetic month name is used, this is processed first and must be at least three characters. Numeric components are then assumed to be in the order of the remaining elements of the conversion code. Thus with a code of EDMY, '1 Jun 94' and 'Jun 1 94' would both be treated as 1 June 1994.

For dates entered as two digits, year number values in the range 30 to 99 are assumed to be 1930 to 1999 and 0 to 29 are assumed to be 2000 to 2029. This 100 year window can be moved using the \texttt{YEARBASE} configuration parameter. and the order of these defaults to day.

The time must be entered in the form \texttt{hh:mm:ss} or \texttt{hh:mm} where the colon may be any non-numeric character. If the seconds are omitted, a default of zero is used.

See also: \texttt{Date, Times and Epoch Values, EPOCH(), MVDATE(), MVDATE.TIME(), MVEPOCH(), MVTIME(), SET.TIMEZONE}
F-Correlative Conversion (F)

The F conversion code allows expressions formed as F-correlatives to be used as conversions. These are provided to ease migration from other systems and their use is not recommended for new developments.

The full format of this conversion code is

\[ F; \text{expression} \]

where

\[ \text{expression} \] is constructed in exactly the same way as an F-correlative.

A F-correlative used in a conversion code is executed internally in an interpretive manner whereas an F-correlative in field 8 of a Pick style dictionary record is compiled for optimum performance. The compiled version of the expression is stored in the dictionary in fields 15 onwards though the ED and SED editors will hide it.
Group Conversion (G)

The group conversion code treats the source data as being formed from a number of components separated by a delimiter character and extracts specified components. It works identically on input and output conversions.

- \( G_{cn} \) Returns the first \( n \) components of the source data delimited by character \( c \).
- \( G_{scn} \) Skips \( s \) components and then returns the next \( n \) components of the source data delimited by character \( c \).

Example

Consider a date stored in character form as 03/10/98. The G conversion code could be used to extract components of this date:

- \( G/1 \) returns 03
- \( G/2 \) returns 03/10
- \( G1/1 \) returns 10
HTML Data Conversion (HD)

The HD conversion code translates HTML reserved characters to/from their escape codes.

Used as an output conversion, the minimum set of reserved characters are replaced with their HTML escape codes. These are

```
&     &amp;
<     &lt;
>     &gt;
"     &quot;
'     &apos;
```

In addition, character 128 is converted to &nbsp; (non-breaking space) and the internal representation of the Euro currency symbol is converted to &euro; The Unicode character set defines the Euro symbol as codepoint 0x20AC and the ECS version of QM supports this. On a non-ECS system, the commonly adopted but not strictly defined use of character 160 as the Euro symbol is adopted.

Used as an input conversion, the full set of HTML escape codes is recognised.
HTML URL Conversion (HU)

The HU conversion code translates reserved characters in HTML URL strings to/from their escape codes.

Used as an output conversion, reserved characters are replaced with their hexadecimal escape codes. For example, a comma is replaced by %2c. Spaces are replaced by + signs.

Used as an input conversion, hexadecimal escape codes are translated to their equivalent characters and + signs are replaced by spaces.
Integer Conversion (IS, IL)

The integer conversion codes convert integer values between numeric form and hardware specific integer representation.

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS</td>
<td>Short integer (16 bit value).</td>
</tr>
<tr>
<td>IL</td>
<td>Long integer (32 bit value).</td>
</tr>
</tbody>
</table>

Used with the ICONV() function, the conversion code translates a QMBasic integer numeric value to the equivalent hardware specific representation of that integer. Used with the OCONV() function, the conversion code translates a hardware specific representation of an integer value to its equivalent QMBasic numeric form.

By default, these conversions adopt the byte ordering of the machine on which the program is being executed. Adding an optional L to the end of the conversion code (ISL, ILL) causes conversion to assume a low byte first format for the hardware representation of the value. Similarly, adding an optional H to the end of the conversion code (ISH, ILH) causes conversion to assume a high byte first format for the hardware representation of the value.

These codes should not be used to encode numeric values to be stored in database files as the hardware specific representation may include bytes that will be interpreted as mark characters. These conversions are intended for use in, for example, applications that need to generate hardware specific data for transmission over communications networks.

Note that when used in this way, the input and output conversions may appear to be reversed from the expected use. An application would use ICONV() to transform a numeric value to the byte sequence for transmission over the network and OCONV() to transform received data from a byte sequence to a number. These codes were originally defined in another multivalue product and their definition has been maintained.
Length Conversion (L)

The length conversion code performs string length constraint checks. It works identically on input and output conversions and has three forms:

- **L**: Returns the length of the string being converted.
- **Ln**: Returns the original string if its length is less than or equal to \( n \). Otherwise it returns a null string.
- **Ln,m**: Returns the original string if its length is greater than or equal to \( n \) and less than or equal to \( m \). Otherwise it returns a null string.

- **LW**: Returns the display length of the string being converted.
- **LWn**: Returns the original string if its display length is less than or equal to \( n \). Otherwise it returns a null string.
- **LWn,m**: Returns the original string if its display length is greater than or equal to \( n \) and less than or equal to \( m \). Otherwise it returns a null string.

Note that on ECS systems, the default behaviour of the L conversion code checks character counts, not display width. This may be significant if the string includes double width characters. Use of the optional W qualifier after the L causes the conversion code to use the string's display width. On non-ECS systems, the W qualifier is ignored.

### Examples

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Data</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>L</td>
<td>1234</td>
<td>4</td>
</tr>
<tr>
<td>L3</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>L3</td>
<td>AB</td>
<td>AB</td>
</tr>
<tr>
<td>L3</td>
<td>ABC</td>
<td>ABC</td>
</tr>
<tr>
<td>L3</td>
<td>ABCD</td>
<td></td>
</tr>
<tr>
<td>L2,3</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>L2,3</td>
<td>AB</td>
<td>AB</td>
</tr>
<tr>
<td>L2,3</td>
<td>ABC</td>
<td>ABC</td>
</tr>
<tr>
<td>L2,3</td>
<td>ABCD</td>
<td></td>
</tr>
</tbody>
</table>
Character Conversion (MCx)

The character conversion codes perform various character based conversions.

- **MCA**: Delete all non-alphabetic characters.
- **MC/A**: Delete all alphabetic characters.
- **MCAN**: Delete all non-alphanumeric characters.
- **MC/AN**: Delete all alphanumeric characters.
- **MCL**: Convert to lower case.
- **MCN**: Replace non-printing characters by character c, using period if c is omitted. The substitute character may optionally be enclosed in single or double quotes. An empty set of quotes removes non-printing characters instead of replacing them.
- **MC/N**: Delete all numeric characters. The optional minus sign treats a leading minus sign in the data to be converted as a numeric character.
- **MCP**: Convert first alphabetic character in each sentence to uppercase, leaving all other characters unchanged.
- **MCS**: Convert first alphabetic character in each sentence to lowercase, leaving all other characters unchanged.
- **MCT**: Capital initial all words (see below).
- **MCU**: Convert to uppercase.

These conversion codes behave identically for both input and output conversion.

Character types (digit, alphabetic, printing, etc) are determined by the currently selected character map. See [Extended Character Set Support](Character Maps on Non-ECS Systems) for more information relating to ECS mode systems or the Character Maps on Non-ECS Systems subsection.

The MCT conversion code is implemented differently across various multivalue products. The default behaviour of QM is to match D3 and other Pick style products where the first letter after a non-alphabetic character is converted to uppercase, all others to lowercase. Use of the SPACE.MCT mode of the `OPTION` command enables the behaviour found in Information style products such as UniVerse whereby letters immediately following a space are converted to uppercase, all others to lowercase. In both modes, the first character of the string is converted to uppercase.

### Examples

<table>
<thead>
<tr>
<th>Data</th>
<th>Code</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>267PS-A17</td>
<td>MCA</td>
<td>PSA</td>
</tr>
<tr>
<td>267PS-A17</td>
<td>MC/A</td>
<td>267-17</td>
</tr>
<tr>
<td>267PS-A17</td>
<td>MCAN</td>
<td>267PSA17</td>
</tr>
<tr>
<td>267PS-A17</td>
<td>MC/AN</td>
<td>-</td>
</tr>
<tr>
<td>267PS-A17</td>
<td>MCN</td>
<td>26717</td>
</tr>
<tr>
<td>267PS-A17</td>
<td>MC/N</td>
<td>PS-A</td>
</tr>
<tr>
<td>-123-456</td>
<td>MCN</td>
<td>123456</td>
</tr>
<tr>
<td>-123-456</td>
<td>MCN-</td>
<td>-123456</td>
</tr>
<tr>
<td>-123-456</td>
<td>MC/N-</td>
<td>-</td>
</tr>
<tr>
<td>Red pencil</td>
<td>MCL</td>
<td>red pencil</td>
</tr>
<tr>
<td>Red pencil</td>
<td>MCU</td>
<td>RED PENCIL</td>
</tr>
<tr>
<td>------------</td>
<td>-----------</td>
<td>------------</td>
</tr>
<tr>
<td>Red pencil</td>
<td>MCT</td>
<td>Red Pencil</td>
</tr>
<tr>
<td>123FM456</td>
<td>MCP</td>
<td>123.456</td>
</tr>
<tr>
<td>123FM456</td>
<td>MCP?</td>
<td>123?456</td>
</tr>
<tr>
<td>123FM456</td>
<td>MCP&quot;x&quot;</td>
<td>123x456</td>
</tr>
<tr>
<td>123FM456</td>
<td>MCP&quot;&quot;</td>
<td>123456</td>
</tr>
<tr>
<td>abc. dE. 1f</td>
<td>MCS</td>
<td>Abc. DE. 1F</td>
</tr>
</tbody>
</table>
Masked Decimal Conversion (MD, ML, MR)

The masked decimal conversion codes convert a number between its internal and external forms. The formats available provide scaling, currency symbol insertion, thousands separation and a variety of methods for handling negative numbers.

Scaling provides a means by which items such as currency values which are usually written as pounds and pence (or dollars and cents) can be handled internally as integer numbers of pence (or cents) for faster and precise calculation. Scaling is performed by specifying the position of an assumed decimal point.

Input conversion allows for some degree of flexibility in the exact format used. For example, any of the negative value representations may be used regardless of the method actually defined in the conversion code.

The three masked decimal conversion codes are

- **MD**: Convert without regard to justification
- **ML**: Left justify the converted result
- **MR**: Right justify the converted result

The full format of this conversion code is

```
ML n {f} {,} {$} {s} {[[intl]]} {P} {Z} {T} {x{c}} {fx}
```

where

- **n** is a digit in the range 0 to 9 specifying the number of digits to appear to the right of the decimal point. Rounding occurs on output conversion in the fractional part and, if the result is an integer, the decimal point does not appear.

- **f** is a digit in the range 0 to 9 specifying the position of the implied decimal point in the data to be converted. For example, if the value supplied to an output conversion is 12345 and \( f \) is 2, the result is 123.45. Conversely, if the value supplied to an input conversion is 123.45 and \( f \) is 2, the result is 12345. If omitted, \( f \) defaults to the same value as \( n \).

- **,** specifies that the national language convention thousands separator is to be inserted between every third digit to the left of the decimal point. The default delimiter is a comma but this may be changed by use of the **NLS** command or the **SETNLS** QMBasic statement.

- **$** specifies that the national currency symbol should be used as a prefix to the converted data on output conversion and may be present on input conversion. The default currency symbol is a dollar sign but this may be changed by use of the **NLS** command or the **SETNLS** QMBasic statement.

- **s** specifies the handling of the numeric sign of the value.

- **+** places a + or - sign to the right of the converted data.

- **-** places a - sign to the right of negative values or a space to the right of positive values.

- **(** encloses negative values in round brackets. A positive value has a space placed to its right. This element is not allowed when the PICK.ML.CONV.MASK mode of the **OPTION** command is in effect.
< encloses negative values in angle brackets. A positive value has a space placed to its right.

C places the letters cr to the right of negative values or two spaces to the right of positive values. Use the CRDB.UPCASE keyword of the OPTION command to change this to CR.

D places the letters db to the right of negative values or two spaces to the right of positive values. Use the CRDB.UPCASE keyword of the OPTION command to change this to DB. Note that this qualifier is implemented inconsistently across different multivalue products. QM follows the Prime Information semantics. Most other products apply the db suffix to positive values.

Input conversion accepts any of these representations of a negative value regardless of the actual conversion code used.

[intl] specifies alternative international handling of currency symbols and separators. intl consists of up to four comma separated items which specify the prefix, thousands separator, decimal separator and suffix to be applied to the converted value. These components should be quoted if there is any potential confusion.

Z specifies that a zero value should be represented by a null string on output conversion. This option is ignored on input conversion.

T specifies that trailing decimal places should be truncated rather than rounded. This option is ignored on input conversion.

\( x\{c\} \) specifies that the result of an output conversion is to be a field of \( x \) characters, left or right justified as specified by use of ML or MR. If the converted data is longer than \( x \) characters it will be truncated to fit. If it is less than \( x \) characters it is padded using character \( c \) or spaces if \( c \) is not specified. The value of \( x \) may be one or two digits.

On input conversion, the value of \( x \) is ignored and all occurrences of character \( c \), or spaces if \( c \) is not specified, within the data are ignored.

\( fx \) is an alternative style of padding specification. It cannot be used with \( x\{c\} \).

- \( f \) specifies the padding character to be used; * for asterisk, # for space, % for zero.
- \( x \) specifies the field width.

The \( fx \) element may be a complete mask as used in format specifications (e.g. #3-#4). For compatibility with Pick style systems, when the PICK.ML.CONV.MASK mode of the OPTION command is in effect, the \( fx \) element of ML and MR conversions may be enclosed in round brackets.

### Masked Decimal Output Conversion

If the input data is a null string, the ML and MR codes normally treat this as zero and return an appropriately formatted conversion of the number zero. If the PICK.NULL mode of the OPTION command is active, a null string will be returned.

The following table sets out a variety of combinations of masked decimal output conversion features.

<table>
<thead>
<tr>
<th>Value</th>
<th>Conversion</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MD0</td>
<td>'0'</td>
</tr>
<tr>
<td>Value</td>
<td>Option</td>
<td>Format</td>
</tr>
<tr>
<td>---------</td>
<td>--------</td>
<td>-----------------------</td>
</tr>
<tr>
<td>0</td>
<td>MD0Z</td>
<td>&quot;</td>
</tr>
</tbody>
</table>

### Sign handling

<table>
<thead>
<tr>
<th>Value</th>
<th>Option</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>MD0</td>
<td>'12345678'</td>
</tr>
<tr>
<td>-12345678</td>
<td>MD0</td>
<td>'-12345678'</td>
</tr>
<tr>
<td>12345678</td>
<td>MD0+</td>
<td>'12345678+'</td>
</tr>
<tr>
<td>-12345678</td>
<td>MD0+</td>
<td>'-12345678-'</td>
</tr>
<tr>
<td>12345678</td>
<td>MD0-</td>
<td>'12345678'</td>
</tr>
<tr>
<td>-12345678</td>
<td>MD0-</td>
<td>'-12345678-'</td>
</tr>
<tr>
<td>12345678</td>
<td>MD0&lt;</td>
<td>'12345678'</td>
</tr>
<tr>
<td>-12345678</td>
<td>MD0&lt;</td>
<td>'&lt;12345678&gt;'</td>
</tr>
<tr>
<td>12345678</td>
<td>MD0()</td>
<td>'12345678'</td>
</tr>
<tr>
<td>-12345678</td>
<td>MD0()</td>
<td>'(12345678)'</td>
</tr>
<tr>
<td>12345678</td>
<td>MD0C</td>
<td>'12345678'</td>
</tr>
<tr>
<td>-12345678</td>
<td>MD0C</td>
<td>'12345678CR'</td>
</tr>
<tr>
<td>12345678</td>
<td>MD0D</td>
<td>'12345678'</td>
</tr>
<tr>
<td>-12345678</td>
<td>MD0D</td>
<td>'12345678DB'</td>
</tr>
</tbody>
</table>

### Scale factors

<table>
<thead>
<tr>
<th>Value</th>
<th>Option</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>12345678</td>
<td>MD22</td>
<td>'123456.78'</td>
</tr>
<tr>
<td>12345678</td>
<td>MD22</td>
<td>'12345.68'</td>
</tr>
<tr>
<td>-12345678</td>
<td>MD22</td>
<td>'-123456.78'</td>
</tr>
<tr>
<td>12345678</td>
<td>MD02</td>
<td>'123457'</td>
</tr>
<tr>
<td>12345678</td>
<td>MD02T</td>
<td>'123456'</td>
</tr>
</tbody>
</table>

### Thousands separators and currency symbols

<table>
<thead>
<tr>
<th>Value</th>
<th>Option</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234567</td>
<td>MD0,</td>
<td>'1,234,567'</td>
</tr>
<tr>
<td>12345678</td>
<td>MD2,$</td>
<td>'$123,456.78'</td>
</tr>
</tbody>
</table>

### Left justification and padding

<table>
<thead>
<tr>
<th>Value</th>
<th>Option</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ML004*</td>
<td>'0***'</td>
</tr>
<tr>
<td>12345678</td>
<td>ML0010*</td>
<td>'12345678**'</td>
</tr>
<tr>
<td>12345678</td>
<td>ML0+12*</td>
<td>'12345678+****'</td>
</tr>
<tr>
<td>-12345678</td>
<td>ML0+12*</td>
<td>'-12345678-****'</td>
</tr>
<tr>
<td>12345678</td>
<td>ML0-12*</td>
<td>'12345678 ***'</td>
</tr>
<tr>
<td>12345678</td>
<td>ML0(12*</td>
<td>'12345678 ****'</td>
</tr>
<tr>
<td>-12345678</td>
<td>ML0(12*</td>
<td>'(12345678)**'</td>
</tr>
<tr>
<td>12345678</td>
<td>ML0C12*</td>
<td>'12345678 **'</td>
</tr>
</tbody>
</table>
Masked Decimal Input Conversion Examples

The behaviour of a masked decimal input conversion with invalid data provides close compatibility with other multivalue products but may not be exactly identical as there is some variation. The MD code will return a null string as its result. The ML and MR codes return the converted form of the leading numeric part of the data, ignoring any further characters. If there is no leading numeric part, the original data is returned unconverted. The QMBasic ICONV() function will return a status code of 1 when attempting to convert invalid data with the MD, ML and MR codes.

The following table sets out a variety of combinations of masked decimal conversion features.

<table>
<thead>
<tr>
<th>Value</th>
<th>Conversion</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>''</td>
<td>MD0</td>
<td>''</td>
</tr>
<tr>
<td>'0'</td>
<td>MD0</td>
<td>'0'</td>
</tr>
</tbody>
</table>

Sign handling

<table>
<thead>
<tr>
<th>Value</th>
<th>Conversion</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>'12345678'</td>
<td>MD0</td>
<td>'12345678'</td>
</tr>
<tr>
<td>'-12345678'</td>
<td>MD0</td>
<td>'-12345678'</td>
</tr>
<tr>
<td>'12345678-'</td>
<td>MD0</td>
<td>'-12345678'</td>
</tr>
<tr>
<td>'&lt;12345678&gt;'</td>
<td>MD0</td>
<td>'-12345678'</td>
</tr>
<tr>
<td>'(12345678)'</td>
<td>MD0</td>
<td>'-12345678'</td>
</tr>
<tr>
<td>'12345678CR'</td>
<td>MD0</td>
<td>'-12345678'</td>
</tr>
<tr>
<td>'12345678DB'</td>
<td>MD0</td>
<td>'-12345678'</td>
</tr>
</tbody>
</table>

Scale factors

<table>
<thead>
<tr>
<th>Value</th>
<th>Conversion</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>'123456.78'</td>
<td>MD2</td>
<td>'12345678'</td>
</tr>
<tr>
<td>'123456.78'</td>
<td>MD23</td>
<td>'1234567.8'</td>
</tr>
</tbody>
</table>

Currency symbols and thousands separators
'12,345' MD0, '12345'
'$123,456.78' MD2,$ '12345678'
'($123456.78)' MD22S(' '12345678')
Time Conversion (MT)

The time conversion code converts a time from its internal representation (number of seconds since midnight) to a string representing hours, minutes and seconds or vice versa.

The full format of this conversion code is

\[
\text{MT} \{\text{M}\} \{\text{H}\} \{\text{S}\} \{c\}
\]

where

\(\text{M}\) specifies that the time value to be converted is in milliseconds (output conversion only).

\(\text{H}\) specifies that the time is to appear in 12-hour format with either "am" or "pm" appended. If \(\text{H}\) is not specified, 24-hour conversion is used for output conversion and assumed for input conversion. If the AMPM.UPCASE mode of the \text{OPTION} command is active, the appended suffix is in uppercase (AM or PM).

\(\text{S}\) specifies that output conversion is to include the seconds field. If the \(\text{M}\) option is also present, a period and the three digit fractional seconds value will be included in the converted output. Input conversion determines whether seconds are included by examination of the data to be converted and may not include a fractional element.

\(c\) is the character to separate the hours, minutes and seconds fields. If omitted, a colon is used. The separator character should be enclosed in quotes if required to avoid syntactic ambiguity (for example \text{MT}'h' for French format times such as 09h30). A pair of adjacent quotes can be used to specify that no separator is to be inserted.

The optional \(\text{M}\), \(\text{H}\) and \(\text{S}\) qualifiers may be in any order.

Examples

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Data</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT</td>
<td>0</td>
<td>00:00</td>
</tr>
<tr>
<td>MT</td>
<td>31653</td>
<td>08:47</td>
</tr>
<tr>
<td>MT</td>
<td>63306</td>
<td>17:35</td>
</tr>
<tr>
<td>MTH</td>
<td>0</td>
<td>12:00am</td>
</tr>
<tr>
<td>MTH</td>
<td>31653</td>
<td>08:47am</td>
</tr>
<tr>
<td>MTH</td>
<td>63306</td>
<td>05:35pm</td>
</tr>
<tr>
<td>MTS</td>
<td>31653</td>
<td>08:47:33</td>
</tr>
<tr>
<td>MTS</td>
<td>63306</td>
<td>17:35:06</td>
</tr>
<tr>
<td>MTHS</td>
<td>63306</td>
<td>05:35:06pm</td>
</tr>
<tr>
<td>MTS.</td>
<td>63306</td>
<td>17:35:06</td>
</tr>
<tr>
<td>MTMS</td>
<td>33888250</td>
<td>09:24:48.250</td>
</tr>
<tr>
<td>MT''h''</td>
<td>63306</td>
<td>17h35</td>
</tr>
<tr>
<td>MT''''</td>
<td>63306</td>
<td>1735</td>
</tr>
</tbody>
</table>

See also:
Dates, Times and Epoch Values
Thousands Conversion (MK)

The thousands conversion code converts an integer numeric value to a rounded thousands form. It behaves identically when used as an input or output conversion.

The full format of this conversion code is

\[ \text{MK}\{n\} \]

where

\[ n \]

specifies the width of the result and defaults to 3 if omitted.

The MK conversion code constrains an integer value to a given width, rounding to a number of thousands, millions or billions using suffixes of K, M or G respectively if needed. If the data cannot be represented in the requested width, an asterisk is returned.

Examples

<table>
<thead>
<tr>
<th>Value</th>
<th>MK3</th>
<th>MK4</th>
<th>MK6</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>123</td>
<td>123</td>
<td>123</td>
<td>123</td>
</tr>
<tr>
<td>1234</td>
<td>1K</td>
<td>1234</td>
<td>1234</td>
</tr>
<tr>
<td>12345</td>
<td>12K</td>
<td>12K</td>
<td>12345</td>
</tr>
<tr>
<td>123456</td>
<td>*</td>
<td>123K</td>
<td>123456</td>
</tr>
<tr>
<td>1234567</td>
<td>1M</td>
<td>1M</td>
<td>1235K</td>
</tr>
<tr>
<td>12345678</td>
<td>12M</td>
<td>12M</td>
<td>123346K</td>
</tr>
<tr>
<td>123456789</td>
<td>*</td>
<td>123M</td>
<td>123M</td>
</tr>
<tr>
<td>1234567890</td>
<td>1G</td>
<td>1G</td>
<td>1235M</td>
</tr>
<tr>
<td>12345678901</td>
<td>12G</td>
<td>12G</td>
<td>12346M</td>
</tr>
<tr>
<td>123456789012</td>
<td>*</td>
<td>123G</td>
<td>123G</td>
</tr>
<tr>
<td>1234567890123</td>
<td>*</td>
<td>*</td>
<td>1235G</td>
</tr>
<tr>
<td>12345678901234</td>
<td>*</td>
<td>*</td>
<td>12346G</td>
</tr>
<tr>
<td>999000</td>
<td>1M</td>
<td>999K</td>
<td>999000</td>
</tr>
</tbody>
</table>
Radix Conversion (MB, MO, MX)

The radix conversion codes convert a number to/from binary (MB), octal (MO) or hexadecimal (MX).

Input Conversion

The MB, MO and MX conversions take a number represented by a character string of binary, octal or hexadecimal digits and converts it to an internal integer value. The Z suffix described below for output conversion is ignored on input conversion.

Addition of the 0C suffix to these codes (MB0C, MO0C, MX0C) takes a character string holding a series of binary, octal or hexadecimal digits and translates each group of 8, 3 or 2 digits to the corresponding ASCII character. If the source data is not an exact multiple of 8, 3 or 2 digits in length, as appropriate to the conversion type, implied leading zeros are added.

The MBUC, MOUC and MXUC conversion codes are similar to MB0C, MO0C and MX0C but take a character string holding a series of binary, octal or hexadecimal digits and translates each group of 16, 6 or 4 digits to the corresponding character. If used on a non-ECS mode system, attempting to convert data that would yield characters outside the 8 bit set will fail.

Output Conversion

The MB, MO and MX conversions convert a number to binary, octal or hexadecimal form as a character string. Non-integer values are truncated towards zero. Negative values are treated as unsigned 32 bit values. Leading zeros are suppressed by default. If the Z suffix is added to the conversion code (e.g. MXZ) leading zeros are added to form a 32 bit value or, if the source data is over 32 bits, a 64 bit value.

The addition of the 0C suffix to any of these conversion codes treats the source data as a character string and converts each character to its eight digit binary, three digit octal or two digit hexadecimal representation.

The MBUC, MOUC and MXUC conversion codes are similar to MB0C, MO0C and MX0C but return the data for each character as a sixteen digit binary, six digit octal or four digit hexadecimal value. Although supported on all versions of QM, these are of particular use of ECS mode systems where MB0C, MO0C and MX0C will not produce the correct result for characters outside the 8 bit set.

Examples

<table>
<thead>
<tr>
<th>Source data</th>
<th>Conversion</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1110</td>
<td>MB</td>
<td>14</td>
</tr>
<tr>
<td>37777777762</td>
<td>MO</td>
<td>-14</td>
</tr>
<tr>
<td>97AB</td>
<td>MX</td>
<td>38827</td>
</tr>
<tr>
<td>0100000101000010010000</td>
<td>MB0C</td>
<td>ABC</td>
</tr>
<tr>
<td>11</td>
<td>MO0C</td>
<td>ABC</td>
</tr>
<tr>
<td>101102103</td>
<td>MO0C</td>
<td>ABC</td>
</tr>
<tr>
<td>414243</td>
<td>MX0C</td>
<td>ABC</td>
</tr>
</tbody>
</table>
### Output conversion:

<table>
<thead>
<tr>
<th>Source data</th>
<th>Conversion</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>MB</td>
<td>10011</td>
</tr>
<tr>
<td>19</td>
<td>MO</td>
<td>23</td>
</tr>
<tr>
<td>19</td>
<td>MX</td>
<td>13</td>
</tr>
<tr>
<td>19</td>
<td>MBZ</td>
<td>0000000000000000000000010011</td>
</tr>
<tr>
<td>19</td>
<td>MOZ</td>
<td>00000000023</td>
</tr>
<tr>
<td>19</td>
<td>MXZ</td>
<td>00000013</td>
</tr>
<tr>
<td>ABC</td>
<td>MB0C</td>
<td>010000010100001001000111</td>
</tr>
<tr>
<td>ABC</td>
<td>MO0C</td>
<td>101102103</td>
</tr>
<tr>
<td>ABC</td>
<td>MX0C</td>
<td>414243</td>
</tr>
<tr>
<td>AB</td>
<td>MBUC</td>
<td>0000000000100000100000000000100001000010</td>
</tr>
<tr>
<td>AB</td>
<td>MOUC</td>
<td>000101000102</td>
</tr>
<tr>
<td>AB</td>
<td>MXUC</td>
<td>00410042</td>
</tr>
</tbody>
</table>
Radix Conversion (MCDX, MCXD)

Used as an output conversion, MCDX converts a number from decimal to hexadecimal and MCXD converts from hexadecimal to decimal. Used as an input conversion, the roles of these two codes are reversed.

The final character of the conversion code name is optional. They can be written as MCD and MCX.
Packed Decimal Conversion (MP)

The MP conversion code transforms a numeric value to or from packed decimal form (also known as Binary Coded Decimal, BCD).

The full format of this conversion code is

    MP

The MP conversion code used as an input conversion transforms a numeric value to packed decimal form where two digits are packed into each byte of the result string. Used as an output conversion, the MP conversion code expands a packed decimal value into an integer.

A negative value is indicated by the low order 4 bits of the final byte having the value 13 (hexadecimal D).

Because a packed decimal value always contains an even number of 4-bit values, a leading zero value is added at the start of the item if it would otherwise have an odd number of values.
Roman Numeral Conversion (NR)

The NR conversion code transforms a numeric value to or from Roman numeral notation.

The full format of this conversion code is

NR

The NR conversion code used as an output conversion transforms a numeric value to Roman numeral notation. A value of zero will be transformed to "N" in accordance with practice introduced around 725AD. Other positive values will be formed using the M, D, C, L, X, V and I symbols. Note that numbers over 4999 will have a long sequence of leading M characters.

Negative values will cause the output conversion to fail.

Used as an input conversion, the NR code transforms a number from Roman numeral notation.

Examples (Output conversion)

<table>
<thead>
<tr>
<th>Data</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>N</td>
</tr>
<tr>
<td>17</td>
<td>XVII</td>
</tr>
<tr>
<td>275</td>
<td>CCLXXV</td>
</tr>
<tr>
<td>1999</td>
<td>MCMXCIX</td>
</tr>
</tbody>
</table>
Pattern Matching Conversion (P)

The P conversion code attempts to match the supplied data against one or more pattern templates. The full format of this conversion code is

\[ P(\text{template})\{;\text{(template)}\ldots\} \]

where

\[ \text{template} \]

is a pattern template as for the MATCHES operator and must be enclosed in brackets. If more than one \text{template} is supplied, they may be separated by semicolons (;) or forward slashes (/).

The P conversion code tests whether the input data matches \text{template}. If a match is found, the original data is returned. If no match is found, a null string is returned.

If more than one \text{template} is provided, the input value is tested against each in turn.

A null input value always returns a null result.

**Examples**

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Data</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>P(3A)</td>
<td>123</td>
<td></td>
</tr>
<tr>
<td>P(3A)</td>
<td>ABC</td>
<td>ABC</td>
</tr>
<tr>
<td>P(1-2N)</td>
<td>74</td>
<td>74</td>
</tr>
<tr>
<td>P(1-2N)</td>
<td>123</td>
<td></td>
</tr>
<tr>
<td>P(2A);(3N)</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>P(2A);(3N)</td>
<td>AB</td>
<td>AB</td>
</tr>
<tr>
<td>P(2A);(3N)</td>
<td>123</td>
<td>123</td>
</tr>
<tr>
<td>P(2A)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Range Check Conversion (R)

The **R** conversion code checks whether a numeric value is within a specified range.

The full format of this conversion code is

\[
R_{n,m} \{ n_{1},m_{1} ... \}
\]

where

\[ n, m \]

specifies a range of numeric values. Negative values are allowed. If more than one \( n, m \) pair is supplied, they may be separated by semicolons (:) or forward slashes (/).

The **R** conversion code tests whether the input data is an integer value in the range \( n \) to \( m \). If the value is within the specified range, the conversion returns the input data. If the value is outside the range, a null string is returned.

If more than one \( n, m \) pair is provided, the input value is tested against each in turn.

A null input value always returns a null result regardless of the values of \( n \) and \( m \).

A non-numeric data item will return the original data. The **STATUS** function would then return 1 to indicate an error.

**Examples**

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Data</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>R3,5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>R3,5</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>R-2,0</td>
<td>-3</td>
<td></td>
</tr>
<tr>
<td>R-2,0</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>R1,3;6,8</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>R1,3;6,8</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>R1,3</td>
<td>0002</td>
<td>0002</td>
</tr>
<tr>
<td>R1,3</td>
<td>A</td>
<td>A</td>
</tr>
</tbody>
</table>
Soundex Conversion (S)

The S (Soundex) conversion code returns the Soundex code for a supplied character string.

The full format of this conversion code is

\[ S \]

The S code applies the Soundex algorithm to the data string, returning a four character sound code. For details of the Soundex algorithm, see the SOUNDEX() function.

Example

A file has a customer name. The S conversion can be used to show the equivalent sound code.

<table>
<thead>
<tr>
<th>Name</th>
<th>Soundex</th>
</tr>
</thead>
<tbody>
<tr>
<td>Smith</td>
<td>S530</td>
</tr>
<tr>
<td>Jones</td>
<td>J520</td>
</tr>
<tr>
<td>Robinson</td>
<td>R152</td>
</tr>
</tbody>
</table>
Substitution Conversion (S)

The S (substitution) conversion code allows an application to handle zero or null data items as a special case.

The full format of this conversion code is

$$S;\text{value1};\text{value2}$$

The S code returns a value determined by value1 if the source data is not zero or null, or the value determined by value2 if the source data is zero or null.

The value1 and value2 items may be:

- A number specifying the field from @RECORD (the current record being processed in a query) to be returned. A value of zero returns the content of @ID.
- A literal string enclosed in single quotes, double quotes or backslashes.
- An asterisk indicating that the original data is to be returned.

If either value is omitted, it defaults to a null string.

Examples

A file has a date field that contains zero when it is not significant. Using a date conversion would return this as 31 Dec 1967. The S conversion code could be used to replace the zero by a null string before applying the date conversion. The dictionary conversion code field could be

$$S;\ast;'''\text{VM}D4\text{DMY}$$

The following table shows a variety of conversions for a data record of "F1FMF2" and a record id of "ID".

<table>
<thead>
<tr>
<th>Conversion</th>
<th>Data</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>S;1;2</td>
<td>0</td>
<td>F2</td>
</tr>
<tr>
<td>S;1;2</td>
<td></td>
<td>F2</td>
</tr>
<tr>
<td>S;1;2</td>
<td>5</td>
<td>F1</td>
</tr>
<tr>
<td>S;1;2</td>
<td>XX</td>
<td>F1</td>
</tr>
<tr>
<td>S;\ast;ZZ'</td>
<td>0</td>
<td>ZZ</td>
</tr>
<tr>
<td>S;\ast;ZZ'</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>S;0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>S;0</td>
<td>1</td>
<td>ID</td>
</tr>
<tr>
<td>S;;0</td>
<td>0</td>
<td>ID</td>
</tr>
</tbody>
</table>
**File Translation Conversion (T)**

The T conversion code uses the source data as a record id to the named file and returns data from this record.

The format of the conversion code is

\[ T_{file}cvio \]

where

- `file` is the file name. This may be prefixed by either DICT followed by a space or by an asterisk with no space to reference the dictionary part of the file. Where necessary to avoid ambiguity, the file name may be enclosed in quotes.

- `c` identifies the action to be taken if the requested record does not exist or the field to be extracted is null. This is a single letter:
  - C Returns the record id.
  - V Displays a warning message and returns a null value.
  - X Returns a null value
  - I Like V for input conversion, C for output conversion
  - O Like C for input conversion, V for output conversion

- `v` specifies the value position to be extracted from the returned data. If omitted, all values are returned with value and subvalue marks replaced by spaces.

- `i` specifies the field position of the data to be returned when performing an input conversion.

- `o` specifies the field position of the data to be returned when performing an output conversion.

This conversion code is closely related to the TRANS() function. Where possible, TRANS() should be used in preference to the T conversion code for best performance.
Text Substring Conversion (T)

The text substring extraction code returns a portion of the source data.

The format of the conversion code is

\[
T_{m,n} \\
T_m
\]

This code behaves identically with ICONV() or OCONV(). The first form returns \( n \) characters starting at character \( m \) from the source data. The second form returns the last \( m \) characters of the source data.

When used as the conversion code in a dictionary item as part of a query, the behaviour of the \( T_m \) form is dependent on the justification specified in the dictionary. For left aligned items, the first \( m \) characters of the source data are returned. For right justified items, the last \( m \) characters are returned.
User Defined Conversions (U)

Users may add their own conversion codes to the system by writing a QMBasic subroutine to perform the conversion. Pick style user exits are provided in this way.

The format of the conversion code is

- \textbf{U} \textit{subrname}
- \textbf{U} \textit{subrname.extension}

where

- \textit{subrname} is the catalogue name of the subroutine to be called. To allow creation of substitutes for Pick user exits, this name may commence with a digit.
- \textit{extension} is optional qualifying information for the conversion code. This can be accessed within the subroutine in the \texttt{@CONV} variable. The separator before the \textit{extension} may be a period, a comma or a semicolon.

The subroutine should have four arguments:

\textit{subrname}(\textit{result}, \textit{src}, \textit{status}, \textit{oconv})

where

- \textit{subrname} is the name of the subroutine. This is the same as the conversion code but with the U prefix removed.
- \textit{result} should be set to the result of the conversion.
- \textit{src} is the item to be converted.
- \textit{status} is the value to be set for the \texttt{STATUS()} function. This defaults to 1 if not updated by the subroutine.
- \textit{oconv} indicates whether this is an input (0) or output (1) conversion.

User Exits

See the U50BB subroutine in the BP file of the QMSYS account for an example of a Pick user exit routine. There are many other user exit functions in this file though it is recommended that their use should be replaced by the equivalent in-line code when migrating an application to QM.

Handling Undefined Conversion Codes

The U conversion code prefix is primarily intended for emulation of the "user exits" found in Pick style systems. QM provides a more generalised mechanism for handling undefined conversion codes. If a conversion is attempted using a conversion code that QM does not recognise, including U codes for which there is no catalogued handler, it will check for the presence of a catalogued function named UNDEFINED,CONV. If this is present, it is called in a similar manner to the user exit handlers described above but with a fifth argument that contains the conversion code.
UNDEFINED.CONV(result, src, status, oconv, conv.code)

where

Example

The following example subroutine implements a UDT conversion code which converts a combined date/time value as returned by SYSTEM(1005) to a text representation or vice versa.

```fortran
SUBROUTINE DT(result, src, state, is.oconv)
    IF src = '' THEN
        result = ''
    END ELSE IF IS.OCONV THEN
        D = IDIV(src, 86400)
        T = REM(src, 86400)
        result = OCONV(D, 'D2/DMY') : ' ' : OCONV(T, 'MTS')
    END ELSE
        D = FIELD(src, ' ', 1)
        T = FIELD(src, ' ', 2)
        result = ICONV(D, 'D2/DMY') * 86400 + ICONV(T, 'MTS')
    END
    state = 0
    RETURN
END
```
Character Encoding Conversion (X)

The X conversion code applies a character encoding.

The format of the conversion code is

\[ X_{\text{name}} \]
\[ X_{\text{name}}.\text{modes} \]

where

\text{name} is the name of the encoding to be applied.

\text{modes} is a list of case insensitive mode qualifiers to the specified encoding, each represented by a single character. The \text{modes} are separated from the \text{name} by a period.

Used as an output conversion, the X conversion code takes internal data and encodes it to an external representation such as UTF-8. Used as an input conversion, the data is transformed from its external representation to a QM character string. Although primarily intended for use on ECS mode systems, the X conversion code is supported on non-ECS mode systems but limited to the 8 bit character set.

See Character Encodings for a description of the built-in encodings.

Users can add their own encodings by cataloguing a pair of QMBasic subroutines named ENCODE.name and DECODE.name where \text{name} is the encoding name to be defined. These subroutines should be declared as

\begin{verbatim}
SUBROUTINE ENCODE.name(RESULT, STR, MODES, STATUS)

and

SUBROUTINE DECODE.name(RESULT, STR, MODES, STATUS)
\end{verbatim}

where

\begin{align*}
\text{RESULT} & \quad \text{is the returned encoded / decoded data} \\
\text{STR} & \quad \text{is the data string to transform} \\
\text{MODES} & \quad \text{is the modes qualifier from the conversion code} \\
\text{STATUS} & \quad \text{is set to the value to be returned by the STATUS() function after the conversion.}
\end{align*}

Encodings defined in this way can be used in all places where a character encoding name is used by QM.

The status value returned by the subroutine should follow the standard rules for conversion codes:

\begin{itemize}
\item 0 \quad \text{Successful}
\item 1 \quad \text{Data cannot be converted}
\item 2 \quad \text{Conversion code not recognised}
\item 3 \quad \text{Data converted may be faulty. This might be used, for example, where an input conversion on a non-ECS system yields a character outside the 8 bit set that has been replaced by some substitute character.}
\end{itemize}
Field Extraction ($<f,v,s>$)

The field extraction code, only supported for output conversion, extracts a field, value or subvalue from the source data. It is used most frequently in conjunction with other conversion codes.

$<f>$ Extracts field $f$.
$<f,v>$ Extracts field $f$, value $v$.
$<f,v,s>$ Extracts field $f$, value $v$, subvalue $s$. 
3.11 Format Specifications

Format specifications appear in field 5 of C-type, D-type and I-type dictionary items to determine the default format of output from the LIST and SORT query processor commands. They are also used in the QMBasic FMT() function and with the query processor FMT keyword.

The full form of a format code is

\[
\{field.width\} \{fill.char\} justification \{n\{m\}\} \{conv\} \{mask\}
\]

where

- \(field.width\) is the width of the field into which the data is to be formatted. If \(field.width\) is omitted, \(mask\) must be specified.
- \(fill.char\) is the character to be used to expand the string to \(field.width\) characters. If omitted, a space is used by default. Where \(fill.char\) is a digit, it must be enclosed in single or double quotes.
- \(justification\) indicates the justification mode to be applied. It takes one of the following values:
  - \(C\) specifies centered justification. The data is centered in a field of \(field.width\) characters, additional \(fill.char\) characters being added to either side if the data is shorter than \(field.width\). If the data is longer than \(field.width\), text marks are inserted at intervals of \(field.width\) from the start of the data.
  - \(L\) specifies left justification. The data is left aligned in a field of \(field.width\) characters, additional \(fill.char\) characters being appended if the data is shorter than \(field.width\). If the data is longer than \(field.width\), text marks are inserted at intervals of \(field.width\) from the start of the data.
  - \(R\) specifies right justification. The data is right aligned in a field of \(field.width\) characters, additional \(fill.char\) characters being inserted at the start if the data is shorter than \(field.width\). If the data is longer than \(field.width\), text marks are inserted at intervals of \(field.width\) from the start of the data.
  - \(T\) specifies text justification. Text marks are inserted to break the data into fragments of no more than \(field.width\) characters, aligning breaks onto the positions of spaces in the data. Where there is no suitable space at which to break the data, the text mark is inserted \(field.width\) characters after the last break position. The final fragment is padded using \(fill.char\) to be \(field.width\) characters in length.
    When the data is displayed, output moves to a new line where a text mark is present in the formatted data.
  - \(U\) specifies left justification and is treated identically to the \(L\) code by the QMBasic FMT() function. Within the query processor, data formatted with this code that is wider than...
field.width is not wrapped over multiple lines but extends into the space to its right, possibly overwriting whitespace in later columns.

$n$ specifies the number of decimal places to appear in the result when formatting numeric data. The value is rounded in the normal manner. If $n$ is zero, the value is rounded to an integer.

$m$ specifies the scaling factor to be applied. The value being formatted is scaled by moving the decimal point $m - p$ places to the left, where $p$ is the current precision value.

$conv$ is any meaningful combination of the following codes:

- $\$ specifies that the national currency symbol should be used as a prefix to the converted data on output conversion and may be present on input conversion. The default currency symbol is a dollar sign but this may be changed by use of the NLS command or the SETNLS QMBasic statement.
- $,$ indicates that the national language convention thousands delimiter is to be inserted every third digit to the left of the decimal point when converting numeric data. This delimiter defaults to a comma.
- $B$ appends db to negative numbers, two spaces to positive numbers. Use the CRDB.UPCASE keyword of the OPTION command to change this to DB.
- $C$ appends cr to negative numbers, two spaces to positive numbers. Use the CRDB.UPCASE keyword of the OPTION command to change this to CR.
- $D$ appends db to positive numbers, two spaces to negative numbers. Use the CRDB.UPCASE keyword of the OPTION command to change this to DB.
- $E$ encloses negative number in angle brackets (<…>). Positive numbers are followed by a single space.
- $M$ appends a minus sign to negative numbers.
- $N$ suppresses any sign indicator.
- $Z$ indicates that a value of zero should be represented by a null string.

$mask$ specifies a mask to be used to format the data. If omitted, field.width must be specified. Both can be used together.

The mask consists of a character string containing #, * or % characters and other characters. Each #, * or % is substituted by one character from the source data. Other characters are copied directly to the result string. Multiple #, * or % characters may be represented by a single #, * or % followed by a number indicating the number of characters to be inserted. Characters having special meaning within the format string may be prefixed by a backslash () to indicate that they are to be treated as text.
The value 1234567 with a format specification of 9L#2-#3-#2 would return 12-345-67.

Where the mask specifies more characters than in the data being converted, positions corresponding to # characters in the mask are replaced by the fill.char, positions corresponding to * characters in the mask are replaced by asterisks and positions corresponding to % characters in the mask are replaced by zeros. If the data is left aligned, the padding is inserted in the rightmost positions. If the data is right aligned, the padding is inserted in the leftmost positions.

If the mask specifies fewer characters than in the data being converted, part of the source data will be lost. A left aligned format will truncate the source data and a right aligned format will lose data from the start of the source.

Data formatting attempts to handle the data as a number if the decimal places, currency symbol, comma insertion or null zero features are included in the format specification. If these features are all absent, or if the data cannot be converted to a number, it is handled as a string. The difference in handling is relevant when processing data such as a string with leading zeros.

**Special Case**

For compatibility with other multivalue products, QM supports a special case of a format code that is just a number. If this is a single digit, it is treated as being the number of decimal places ($n$ from the descriptions above). If it is two digits, the first is the number of decimal places and the second is the scale factor ($m$ above). No other formatting is applied.

**Format Code Examples**

The following table shows some uses of format codes.

<table>
<thead>
<tr>
<th>Value</th>
<th>Format code</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>'ABCD'E</td>
<td>'8L'</td>
<td>'ABCD    '</td>
</tr>
<tr>
<td>'ABCD'E</td>
<td>'8R'</td>
<td>'  ABCDE   '</td>
</tr>
<tr>
<td>'ABCD'E</td>
<td>'8**L'</td>
<td>'ABCD***    '</td>
</tr>
<tr>
<td>'0012345'</td>
<td>'8R'</td>
<td>'  12345'</td>
</tr>
<tr>
<td>'0012345'</td>
<td>'8RZ'</td>
<td>'  12345'</td>
</tr>
<tr>
<td>'0000000'</td>
<td>'8RZ'</td>
<td>'  000000'</td>
</tr>
<tr>
<td>'12345'</td>
<td>'80*R'</td>
<td>'  12345'</td>
</tr>
<tr>
<td>'1234567'</td>
<td>'15P'</td>
<td>'1234567.00'</td>
</tr>
<tr>
<td>'1234567'</td>
<td>'15P$,,'</td>
<td>'$1, 234, 567.00'</td>
</tr>
<tr>
<td>'12345, 67'</td>
<td>'15*P$,,'</td>
<td>'*****$12, 345, 67'</td>
</tr>
<tr>
<td>'1234567'</td>
<td>'14L2'</td>
<td>'1234567.00'</td>
</tr>
<tr>
<td>'43'</td>
<td>'L####n'</td>
<td>'43 m'</td>
</tr>
<tr>
<td>'43'</td>
<td>'R####n'</td>
<td>'43m'</td>
</tr>
<tr>
<td>'43'</td>
<td>'&quot;0&quot;R####n'</td>
<td>'043m'</td>
</tr>
<tr>
<td>'1234567890'</td>
<td>'L####-#####'</td>
<td>'123-4567890'</td>
</tr>
<tr>
<td>'123456789'</td>
<td>'L#3-#3-#3'</td>
<td>'123-456-789'</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>'12345'</td>
<td>'L#4'</td>
<td>'1'</td>
</tr>
<tr>
<td>'12345'</td>
<td>'R#5'</td>
<td>'5'</td>
</tr>
<tr>
<td>'123456789'</td>
<td>'L#6'</td>
<td>'12345'</td>
</tr>
<tr>
<td>'123456789'</td>
<td>'R#5'</td>
<td>'56789'</td>
</tr>
<tr>
<td>'12345'</td>
<td>'L#6'</td>
<td>'12345'</td>
</tr>
<tr>
<td>'12345'</td>
<td>'R#6'</td>
<td>'12345'</td>
</tr>
<tr>
<td>'A LONG LINE'</td>
<td>'6T'</td>
<td>'A LONG LINE'</td>
</tr>
<tr>
<td>'A LONG LINE'</td>
<td>'7T'</td>
<td>'A LONG LINE'</td>
</tr>
<tr>
<td>'A LONG LINE'</td>
<td>'8T'</td>
<td>'A LONG LINE'</td>
</tr>
<tr>
<td>'A LONG LINE'</td>
<td>'8R'</td>
<td>'A LONG LINE'</td>
</tr>
<tr>
<td>'BANANAS'</td>
<td>'3T'</td>
<td>'BANANAS'</td>
</tr>
<tr>
<td>'1.236'</td>
<td>'2'</td>
<td>'1.24'</td>
</tr>
</tbody>
</table>
3.12 Locks

In order to prevent undesirable interaction between processes, QM provides a system of locks. Consider, for example, a process that reads a record, decrements a value in that record and writes it back to the file. There is no problem here if only one process is operating. If, however, there is a second process performing a similar operation on the file, there is a danger that both processes read the record, then both write back an updated copy. Only one of the updates will actually occur as they both started from an identical version of the original data, unaware that there was another process updating the record.

Three types of lock are available on files: file locks, read locks and update locks. Each has a different role and care should be taken to use them correctly.

A **file lock** applies to the entire file and prevents any other user from obtaining any sort of lock on the file or the records therein. File locks are usually only needed during operations that must handle the file in a consistent "snap shot" manner when, for example, summing the values of some field from all records. A process can only acquire a file lock if no other process has any locks on the file or its records. Conversely, no other process can obtain any other sort of lock within the file while the file lock is active. Use of file locks can have a severe effect on performance and hence they should only be used where absolutely necessary.

A file lock is obtained by the **FILELOCK** statement and is released by the **FILEUNLOCK** or **RELEASE** statements, or on closing the file.

A **read lock** (sometimes called a **shared lock**) applies to an individual record and prevents other processes from obtaining the file lock or an update lock on the same record. Any number of processes may acquire read locks on the same record at the same time. An attempt to obtain a read lock will fail if another process has the file locked or has an update lock on the record. Read locks can be used to ensure that a program sees a consistent view of a set of records, without the risk that some other process has changed any of these records.

An **update lock** also applies to an individual record and prevents other processes from obtaining the file lock or either type of record lock on the same record. Only one process may acquire an update lock on any individual record at a time. An attempt to obtain an update lock will fail if another process has the file is locked or has a read or update lock on the record.

A read or update lock may be acquired on a record that does not exist in the file. This provides a means of locking a record that is about to be written for the first time.

Read and update locks are obtained by the **READL** or **READU** statements (and others). These locks are released on closing the file, by the **RELEASE** statement or on return to the command prompt. Additionally, read and update locks are normally released by writing or deleting the locked record, however, the **WRITEU** and **DELETEU** statements provide a means of writing or deleting without releasing the lock.

Although it is possible to hold very many record locks at a time, this tends to indicate poor application design and may have an adverse effect on performance. The system wide limit on the number of concurrent locks is determined by the **NUMLOCKS** configuration parameter. If this limit is reached, the program will behave as though the record is locked by another user, taking the **LOCKED** clause of the relevant QMBasic statement with a **STATUS()** value of 0 or, if no **LOCKED** clause is present, waiting for space to become available. A message will be written to the system error log file. The **MAXRLOCKS** configuration parameter limits the number of record locks that a single process may hold at one time. Use of this optional limit can prevent run-away processes filling the internal lock tables.
Where an attempt to obtain a lock fails, the QMBasic language provides two methods of handling the situation. A program may either wait automatically for the lock to be released or it may regain control and take some action of its own (see the LOCKED clause of the QMBasic file handling statements). For compatibility with some other multivalue database products, the LOCK.BEEP mode of the OPTION command can be used to cause the system to emit a beep at the terminal once per second while waiting for a record lock or file lock.

A process is not affected by its own locks. Thus it is possible to take a record lock on a record from a file for which the file lock is held or, conversely, to take the file lock while holding one or more record locks. Taking a read lock on a record for which an update lock is held or vice versa will convert the lock type.

Locks are associated with the underlying operating system file, not the VOC reference to the file. QM will correctly track locks relating to the same file however it was opened, including when a file uses soft links on Linux and has multiple pathnames. Where a file has been opened more than once simultaneously by a single process, the locks are released on the final close of the file.

Locks are also associated with the particular file variable referenced when they are acquired. Thus, if an application opens the same file more than once, closing one of the file references will automatically release any locks acquired using that file variable but leave other locks in place. Similarly, use of the form of the RELEASE statement that takes only a file variable will release locks associated with that file variable. This mechanism ensures that developers do not need to be aware of how other modules within the application operate.

**Lock Rule Enforcement**

The lock handling operations of QMBasic only operate with correctly structured applications. The non-locking versions of the READ and WRITE operations, etc, take no part in the locking system and hence can access a file regardless of its lock state. The individual statement descriptions give more information.

A correctly written application never writes or deletes a record without locking it first, however, for compatibility with other multivalue database products, this is not enforced by default except within transactions. A poorly written program that uses READ in place of READU and then writes the record could overwrite a record that is locked by another user. The MUSTLOCK configuration parameter can be used to enforce tight control of locks, eliminating this potential problem.

Enabling lock rule enforcement may not be easy when porting existing applications to QM. Because multivalue systems have not enforced these rules in the past, programmers sometimes omitted use of locks in situations where there would never be contention. For example, overnight processing might not use locks because the developer knew that it is run at a time when there are no other users on the system. Programs that write a record that is known not to exist also appear not to need locks. Both of these examples actually represent bad programming practice as the assumption made by the developer may subsequently turn out not to be true due to changes in business operation.

**Deadlocks**

A deadlock occurs when two processes are each waiting for a lock held by the other process. For example, process 1 locks record A and process 2 locks record B. Process 1 then goes on to lock record B, causing it to wait for the lock to be released by process 2. If at this stage process 2 tries to lock record A, the two processes are waiting for each other. Deadlocks can be more complex, involving multiple processes.

Deadlocks are totally avoidable by careful application design. If in the above example a rule had been adopted such that record A must always be locked before record B, a deadlock cannot occur. Setting a sequence in which locks must be acquired (e.g. order, customer, part) is gives a simple solution but
may not always be practical. An order entry system in which locks on the part records are maintained until entry of the order is complete will expose the system to deadlocks if, for example, one customer orders parts 101 and 102 at the same time as another customer orders parts 102 and 101. In this case, the best solution is to devise a scheme where part records do not remain locked.

The default behaviour of QM is to allow deadlocks to happen, leaving it to the developer to analyse and resolve the cause. Setting the DEADLOCK configuration parameter to a non-zero value causes QM to check whether a deadlock would occur if the process waited for a record lock that is held by another process. If a deadlock would occur, the process attempting to get the lock is aborted. A diagnostic messages showing the locks involved is displayed.

Setting the DEADLOCK configuration parameter to 2 extends the diagnostics by recording the state of the locking system in an item named deadlock.\( n \) in the temp subdirectory of the QMSYS account where \( n \) is the QM user number. The information written has four lines for each lock and a blank line between locks. Line 1 has the QM user number and user name. Line 2 has the internal file number and its path name. Line 3 has the lock type and, for a record lock, the record id. On an ECS mode system, the record id is encoded to UTF-8 form. Line 4 shows the date/time at which the lock was acquired.

Note that resolution of a deadlock by aborting a process carries a risk of data integrity errors if the process was performing a related series of updates some of which had already been written when the deadlock occurred. Use of transaction programming would ensure that the entire update sequence could be rolled back at the abort.

**Task Locks**

QM also provides task locks, sometimes known as process synchronisation locks, that are not related to any particular file and are typically used to ensure that some task cannot be performed by two users simultaneously.

A task lock is simply a numbered entry in a 64 element locking table. A process acquires a task lock using the LOCK command or the QMBasic LOCK statement. If the lock is already owned by another user, the process either waits for it to be released or handles the situation for itself. On completion of the task, the process can release the lock using the CLEARLOCKS command or the QMBasic UNLOCK statement.

Task locks can be difficult to use because the lock is not related to a file, record, etc and the application designer must choose one that is not also used for some other purpose. Because task locks are shared across all accounts, this implies a possible unwanted interaction between different unrelated applications.
3.13 Select Lists

Select lists are lists of things to be processed, usually record keys from a file. The list may contain all record keys or only those where the key or record data meets some specified criteria. Using select lists simplifies and speeds up many data processing operations using commands or within QMBasic programs.

There are two types of select list; numbered lists and select list variables.

Numbered Select Lists

The eleven numbered select lists are numbered 0 to 10 and are visible to all parts of a QM session. List 0 is referred to as the default select list and is used by some verbs such as COPY to determine the records (or files for some other verbs) to be processed.

With QM in its default modes, numbered select lists are created by the SELECT or SSELECT query processor commands. The SELECT command builds a list of keys of records meeting the specified criteria but with no apparent ordering to the list. The SSELECT command is similar but the list is in order of record key value. The SELECT command is faster both during generation of the list and subsequent processing of records as its order reflects the placement of records within the file.

Numbered select lists can also be created by the QMBasic SELECT statement. This statement builds a list of all records in the file and provides no means of including or excluding records by selection criteria. Programs can then read keys sequentially from the list using the READNEXT statement.

Numbered select lists may also be saved to records in the $SAVEDLISTS file using SAVE.LIST and later restored using GET.LIST. Lists that have been written to other files by QMBasic programs may be restored using FORM.LIST. The EDIT.LIST command allows editing of select lists. Saved select lists may be copied using COPY.LIST, deleted using DELETE.LIST or merged using MERGE.LIST.

An active numbered select list can be viewed and edited by use of the SHOW.LIST command.

Because the numbered select lists are visible to all parts of a QM session, a list created in the command language may be used by a QMBasic program, or vice versa. Named lists saved to the $SAVEDLISTS file can be shared across separate QM processes. They are often used to create a list of records that may be processed many days later. For example, an application might generate a list of overdue accounts and immediately use this list to send letters to the relevant clients. A saved copy of this list might then be used two weeks later to check if the overdue accounts have now been paid.

A select list represents a "snapshot" of the file at the time when it was generated. Adding, deleting or modifying records will not affect the select list. Thus, if a file may be modified by another process between generation of the select list and retrieval of records for processing, the program must allow for records that have been deleted or no longer meet the selection criteria.

A special type of select list, an exploded list, is constructed using the BY.EXP or BY.EXP.DSND keywords of the query processor. In an exploded select list, the multivalued field from which it was created generates a separate entry for each value or subvalue. The internal representation of this list includes information to identify the value and subvalue positions of the corresponding data element. This can be retrieved by the QMBasic READNEXT statement and is used automatically by some operations within the query processor.

The CLEAR.SELECT mode of the OPTION command causes the default select list (list 0) to be cleared on return to the command processor unless the program executed by the command uses the
**KEEP.SELECT** statement to indicate that the list should be retained. This is needed in programs whose role is to generate a select list for processing by subsequent commands.

**Select List Variables**

A QMBasic program can create a select list in a variable by using the **SELECTV** statement. This can be processed item by item within the application by use of **READNEXT** in the same way as a numbered list. There is no practical limit to the number of separate select list variables that may exist at one time.

**Directory Files**

When building a select list from a directory file, items that are considered to be hidden files by the operating system are normally included. The **DIR.SEL.OMIT.HIDDEN** mode of the **OPTION** command can be enabled to omit the hidden items.

**Pick Compatibility Options**

To ease migration from Pick style multivalue database products, the **PICK.SELECT** mode of the **$MODE** compiler directive can be used to make **SELECT** produce select list variables instead of numbered lists.

**Partial Select**

The QMBasic **SELECT, SELECTN** and **SELECTV** statements use an optimised method for processing hashed files such that each group is examined only when the record keys are extracted from the select list. This reduces disk transfers and gives better application performance than constructing the entire list in one operation. It also ensures that very large select lists do not require the entire list to be in memory. This overlapping of processing with record selection means that, if the application writes new records while the select list is being processed, these new records may be seen later in the operation.

A similar partial selection mechanism is applied to directory files, reading ids from the underlying operating system directory in burst of no more than the value of the **DIRSELSZ** configuration parameter.

QMCClient also provides the ability to perform partial selects.

Some operations such as the QMBasic **READLIST** statement or building a select list in a transaction require that the entire list is available and will force immediate completion of the selection process.
3.14 Alternate Key Indices

An alternate key index provides a method to access data file other than by the primary key (record id).

Consider a file holding information about orders with, for example, 100,000 records in it. For simplicity, assume that these records are made up of 10 orders from each of 10,000 customers. To locate all the orders placed by a specific customer would require all 100,000 records to be processed to find the 10 that we want.

Using an alternate key index on the customer number field of the order records, QM can look up the customer in the index and then go directly to the 10 order records. The application goes 10,000 times faster.

An alternate key index is created using the CREATE_INDEX command. This defines the index content but does not populate it. The BUILD_INDEX command builds the actual index and activates it. From that point forwards, QM will maintain the index automatically and the query processor will use it automatically. No changes are required to application software.

The functions of the CREATE_INDEX and BUILD_INDEX commands are combined in the MAKE_INDEX command.

Once an index has been built, it is maintained totally automatically by QM such that it is impossible to write or delete a record without the corresponding index updates being applied. It is essential that applications lock records (or the entire) file when writing or deleting in a file that uses indices. If this rule is obeyed, the only situation where it could be possible for an index not to correctly reflect the data in the file is after a power failure where some data may not have been flushed to disk. In this case, the REBUILD_ALL_INDICES command can be used to rebuild all indices in an account.

The query processor will also use the index totally automatically where it appears relevant. If the query processing speed suggests that it is not using an index where use was expected, the REQUIRE_INDEX keyword can be added to the query. If an index cannot be used, a diagnostic message is displayed to show the reason.

Indices can be built on real data stored in the file (dictionary D-type, A/S-type without a correlative, or E-type) or on calculated values (dictionary C/I-type or A/S-type with a correlative). When using calculated values, it is essential that the expression relies only on data from the file to which the index applies and is not time variant. Thus an index using data retrieved from another file using the TRANS() function, a T-conversion or a subroutine that performs a read will fail because the index will not be updated if the remote file is modified. Similarly, an index built using a calculation that uses the date or time (age calculated from a date of birth, for example) will fail because the expression does not always return the same output value for the same input.

These are both examples of the one and only rule that determines whether an index based on a calculated value will work: The virtual attribute expression must always return the same value when evaluated for the same input data.

When using a C-type index or an I-type that calls a subroutine, the index expression must not cause any write or delete operations to occur in the file to which the index applies.

For an index to be effective, each entry in the index should lead to a very small proportion of the data in the file. Index entries that lead to very large numbers of records are less effective and may also be costly to access or update. The worst case of this is indexing on a simple yes/no value.

A file may have up to 32 indices. The more indices there are, the longer it will take to update a record in the data file though this should be outweighed by the advantage of being able to use the
indices in queries. Also, it should be remembered that indexing on a multivalued field may require many index entries to be updated when writing a data record.

Indices can be deleted using the **DELETE.INDEX** command if they are no longer wanted. A report of any or all indices on a file can be produced with the **LIST.INDEX** command.

A large data import or other process involving extensive updates to a file may run faster with indexing disabled. A subsequent use of **BUILD.INDEX** may be significantly faster than the time saved by disabling the index because it can process the data in sorted order, optimising the way in which the underlying index structures are built. To enable this mode of operation, use the **DISABLE.INDEX** command to turn off the indices, perform the data import or processing, and then use **BUILD.INDEX** to reconstruct the indices. QM does not support the **ENABLE.INDEX** and **UPDATE.INDEX** commands found in some other multivalue products as the possibility of enabling the index without reconstructing it means that the index will not reliably reflect the stored data. QM’s implementation of **DISABLE.INDEX** allows individual indices to be disabled. Also note that while an index is disabled, it will not be used by the query processor.

**Index Structure and Case Sensitivity**

Internally, an AK is stored as a balanced tree structure in which the index keys (data field values) are in sorted order. When the index is built in case sensitive mode (the default), the sort order of the indexed data is in accordance with the ASCII character set. Thus an index might contain, in sorted order:

Lucy
MICHAEL
Mick
Peter
martin

For languages that use characters from the upper half of the character set (e.g. accented characters), this may not correspond to the conventional way in which the alphabet is sequenced.

An index can also be constructed in case insensitive form. Internally, this is performed by converting the indexed values to uppercase. Thus, the above example would be stored internally as

LUCY
MARTIN
MICHAEL
MICK
PETER

Notice how the sort order has also changed. Because of this, a case sensitive index cannot be used to resolve query processor selection clauses that use case insensitive operators and vice versa.

**Equivalent Indices**

An index is not identified internally by its name but rather by what it indexes. If two dictionary items have the same type, field number or expression, justification and single/multivalue status an index built on one item can be used in resolving selection criteria for the other. Thus, for example, if a file has two dictionary entries that reference the same date field but display it with a different conversion, and index built using one of the field names is equally applicable to the other. The **CREATE.INDEX** and **MAKE.INDEX** commands will not allow separate indices to be created for equivalent items.

**Changing an Index**
If the definition of the indexed data changes as a result of, for example, modifying an I-type expression that derives the indexed value, it is necessary to delete, recreate and build the index. This is because the definition of the index item is stored in the file at the point when the index is created. Subsequently modifying the dictionary will not affect the index. To simplify this process, both the \texttt{CREATE.INDEX} and \texttt{MAKE.INDEX} commands have a \texttt{DELETING} option that deletes any old index before creating the new index.

\textbf{Use of Indices in QMBasic}

A program can determine whether a file has indices by use of the \texttt{FILEINFO()} function with action key FL$AK:

\begin{verbatim}
IF FILEINFO(FVAR, FL$AK) THEN DISPLAY 'File has indices'
\end{verbatim}

The names of the indices that exist for a file can be determined using the \texttt{INDICES()} function:

\begin{verbatim}
NAMES = INDICES(FVAR)
\end{verbatim}

An extended form of the \texttt{INDICES()} function can be used to retrieve the details of a specific index by name:

\begin{verbatim}
INDEX.DATA = INDICES(FVAR, NAME)
\end{verbatim}

Programmers can gain access to the index itself using the QMBasic \texttt{SELECTINDEX} statement and advanced index scanning operations can be performed using \texttt{SELECTLEFT}, \texttt{SELECTRIGHT}, \texttt{SELE}$\text{\texttt{LEFT}}$ and \texttt{SETRIGHT} as described below.

\textbf{Scanning an Index}

Because the index keys (data field values) are in sorted order, the index can be used to find records where the indexed data field value lies within a specified range of values.

The \texttt{SELECTINDEX} statement is used by QMBasic programs to build a select list of records that have a specific value in an indexed field. As a side effect, this operation leaves a pointer into the index structure at the position identified by the value referenced in the \texttt{SELECTINDEX}. If there are no records with the requested indexed value, the pointer is left positioned where such a value should go. Thus, if the index contained references to records with indexed values 1, 3, 5 and 7, a \texttt{SELECTINDEX} operation to build a list of records with an indexed value of 4 would return an empty list but would leave the index pointer between the entries for 3 and 5.

Alternatively, the \texttt{SELECTLEFT} or \texttt{SETRIGHT} statements may be used to position the pointer at the extreme left or right of the indexed values.

The \texttt{SELECTRIGHT} or \texttt{SELECTLEFT} statements can be used to walk through the index, starting at the position of the index pointer and moving to the right (ascending order of indexed value) or left (descending order of indexed value). This operation creates a select list of records with the next indexed value in the direction of the scan.

Thus, to walk through an index from beginning to end in ascending order, a program would use \texttt{SELE}$\text{\texttt{LEFT}}$ to position at the start of the data and then perform a series of \texttt{SELECTRIGHT} operations until no further data is available. For example, if we have an orders file open as ORD.F and there is an index named CUST.NO for the customer number, we could process orders in ascending customer number with a loop such as:

\begin{verbatim}
SETLEFT 'CUST.NO' FROM ORD.F
LOOP
  SELECTRIGHT 'CUST.NO' FROM ORD.F SETTING CUST
\end{verbatim}
The outer loop walks through the index returning a select list of orders for each customer, also setting CUST to the corresponding customer number. The inner loop processes the orders for the customer.

There is a sample QMBasic class module named INDEX.CLS in the BP file of the QMSYS account, catalogued as !INDEX.CLS. This class allows an application to scan an index one record id at a time instead of one indexed value at a time. Full details of its use and internal operation can be found in the source code.

**Partial Key Look-up**

Sometimes it is useful to allow a user to type in a text item and, as each character is entered, display a list of items that start with as much as has been typed. The following code shows the principles behind a subroutine that provides this functionality.

```qmbasic
SUBROUTINE LOOKUP(FVAR, INDEX.NAME, WIDTH)
* FVAR = file variable of file to search
* INDEX.NAME = name of indexed field
* WIDTH = maximum number of characters in search value

$INCLUDE KEYIN.H

CRT @(-1) :

* Loop around gathering characters for the look-up key
KEY = ''
LOOP
    DISPLAY @(0,22) : '[': FMT(KEY, WIDTH:'L') : ']' :
    DISPLAY @(1+LEN(KEY), 22) :
    C = KEYCODEV()
UNTIL C = KV$RETURN ; * Terminate at return key

BEGIN CASE
CASE C = KV$BACKSPACE ; * Backspace
    KEY = KEY[1,LEN(KEY)-1]
CASE C >= 32 ; * Data character - add to key
    IF LEN(KEY) < WIDTH THEN KEY := CHAR(C)
END CASE

* Show data matching partial key
LN = 2
IF LEN(KEY) THEN
    SELECTINDEX INDEX.NAME, KEY FROM FVAR
    ACTUAL.KEY = KEY
    LOOP
        IF @SELECTED THEN
            DISPLAY @(0,LN) : ACTUAL.KEY
            LN += 1
```
In the code under the "Show data matching partial key" comment, the SELECTINDEX statement attempts to find a perfect match for the data in the KEY variable, building a select list of records that have that data. As a side effect the SELECTINDEX sets the scan position at the item that it found. If, as is more likely, it does not find a perfect match, an empty select list will be returned and the scan position is set to where the item would have been if it were present.

The loop under the SELECTINDEX then walks through the index until an item is encountered that no longer matches the entered text. On each cycle, it displays the indexed value that has been found.

Index Encryption

The ENCRYPT option to CREATE.IND or MAKE.IND creates an encrypted index. This is only possible when the data file being indexed uses record level encryption. The same encryption key is used for the indices and the data file. There is a significant performance penalty in using encrypted indices.

Tracking Sequential Record Ids

Sometimes an application may build an AK on the primary key. This can be useful where, for example, the ids are not simple sequential numbers. For this discussion, we will imagine a log file for which the record id is a timestamp value.

Using a left to right scan loop similar to that above would allow the application to walk through the log records. Instead of terminating the loop when we reach the end of the index data, we could use SLEEP to pause for a while and then try again. This would allow us to process new records as they are added to the log.

Use of SETRIGHT positions after the last record in ascending sort order. Instead of scanning the entire log from left to right, we could use SETRIGHT to position at the extreme right and then process only records added after our program starts.
3.15 Triggers

A trigger is an optional user-written subroutine associated with a QM data file and configured to be executed when certain file operations are performed. Executed before a write or delete, the trigger can be used to apply data validation. Executed after a record is written or deleted, the function can trigger other events such as related file updates. Trigger functions can also be executed after a read and before or after a clear file operation.

The trigger function is simply a catalogued QMBasic subroutine which is automatically executed as part of the file operation. The subroutine is passed a mode flag to indicate the action being performed, the record id, the record data (read or write operations) and a flag indicating whether the QMBasic ON ERROR clause is present. The subroutine may do whatever processing the application designer wishes. If the write or delete is to be disallowed, the pre-write or pre-delete trigger function should set the @TRIGGER.RETURN.CODE variable to a non-zero value such as an error number or an error message text to cause the write or delete to take its ON ERROR clause if present or to abort if omitted. The STATUS() function will return ER$TRIGGER when executed in the program that initiated the file operation. Programs should test STATUS() rather than testing for @TRIGGER.RETURN.CODE being non-zero to determine whether the trigger function has disallowed the write or delete as @TRIGGER.RETURN.CODE is only updated when the error status is set.

The trigger function name is limited to 32 characters and is set up using the SET.TRIGGER command. After it has been set up, the trigger function is loaded into memory when first needed and is called for all operations defined by the mode settings in the SET.TRIGGER command. Setting or removing a trigger function or modifying and re-cataloguing the trigger function will take immediate effect even if the file is open.

A trigger function on a hashed file will be called by all updates to the file in the modes for which the trigger is active. A trigger function on a directory file will be called only by updates from within QM and not for use of sequential file operations (WRITESEQ, WRITEBLK, etc), OSWRITE or OSDELETE. Some implications of this are that query processor CSV or delimited reports directed to a file and QMBasic compiler listing files will not call the trigger function.

If the trigger function is not in the catalogue or has the incorrect number of arguments, no error occurs until the first action that would call the function. Note that the trigger function must be visible to all accounts that may reference the file. Where a file is used by multiple accounts, this can be achieved by using global cataloguing, sharing a private catalogue, or ensuring that the VOC entry for a locally catalogued trigger function is present in each account. Although it would be possible for a shared file to use a different trigger function depending on the account from which it is referenced, this is not recommended.

**Important Note:** Because information about the trigger is stored in the file, copying the directory that represents a QM file that uses triggers will result in the new file also using the trigger.

The interface into a trigger function is:

```
SUBROUTINE name(mode, id, data, on.error, fvar)
```

where

- **name** is the trigger subroutine name.
- **mode** indicates the point at which the trigger function is being called:
  1. 0x01 FL$TRG.PRE.WRITE before writing a record
  2. 0x02 FL$TRG.PRE.DELETE before deleting a record
4 0x04 FL$TRG.POST.WRITE  after writing a record
8 0x08 FL$TRG.POST.DELETE after deleting a record
16 0x10 FL$TRG.READ after reading a record
32 0x20 FL$TRG.PRE.CLEAR before clearing the file
64 0x40 FL$TRG.POST.CLEAR after clearing the file

Other values may be used in the future. Trigger functions should be written to ignore unrecognised values. Use of the hexadecimal values (e.g. 0x04) makes it easier to form composite mode values as described below.

**id**
is the id of the record to be written or deleted.

**data**
is the data. This is a null string for a delete or clearfile action. When using triggers on data collection files, the data argument will be a data collection.

**on.error**
indicates whether the program performing the file operation has used the ON ERROR clause to catch aborts.

**fvar**
is the file variable that can be used to access the file. Beware that reading, writing or deleting records via this file variable may cause a recursive call to the trigger function. This argument can be omitted for compatibility with earlier releases.

When writing trigger functions, the original data of the record to be written or deleted can be examined by reading it in the usual way. Trigger functions should not attempt to write the record for which they are called. Neither should they release the update lock on this record as this could cause concurrent update of the record.

If a pre-write trigger modifies the value of the data argument, the modified data is written to the file but the variable used in the write operation that fired the trigger is not updated. In a post-read trigger, the data argument on entry to the trigger function contains the data read from the file and any modification made to data will be seen in the data returned from the read operation that fired the trigger. In all modes, changes to the value of id will not affect database updates in any way.

Trigger functions may perform all of the actions available to other QMBasic subroutines including performing updates that may themselves cause trigger functions to be executed.

The mode values correspond to bit positions in a binary value and hence a condition such as

```
IF MODE = 4 OR MODE = 8 THEN ...
```

is equivalent to

```
IF BITAND(MODE, 12) THEN ...
```

which can simplify some trigger functions. Writing this with the mode value in hexadecimal form can make it easier to combine modes.

```
IF BITAND(MODE, 0x0C) THEN ...
```

**Example**

The following simple trigger function could be used to capture all writes and deletes, logging the new record data in a "replication log" file which can then be used to maintain a copy of the data on a separate server. The same trigger function can be applied to many files. Whilst this is a valid example of the use of triggers, QM's data replication facilities provide a better solution.

```
SUBROUTINE REPTRIGGER(MODE, ID, DATA, ON.ERROR, FVAR)
```
$CATALOGUE GLOBAL

$INCLUDE KEYS.H

COMMON /REPLOG/RLG.F ;* REPLOG file variable, persistent across calls

FN = FILEINFO(FVAR, FL$VOCNAME)

IF NOT(FILEINFO(RLG.F, FL$OPEN)) THEN
    OPEN REPLOG TO RLG.F ELSE
        LOGMSG 'Unable to open REPLOG'
        RETURN
    END
END

BEGIN CASE
    CASE MODE = FL$TRG.PRE.WRITE
        RECORDLOCKU RLG.F, FN:' ':ID
        WRITE DATA TO RLG.F, FN:' ':ID
    CASE MODE = FL$TRG.PRE.DELETE
        RECORDLOCKU RLG.F, FN:' ':ID
        WRITE '' TO RLG.F, FN:' ':ID
END CASE
END

On first call, this trigger routine will open a file named REPLOG which is used for logging. Because the file variable is stored in a common block, the file will remain open for subsequent calls to the trigger function.

For write operations (mode FL$TRG.PRE.WRITE), the trigger routine writes a copy of the data to a record in REPLOG with its id constructed from the file name and record id of the write that is being replicated.

For delete operations (mode FL$TRG.PRE.DELETE), the trigger routine writes a null record to REPLOG with its id constructed in the same way.

A separate program, perhaps running via QMNet from another server, could periodically read all records from the REPLOG file and apply the changes to a copy of the original data, deleting the REPLOG entry. Note that the REPLOG is not a sequential audit trail but stores only the last update to any record. Thus a long series of updates will only ever produce a single REPLOG record.

There are two assumptions made by this example. Firstly, the data files being replicated never contain null records. We can, therefore, recognise a delete operation from the presence of a null record in the REPLOG. If this were not a safe assumption, we would need to add an extra field to the REPLOG data to say whether this was a write or a delete.

Secondly, there is an assumption that a file only has a single VOC entry defining it. If this were not the case, a combination of the filename (FN variable) and record id (ID argument) would not be a unique reference to the record. If this assumption was not valid, it would be possible to use FILEINFO() to get the pathname of the file and use this instead.

Use of a THEN/ELSE Clause in WRITE, MATWRITE or DELETE

For compatibility with the way in which triggers operate in some other multivalue products, the WRITE, MATWRITE and DELETE statements have an optional THEN/ELSE clause. Because
this would otherwise lead to a syntactic ambiguity, compilation of programs that use this clause requires the WRITE.DELETE.THEN.ELSE option of the $MODE compiler directive to be enabled. Once this is done, the optional THEN/ELSE clauses can be included in their usual position, after the ON ERROR clause. The on.error argument to the trigger function will be True if there is either an ON ERROR or THEN/ELSE clause, indicating that trigger errors are handled programmatically.

When a WRITE, MATWRITE or DELETE statement has a THEN/ELSE clause, a failure returned from a trigger function, typically as a result of a pre-write or pre-delete data validation error, will cause the ELSE clause to be executed instead of the ON ERROR clause.

**A Further Example**

Triggers can be a useful tool when trying to track down application errors. The example below checks whether the application is writing or deleting a record for which it does not have a lock.

```
SUBROUTINE TEST.TRG(MODE, ID, DATA, ON.ERROR, FVAR)
$CATALOG
  IF BITAND(MODE, 3) THEN
    IF RECORDLOCKED(FVAR, ID) # 2 THEN
      WHERE = SYSTEM(1002)
      LOGMSG 'No lock at line ' : WHERE<2, 2, 2> : ' of ' : WHERE<2, 1>
      END
  END
RETURN
END
```
3.16 Data Integrity Constraints

Multivalue database products do not normally impose validation rules when writing data to files. Instead, they largely rely on validation being performed when data is entered into the system. In some products, developers who wish to apply validation can do so via a pre-write trigger, rejecting the write request if the validation fails.

QM allows use of a pre-write trigger but also provides an alternative mechanism for hashed files where the validation rules for each field are defined in the dictionary and encoded into the data file in a manner that makes it impossible to skip the validation. This feature is known as Data Integrity Constraints. If both a pre-write trigger and data integrity constraints are present, the trigger is executed before the constraint validation.

The constraints are defined by an expression stored in field 12 of the A/D/S-type dictionary record for each field that is to have rules applied. Where multiple dictionary items define different views of the same field, the constraints expression must appear in only one of them. Constraints apply to the actual internal format data stored in the record, not calculated values from I-type records or A/S-type records with a correlative expression. Fields for which no constraints are defined are written with no validation.

Failure of constraint validation can be handled in several ways:

- If the full constraint validation system is enabled, the write operation is abandoned, taking the ON ERROR clause of the WRITE, if present, or the optional ELSE clause. If neither of these is present, QM throws a SYS.FILESYS.CONSTRAINTS exception or, if there is no suitable exception handler, aborts.
- Validation can operate in logging mode where details of a validation error is written to the error log but the write is allowed to continue.
- Validation can be disabled, allowing potentially invalid data to be written and subsequently reported using the VERIFY.CONSTRAINTS command described below.

These three modes can be selected separately on a file by file basis.

If a validation error occurs, the field number will be in @FNO and the corresponding field name will be in @FNAME.

Data integrity constraints are fully compatible with both record level and field level encryption. In the latter case, the constraints validation is skipped for any field that is not accessible by the user applying the update.

When using data replication, the data integrity constraints validation does not occur on the subscriber system as it would have already been done on the publisher.

Defining Constraints

A constraint definition is an expression similar to an I-type with some restrictions:

- The SUBR() function to execute a catalogued subroutine is not allowed.
- The TRANS(), RTRANS() and XLATE() functions or the equivalent T-conversions are not allowed.
- References to evaluated dictionary expressions (I-type, C-type, correlatives) are not allowed.

Three special data names are provided:

- The data for the field being validated is available as @DATA
- The record id is available as @ID
- The entire record is available as @RECORD
Note that these names are private to the constraints validation expression and are not the same as use of these names in other contexts such as @RECORD in a query processor command.

When validating a field that is defined as multivalued in the dictionary, **multivalued functions** such as the **EQS()** function should be used. If the result of the expression is multivalued, all values must comply with the constraint rule.

Compound expressions are supported with intermediate results stored as @1, @2, etc exactly as in an I-type expression.

Five special functions available only in constraints expressions are provided to handle common validation criteria:

### Integer with optional low/high range limits

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER(low, high)</td>
<td>INTEGER(1, 10)</td>
<td>Range of integer values</td>
</tr>
<tr>
<td>INTEGER()</td>
<td>INTEGER()</td>
<td>Integer, no range check</td>
</tr>
<tr>
<td>INTEGER(low)</td>
<td>INTEGER(4)</td>
<td>Integer, minimum value 4</td>
</tr>
<tr>
<td>INTEGER()</td>
<td>INTEGER(8)</td>
<td>Integer, maximum value 8</td>
</tr>
</tbody>
</table>

### Numeric with optional low/high range limits

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMBER(low, high)</td>
<td>NUMBER(0.5, 100)</td>
<td>Range of numeric values</td>
</tr>
<tr>
<td>NUMBER()</td>
<td>NUMBER()</td>
<td>Number, no range check</td>
</tr>
<tr>
<td>NUMBER(low)</td>
<td>NUMBER(0.5)</td>
<td>Number, minimum value 0.5</td>
</tr>
<tr>
<td>NUMBER()</td>
<td>NUMBER(40.5)</td>
<td>Number, maximum value 40.5</td>
</tr>
</tbody>
</table>

### String, minimum/maximum length limits

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>STRING(low, high)</td>
<td>STRING(1, 200)</td>
<td>String, length 1 to 200 chars</td>
</tr>
<tr>
<td>STRING(len)</td>
<td>STRING(10)</td>
<td>String, exact length 10 chars</td>
</tr>
</tbody>
</table>

### Pattern match, any number of patterns

MATCHING(pattern, pattern, ...) MATCHING('1-3N', '2N2A') Passes if matches either pattern

### Test presence of record

EXISTS(filename, id) EXISTS('STOCK', @DATA) Inter-file data integrity check

Note that all but the last of the functions listed above operate against the @DATA item without this being named in the function call.

If the field being validated is defined as multivalued in the dictionary, these functions operate on all values, returning a multivalued list of True/False results.

### Examples

A simple constraints validation expression to check that the field is five digits might be written as

@DATA MATCHES '5N'

This could also be written as

MATCHING ('5N')

A stock file might have both a cost price and selling price field. The validation for the selling price might be

INTEGER(100, 50000) AND @DATA > COST
to impose a range of prices from 1.00 to 500.00 (using the internal data format) and checking that the selling price is greater than the cost price.

A file might contain a multivalued list of dates that must be in January or July. This can be enforced with an expression that uses multivalued functions:

\[
\text{OCONVS(@DATA, "DM"); ORS(EQS(@, REUSE(1)), EQS(@, REUSE(7)))}
\]

### Compiling the Constraint Definitions

The data integrity constraint definitions in the dictionary must be compiled and stored in the data file before they take effect. This is done using the `COMPILE.CONSTRAINTS` command:

```
COMPILE.CONSTRAINTS file.name
```

If `file.name` is a multi-file, an individual subfile can be updated

```
COMPILE.CONSTRAINTS file.name, subfile.name
```

or all subfiles can be updated together by giving the subfile name as an asterisk:

```
COMPILE.CONSTRAINTS file.name,*
```

The default behaviour of the `COMPILE.CONSTRAINTS` command is to enable full validation, failing any attempt to write faulty data. The `LOGGING` and `DISABLE` keywords to this command select the other two styles of validation described above.

### Verifying Constraints

When constraints are first defined or updated for an existing file, it may be useful to verify that all records already in the file meet the constraint conditions. The `VERIFY.CONSTRAINTS` command does this:

```
VERIFY.CONSTRAINTS file.name
```

As with the `COMPILE.CONSTRAINTS` command, all subfiles of a multi-file can be processed in a single command by use of an asterisk as the subfile name.

The `VERIFY.CONSTRAINTS` command shows the record ids and failing field number of all records that fail validation. On completion of the command, the `@SYSTEM.RETURN_CODE` variable is zero for success, negative for a command error or a positive number of records that failed.

An application can use the QMBasic `VALIDATE()` function to verify whether a record meets the constraint rules without attempting to write it.

```
OK = VALIDATE(fvar, id, data)
```
3.17 Data Encryption

QM supports three data encryption methods:

Ad hoc encryption is provided by three QMBasic functions, ENCRYPT(), ENCRYPTX() and DECRYPT(). These take two arguments; the string to be processed and the encryption key (not the name of a key defined with CREATE.KEY as discussed below). Internally, these use the AES 128 bit encryption algorithm but the encrypted data is further processed to ensure that it can never contain the mark characters or ASCII C0 control characters and can, therefore, be stored as a field within a data record or in a text file. The ENCRYPT() function uses a fixed initialisation vector whereas ENCRYPTX() uses a random initialisation vector for improved security but returns a slightly longer result. The DECRYPT() function handles both forms automatically.

Because encryption is a byte level operation, on an ECS mode system, the data to be encrypted must first be transformed to a byte string using the BS conversion code if it may contain ECS characters. Conversely, the decrypted data must be transformed back to ECS form using the same conversion code. It is, therefore, necessary for an ECS capable application to know whether the ECS to byte string transformation has been done. An important point about this process is that the encrypted form of data is identical on ECS mode and non-ECS mode systems so long as the plain text form contains only data from the 8-bit character set.

It is the user’s responsibility to provide a mechanism to prevent disclosure of the key string. The key provided by the user can be of any length and may contain any characters. QM will automatically transform this into a form that is valid for use with the AES algorithm. For best security, avoid very short encryption keys which could be determined by repeated attempts to decrypt the data.

Because ad hoc encryption is performed outside of QM’s control, it is unlikely to be practical to build alternate key indices on encrypted data.

Record level encryption encrypts an entire record using a single, user defined key and the AES 128, 192 or 256 bit encryption algorithm. Because this encryption occurs deep inside the QM file system, it is possible to have alternate key indices on fields within a file that uses record level encryption. Note, however, that unless the index file itself is also encrypted, indexed fields are partially exposed outside of the encryption system.

Field level encryption allows users to encrypt specific fields within a file, possibly using different keys or algorithms for each encrypted field. It is not possible to build an alternate key index on an encrypted field mainly because not all users might have access to the field and hence the index cannot be maintained. An index built on a non-encrypted field may itself be encrypted. Field level encryption results in a slight increase in record size because of a transformation performed to ensure that the encryption process can never produce data that contains the mark characters.

With either level of encryption, an application may use many different encryption keys and each key can be made accessible to a selected set of users. A user cannot open a file that uses record level encryption unless they have access to the key. When using field level encryption, read operations will return null strings for fields to which the user has no access and write operations will preserve the previous content of these fields when updating an existing record.

The encryption system is managed by a user (or multiple users) known as the security administrator. This user is responsible for management of the key vault, a file that defines the names and actual values of all encryption keys used on the system. The key vault is itself encrypted using the master key which should be known only by the security administrator. This key value is entered when the key vault is created by first use of CREATE.KEY. If the key vault is moved to another system or a new licence is applied, the master key must be re-entered either via the licence entry screen or by use of the RESET.MASTER.KEY command. Note that there is no way to
recover the master key or the data keys if they are lost. **Always keep a written copy of these keys in a secure off-site location.**

Because QM uses key names rather than the actual keys in operations such as creating files or setting encryption rules, there is no need to restrict knowledge of the key names. The security administrator sets the actual encryption key value when the key is entered into the key vault and this can subsequently be applied to data files without knowing what encryption is being used. For example, the security administrator might define a key named CARDNO for use on fields containing credit card numbers. Developers can freely apply this without knowing the encryption key. To ensure that data is not lost in the event of a major system failure, the master key and details of each key name / key value pair should be stored securely off-site in some way. All security administrator commands require entry of the master key though this is remembered for the duration of the user's session.

A new encryption key is created using the `CREATE.KEY` command, available only to users with administrative rights in the QMSYS account. This command assigns the actual encryption key and the algorithm to be associated with the key name. The AES 128, 192 and 256 bit algorithms require a key length of 16, 24 or 32 bytes respectively, however, to enable administrators to use convenient keys of any length up to 64 characters, QM will transform the key entered by the user into a form that is valid for the AES algorithms. Specifying a key that is approximately the required internal length or a multiple thereof gives best security. Key names are case insensitive but the encryption key itself is case sensitive.

Defining a key name makes it accessible to the user that defined it. To make it accessible to other users, the security administrator uses the `GRANT.KEY` command, specifying the key name and the login names of the users or user groups (not Windows) to whom access is to be granted. Access to a key can subsequently be removed using the `REVOKE.KEY` command. If a key is no longer used, it can be removed from the key vault using `DELETE.KEY`.

Keys created using the `CREATE.SECURE.KEY` command can only be accessed after entry of the associated password. This can be done using the `ENABLE.KEY` command or the corresponding `ENABLE.KEY` QMBasic statement. If required, access can be denied again by use of the `DISABLE.KEY` command or the corresponding `DISABLE.KEY` QMBasic statement. A user with administrative rights can update the password by use of the `CHANGE.KEY.PASSWORD` command.

The security administrator can use the `LIST.KEYS` command to report the name, algorithm and access rights of each encryption key in the key vault. There is no way to report the key string associated with the key. This same command can be used with a filename to report the encryption key names used by the file.

The actual encryption process uses the key string defined in the key vault. If a file that uses encryption is moved to another system, the data in that file will be accessible if the encryption key names used by the file are added to the new system's key vault with the same encryption algorithm and key string.

The master key is used only to encrypt the key vault. The master key must be re-entered if the key vault is moved to another system or possibly as a result of relicensing the system. This is an automatic part of the QM licensing process and helps to ensure security if, for example, a backup tape of the system is stolen. If the master key is forgotten, there is no way to retrieve it from the key vault and hence the vault and all encrypted files would become inaccessible. It is therefore essential that a paper copy of the encryption keys is stored securely.

A file is created for record level encryption by use of the `ENCRYPT` keyword to the `CREATE.FILE` command, specifying the name of the encryption key to be used. This style of encryption is supported for hashed files and directory files but note that when applied to directory files, the data cannot be read from outside of QM and any record written from outside will be meaningless within a QM session as the system will attempt to decrypt an unencrypted record. Also,
encryption and replication cannot be used together on a directory file. The $ENCR X-type VOC record can be used to apply record level encryption automatically when creating a hashed file. See CREATE.FILE for more details.

For field level encryption, the developer creates the file, populates the dictionary and then uses the ENCRYPT.FILE command to specify the names of each field to be encrypted and the name of the key to be applied. Field level encryption is supported only for hashed files. If the file is not empty, the newly defined encryption is applied to the data. This command can also be used to apply record level encryption to an existing file or to many files in a single operation.

QM also provides optional protection that makes the data inaccessible even if the entire system is stolen - a situation that is more likely with portable computers than with corporate servers. This optional feature requires that the master key is entered every time that QM is started (not for each user entry into the system). Whilst potentially inconvenient, this mode of operation gives the highest level of security and is recommended for portable systems. The process of creating the master key includes a prompt asking if it must be entered to unlock the key vault after system restart. When this mode is enabled, any user with administrator rights can use the UNLOCK.KEY.VAULT command to enable access to encrypted files. Until access is unlocked, files using either field or record level encryption cannot be opened. The need for use of UNLOCK.KEY.VAULT can be enabled or disabled at any time by using RESET.MASTER.KEY.

To allow files to be moved to systems where the encryption key name clashes with a name already defined on that system, the SET.ENCRYPTION.KEY.NAME command can be used to update the key name index stored in the encrypted data file.

When used with QMClient or QMNet, access to encrypted data is based on the access rights of the user name under which the process connects to the server system. Because QMClient is a general library that can be used in various programming environments, it is not possible to apply the encryption at the client side. Although the data will be decrypted on the remote system (assuming that the user has access to it), the actual network traffic of QMClient and QMNet uses its own separate encryption.

ACCOUNT.SAVE, FILE.SAVE and T.DUMP operate within the security rules imposed by use of QM's encryption features. Files that use record level encryption will not be saved if the user performing the save does not have access to the encryption key. The data saved for files that use field level encryption will have all fields for which the user is denied access set to null strings. All saved data is recorded in decrypted form and hence storage of media created using these tools may reduce system security. The media format is an industry standard that does not provide a way to record details of data encryption. Restoring a save in which encrypted fields have been omitted is unlikely to yield a usable file. Backup of accounts that include encrypted data should be performed using operating system level tools.

Encryption can be removed from a file or specific field by use of the DECRYPT.FILE command. This requires that the user has access to the data to be decrypted and also prompts for the master key for improved security.

**Disaster Recovery with Encryption**

If it is necessary to move a QM system that uses encryption to a different system, the following steps are required:

Copy the data files to the new system, probably restoring them from a backup.

Then do one of the following:
• If the new system does not already use encryption, copy/restore the $VAULT subdirectory of the QMSYS account to the new system. Then use the RESET.MASTER.KEY to update the key vault.

• If the new system already uses encryption but not with the same key names use CREATE.KEY to create keys to match those of the original system.

• If the new system already uses encryption but has the same key names referencing different encryption details, use CREATE.KEY to create replacements for the original encryption keys and use SET.ENCRIPTION.KEY.NAME to amend the key name details stored in each affected file.

In all cases, it may be necessary to use GRANT.KEY to amend access rights to encryption keys if the user names or groups on the new system do not match those of the original system.

Recovery from Loss of the Key Vault

If the key vault is accidentally deleted or damaged in some way, it can be reconstructed as follows:
1. Shut down QM.
2. Delete the $VAULT directory from the QMSYS account directory.
3. Restart QM.
4. Use the CREATE.KEY command to re-enter the data keys using the same key names, encryption modes and key strings as before the failure. When entering the first key, you will be prompted to set the master key. This does not have to be the same as before the failure.
3.18 Transactions

A **transaction** is a group of related database updates to be treated as a unit that must either happen in its entirety or not at all. From a programmer's point of view, the updates are enclosed between two QMBasic statements, **BEGIN TRANSACTION** and **END TRANSACTION**. All writes and deletes appearing during the transaction are cached and only take place when the program executes a **COMMIT** statement. The program can abort the transaction by executing a **ROLLBACK** statement which causes all updates to be discarded.

An alternative transaction syntax is available using the **TRANSACTION START**, **TRANSACTION COMMIT** and **TRANSACTION ABORT** statements. The two styles may be mixed in a single application.

Transactions apply the ACID criteria:

**Atomicity.** Updates applied within a transaction are treated as a single action. They are all applied at commit or all discarded at roll-back.

**Consistency.** The database remains in a consistent state. Each committed transaction moves the database forwards from one valid state to another.

**Isolation.** At no point are partially committed transactions visible to other processes so long as they follow the standard locking rules.

**Durability.** The updates applied by a transaction are permanent unless undone by a further transaction. QM also support non-durable transactions where updates from a child transaction started inside another transaction are inherited by the parent transaction at commit and will be rolled back if the parent transaction does not also commit.

Even though updates made within a transaction are not written until the transaction is committed, attempting to read a record that has been updated within the same transaction or a parent transaction will return the updated data from the transaction cache. Creation of a select list inside a transaction will reflect the state of the file as though the transaction had been committed. Note that the alternate key indices are only updated when the transaction commits. The query processor will ignore alternate key indices in a transaction if the transaction has applied updates to the file referenced by the query command.

Transactions affect the operation of file and record locks. Outside a transaction, locks are released when a write or delete occurs. Transactional database updates are deferred until the transaction is committed and all locks acquired inside the transaction are held until the commit or roll-back. Because of this change to the locking mechanism, converting an application to use transactions is usually rather more complex than simply inserting the transaction control statements into existing programs. The retention of locks can give rise to deadlock situations.

There are some restrictions on what a program may do inside a transaction. In general, QM tries not to enforce prohibitive rules but leaves the application designer to consider the potential impact of the operations embedded inside the transaction. Note carefully that developers should try to avoid user interactions (e.g. **INPUT** statements) inside a transaction as these can result in locks being held for long periods if the user does not respond quickly.

Testing the value of the **STATUS()** function immediately after the **END TRANSACTION** statement will return 0 if the transaction committed successfully, **ERS$COMMIT.FAIL** if an error occurred during the commit action, or **ERS$ROLLBACK** if the transaction was rolled back.

Sometimes an application may open a file for which updates are not to be treated as part of any transaction within which they occur. The **OPEN** and **OPENPATH** statements both have an option to open the file in a non-transactional manner. Alternatively, the presence of the N option in field 6 of the **F-type** VOC entry will cause a non-transactional open.
Example

BEGIN TRANSACTION
   READU CUST1.REC FROM CUST.F, CUST1.ID ELSE ROLLBACK
   CUST1.REC<C.BALANCE> -= TRANSFER.VALUE
   WRITE CUST1.REC TO CUST.F, CUST1.ID

   READU CUST2.REC FROM CUST.F, CUST2.ID ELSE ROLLBACK
   CUST2.REC<C.BALANCE> += TRANSFER.VALUE
   WRITE CUST2.REC TO CUST.F, CUST2.ID
   COMMIT
END TRANSACTION

The above program fragment transfers money between two customer accounts. The updates are only committed if the entire transaction is successful.
3.19 Replication

Data replication provides a mechanism whereby updates to selected files are automatically exported to one or more other files, usually on a different system in order to provide resilience to hardware failure. In normal usage, the intention is to maintain an identical copy of the original files, either so that the second system can take over or for use as a reporting server. It is, however, possible to use the replication facility to copy data from disparate sources into a single file, perhaps merging data from satellite offices into a single file at head office.

Basic Principles

The system that exports data is known as the publisher. The system receiving the data is known as the subscriber. Normally this would be a simple relationship with just one publisher/subscriber pair, however, QM imposes few restrictions on how replication is used.

Each published file may be exported to multiple subscribers. A subscriber system can publish these files onwards to further subscribers, cascading data through a series of servers. The only rule is the sequence of exports must not contain a loop where a file is replicated back to itself. There is an exception to this rule as described below under "symmetrical replication".

A subscriber may receive data from any number of publishers.

QM does not prevent other updates to the replicated files on the subscriber system. Although use as a standby system probably requires that updates are not allowed, some other uses of the replication mechanism can require the ability to apply local updates. It is the application designer's responsibility to ensure that appropriate rules are imposed.

Replication can be used with hashed files and directory files but, because QM is unaware of changes made to directory files from outside of QM, only updates made by QM applications will be replicated. The subscriber systems may import the data into any type of file. Also, changes made to directory files using sequential file processing (WRITESEQ, WRITEBLK, etc), OSWRITE or OSDELETE are not replicated. Some implications of this are that query processor CSV or delimited reports directed to a file and QMBasic compiler listing files are not replicated. As a general rule, replication should not be used to publish files from the QMSYS account.

Note that replication of a file that uses case insensitive record ids into a file that uses case sensitive record ids is unlikely to work correctly as the update applied at the subscriber will have the casing of the record id as used in the update on the publisher system. Thus updates to a record named "ABC" and a record named "abc" on the publisher in a file with case insensitive ids will reference the same record but these updates replicated to a subscriber file with case insensitive ids will access different records. This is no different from the similar problem that can occur when copying records between case insensitive files and case sensitive files on the one system.

The replication export is not instantaneous. The subscribers poll for updates at an interval that is configurable in the range 1 to 300 seconds, defaulting to five seconds. At this default rate, the subscriber system will be on average about three seconds behind the publisher though peak loads may increase this time due to network delays. For a true hot standby where the application does not continue until the update has been committed on the standby system, use QM triggers. The network delays associated with this approach will have a substantial effect on application performance.

A summary report of the replication system status can be produced by use of the RPL.STATUS command. This is normally only present in the VOC of the QMSYS account but can be copied to other accounts if needed.
Setting up the Publisher

The publisher side of replication can only be enabled if it is included in your QM licence.

There are four configuration parameters related to replication on the publisher system.

- The mandatory **REPLDIR** parameter specifies the directory where QM will write the log files that contain data waiting for transfer to the subscribers. This directory will be created automatically when QM is started if it does not exist. There will be a separate subdirectory under this location for each subscriber.

- The **REPLMAX** parameter specifies the maximum number of simultaneous replication targets that are allowed. Publishing a single file to a large number of targets may have a significant performance impact.

- The **REPLPORT** parameter specifies the port on which QM will listen for incoming connections from subscriber systems. This defaults to port 4244 if omitted which is the default used by the subscriber.

- The **REPLSIZE** parameter specifies the approximate maximum size for a replication log file in Mb before a new log file is started. The default value is 10 which should be appropriate for most systems.

A file is published by use of the **PUBLISH** command by a user with administrator rights:

```
PUBLISH {DICT} filename {DICT} server:account:filename
```

where the first filename references the file to be published and the **server:account:filename** references the file to which updates are to be exported. Multiple targets may be specified in a single **PUBLISH** command. The subscriber system identified by **server** must have been defined with **SET.SERVER** as the command will check for the existence of the target file, however, it is not necessary to enable extended filename syntaxes (see the **FILERULE** configuration parameter).

The **PUBLISH ACCOUNT** command can be used to simplify publishing a large number of files to the same target server/account.

```
PUBLISH ACCOUNT server:account
```

This command will show a list of files in the current account that are not already published to the specified target. The user can then select files from this list that are to be published.

From the point when the **PUBLISH** command is executed, all updates to the published files will be logged in the directory specified by the **REPLDIR** parameter.

See [Replication Example](#) for a step by step example of setting up replication.

Setting up the Subscriber

The subscriber side of replication is not separately licensed.

When using replication to create a standby system, it is essential that the target files correctly match the corresponding files on the publisher system before enabling subscription. The easiest way to do this is to copy the file at the operating system level while not in use and before setting it up for publishing. Alternatively, copy the file using the QM **COPY** command after setting it up for publishing. In the latter case, updates to the publisher system that occur during or after the copy will be applied when subscription is enabled.
The **REPLSRVR** configuration parameter must be set to specify the name of the subscriber server. This must match the server name used in the **PUBLISH** commands but does not need to be the same as the network name of the subscriber system.

The **NUMFILES** configuration parameter must have a value that allows all subscribed files on the system to be open simultaneously.

The **SUBSCRIBE** command is used by a user with administrator rights to apply updates from the publisher to the subscriber. This command starts a phantom process that manages the actual data transfer and is probably best initiated from a **STARTUP** configuration parameter.

```
SUBSCRIBE server{:<port>} username password {options}
```

The **server** element of this command may be the network name or ip address of the publisher. The **port** number defaults to 4244 if omitted. The user name and password must be valid for connection to the publisher system. The password should preferably be encrypted using the **AUTHKEY** command for security and specified with a prefix of ENCR: in the command. A server process running as this user will be established on the publisher.

See the **SUBSCRIBE** command for details of the **options**. See Replication Example for a step by step example of setting up replication.

**Important Note:** Because information about the replication targets is stored in the file, copying the directory that represents a QM file that uses replication will result in the new file also replicating to the same targets.

**Transactions**

Updates that were part of a transaction on the publisher will be applied to the subscriber as a group but not strictly as a transaction. There should be no impact on the replication process or data integrity.

**Encryption**

Data traffic between the publisher and the subscriber is encrypted though there is an option to suppress this, perhaps when the data path is entirely within a secure network.

If files that use record or field level data encryption are published, the data is exported in encrypted form and imported at the subscriber without change. The implication of this is that the target files on the subscriber system must use the same encryption level and key strings as the source files. This is a logical step if replication is not to weaken system security. Also, the user name of the subscriber phantom must have full access to all relevant encryption keys. Encryption and replication cannot be used together on a directory file.

**Alternate Key Indices**

Because the **SUBSCRIBE** command writes data in the same way as any other QM process, indices on the subscriber system will be maintained automatically. The indices defined on the subscriber can be different from those on the publisher.

**Triggers**
Trigger functions are not applied on updates at the subscriber. Where triggers are used for data validation on the publisher, the data does not require revalidation on the subscriber. Where triggers are used to trigger other events on the publisher, any updates from these events to published files will also be replicated.

Permissions

All QM users who may update exported files must have write access to the directory specified by the REPLDIR parameter on the publisher system. The replication system will set the access rights automatically as each new log file is created.

The user name under which the subscription server process runs on the publisher must have read access to all files that are to be published. Because encrypted data is transferred in encrypted form, this user does not need to have access to encryption keys.

The user name under which the SUBSCRIBE command runs on the subscriber system must have full access to all subscribed files. When replicating to directory files on the subscriber on a Linux system, new data records will be created with the default umask setting for the system (not one set by the Linux profile script as this does not run for phantom processes). It may be useful to add a UMASK command to the LOGIN paragraph of the QMSYS account.

Cancelling Replication

Replication of a single file can be cancelled by using the CANCEL keyword of the PUBLISH command. Replication of an entire account can be cancelled using the CANCEL keyword with PUBLISH.ACCOUNT.

Copying Replicated Files

The details of the replication targets for a published file are stored within the file. If a published file is copied using operating system tools or restored from a backup to a new location, it will usually be necessary to cancel replication of the new file.

Failure Situations

Where replication is used to provide resilience, it is important to understand the failure situations for which it provides protection and how to recover from them.

1. Failure of the publisher

If the publisher system fails, the subscriber system can take over. Updates that are currently logged on the publisher but have not been exported to the subscriber will not be present.

The recovery procedure depends on how the subscriber system has been used during the failure period.

If no changes have been made to replicated files on the subscriber system, no action is necessary. The subscriber system will periodically attempt to re-establish connection with the publisher and, when this is successful, the pending updates will be applied.

If users have logged into the subscriber as a standby system and made updates to replicated files, recovery requires a little work.
a) Terminate the subscriber process before restarting the publisher so that it does not attempt to reconnect.

b) Consider whether any updates in the replication log files on the publisher are to be allowed to be applied, possibly overwriting change made on the subscriber or whether they should be discarded. If updates are to be discarded, login on the publisher when it is restarted, shutdown any QM processes that may be accessing published files (e.g. phantoms created using the STARTUP configuration parameter), and delete the content of the relevant log file directory. This is a subdirectory of the same name as the subscriber system under the directory identified by the REPLDIR configuration parameter. Delete only the files, not the directory.

c) With no QM users logged in on the subscriber system, copy the relevant files back to the publisher using the QM COPY command, not an operating system level copy.

d) Allow users back into QM on the publisher.

e) Restart the subscriber process.

2. Failure of the subscriber

If the subscriber system fails, the publisher will continue to run with updates queued in the replication log files. When the subscriber comes back online, these updates will be applied automatically.

Updates are queued in log files that are limited to approximately 10Mb, switching to a new log file when this limit is reached. Log files are discarded when they have been completely processed. If the subscriber fails (or the subscription process is not running), the publisher system will generate a message in the error log file if the backlog of updates has three or more files (approximately 30Mb). This message will be repeated at hourly intervals until the situation is resolved.

3. Failure of the network

Network failure is essentially no different from subscriber failure except that the data is still available for reporting purposes. Updates will restart automatically shortly after the network connection is restored.

Symmetrical Replication

Where replication is used in a simple one publisher / one subscriber configuration, symmetrical replication can significantly simplify the failure and recovery process.

The two systems are configured to publish files to each other. Updates made on either system will be replicated on the other but the replication process will not copy an update from the subscriber back to the publisher from which it came, thus avoiding an endless cycle of updates between the two systems.

If the primary publisher system fails, users login to the subscriber standby system and continue work. Updates that they make will be logged for replication back to the primary system and will be applied when it comes back online. The same considerations apply as described above regarding updates that were logged on the primary system but did not get sent to the standby system immediately prior to the failure.

On return of the primary system, users can be moved back at a convenient moment. No data files need to be copied manually.
Whilst symmetrical replication provides a simple recovery strategy, it is very important to ensure that updates are not made to the same records on both systems at the same time as there is no locking between the two systems to control the sequence in which updates would get applied.

Disabling Publishing

The DISABLE.PUBLISHING command can be used to suspend recording of updates to published files in the replication log area. It is intended for use when synchronising the publisher and subscriber systems. Updates applied to published files while publishing is disabled will not be replicated. Account replication (see below) is also disabled by this command. Publishing can be resumed using the ENABLE.PUBLISHING command and resumes automatically if QM is restarted.

Moving the Replication Directory

If it is necessary to move the directory that stores replication log files to a new location, perhaps for load balancing across disk drives, this can be accomplished simply by disabling publishing, modifying the REPLDIR configuration parameter, moving the directory structure, and re-enabling publishing.

Identifying Replicated Files

The files for which replication is enabled can be shown using the LIST.REPLICATED command. The LISTF, LISTFL and LISTFR commands also report files with replication but do not show dictionaries.

Account Replication

When using replication to maintain a standby system, it may be useful for the action of some commands executed on the publisher to be replicated on one or more subscriber systems. This capability is provided by an extension to the replication system that is largely independent of the mechanisms discussed above.

To activate account replication, create an X-type VOC item named $REPLICATE in the relevant account(s) on the publisher system. Field 2 of this record should hold the name of the account in which command actions are to be replicated in the form

servername:account

where servername is an item previously set up with the CREATE.SERVER command and account is an account on this server. Multiple targets may be specified separated by value marks. The QMCLIENT configuration parameter must be set to zero (its default value) on the remote system. Beware that this may weaken system security.

Fields 3 onwards contain a list of the commands that are to be replicated on the remote system(s) from the list below.

- ADD.DF
- BUILD.INDEX
- CATALOGUE
- CREATE.FILE
- CREATE.INDEX
- DELETE.CATALOGUE
- DELETE.FILE
- DELETE.INDEX

Add an element to a distributed file
Build an alternate key index
Add program to the system catalogue. The American spelling may be used.
Create a file
Create an alternate key index
Delete a program from the system catalogue. The American spelling may be used.
Delete a file
Delete an alternate key index
GRANT.KEY                  Grant access to an encryption key
MAKE.INDEX                Create and build an alternate key index
REMOVE.DF                 Remove an element of a distributed file
REVOKE.KEY                Remove access to an encryption key
SET.FILE                  Set a Q-pointer to a remote file

With the exception of the two alternative spellings referenced above and the interchangeability of hyphens and periods in the names, the account replication system replicates only the commands as listed in the $REPLICATE record. Use of a command that is a user added synonym for one of the above list will not be replicated unless the synonym is also added to the $REPLICATE record. In this case, the synonym must also be present on the remote system.

Because synonyms not in the above list are not replicated, it is possible to create synonyms for replicated commands that will themselves not be replicated. For example, copying the CREATE.FILE VOC record to CREATE.LOCAL.FILE would allow a choice between CREATE.FILE for file creation that is to be replicated and CREATE.LOCAL.FILE to be used for file creation that is not to be replicated.

Any command name in the $REPLICATE record may be followed by a list of space separated mode options. Currently, the only option supported is QUERY which causes the user to be prompted for confirmation that the command is to be replicated. Beware that this option would cause the command to fail in a phantom process and may cause other problems in a QMClient session.

Replication of the CATALOGUE command also updates the object code in the appropriate file on the target system.

In addition, presence of
    AUTO.PUBLISH
in the command list will cause files created using a replicated CREATE.FILE command to be published automatically to a file of the same name in the remote account(s).

Changes to the $REPLICATE record take effect only on next entry to the account, either as a result of a new login or by use of LOGTO.

See also:
Replication Example
Replication Example

This example gives a step by step guide to setting up a simple replication system. It is based on the simple sales processing database created by the \texttt{SETUP.DEMO} command, creating a reporting server that mirrors the data from the \texttt{STORE} account on the live production system. Pathnames below are as appropriate for a Linux system but can be substituted with Windows pathnames.

The Reporting Server - Part 1

1. The reporting server should be set up with a copy of the \texttt{STORE} account from the live server, typically by restoring a backup.

2. Use \texttt{EDIT.CONFIG} or a text editor to set the \texttt{NETFILES} configuration parameter to allow incoming QMNet connections and set \texttt{REPLSRVR} to the name that will be used by the live server to reference the reporting server.

\begin{verbatim}
NETFILES=2
REPLSRVR=REPORTING
\end{verbatim}

3. Restart QM.

The Live Server

1. Log to the QMSYS account

\begin{verbatim}
LOGTO QMSYS
\end{verbatim}

2. Add a QMNet server definition that allows access to the reporting server using the server name set in the \texttt{REPLSRVR} parameter above.

\begin{verbatim}
SET.SERVER REPORTING
\end{verbatim}

Enter appropriate network and user authentication details.

3. Use \texttt{EDIT.CONFIG} or a text editor to add the \texttt{REPLDIR} configuration parameter that identifies where replication log files will be stored.

\begin{verbatim}
REPLDIR=/usr/replication
\end{verbatim}

4. Restart QM.

5. Log to the account holding the files to be published.

\begin{verbatim}
LOGTO STORE
\end{verbatim}
6. Publish the data files that are to be replicated
   
   PUBLISH STOCK REPORTING:STORE:STOCK
   PUBLISH SALES REPORTING:STORE:SALES
   PUBLISH CUSTOMERS REPORTING:STORE:CUSTOMERS

   If many files are to be published, use of the PUBLISH.ACCOUNT command will be simpler.

7. If the files to be published may have changed on the live server since the copy was created on the reporting server, copy the current versions of these files across.

8. Ensure that the port to be used by replication (set by REPLPORT parameter on the reporting server, default 4244) is open in any firewall.

The Reporting Server - Part 2

1. Determine the user name to be used for the subscriber phantom process. This user must have full access to all files that are being updated by the replication process.

2. Ensure that any subroutines called from I-type dictionary items used by alternate key indices on replicated files are either globally catalogued or catalogued within the QMSYS account.

3. Use the AUTHKEY command in the QMSYS account to form the encrypted password for this user.

   AUTHKEY password

4. Use EDIT.CONFIG or a text editor to add a STARTUP configuration parameter that will start the subscriber phantom automatically whenever QM is started.

   STARTUP=SUBSCRIBE 123.123.123.123 username ENCR:password

   where 123.123.123.123 is the network address or name of the live server, username is the user name for the subscriber phantom and password is the encrypted password.

5. Either restart QM or start the subscriber phantom manually using the command from the STARTUP configuration parameter above.

   All updates to published files on the live system should now be applied automatically to the same files on the reporting server.
Part 4

QM Commands
4 QM Commands

* $ECHO
  Paragraph tracing
  Synonym for SH

- ABORT
  Abort processing and return to command prompt

ACCOUNT.RESTORE
  Restore a Pick style ACCOUNT-SAVE tape

ACCOUNT.SAVE
  Save an account to tape in Pick compatible format

ADD_DF
  Add an element to a distributed file

ADMIN
  Administrative command menu

ADMIN_SERVER
  Manage security rules for QMNet or VFS server connections

ADMIN.USER
  User name administration for network connections

ALIAS
  Create a temporary alias for a command

ANALYSE.FILE
  Analyse structure and usage of a file

ANALYZE.FILE
  Synonym for ANALYSE.FILE

AUTHENTICATE
  Reduce privileges of a startup process

AUTHKEY
  Encrypt password for AUTHENTICATE

AUTOLOGOUT
  Set inactivity timer

BASIC
  Compile QMBasic programs

BELI
  Enable or disable audible alarm

BLOCK.PRINT
  Print text using large characters

BLOCK.TERM
  Display text using large characters

BREAK
  Enable, disable or query break key

BUILD_INDEX
  Build an alternate key index

CATALOG
  Synonym for CATALOGUE

CATALOGUE
  Add program to system catalogue

CD
  Synonym for COMPILE.DICT

CHANGE.KEY.PASSWORD
  Change the password for a secure protected encryption key

CLEAN.ACCOUNT
  Remove records from $HOLD, $COMO and $SAVEDLISTS

CLEAR.ABORT
  Clear the abort status in an ON.ABORT paragraph

CLEAR.DATA
  Clear the data queue

CLEAR.FILE
  Remove all records from a file

CLEAR.INPUT
  Clear keyboard type-ahead

CLEAR.LOCKS
  Release task locks

CLEAR.PROMPTS
  Clear inline prompt responses

CLEAR.SELECT
  Clear one or all select lists

CLEAR.STACK
  Clear the command stack

CLEARDATA
  Synonym for CLEAR.DATA

CLEARINPUT
  Synonym for CLEAR.INPUT

CLEARPROMPTS
  Synonym for CLEAR.PROMPTS

CLEARESELECT
  Synonym for CLEAR.SELECT

CLOSE.ECI
  Close all External Call Interface child processes

CLR
  Clear display

CNAME
  Rename a file or record within a file

COMO
  Activate or deactivate command output files

COMPARE.FILES
  Compare content of two files

COMPILE.CONSTRAINTS
  Compile data integrity constraints rules

COMPILE.DICT
  Compile I-types in a dictionary

CONFIG
  Display licence and configuration parameters

CONFIGURE.FILE
  Change file configuration parameters

COPY
  Copy records

COPYI
  Synonym for COPY using the Information style syntax

COPY.LIST
  Copy a saved select list
COPY.LISTP
Copy a saved select list using Pick style syntax

COPYP
Copy records using Pick style syntax

COUNT
Count records

CREATE.ACCOUNT
Make a new QM account

CREATE.FILE
Create a file

CREATE.INDEX
Create an alternate key index

CREATE.KEY
Creates a data encryption key

CREATE.USER
Create user name for network connection

CS
Synonym for CLR

CT
Display records from a file

CWD
Change working directory

DATA
Add text to the data queue for associated verb or program

DATE
Display the date and time

DATE FORMAT
Selects default date format

DEBUG
Debug QMBasic program

DECATALOG
Delete a local catalogue entry and the associated object code

DECRYPT.FILE
Decrypt a file

DELETE
Delete records from a file

DELETE.ACCOUNT
Delete a QM account

DELETE.CATALOG
Synonym for DELETE.CATALOGUE

DELETE.CATALOGUE
Delete a program from the system catalogue

DELETE.COMMON
Delete a named common block

DELETE.DEMO
Delete demonstration files

DELETE.FILE
Delete a file

DELETE.INDEX
Delete an alternate key index

DELETE.KEY
Delete a data encryption key

DELETE.LIST
Delete a saved select list

DELETE.PRIVATE.SERVER
Delete a QMNet private server definition

DELETE.SERVER
Delete a QMNet or VFS server definition

DELETE.USER
Delete user name for network connection

DISABLE.KEY
Disable access to a password protected secure encryption key

DISABLE.INDEX
Disable update and use of one or more alternate key indices

DISPLAY
Display text

DO.REMOTE
Execute a command on a remote server

DUMP
Display records from a file in hexadecimal and character format

ECHO
Disable or enable keyboard echo

ED
Line editor

EDIT
Synonym for ED

EDIT.CONFIG
Configuration parameter editor

EDIT.LIST
Edit a saved select list

EDIT.MAP
Edit an ECS map

ENABLE.KEY
Enable access to a password protected secure encryption key

 ENCRYPT.FILE
 Applies encryption to a file

 EXECUTE.LIST
 Execute one or more commands for each entry in a select list

 FILE.SAVE
 Save all accounts to tape in Pick compatible format

 FILE STAT
 Report a summary of all files in one or more accounts

 FIND ACCOUNT
 Find an account on a FILE.SAVE tape

 FIND FILE
 Report summary information about a file

 FIND PROGRAM
 Locate program in catalogue

 FORMAT
 Apply conventional formatting to a QMBasic program

 FORM.LIST
 Create a select list from a file record
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ESTAT</td>
<td>Collect and report file statistics</td>
</tr>
<tr>
<td>GENERATE</td>
<td>Generate a QMBasic include record from a dictionary</td>
</tr>
<tr>
<td>GET.LIST</td>
<td>Retrieve a previously saved select list</td>
</tr>
<tr>
<td>GET.STACK</td>
<td>Restore a saved command stack</td>
</tr>
<tr>
<td>GO</td>
<td>Jump to a label within a paragraph</td>
</tr>
<tr>
<td>GRANT.KEY</td>
<td>Grants access to a data encryption key</td>
</tr>
<tr>
<td>HELP</td>
<td>Display help</td>
</tr>
<tr>
<td>HSM</td>
<td>Hot Spot Monitor performance monitoring tool</td>
</tr>
<tr>
<td>HUSH</td>
<td>Disable or enable display output</td>
</tr>
<tr>
<td>IF</td>
<td>Conditional execution in paragraphs</td>
</tr>
<tr>
<td>INHIBIT.LOGIN</td>
<td>Enables or disables login of new QM sessions</td>
</tr>
<tr>
<td>LIST</td>
<td>List records from a file</td>
</tr>
<tr>
<td>LIST.COMMON</td>
<td>List named common blocks</td>
</tr>
<tr>
<td>LIST.DF</td>
<td>List elements of a distributed file</td>
</tr>
<tr>
<td>LIST.DIFF</td>
<td>Form difference of two saved select lists</td>
</tr>
<tr>
<td>LIST.FILES</td>
<td>List details of open files</td>
</tr>
<tr>
<td>LIST.INDEX</td>
<td>List details of an alternate key index</td>
</tr>
<tr>
<td>LIST.INTER</td>
<td>Form intersection of two saved select lists</td>
</tr>
<tr>
<td>LIST.ITEM</td>
<td>List records from a file in internal format</td>
</tr>
<tr>
<td>LIST.KEYS</td>
<td>Lists details of encryption keys</td>
</tr>
<tr>
<td>LIST.LABEL</td>
<td>List records from a file in address label format</td>
</tr>
<tr>
<td>LIST.LOCKS</td>
<td>List task lock status</td>
</tr>
<tr>
<td>LIST.PHANTOMS</td>
<td>List currently active phantom processes</td>
</tr>
<tr>
<td>LIST.READU</td>
<td>List file, read and update locks</td>
</tr>
<tr>
<td>LIST.REPLICATED</td>
<td>Lists files that use replication with the target file details</td>
</tr>
<tr>
<td>LIST.SERVERS</td>
<td>List defined QMNet and VFS server names</td>
</tr>
<tr>
<td>LIST.TERM.TYPES</td>
<td>Display list of terminal types</td>
</tr>
<tr>
<td>LIST.TRIGGERS</td>
<td>List trigger functions used in the account</td>
</tr>
<tr>
<td>LIST.UNION</td>
<td>Form union of two saved select lists</td>
</tr>
<tr>
<td>LIST.USERS</td>
<td>List user names for network connection</td>
</tr>
<tr>
<td>LIST.VARS</td>
<td>List user @-variables</td>
</tr>
<tr>
<td>LIST.DICT</td>
<td>List a dictionary in Pick style format</td>
</tr>
<tr>
<td>LISTF</td>
<td>List all files defined in the VOC</td>
</tr>
<tr>
<td>LISTFL</td>
<td>List all local files defined in the VOC</td>
</tr>
<tr>
<td>LISTFR</td>
<td>List all remote files defined in the VOC</td>
</tr>
<tr>
<td>LISTK</td>
<td>List all keywords defined in the VOC</td>
</tr>
<tr>
<td>LISTM</td>
<td>List all menus defined in the VOC</td>
</tr>
<tr>
<td>LISTPA</td>
<td>List all paragraphs defined in the VOC</td>
</tr>
<tr>
<td>LISTPH</td>
<td>List all phrases defined in the VOC</td>
</tr>
<tr>
<td>LISTPQ</td>
<td>List all PROCs defined in the VOC</td>
</tr>
<tr>
<td>LIST.PRINTERS</td>
<td>List all printer names known to the operating system</td>
</tr>
<tr>
<td>LISTQ</td>
<td>List all indirect file references in the VOC</td>
</tr>
<tr>
<td>LISTR</td>
<td>List all remote items defined in the VOC</td>
</tr>
<tr>
<td>LISTS</td>
<td>List all sentences defined in the VOC</td>
</tr>
<tr>
<td>LISTU</td>
<td>List users currently in QM, sorted by user number</td>
</tr>
<tr>
<td>LISTV</td>
<td>List all verbs defined in the VOC</td>
</tr>
<tr>
<td>LOAD.LANGUAGE</td>
<td>Install a new message text language</td>
</tr>
<tr>
<td>LOCK</td>
<td>Set a task lock</td>
</tr>
<tr>
<td>LOCK.TRACER</td>
<td>Enable/disable/report lock tracing</td>
</tr>
<tr>
<td>LOGIN.PORT</td>
<td>Login a serial port from within another QM session</td>
</tr>
<tr>
<td>LOGMSG</td>
<td>Write a message to the error log</td>
</tr>
<tr>
<td>LOGOUT</td>
<td>Terminate a phantom process</td>
</tr>
<tr>
<td>LOGTO</td>
<td>Change to an alternative account</td>
</tr>
<tr>
<td>LOOP / REPEAT</td>
<td>Defines loop within paragraph</td>
</tr>
<tr>
<td>MAKE.INDEX</td>
<td>Create and build an alternate key index</td>
</tr>
<tr>
<td>MAP</td>
<td>Display a list of the catalogue contents</td>
</tr>
<tr>
<td>MED</td>
<td>Edit a menu definition</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>MERGE.LIST</td>
<td>Create a select list by merging two other lists</td>
</tr>
<tr>
<td>MESSAGE</td>
<td>Send a message to selected other users</td>
</tr>
<tr>
<td>MODIFY</td>
<td>Modify records in a file</td>
</tr>
<tr>
<td>NLS</td>
<td>Set or report national language support values</td>
</tr>
<tr>
<td>NSELECT</td>
<td>Remove items from a select list</td>
</tr>
<tr>
<td>OFF</td>
<td>Synonym for QUIT</td>
</tr>
<tr>
<td>OPTION</td>
<td>Set, clear or display options</td>
</tr>
<tr>
<td>OSCOPY</td>
<td>Copy and operating system file or directory</td>
</tr>
<tr>
<td>PACKAGE.LICENCE</td>
<td>Maintain package licence details</td>
</tr>
<tr>
<td>PASSWORD</td>
<td>Change user password for network connection</td>
</tr>
<tr>
<td>PAUSE</td>
<td>Display &quot;Press return to continue&quot; prompt</td>
</tr>
<tr>
<td>PDEBUG</td>
<td>Runs the phantom debugger</td>
</tr>
<tr>
<td>PDUMP</td>
<td>Generate a process dump file</td>
</tr>
<tr>
<td>PHANTOM</td>
<td>Initiate a background process</td>
</tr>
<tr>
<td>PRINTER</td>
<td>Administer print units</td>
</tr>
<tr>
<td>PROFILER</td>
<td>Enable/disable/report application event profiling</td>
</tr>
<tr>
<td>PSTAT</td>
<td>Report process status</td>
</tr>
<tr>
<td>PTERM</td>
<td>Set or display terminal characteristics</td>
</tr>
<tr>
<td>PUBLISH</td>
<td>Set up, cancel or query publishing of a file for replication</td>
</tr>
<tr>
<td>PUBLISH.ACCOUNT</td>
<td>Select files to set up or cancel for replication</td>
</tr>
<tr>
<td>QSELECT</td>
<td>Construct a select list from the content of selected records</td>
</tr>
<tr>
<td>QUIT</td>
<td>Terminate session or revert to lower command level</td>
</tr>
<tr>
<td>REBUILD.ALL.INDICES</td>
<td>Rebuild all alternate key indices in an account</td>
</tr>
<tr>
<td>REDO</td>
<td>Repeat a command at specified intervals</td>
</tr>
<tr>
<td>RELEASE</td>
<td>Release record or file locks</td>
</tr>
<tr>
<td>REMOVE.DF</td>
<td>Remove an element of a distributed file</td>
</tr>
<tr>
<td>RENAME</td>
<td>Synonym for CNAME</td>
</tr>
<tr>
<td>REPORT.SRC</td>
<td>Display @SYSTEM.RETURN.CODE at command prompt</td>
</tr>
<tr>
<td>REPORT.STYLE</td>
<td>Sets the default style for query processor reports</td>
</tr>
<tr>
<td>RESET.MASTER.KEY</td>
<td>Resets the master encryption key</td>
</tr>
<tr>
<td>RESTORE.ACCOUNTS</td>
<td>Restore all accounts from a FILE.SAVE tape</td>
</tr>
<tr>
<td>REVVOKE.KEY</td>
<td>Removes access to a data encryption key</td>
</tr>
<tr>
<td>RPL.STATUS</td>
<td>Show summary report of replication system status</td>
</tr>
<tr>
<td>RUN</td>
<td>Run a compiled QMBasic program</td>
</tr>
<tr>
<td>SAVE.LIST</td>
<td>Save a select list</td>
</tr>
<tr>
<td>SAVE.STACK</td>
<td>Save the command stack</td>
</tr>
<tr>
<td>SCAN</td>
<td>Search for records containing specified text</td>
</tr>
<tr>
<td>SCRB</td>
<td>Create or edit a screen definition</td>
</tr>
<tr>
<td>SEARCH</td>
<td>Search file for records containing string(s)</td>
</tr>
<tr>
<td>SECURITY</td>
<td>Enable, disable or report system security</td>
</tr>
<tr>
<td>SED</td>
<td>Screen editor</td>
</tr>
<tr>
<td>SELECT.RESTORE</td>
<td>Selective restore from an ACCOUNT.SAVE or FILE.SAVE tape</td>
</tr>
<tr>
<td>SELECT</td>
<td>Select records meeting criteria</td>
</tr>
<tr>
<td>SELECT.DIR</td>
<td>Build a select list of file or subdirectory names under a given pathname</td>
</tr>
<tr>
<td>SELECTINDEX</td>
<td>Builds a select list of all indexed values in a specified alternate key index</td>
</tr>
<tr>
<td>SET</td>
<td>Set a user @variable</td>
</tr>
<tr>
<td>SET.DEVICE</td>
<td>Attach a tape device</td>
</tr>
<tr>
<td>SET.ENCRYPTION.KEY.NAME</td>
<td>Updates encryption key names for a file</td>
</tr>
<tr>
<td>SET.EXIT.STATUS</td>
<td>Set final exit status value</td>
</tr>
<tr>
<td>SET.FILE</td>
<td>Set a Q-pointer to a remote file</td>
</tr>
<tr>
<td>SET.LANGUAGE</td>
<td>Select a language for message display</td>
</tr>
<tr>
<td>SET.MODE</td>
<td>Set access permissions for records in a directory file.</td>
</tr>
<tr>
<td>SET.PRIVATE.SERVER</td>
<td>Define a QMNet private server</td>
</tr>
</tbody>
</table>
SET QUEUE
Define a Pick style form queue

SET SERVER
Define a QMNet server

SET TRIGGER
Set, remove or display trigger function for a file

SET VFS SERVER
Define a VFS server

SETPORT
Set communications parameters of a serial port

SETPTR
Set print unit characteristics

SETUP DEMO
Create demonstration files

SH
Execute operating system shell command

SHOW
Build select list interactively

SHOW LIST
Display an active numbered select list, allowing editing

SLEEP
Suspend process until specified time

SORT
List records sorted by record key

SORT ITEM
List records sorted by record key in internal format

SORT LIST
List records in address label format, sorted by record key

SORTU
Sort a saved select list

SP ASSIGN
List users currently in QM, sorted by user name

SP CLOSE
Set printer options using a Pick style form queue

SP OPEN
Close a print unit previously in "keep open" mode

SP VIEW
Open a print unit in "keep open" mode

SPOOL
View and print records from $HOLD or other files

SSELECT
Send record(s) to the printer

STATUS
Select records meeting criteria, sorting list by record key

STOP
Terminate an active paragraph

SUBSCRIBE
Display list of active phantom processes

SUM
Terminate an active paragraph

T ATT
Establish replication subscriber data transfer

T DET
Report total of named fields

T DUMP
Synonym for SET DEVICE

T EOD
Detach a previously assigned tape device

T FWD
Save a file

T LOAD
Position a tape device to the end of the recorded data

T RDLBL
Move a tape device forward by one file

T READ
Restore a file

T REW
Read the label from a tape device

T STAT
Read data from a tape device

T STAT
Rewind a tape device

T WEOF
Report the status of a tape device

T WEOF
Write end of file marker to a tape device

TANDEM
Monitor the screen of another QM session

TERM
Set or display terminal window size

TIME
Display date and time

UMASK
Set access rights to be applied at file creation

UNLOCK
Unlock a record or file

UNLOCK KEY VAULT
Enable access to encryption key vault

UPDATE ACCOUNT
Update VOC items from NEWVOC

UPDATE LICENCE
Apply new licence information

UPDATE RECORD
Database maintenance tool

VERIFY CONSTRAINTS
Verify data integrity constraints compliance

WEB SVC
Simple web services listener

WHO
Display user number and account name

WHERE
Display pathname of current account
4.1 *(Comment)*

The * (comment) command allows comment text to be embedded in sentences and paragraphs.

**Format**

```
* {text}
```

A comment line has its first non-space character as an asterisk. There must be at least one space separating the asterisk from any `text`. The comment is ignored except that any inline prompt sequences within `text` will be executed. This enables prompts to be resolved in a convenient and logical order ahead of the need to use the responses.

Although comments are mainly used within stored paragraphs, they may be entered at the keyboard in response to the command prompt. The comment will be recorded in any active como file.

The value of `@SYSTEM.RETURN.CODE` is not affected by a comment.
4.2  $ECHO

The $ECHO directive inserted in a paragraph enables or disabled paragraph tracing.

**Format**

\[
\begin{align*}
&ECHO \{ON\} \\
&ECHO \text{ OFF}
\end{align*}
\]

The $ECHO directive (optionally with a qualifier of ON) enables paragraph tracing. When this mode is active, the command processor displays the paragraph name, line number and sentence for each line executed.

The $ECHO \text{ OFF} directive disables paragraph tracing.

$ECHO$ is not a command, but a control directive for the paragraph interpreter. Consequently it cannot be entered at the command line and it cannot be executed from a QMBasic program.

$ECHO \{ON\}$ if not followed by $ECHO \text{ OFF}$ in the same paragraph, turns on echo mode for all subsequent paragraphs, whether or not they also contain $ECHO$ directives. If placed in the LOGIN paragraph, it will turn on echo mode for all paragraphs throughout the account. This can be done selectively by, for example, testing the login name of the user. In this way, echo mode can be limited to developers who need it for debugging purposes.

```
IF @LOGNAME # 'TESTER' THEN GO FINISH
$ECHO ON
FINISH:
```

This fragment of paragraph logic turns echo on for user TESTER only. A developer who has a need to see paragraph commands echoed will log in with that user name. All other users will see paragraphs acting as usual.

**Example**

The following paragraph might be used to delete all items in the BP.OUT file for which there is no corresponding source record in the BP file.

```
VOC CLEAN.BP
1: PA
2: $ECHO
3: SELECT BP.OUT
4: NSELECT BP
5: IF @SELECTED = 0 THEN STOP
6: DELETE BP.OUT
7: DATA Y
```

Running this with the $ECHO on line 2 shows trace of each command prior to execution:

```
CLEAN.BP 3: SELECT BP.OUT
223 record(s) selected to list 0
CLEAN.BP 4: NSELECT BP
17 record(s) selected to select list 0
CLEAN.BP 5: IF @SELECTED = 0 THEN STOP
CLEAN.BP 6: DELETE BP.OUT
```
Use active select list (First item 'J7')? Y
17 record(s) deleted
4.3 ABORT

The **ABORT** command terminates all active processing and returns to the command prompt.

**Format**

```
ABORT{text}
```

where

- `text` is the optional message text to be displayed.

The **ABORT** command is intended for use where a paragraph detects an application fault and needs to terminate all active processing. All active commands, sentences, paragraphs, menus, etc. are discarded. Unless the command was run using the QMBasic **EXECUTE** statement with the TRAPPING ABORTS option, just before return to the command prompt, QM checks for an executable item (usually a paragraph) in the VOC named **ON.ABORT** and, if this is found, runs it.

The **ON.ABORT** paragraph may examine the `@ABORT_CODE` variable to determine why it was invoked. The **ABORT** command sets this variable to 1. The `text`, if present, will be stored in `@ABORT.MESSAGE`.

The value of `@SYSTEM.RETURN_CODE` is not affected by the **ABORT** command.

See also:  
**CLEAR.ABORT, STOP**
4.4 ACCOUNT.RESTORE

The **ACCOUNT.RESTORE** command restores a Pick style ACCOUNT-SAVE tape.

**Format**

```
ACCOUNT.RESTORE {options}
```

where

*options* is any combination of the following:

- **BINARY**
  Suppresses translation of field marks to newlines when restoring directory files. Use this option when restoring binary data.

- **DET.SUP**
  Suppresses display of the name of each file as it is restored.

- **DIRECTORY**
  Causes new files to be created as directory files. Existing files are not reconfigured.

- **NO.CASE**
  Causes new files to be created with case insensitive record ids. Existing files are not reconfigured.

- **NO.INDEX**
  Do not create alternate key indices.

- **NO.OBJECT**
  Omits restore of object code. This is particularly useful when migrating to QM from other environments.

- **POSITIONED**
  Assumes that the tape is already positioned at the start of the data to be restored.

The **ACCOUNT.RESTORE** command processes a Pick style "compatible mode" tape or pseudo tape and restores data from it into a QM system.

The tape to be restored must first be opened to the process using the **SET.DEVICE** command.

The command prompts for the name of the target account.

Items in the save that originated in the MD file are restored to a file named MD-RESTORE from where the user can then determine which are relevant to QM and require transfer to the VOC file. Similarly, if a save includes a VOC file, this will be restored to VOC-RESTORE.

Note that restoring a file from a different database product may take considerably longer than restoring the same save on the other product because the file hashing order will be different and the data will not appear in group by group order.

The format of ACCOUNT.SAVE tapes varies between multivalue products. A Pick style "compatible mode" (R83 format) tape commences with
```
Label
EOF block
Label
Descriptor block
EOF block
Label
```
Data.....

ACCOUNT.RESTORE therefore normally starts by skipping forwards to the third label block. On some systems, the tape commences simply with a label block followed immediately by the data. To allow for the possibility of this and other formats, the T.RDLBL and T.READ commands can be used to position the tape before the first data block. Use of the POSITIONED option in ACCOUNT.RESTORE will then omit all other positioning from within ACCOUNT.RESTORE itself.

ACCOUNT.RESTORE reads the account name and other information from the label it finds on the tape. This account name is then offered as the default account name in a confirmation dialogue. If the name supplied in this dialogue exists in the accounts register, the tape is restored into that account.

If the account name supplied is not found in the accounts register, ACCOUNT.RESTORE prompts for a system pathname to be used as the path to the account. Therefore one can restore a backup of an existing account by issuing an ACCOUNT.RESTORE command and then supplying a different name for the account. Any required items can then be copied into another account via a Q-pointer.

See also: ACCOUNT.SAVE, FILE.SAVE, FIND.ACCOUNT, QMSAVE, RESTORE.ACCOUNTS, SEL.RESTORE, SET.DEVICE, T.ATT, T.DUMP, T.LOAD, T.xxx
4.5 ACCOUNT.SAVE

The ACCOUNT.SAVE command creates a Pick style ACCOUNT-SAVE tape.

Format

ACCOUNT.SAVE {account.name} {options}

where

account.name is the name of the account to be saved. If omitted, the command prompts for the account name.

options specifies options processing features:

- **BINARY** suppresses translation of newlines to field marks when saving directory files. Use this option when saving binary data.
- **DET.SUP** suppresses display of the names of files saved.
- **EXCLUDE.REMOTE** causes remote files to be omitted as described below.
- **INCLUDE.REMOTE** causes remote files to be saved as described below.

The ACCOUNT.SAVE command creates a Pick style "compatible mode" pseudo tape and saves a QM account to it.

**Note carefully:** Although it would be possible to use this command to create a backup of a QM account, it is recommended that operating system level tools are used for this purpose. The media format used by ACCOUNT.SAVE cannot save ECS mode files or information related to triggers, alternate key indices, replication or encryption (see below). For this reason, ACCOUNT.SAVE is not considered suitable for routine system backups. It is provided primarily as a way to return data to a Pick style system during migration of an application.

The media format also has no way to represent a distributed file. These will be omitted and a warning message will be displayed. The data elements of the distributed file will be saved in the usual way so no data should be lost, however, it will be necessary to recreate the distributed file itself if the data is restored.

The tape to be created must first be opened to the process using the SET.DEVICE command.

The command reports its progress by displaying the name of each file as it is saved unless the DET.SUP option is used.

ACCOUNT.SAVE normally saves all files referenced by F-type records in the VOC of the account being saved. There is a three level mechanism by which files can be excluded:

1. Field 5 of the F-type VOC entry can contain
   - **D** Save the dictionary but omit the data element
   - **E** Exclude this file from an ACCOUNT.SAVE or FILE.SAVE
Include this file in an ACCOUNT.SAVE or FILE.SAVE

2. If field 5 of the VOC record does not specify any of the above flags, the EXCLUDE.REMOTE and INCLUDE.REMOTE options are used to determine whether remote files (those with a directory delimiter in their pathnames) are to be saved.

3. If neither of the above methods of file selection is used, the value of the EXCLREM configuration parameter is used to determine whether remote files are to be saved.

By use of a combination of the above methods, it should be possible to achieve total control of what is included in a save.

Encryption

ACCOUNT.SAVE operates within the security rules imposed by use of QM's encryption features. Files that use record level encryption will not be saved if the user performing the save does not have access to the encryption key. The data saved for files that use field level encryption will have all fields for which the user is denied access set to null strings. All saved data is recorded in decrypted form and hence storage of ACCOUNT.SAVE media may reduce system security. The media format used by ACCOUNT.SAVE is an industry standard that does not provide a way to record details of data encryption. Restoring a save in which encrypted fields have been omitted is unlikely to yield a usable file. Backup of accounts that include encrypted data should be performed using operating system level tools.

See also: ACCOUNT.RESTORE, FILE.SAVE, FIND.ACCOUNT, QMSAVE, RESTORE.ACCOUNTS, SEL.RESTORE, SET.DEVICE, T.ATT, T.DUMP, T.LOAD, T.xxx
4.6 ADD.DF

The ADD.DF command adds a part file to a distributed file.

Format

```
ADD.DF dist.file part.file part.no {algorithm} {RELATIVE}
```

where

- `dist.file` is the name of the distributed file.
- `part.file` is the name of the part file to be added.
- `part.no` is the part number associated with this part file.
- `algorithm` is the name of the partitioning algorithm in the dictionary of `part.file`.

First use of the ADD.DF command will create the distributed file, compiling the partitioning algorithm and adding the named part file. The newly created distributed file will share the dictionary of the first part file. Subsequent use of the ADD.DF command will add further part files. The algorithm need not be specified for the second and subsequent parts and will be ignored if present.

If the `part.file` is stored in the same directory as `dist.file` or a subdirectory of it, the RELATIVE keyword causes the `part.file` pathname to be stored internally relative to that directory. This can simplify copying distributed files.

When creating a new distributed file, the dictionary reference in the VOC F-pointer created to define this file is copied from the VOC entry for `part.file`. Again, use of the RELATIVE keyword will create the VOC entry with a relative pathname for the dictionary if it is in the same directory as `dist.file` or a subdirectory of it.

Note that the partitioning algorithm is copied into the distributed file. Changing the expression in the dictionary will not have any effect. It will be necessary to reconstruct the distributed file.

Example

```
ADD.DF ORDERS ORDERS-08 8 PART.ALG
ADD.DF ORDERS ORDERS-09 9
```

This pair of commands creates a distributed file named ORDERS that consists of two parts, ORDERS-08 and ORDERS-09, perhaps corresponding to orders taken in a specific year. The part numbers are 8 and 9 respectively. The partitioning algorithm is named PART.ALG in the dictionary of ORDERS-08.

See also:
Distributed files, LIST.DF, REMOVE.DF
4.7 ADMIN

The ADMIN command provides a simple menu of administrative commands.

Format

ADMIN

The ADMIN command is a multi-level menu of administrative commands. The structure of this menu and the commands associated with each entry are shown below.

Configuration
  Configuration Parameters  EDIT.CONFIG
  Security Settings        SECURITY
  Update Licence           UPDATE.LICENCE

Accounts
  List Accounts            LIST QM.ACCOUNTS
  Create Account           CREATE.ACCOUNT
  Delete Account           DELETE.ACCOUNT

User Management
  List Registered Users    LIST.USERS
  User Administration     ADMIN.USER

System Monitor
  Process Management
    List Logged-in QM Processes  LISTU
    Phantom Processes           STATUS ALL
    Process Status              PSTAT
    Logout Process              LOGOUT
    Send Message                MESSAGE

Lock Management
  List Task Locks          LIST.LOCKS
  List File and Record Locks LIST.READU
  Clear Task Lock           UNLOCK TASKLOCK
  Clear File Lock           UNLOCK FILELOCK
  Clear Record Lock         UNLOCK

Error Logging
  SP.VIEW QMSYS errlog

File System

QMNet
  List Servers             LIST.SERVERS
  Create New Server Definition CREATE.SERVER
  Server Security Administration ADMIN.SERVER
  Delete Server Definition  DELETE.SERVER

Performance and Statistics
  List Open Files          LIST.FILES
<table>
<thead>
<tr>
<th>Feature</th>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>File Statistics Monitor</td>
<td>FSTAT</td>
</tr>
<tr>
<td>Data Encryption</td>
<td></td>
</tr>
<tr>
<td>List Encryption Keys</td>
<td>LIST.KEYS</td>
</tr>
<tr>
<td>Create Encryption Key</td>
<td>CREATE.KEY</td>
</tr>
<tr>
<td>Grant Access to Key</td>
<td>GRANT.KEY</td>
</tr>
<tr>
<td>Revoke Access to Key</td>
<td>REVOKE.KEY</td>
</tr>
<tr>
<td>Delete Encryption Key</td>
<td>DELETE.KEY</td>
</tr>
<tr>
<td>Replication</td>
<td></td>
</tr>
<tr>
<td>Disable Publishing</td>
<td>DISABLE.PUBLISHING</td>
</tr>
<tr>
<td>Enable Publishing</td>
<td>ENABLE.PUBLISHING</td>
</tr>
</tbody>
</table>
4.8 ADMIN.SERVER

The **ADMIN.SERVER** command allows a system administrator to set security rules on definitions of QMNet public servers and VFS servers.

**Format**

```
ADMIN.SERVER {name}
```

where

- `name` is the name of the server for which the details are to be updated. If this is not present on the command line, a prompt appears.

If `name` does not correspond to an existing server, the **ADMIN.SERVER** command will offer to create a new server definition. This will prompt for the network address and optional port number and then enter the screen described below. If a port number is specified, this may be separated from the network address with a colon or a semicolon. If an IPV6 format address is used, which contains colons as part of its syntax, a semicolon must be used before the port number.

The default behaviour of the **SET.SERVER** or **SET.VFS.SERVER** command is to create a server definition that may be accessed by all users of the system. There is a potential security weakness here because the process started on the remote system to handle the connection runs as the user name specified in the server definition, regardless of the user name of the local user accessing the remote file. Security can be improved by arranging that the user name used for the remote process is dependent on the user name or user group of the local user. This can be achieved by use of the **ADMIN.SERVER** command. Because there is no way in which QM can determine the password for a specific user, it is not possible for the remote server login to automatically using the same username and password as the session from which QMNet or the VFS is used.

The screen display from this command is as shown below.

```
Remote user: george
Local users: gsmith, dave
O/S groups : QM groups :
--------------------------------------------------------------
Remote user: root
Local users:
O/S groups : admin QM groups : admin
--------------------------------------------------------------
Remote user: sales
Local users: ALL
O/S groups :
QM groups :
--------------------------------------------------------------
SALES 193.118.14.97
F1=Help
Enter remote user name
```

The display consists of a series of four line entries with a horizontal separator. Each entry identifies the remote user name that will be used for the remote process based on criteria related to the local user accessing the file. When creating a connection, the list is scanned from the top downwards looking for the first entry that is applicable to the user.
Remote user: The user name to be used for the remote process. Changing this name will also prompt for the associated password. Domain style user names should be entered as user@domain.

Local users: A comma separated list of user names on the local system who will connect as the associated remote user name. Specifying this field as ALL, allows connection by all users.

O/S groups: A comma separated list of operating system user group names. If the user is a member of a named group, access is granted with the associated remote user name. This field is ignored on Windows.

QM groups: A comma separated list of QM user group names as set with ADMIN.USER. If the user is a member of a named group, access is granted with the associated remote user name.

In the above example, users logged in to the local system as gsmith or dave will connect to the remote server with user name george. Users who are members of either the operating system user group named admin or the QM user group of the same name will connect as user name root. All other users will connect as user name sales.

If the local user does not meet the conditions set by any entry in the list, connection to the server is not permitted. If a user fits the conditions for more than one entry in the list, the first one found applies.

The default action of the SET.SERVER command is to create a server definition in which the remote user is as specified in the command and the local users field is set to ALL.

To move through the entries in the displayed list, use any of the following keys:

- Ctrl-N: Cursor down
- Ctrl-P: Cursor up
- Ctrl-Z: Page down
- Ctrl-A: Home
- Ctrl-B: Cursor left
- Ctrl-D: Delete
- Ctrl-E: End
- Ctrl-F: Cursor right
- Ctrl-H: Backspace
- Ctrl-K: Insert
- Ctrl-I: Move down to next line
- Ctrl-U: Move up to previous line
- Ctrl-Page down: Move down one page
- Ctrl-Page up: Move up one page

The amend a line, simply type new data or use any of the standard editing keys:

- Ctrl-A: Home
- Ctrl-B: Cursor left
- Ctrl-D: Delete
- Ctrl-E: End
- Ctrl-F: Cursor right
- Ctrl-H: Backspace
- Ctrl-K: Insert
- Ctrl-I: Move down to next line
- Ctrl-U: Move up to previous line
- Ctrl-Page down: Move down one page
- Ctrl-Page up: Move up one page

The amend a line, simply type new data or use any of the standard editing keys:

- Ctrl-A: Home
- Ctrl-B: Cursor left
- Ctrl-D: Delete
- Ctrl-E: End
- Ctrl-F: Cursor right
- Ctrl-H: Backspace
- Ctrl-K: Insert
- Ctrl-I: Move down to next line
- Ctrl-U: Move up to previous line
- Ctrl-Page down: Move down one page
- Ctrl-Page up: Move up one page

The amend a line, simply type new data or use any of the standard editing keys:

- Ctrl-A: Home
- Ctrl-B: Cursor left
- Ctrl-D: Delete
- Ctrl-E: End
- Ctrl-F: Cursor right
- Ctrl-H: Backspace
- Ctrl-K: Insert
- Ctrl-I: Move down to next line
- Ctrl-U: Move up to previous line
- Ctrl-Page down: Move down one page
- Ctrl-Page up: Move up one page

The amend a line, simply type new data or use any of the standard editing keys:

- Ctrl-A: Home
- Ctrl-B: Cursor left
- Ctrl-D: Delete
- Ctrl-E: End
- Ctrl-F: Cursor right
- Ctrl-H: Backspace
- Ctrl-K: Insert
- Ctrl-I: Move down to next line
- Ctrl-U: Move up to previous line
- Ctrl-Page down: Move down one page
- Ctrl-Page up: Move up one page

The amend a line, simply type new data or use any of the standard editing keys:

- Ctrl-A: Home
- Ctrl-B: Cursor left
- Ctrl-D: Delete
- Ctrl-E: End
- Ctrl-F: Cursor right
- Ctrl-H: Backspace
- Ctrl-K: Insert
- Ctrl-I: Move down to next line
- Ctrl-U: Move up to previous line
- Ctrl-Page down: Move down one page
- Ctrl-Page up: Move up one page

The amend a line, simply type new data or use any of the standard editing keys:

- Ctrl-A: Home
- Ctrl-B: Cursor left
- Ctrl-D: Delete
- Ctrl-E: End
- Ctrl-F: Cursor right
- Ctrl-H: Backspace
- Ctrl-K: Insert
- Ctrl-I: Move down to next line
- Ctrl-U: Move up to previous line
- Ctrl-Page down: Move down one page
- Ctrl-Page up: Move up one page

The amend a line, simply type new data or use any of the standard editing keys:
F2 Move current entry up by one place
F3 Move current entry down by one place
F4 Import security settings from another server. A prompt box appears asking for the server name. Entry of a blank response aborts the action.

Clearing the remote user name deletes the associated entry.

To insert a new entry, navigate to the bottom of the list and type in new data. The entry can be moved up if necessary with the F2 key.

To terminate the edit, optionally saving changes, press the Esc key.

See also:
The Virtual file system, QMNet, DELETE.SERVER, LIST.SERVERS, SET.SERVER
4.9 **ADMIN.USER**

The **ADMIN.USER** command allows management of the register of user names for network connections.

**Format**

```
ADMIN.USER
```

The **ADMIN.USER** command is built around a form filling interface. Initially it displays a request for a new or existing user name. The F2 key will display a pick list of registered users. Entering a blank user name exits from the command.

Selection of an existing user displays their details for possible amendment. Entry of a new user name displays an empty form into which the user's details may be entered.

An entry named DEFAULT (case sensitive) may be created. This will be used by the security system to determine the rights of any user for whom there is no separate entry in the register.

See [Application Level Security](#) for further details.

The fields in this screen are:

- **Last login**: Displays the date and time of last login in the local time zone of the user executing the command.
- **Owner details**: Unused by QM internally and may be used for any purpose.
- **Min password** (Windows USB and user mode installations on Linux/Unit only): Minimum acceptable password length. Leave blank to impose no restrictions.
- **Force account**: If set, the user is forced into this account on login except for QMClient processes. If left blank, they will be asked for an account name after entry of their user name and password.
- **Apply to phantoms**: Determines whether the force account option applies to phantom processes.
- **Administrator**: Is this user to have administrator rights?
- **QM user group**: Allows a user name to be associated with a group name (e.g. SALES) for use elsewhere in the QM security system. The group associated with a QM process can be determined from the [@QM.GROUP](#) variable.

**Allow**

Allows user to:

- **Create/delete accounts**
  
  Use [CREATE.ACCOUNT](#), [DELETE.ACCOUNT](#) and create accounts by entering QM in a directory that is not already an account.

- **Define private servers**
  
  Use [SET.PRIVATE.SERVER](#) to create QMNet server
definitions that are private
to the session in which the
command is issued.

Command prompt access  
If this option is not set, the
user cannot access the
interactive command
prompt. System
administrators always have
access.

In a system running with security turned off (see the SECURITY
command), any user not appearing in the user register will be able to
perform any of the actions controlled by the above options.

Valid accounts  
A comma separated list of accounts that the user is allowed to access.
If the "All except those listed below" option is selected, this list is
accounts that the user may not access. In either case, if the list is
empty, no restrictions are imposed.

Action  
Enter A to amend, F to file changes, D to delete this user or X to exit
without saving any changes.

See also:
CREATE.USER, DELETE.USER, LIST.USERS, PASSWORD, SECURITY, Application Level
Security
4.10 ALIAS

The ALIAS command creates a temporary alias for a command.

Format

```
ALIAS command target    Create an alias
ALIAS command          Remove an alias
ALIAS                  List all defined aliases
```

where

- `command` is the alias name.
- `target` is the command to which the alias applies. If this contains spaces, it must be quoted.

The ALIAS command creates an alternative name by which a command can be referenced such that `command` becomes a synonym for `target`. It provides a simple mechanism by which standard commands can be linked to alternative VOC entries as a means of providing improved compatibility with other multivalue database products. For example, the COPY command could be linked to the Pick style variant named COPYP by executing

```
ALIAS COPY COPYP
```

This change affects only the current process and does not modify the VOC. Typically, ALIAS commands would be executed from the LOGIN paragraph. To maintain system security, LOGIN cannot itself be aliased.

The second form of the ALIAS command removes a previously defined alias for `command`.

The third form lists all currently defined aliases.
4.11 ANALYSE.FILE

The **ANALYSE.FILE** command (which may also be entered using the American spelling) reports information regarding the structure and efficiency of a dynamic file. It can also be used to produce a simplified report of a directory file.

**Format**

```
ANALYSE.FILE [DICT] file.name {options}
```

where

*file.name* is the name of the file to be processed. The optional **DICT** prefix indicates that the dictionary portion of the file is to be used.

*options* are chosen from the following.

- **LPTR** `{n}` Directs output to the specified logical print unit. If *n* is omitted, the default printer is used.
- **NO.PAGE** Suppresses paging of the output. This option is ignored if **LPTR** is used.
- **STATISTICS** Extends the analysis to report record and group usage statistics.

Account : C:\QMSYS
File name : MESSAGES
Path name : C:\QMSYS\MESSAGES

Type : Dynamic, version 1
Group size : 1 (1024 bytes)
Large record size : 819
Minimum modulus : 1
Current modulus : 103 (0 empty, 27 overflowed, 1 badly)
Load factors : 80 (split), 50 (merge), 80 (current)
File size (bytes) : 146432 (106496 + 39936), 89905 used
Total records : 1706 (1704 normal, 2 large)

<table>
<thead>
<tr>
<th>Per group:</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Group buffers</td>
<td>1</td>
<td>3</td>
<td>1.28</td>
</tr>
<tr>
<td>Total records</td>
<td>9</td>
<td>28</td>
<td>16.56</td>
</tr>
<tr>
<td>Used bytes</td>
<td>36</td>
<td>1020</td>
<td>824.19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Bytes per record:</th>
<th>Minimum</th>
<th>Maximum</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>All records</td>
<td>16</td>
<td>4029</td>
<td>52.70</td>
</tr>
<tr>
<td>Normal records</td>
<td>16</td>
<td>804</td>
<td>49.82</td>
</tr>
<tr>
<td>Large records</td>
<td>984</td>
<td>4029</td>
<td>2506.50</td>
</tr>
</tbody>
</table>

The above example shows analysis of a dynamic file.

Note that all record size and group space usage figures are in bytes, including on an ECS mode system.
See also:

FILE.STAT
4.12 AUTHENTICATE

The AUTHENTICATE command reduces a user’s privileges in a startup command script.

Format

```
AUTHENTICATE username password
```

where

- `username` is the user name to be used for the revised privileges.
- `password` is the password for `username`.

A QM process started via the STARTUP configuration parameter without specifying a user name runs as SYSTEM on Windows and root on other operating systems. These users have very high levels of privilege that are probably inappropriate to the QM process. The AUTHENTICATE command allows a process running as one of these users to revise its level of access to that associated with some other user known to the operating system.

Typically, the STARTUP configuration parameter is used to run a paragraph in the VOC of the QMSYS account. This might contain, for example,

```
1: PA
2: LOGTO SALES
3: PHANTOM RUN MONITOR
4: PHANTOM RUN WEB.SERVER
```

The phantom processes created by this example would run as SYSTEM or root. Inserting

```
AUTHENTICATE username password
```
as the first command in the paragraph would cause the phantoms to run as the specified user. Alternatively, each phantom could contain its own use of AUTHENTICATE to become a different user.

The AUTHENTICATE command can only be used by SYSTEM and root. Restrictions imposed by the operating system mean that it is not possible for a user process running under any other user name to change its access privileges.

In the above example, the user’s password has been stored in the paragraph as clear text which may reduce system security. It would be possible to adapt this example to retrieve the password from an encrypted file where access to the encryption key is restricted. Alternatively, the AUTHENTICATE command supports direct use of an encrypted password on the command line. To use this, the `password` element of the command is replaced by the encrypted password prefixed with ENCR: (case insensitive). The encrypted form of the password is derived using the AUTHKEY command in the QMSYS account as described below.

The AUTHKEY command is available only in the QMSYS account and is restricted to users with administrator rights. The format of this command is

```
AUTHKEY password
```

The `password` should be enclosed in quotes if it contains spaces. The encrypted password value is displayed on the user's screen and can be transferred easily into the AUTHENTICATE command using cut and paste in the terminal emulator. The encrypted password is further transformed into a hexadecimal encoded form to ensure that non-graphic characters cannot cause problems.
The STARTUP configuration parameter also has an extended form that includes the account name, user name and password. In this case, the password may be encrypted using AUTHKEY in the same way as for AUTHENTICATE and inserted into the STARTUP parameter with a ENCR: prefix.

Example

The earlier example, adapted to use an encrypted password, might appear as

1: PA
2: AUTHENTICATE jsmith encr:6EAE7F355E276053A27FB87E
3: LOGTO SALES
4: PHANTOM RUN MONITOR
5: PHANTOM RUN WEB.SERVER
4.13 AUTOLOGOUT

The AUTOLOGOUT command sets an inactivity period after which a process will automatically be logged out.

**Format**

```
AUTOLOGOUT {period}
```

where

`period` is the inactivity time in minutes. A value of zero disables the timer.

The AUTOLOGOUT command prevents users leaving inactive sessions logged in. If the process is waiting for input for the given time, it will automatically be logged out. The ON.EXIT paragraph will be executed if it exists.

Executing the AUTOLOGOUT command with no `period` option displays the current setting.

Used in a QMClient session, the process will be terminated if no client/server interaction takes place within the autologout period, however, the client will not be aware of this termination until it next attempts a server operation. Note that autologout is not currently supported for QMClient connections established using the QMConnectLocal() function.

**Example**

```
AUTOLOGOUT
 Autologout is disabled

AUTOLOGOUT 4

AUTOLOGOUT
 Autologout period is set to 4 minute(s)
 ...wait...
 Inactivity time expired - Process logged out
 Process terminated
```

In this example, the AUTOLOGOUT command is used to examine the current setting. The inactivity period is then set to four minutes and the setting displayed again. After four minutes of inactivity, the process is automatically logged out.
4.14 BASIC

The BASIC verb runs the QMBasic compiler.

Format

```
BASIC {file.name} {record.name...} {options}
```

where

- `file.name` is the name of the file holding the QMBasic source program. This is usually a directory file. If omitted, the `filename` defaults to BP.
- `record.name` is the name of the record within the file. Multiple record names may be specified. An asterisk as the only `record.name` compiles all programs in the file. If no names are specified and the default select list is active, the list will be used to specify the programs to be compiled.
- `options` are as listed below.

When using a select list or an asterisk to compile all programs, the default action of the BASIC command is to ignore records with a .H or .SCR suffix. This omits include records that use the conventional .H suffix (header files) or .SCR suffix (screen definitions). The $BASIC.IGNORE record described below allows developers to change the rules of which, if any, records are skipped when using a select list.

Unless the KEEP.OLD.OBJECT mode of the OPTION command is active, any old version of the object code is deleted before the compilation commences. This ensures that developers who accidentally miss a compiler error message will not continue testing with the previous version in the incorrect belief that it is the new version.

The following options are accepted by the BASIC verb

- **DEBUGGING** Include debugger control information in the compiled program. A program compiled in debug mode can be executed outside the debugger but will be slightly slower than when compiled without the DEBUGGING option.

- **CHANGED** Compile only if the program is not already in the output file or the date and time of modification of the source program record as given in the operating system directory entry is later than that of the compiled program. Used in conjunction with a select list or the * option, it enables all modified programs to be compiled with a single command. This option requires that `file.name` is a directory file.

- **CONCORDANCE** Generate concordance data (see below).

- **LISTING** Generate a compiler listing record with a .LIS suffix.

- **NO.PAGE** Suppress pagination.
### The Compiler Options

The compiler output file, named as the source file but with a `.OUT` suffix, must be a directory file and will be created automatically if it does not already exist unless the `file.name` is a **Q-pointer**.

@SYSTEM.RETURN.CODE is set to the number of programs successfully compiled. It will contain a negative error code in the event of a fatal error.

Compiling a program signals an event to all QM processes to reload the object code. See the QMBasic **CALL** statement for full details.

### The $BASIC.OPTIONS VOC Record

Compiler options that you wish to use every time you run the QMBasic compiler can be placed in an X-type VOC record named `$BASIC.OPTIONS`. To apply these defaults only to programs stored in a specific file, place the `$BASIC.OPTIONS` record in that file. The compiler looks first in the source file (or the dictionary when compiling a C-type item) and then, if no record has been found, in the VOC.

The first line of the record holds the type code X. The second and subsequent lines of this record should contain compiler option keywords from the list below. The keywords allowed are:

- **CATALOGUE**
  - **LOCAL**
  - **GLOBAL**

(American spelling allowed) Automatically catalogues programs after compilation using the program name as the catalogue name. The LOCAL and GLOBAL keywords may be used to specify that the program is to be catalogued in the given mode instead of in the private catalogue. When using GLOBAL, an optional prefix character, `c`, can be included.

See the **CATALOGUE** command for more information. The **NO.QUERY** keyword can be added to suppress the query prompt that normally occurs if the program is already catalogued in some other mode.

The **$NO.CATALOGUE** compiler directive can be used in specific program modules to override this action. Alternatively, the **$CATALOGUE** compiler directive in
a program can set an alternative catalogue name or mode.

**CONCORDANCE**
Generate concordance data (see below).

**DEBUGGING**
Compiles the program in debug mode.

**DEFINE name {value}**
Defines token name in the same way as the `$DEFINE` compiler directive. The value may be a number or a quoted string. If omitted, the token is assigned a null string as its value.

**INCLUDE {file} id**
Insert the named include record after processing the PROGRAM, SUBROUTINE, FUNCTION or CLASS statement.

**LISTING**
Generates a listing record in the compiler output file.

**MODE option.name**
Sets the given compilation mode as described for the `$MODE` compiler directive. Multiple MODE lines must be used to set more than one option.

**NOCASE.STRINGS**
Compiles the program with case insensitive string operations. See the `$NOCASE.STRINGS` compiler directive for more details. This action is deprecated and replaced by MODE NOCASE.STRINGS.

**NOXREF**
Compiles the program with no cross reference tables. This results in slightly lower memory usage but prevents QM producing detailed messages in the event of an error.

**NO.PAGE or NOPAGE**
Suppress pagination.

**SEARCH filename.list**
Specifies a comma separated list of file names where the compiler will look for include records if no file name is given in the $INCLUDE directive, in the order in which they will be searched. Use of the @ symbol as one of the names searches the file containing the program source record. If no SEARCH list is specified in the $BASIC.OPTIONS record, the default is to look in the file containing the program source record followed by SYSCOM and is equivalent to

```
SEARCH @, SYSCOM
```

**WARNINGS.AS.ERRORS**
Causes the compiler to treat warning messages as fatal errors.
**XREF** Generates a listing record in the compiler output file, including a cross-reference table of all variables and their use.

Unrecognised keywords in this record are ignored.

Of the items listed above, only **DEFINE, MODE, NOCASE, STRINGS** and **WARNINGS, AS, ERRORS** are allowed in C-type items.

The **$BASIC.IGNORE VOC Record**

The default action of the **BASIC** command when using a select list or an asterisk to compile all programs is to ignore source records with a suffix of `.H` or `.SCR`. The **$BASIC.IGNORE** record allows developers to set their own rules controlling which, if any, records will be ignored. As with the **$BASIC.OPTIONS** record, the **$BASIC.IGNORE** record may be in the VOC or in the program file.

The first line of the record holds the type code X. The second and subsequent lines of this record contain pattern matching templates that are applied in turn to the source record id. If any of the templates match the source record id, the record is ignored. The default action of the **BASIC** command is equivalent to having a **$BASIC.IGNORE** record that contains:

```
1: X
2: ...'.H'
3: ...'.SCR'
```

Any record with an id that does not match any of the templates will be compiled. To compile all records, regardless of their name, create a **$BASIC.IGNORE** record that contains just the type code:

```
1: X
```

**Concordance Data**

The QMBasic compiler can create a cross-reference of subroutine usage through an entire application. The compiler will automatically create a file named **BASIC.CONCORDANCE** if it does not already exist. Compiling all programs with either the **CONCORDANCE** option on the command line or via the **$BASIC.OPTIONS** record will then generate the concordance data.

The **BASIC.CONCORDANCE** file will contain a record for each compiled program. The record id in this file is the uppercase form of the name in the **PROGRAM, SUBROUTINE, FUNCTION** or **CLASS** statement or, if no such line is present, the uppercase form of the program source file name. Field 1 (named **REF**) of each record is a multivalued list of all the external modules referenced by the program. In the case of an indirect call, the name appears as the `@` symbol as the compiler cannot determine the actual name.

Using this file, it is easy to use the query processor to find all references to a particular subroutine or other relationships. For example...

Show a simple list of what subroutines are called from each program

```
LIST BASIC.CONCORDANCE
```

Find all the programs that reference subroutine **MYSUB**

```
LIST BASIC.CONCORDANCE WITH REF = "MYSUB"
```

Show a list of which programs call each subroutine
LIST BASIC.CONCORDANCE WITH REF BY.EXP REF BREAK.ON "'OL'" REF @ID ID.SUP

Recompiling a module with the CONCORDANCE option set will update the file. The only time it will be necessary to make manual changes to the file is if a source program is deleted and its corresponding concordance record can therefore also be deleted.

Examples

BASIC PROGRAMS PROG1 LISTING

This command compiles the program in record PROG1 of the PROGRAMS file. A listing record is produced.

SELECT BP
BASIC CHANGED

This sequence of commands compiles all programs in the BP file which have been updated since they were last compiled. Note that the compiler will omit all records with names ending with .H or .SCR, the two standard suffix codes for include records.

See also:
CATALOGUE, DELETE.CATALOGUE, MAP
4.15 BELL

The **BELL** command determines whether the audible alarm is sounded by various QM verbs (typically on encountering error conditions) or by QMBasic programs using the [@SYS.BELL](#) function in a print operation directed to the terminal.

**Format**

- **BELL OFF**  To disable the audible alarm
- **BELL ON**   To enable the audible alarm

[@SYSTEM.RETURN.CODE](#) is returned as

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>after <strong>BELL OFF</strong></td>
</tr>
<tr>
<td>1</td>
<td>after <strong>BELL ON</strong></td>
</tr>
<tr>
<td>-ve</td>
<td>after an error</td>
</tr>
</tbody>
</table>

Printing CHAR(7) to the terminal sounds the audible alarm regardless of the setting of the bell status.
4.16 **BLOCK.PRINT, BLOCK.TERM**

The `BLOCK.PRINT` command prints text on the default printer using large characters. The `BLOCK.TERM` is similar but directs its output to the terminal.

**Format**

```
BLOCK.PRINT text

BLOCK.TERM text
```

where

```
text    is the text to be printed or displayed.
```

The `BLOCK.PRINT` and `BLOCK.TERM` commands print `text` using a large font constructed from a 7 x 7 matrix of characters. On an 80 character printer or terminal, a maximum of 8 characters will fit on a line. The command will wrap long text across multiple lines, breaking on spaces where possible.

By default, the characters in the output are formed from use of the letter "W". There is an X-type control record named "$BIGLTRS" in the VOC of the QMSYS account that can be used to change how the characters are constructed. Field 2 of this record contains a mode value that may be:

0 Use "W" (default)
1 Use uppercase version of the character
2 Use preserved case version of the character
3 Use character shape definitions as in fields 4 to 10.

Adding 8 to the above values centres the text in the available device width.

When using mode 3, the characters will be constructed from the patterns in fields 4 to 10. Field 3 holds a character reference that may make editing easier.
### 4.17 BREAK

The **BREAK** command controls the action taken on use of the break key. It can be used, for example, the suppress quits during critical parts of an application.

**Format**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BREAK OFF</td>
<td>To increment the break inhibit counter</td>
</tr>
<tr>
<td>BREAK ON</td>
<td>To decrement the break inhibit counter</td>
</tr>
<tr>
<td>BREAK RESET</td>
<td>To enable breaks by resetting the break inhibit counter to zero</td>
</tr>
<tr>
<td>BREAK CLEAR</td>
<td>To cancel deferred breaks</td>
</tr>
<tr>
<td>BREAK COUNT</td>
<td>To report the value of the break inhibit counter (the number of active BREAK OFF commands)</td>
</tr>
<tr>
<td>BREAK ON USER n</td>
<td>To enable the break key for the specified user</td>
</tr>
</tbody>
</table>

QM maintains a count of the number of times that breaks are disabled. Each **BREAK OFF** command increments this count. The **BREAK ON** command decrements the count unless it is already zero. The **BREAK RESET** command sets the counter to zero.

The **BREAK COUNT** command reports the current value of the break inhibit counter.

If the break key is pressed whilst breaks are suppressed, the break is deferred until the count returns to zero by a subsequent use of **BREAK ON**. The normal action prompt will then appear. The **BREAK CLEAR** statement cancels any deferred break event.

For all of the above forms, `@SYSTEM.RETURN.CODE` is returned as the current value of the break inhibit counter unless an error occurs, in which case it is set to a negative error code.

The final form, **BREAK ON USER n**, is only available to users registered as administrators and enables the break key for the specified user. `@SYSTEM.RETURN.CODE` is returned as zero unless an error occurs, in which case it is set to a negative error code.

**See also:**
- **BREAK** (QMBasic), **SET.BREAK.HANDLER**, **REMOVE.BREAK.HANDLER**
4.18 BUILD.INDEX

The **BUILD.INDEX** command populates an **alternate key index**

**Format**

```
BUILD.INDEX filename field(s) {CONCURRENT}
BUILD.INDEX filename ALL {CONCURRENT}
```

where

- `filename` is the name of the file for which the index is to be built.
- `field(s)` is one or more field names for which indices have been created.

The **BUILD.INDEX** command deletes any existing index data for the named field(s) and populates the index by processing all records currently in the file. The **ALL** keyword can be used to build all indices that have been created for the file.

**BUILD.INDEX** is best performed immediately after using **CREATE.INDEX** to construct the index. Once the index has been built, it will be maintained automatically whenever changes are made to the file and will be used automatically by the query processor. The **MAKE.INDEX** command combines index creation and build into a single operation.

Except when used with the **CONCURRENT** keyword, the **BUILD.INDEX** command requires exclusive access to the file and may take some time to complete for a very large file. It is therefore best executed at quiet times.

The **CONCURRENT** keyword allows an index to be built while the file is in use. There is a small performance overhead for this and the QMBasic **SELECTINDEX**, **SELECTLEFT** and **SELECTRIGHT** statements may stall waiting for the build to complete.

**See also:**

**CREATE.INDEX, DELETE.INDEX, DISABLE.INDEX, LIST.INDEX, MAKE.INDEX, REBUILD.ALL.INDICES**
4.19 CATALOGUE

The CATALOGUE command (which may be entered with the American spelling for compatibility with other products) adds a compiled QMBasic program to the global or private catalogue file or as a locally catalogued entry in the VOC.

Format

```
CATALOGUE {file.name {catalogue.name}} record.name {options}
```

where

- **file.name** is the name of the directory file holding the program. If omitted, this defaults to BP, the .OUT suffix being added automatically.
- **catalogue.name** is the name by which the program or subroutine is to be catalogued. If omitted, the record.name is used. The catalogue name is translated to uppercase though calls from QMBasic programs are case insensitive. Except as described below, catalogue names must start with a letter. Subsequent characters may be letters, digits, periods, percent signs, dollar signs, hyphens or underscores.

For compatibility with other database systems, the catalogue.name may also commence with a digit though such catalogued items can only be used as user defined conversion codes and not as call names in programs.

- **record.name** is the name of the record within the specified file. The record.name may be specified as an asterisk to catalogue all programs in file.name. If record.name is omitted and the default select list is active, this list will be used to determine the programs to be catalogued.

- **options** are any of the following:
  - **LOCAL** The program is to be catalogued in the VOC.
  - **PRIVATE** The program is to be placed in the private catalogue for use only by this account.
  - **GLOBAL {c}** The program is to be catalogued for use by all accounts. The optional c component of this command is the prefix character to be used if the catalogue.name does not already include a prefix.
  - **NO.PAGE** Suppresses page end prompts when output exceeds page length (e.g. use of select list).
  - **NO.QUERY** Suppresses all confirmation prompts.
The `CATALOGUE` command makes a program or subroutine available for access via the QMBasic `CALL` statement. Catalogued programs can also be executed as command simply by entering their name at the command prompt or within a stored sentence or paragraph.

If none of the `LOCAL`, `PRIVATE` or `GLOBAL` catalogue mode options is present, the default action is to use the private catalogue. The `CATALOGUE.LOCAL` setting of the `OPTION` command changes this default to be the local catalogue. Where the name of the item to be catalogued starts with one of the reserved prefix characters described below, it is always placed in the global catalogue.

Compiling a program signals an event to all QM processes to reload the object code. See the QMBasic `CALL` statement for full details.

**Private Cataloguing**

The program is copied to the private catalogue in the account from which the command is executed and is available only to users of that account. Recompiling the program requires that the program is recatalogued so that the new version is copied to the catalogue directory. A failed compilation, where no object code has been produced, will not impact users of the previously catalogued version.

The private catalogue is normally a subdirectory, cat, under the account directory but can be moved by creating an X-type VOC entry named $PRIVATE.CATALOGUE in which field 2 contains the pathname of the alternative private catalogue directory. This only takes effect when QM is re-entered or on use of the `LOGTO` command. This feature is particularly useful where two or more accounts are to share a common private catalogue. The US spelling, $PRIVATE.CATALOG, may be used instead. If both are present, the British spelling takes priority.

**Global Cataloguing**

The program is copied to the global catalogue file in the QMSYS directory and is available from all accounts. For compatibility with other products, global cataloguing is also implied by adding one of the following prefix characters to the `catalogue.name` of the program:

* User subroutine prefix provided for compatibility with other systems
- User callable system supplied subroutines
! User callable system supplied subroutines
Internal undocumented subroutines

System subroutines, only callable from internal components of QM

Use of a prefix character has the disadvantage that the prefix must also be included in calls to the subroutine as it becomes part of the subroutine name.

Recompiling the program requires that the program is recatalogued so that the new version is copied to the catalogue directory. A failed compilation, where no object code has been produced, will not impact users of the previously catalogued version.

Local Cataloguing

A record is written to the VOC file as a pointer to the object code allowing calls only from the account in which the program is catalogued.

The VOC entry for a locally catalogued program is of type V (verb) and has a dispatch field of CS. The third field holds the full pathname of the executable program. When QM is installed on a USB memory device, the pathname will be transformed internally to a format that allows for the drive letter to be different when the stick is loaded in future. Use of the RELATIVE option to the CATALOGUE command stores a relative pathname so long as the program is in a subdirectory of the account directory. Use of the FULL.PATH option stores the full pathname. If neither option is present, the pathname is saved in the form determined by the setting of the LOCAL.CAT.RELATIVE mode of the OPTION command.

If a locally catalogued program that has a security subroutine defined in its VOC entry is recatalogued, the security subroutine name is carried forward to the new VOC entry.

Recompiling the program does not require the program to be recatalogued as the VOC pointer will still be in place. A failed compilation, where no object code has been produced, may cause other processes to fail. Use of the KEEP.OLD.OBJECT mode of the OPTION command will retain the old object code at a compilation error.

See also:
BASIC, DELETE.CATALOGUE, FIND.PROGRAM, MAP
4.20 CED

The CED command executes the data collection editor.

Format

CED (DICT) file.name {id...}

where

file.name is the name of the file containing the data to be edited.

id is a list of ids for the record(s) to be edited. If omitted, the default select list is used. If this is not active, the user is prompted for the record ids.

The CED command allows creation, viewing or editing of a data collection in a collection file or a JSON string from a normal file. The same functionality for a data collection variable within a QMBasic application is available using the !CED() subroutine.

On entry to the collection editor, the top level item within the collection is displayed, showing the names, data types and values of each element of the item. If the item is an array, the element numbers are displayed as their names. A marker in the leftmost column identifies the currently selected element.

The following keys are available:

Cursor up/down Select previous/next item
Page up/down Move to previous/next page
F1 or ? Display help screen
F3 Rename element
F4 Change element data type
F5 or Ctrl-C Copy selected element
F6 or Ctrl-X Cut selected element
F7 or Ctrl-V Paste previously copied/cut element
Ctrl-Q Quit from any level
Ctrl-S Save changes, continuing edit
Del Delete selected element
Ins or Ctrl-I Insert a new element
Return Dive into selected element
Esc Return to parent element or exit from action on a data element

The F3 (rename) action prompts for the new name and applies it to the selected element.

The F4 (change element type) action prompts for a new data type. Only those that are valid translations of the selected item are offered.

F5, F6 and F7 or their control key equivalents operate in a similar manner to copy/cut/paste in Windows applications. When pasting, a prompt appears asking for the new name.

The Del key deletes the currently selected item without saving it for later pasting.

The Ins key prompts for an item name, data type and value. When inserting a collection or array, an empty item is inserted to which values can be added separately.
Using the Return key to dive into an element which is a further collection or an array displays the elements within that item. The current element path is displayed at the bottom of the screen.

Using the Return key to dive into a data element allows the element to be modified. Use of Return commits the change, use of Esc exits without changing the data.

When editing is complete, entry of Q at the top level item or use of Ctrl-Q at any level exits from the editor, offering to save changes if appropriate.

See also:
Data collections, data collection files, !CED()
4.21 CHANGE.KEY.PASSWORD

The CHANGE.KEY.PASSWORD command changes the password for a secure password protected key. This command can only be executed by users with administrator rights in the QMSYS account.

Format

CHANGE.KEY.PASSWORD {keyname}

where

keyname is the name of the encryption key.

The command prompts for the keyname if it is not supplied on the command line and for the old and new passwords. To cancel password protection, enter a blank password.

A password protected secure encryption key must be enabled using the ENABLE.KEY command or corresponding ENABLE.KEY QMBasic statement in a QM session before it can be used. The CHANGE.KEY.PASSWORD command allows an administrator to change the password associated with the key.

See also:
Data encryption, CREATE.FILE, CREATE.KEY, CREATE SECURE.KEY, DECRYPT.FILE, DELETE.KEY, DISABLE.KEY, DISABLE.KEY (QMBasic), ENABLE.KEY, ENABLE.KEY (QMBasic), ENCRYPT.FILE, GRANT.KEY, LIST.KEYS, RESET.MASTER.KEY, REVOKE.KEY, SET.ENCRIPTION.KEY.NAME, UNLOCK.KEY.VAULT
4.22 CHAR.MAP

The CHAR.MAP command specifies the character map to be used by the QM session on a non-ECS system.

**Format**

```
CHAR.MAP {name}
```

where

*name* is the name of the map to be used.

This command, available only on a non-ECS mode system, allows a QM session to change the character map used to control sort ordering, upper/lower case pairing and character types.

When a QM session is started (login, phantom creation, etc) on a non-ECS system, the character map defined by the CHARMAP configuration parameters is loaded or, if no map is specified in the configuration parameters, a system supplied default map is used. The CHAR.MAP command can be used to select an alternative map within a QM process, for example, to meet local language conventions for character sort order.

Use of the CHAR.MAP command without a map name displays the name of the currently loaded map.

Non-ECS mode systems provide only limited support for character maps. In particular, note that changing character map may invalidate alternate key indices if the data in the file was written when a different map was in use. It may be necessary to rebuild indices.

**See also:**
[Extended Character Set Support](#)
4.23 CLEAN.ACCOUNT

The CLEAN.ACCOUNT command clears the $HOLD, $COMO and $SAVEDLISTS files.

Format

CLEAN.ACCOUNT

The CLEAN.ACCOUNT command is used to remove records from the $HOLD, $COMO and $SAVEDLISTS files. Periodic use of this command will ensure that redundant records are not left in these files. A file cannot be cleaned if there are locks held on records in the file.
4.24 CLEAR.ABORT

The **CLEAR.ABORT** command clears the abort status in an **ON.ABORT** paragraph.

**Format**

```
CLEAR.ABORT
```

When an application generates an abort event, QM discards all programs, paragraphs, menus, etc running in the process and returns to the command processor. Before displaying the command prompt, the system checks for an **ON.ABORT** item in the VOC (usually a paragraph) and, if found, executes it.

The **ON.ABORT** paragraph is intended as a means of preventing a user ever arriving at the command prompt if the application fails. Typically, the **ON.ABORT** paragraph simply terminates the session but it might log the event or perform other processing. If the processing in the **ON.ABORT** paragraph causes a further abort, the session is terminated.

Sometimes, it may be useful to restart the application from the **ON.ABORT** paragraph. In this case, a further abort should probably re-enter the **ON.ABORT** paragraph. The normal action of automatic termination of the session on the second abort can be suppressed by clearing the abort status in the **ON.ABORT** paragraph. It then becomes the developer’s responsibility to avoid endless loops if the **ON.ABORT** action generates a further abort.

Use of the **CLEAR.ABORT** command also sets the `@ABORT_CODE` variable to zero.

**See also:**

**ABORT** command, QMBasic **ABORT** statement
4.25 CLEAR.DATA

The **CLEAR.DATA** command (synonym **CLEARDATA**) clears the data queue created by the **DATA** command or the QMBasic **DATA** statement.

**Format**

```
CLEAR.DATA
```

The data queue is cleared automatically on return to the command prompt. The **CLEAR.DATA** command allows the queue to be cleared within a paragraph, for example, when recovering from premature termination of a program that uses the data queue.

**@SYSTEM.RETURN.CODE** is not affected by this command.

**Example**

```
PA
RUN ARCHIVE.PREVIOUS.YEAR
DATA Y
SELECT ORDERS WITH DATE BEFORE "1 JAN <<@YEAR>>"
DELETE ORDERS
```

The above paragraph might be used to archive old order data at the start of a new business year and then delete the old records from the ORDERS file. The archive program asks the user if a report is to be printed and the **DATA** statement is used to provide the answer to this question from within the paragraph. Because it is using a select list, the **DELETE** command will prompt the user for confirmation before deleting the selected records.

If the archive program fails before asking whether a report is required, the "Y" in the data queue will not be read by this program and will still be in the queue when the **DELETE** command is executed. It will therefore be used erroneously to answer the delete confirmation prompt.

This situation can be avoided by inserting a **CLEAR.DATA** statement before the **SELECT**. In general, it is good practice to insert a **CLEAR.DATA** after any command that uses **DATA** and could terminate without reading the queued data. This action is not automatic because the data queue is frequently used to pass data from one program to another.

**See also:**
**CLEAR.INPUT, CLEARDATA** QMBasic statement
The CLEAR.FILE command deletes all records from a file.

**Format**

```
CLEAR.FILE {DATA | DICT} file.name
```

If neither the DATA keyword nor the DICT keyword is not specified, only the data portion of the file is cleared.

If the DICT keyword is specified, only the dictionary portion of the file is cleared.

In the case of a dynamic file, all overflow space is released and, unless the file has the NO.RESIZE mode enabled, the file returns to its minimum modulus value.
4.27 CLEAR.INPUT

The **CLEAR.INPUT** command (synonym **CLEARINPUT**) clears all unprocessed characters entered at the keyboard.

**Format**

**CLEAR.INPUT**

The **CLEAR.INPUT** command clears all keyboard type-ahead. It may be useful, for example, after a program terminates at an error to ensure that unprocessed keyboard data is not treated as input to the next program.

[@SYSTEM.RETURN.CODE](#) is not affected by this command.

See also:
**CLEAR.DATA**
4.28 CLEAR.LOCKS

The CLEAR.LOCKS command releases task locks.

Format

    CLEAR.LOCKS {lock.number}

where

    lock.number is the number of the task lock (0 to 63) to be released. If omitted, all
    task locks held by the process are released.

@SYSTEM.RETURN.CODE is set to the lock number if releasing a single lock or 64 if releasing all
locks. Errors result in negative error code values.

Examples

    CLEAR.LOCKS 5
    Released task lock 5

This example shows the CLEAR.LOCKS command used to release a specific task lock.

    CLEAR.LOCKS
    All task locks released

This command releases all task locks held by the process.

See also:
LIST.LOCKS, LOCK (command), LOCK (QMBasic), TESTLOCK(), UNLOCK
4.29 CLEAR.PROMPTS

The **CLEAR.PROMPTS** command (synonym **CLEARPROMPTS**) clears all stored inline prompts and responses.

**Format**

```
CLEAR.PROMPTS
```

Inline prompt responses are cleared automatically on return to the command prompt. The **CLEAR.PROMPTS** command allows responses to be cleared within a paragraph should this be necessary for correct processing of later inline prompts.

[@SYSTEM.RETURN.CODE](#) is not affected by this command.

**See also:**

[Inline prompts](#)
4.30 CLEAR.SELECT

The CLEAR.SELECT command (synonym CLEARSELECT) clears an active select list. It is useful to prevent use of a select list by a subsequent command when, for example, the SELECT was performed in error.

Format

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLEAR.SELECT</td>
<td>Clears select list 0, the default list</td>
</tr>
<tr>
<td>CLEAR.SELECT list.number</td>
<td>Clears the specified list (0 to 10)</td>
</tr>
<tr>
<td>CLEAR.SELECT ALL</td>
<td>Clears all select lists</td>
</tr>
</tbody>
</table>

@SYSTEM.RETURN.CODE is returned as the list.number when clearing a specific select list, 11 when clearing all select lists or a negative error code.
4.31 CLEAR STACK

The **CLEAR STACK** command clears the command stack.

**Format**

    CLEAR STACK

The **CLEAR STACK** command removes all entries from the current command stack. **@SYSTEM.RETURN.CODE** is not affected by this command.

**See also:**

    GET STACK, SAVE STACK
4.32 CLOSE.ECI

The `CLOSE.ECI` command terminates all External Call Interface child processes.

**Format**

`CLOSE.ECI`

An External Call Interface handler normally continues to run until the QM process terminates or uses `LOGTO` to move to a different account. The `CLOSE.ECI` command terminates all ECI handler child processes, perhaps to allow the handler to be recompiled.

**See also:**

[External Functions](#)
4.33 CLR

The CLR command (synonym CS) clears the terminal screen.

Format

CLR

@SYSTEM.RETURN.CODE is not affected by this command.
4.34 CNAME

The CNAME command (synonym RENAME) changes the name of a file or record(s) within a file.

**Format**

```
CNAME old.file.name, new.file.name
CNAME old.file.name TO new.file.name
```

where

- `old.file.name` is the current name of the file to be renamed.
- `new.file.name` is the new name of the file to be renamed.

or

```
CNAME {DICT} file.name old.record.id, new.record.id
CNAME {DICT} file.name old.record.id TO new.record.id
```

where

- `file.name` is the name of the file containing the record(s) to be renamed. The optional DICT prefix specifies that the dictionary portion of the file is to be processed.
- `old.record.id` is the current name of the record to be renamed.
- `new.record.id` is the new name for the renamed record.

Used with two file names, the CNAME command renames *old.file.name* to *new.file.name*. The VOC record defining the file is renamed. The underlying operating system directories representing the data file and the dictionary are also renamed if they are the default names and the new names are acceptable operating system directory names. Where the directory names are not the defaults, as would be the case for a file created with the PATHNAME option of the CREATE.FILE command, the directory can be renamed separately using operating system commands but the VOC entry must be manually edited to match this change.

Used with a single file name and two record names, the CNAME command renames a record within the file. In this format, multiple records may be renamed in a single command by repeating the `old.record.id` TO `new.record.id` component of the command.

The command

```
CNAME VOC OLD,NAME TO NEW,NAME
```

is equivalent to

```
CNAME OLD,NAME TO NEW,NAME
```

except that the directory will not be renamed at the operating system level.

**Examples**

```
CNAME CUST.FILE A7194 TO A7149
```
This command renames record A7194 of file CUST.FILE to A7149.

CNAME STOCK, INVENTORY

This command renames the STOCK file to INVENTORY. The VOC record defining the file is renamed. The underlying operating system directories representing the data file and the dictionary are also renamed if they are the default names and the new names are acceptable operating system directory names.

CNAME STOCK A8135,A008135 D4923,D004923

This command renames two data records within the STOCK file.
4.35 COMMAND.LOGGER

The COMMAND.LOGGING command enables or disables command logging.

Format

COMMAND.LOGGING ON       Commence logging of commands executed in the current QM process.

COMMAND.LOGGING OFF       Terminate logging.

The command logging system allows creation of audit logs of every command executed in specific QM processes or globally across all processes. Global logging is controlled by the CMDLOG configuration parameter. If global logging is not enabled, the COMMAND.LOGGING command may be used to create a log specific to the process in which the command is used.

For more details of how logging works, see Command Logging.
## 4.36 COMO

The **COMO** command controls recording of terminal output in a como (command output) file.

### Format

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMO ON ( {filename} \ id \ {APPENDING} )</td>
<td>Commence recording of terminal output in named record.</td>
</tr>
<tr>
<td>COMO SUSPEND</td>
<td>Temporarily suspends logging output in the como file.</td>
</tr>
<tr>
<td>COMO RESUME</td>
<td>Resumes logging out output after a previous use of the SUSPEND option.</td>
</tr>
<tr>
<td>COMO OFF</td>
<td>Terminate recording of terminal output.</td>
</tr>
<tr>
<td>COMO</td>
<td>Report current logging status.</td>
</tr>
</tbody>
</table>

When enabled, all output that is directed to the user's terminal by the process in which the **COMO** command is executed is also written to the named record in the como file.

The **COMO** command is mainly intended as a diagnostic aid during application development but can be used as a general logging mechanism. It can be used, for example, to look back at an error message that was overwritten on the screen before the user had time to read it.

Como records are normally stored in a file named $COMO. This is created automatically when the **COMO ON** command is first used. The default pathname of $COMO under the account directory can be changed by creating the file in an alternative place. The file used for storing como data must be a directory file. On an ECS mode system, the encoding to be used for the $COMO file can be specified in field 7 of the VOC entry in the same way as for any other directory file. If no encoding is specified, the low order 8 bits of each character will be written.

An alternative file can be used by including the \( filename \) element of the **COMO** command. This file must already exist as a directory file on the QM server.

If \( id \) is not valid as an operating system file name, it will be transformed to a valid name in the same way as any other directory file record id.

The **APPENDING** keyword allows data to be appended to an existing como record. If there is no record with the given \( id \), a new record is created.

The **COMO SUSPEND** command increments a counter that inhibits logging of output data. The corresponding **COMO RESUME** command decrements the counter. Output is logged only when the counter is zero. The action of these commands cannot take the counter out of the range 0 to 100. By maintaining a counter instead of a simple on/off toggle, it is possible for stacked operations to manipulate the inhibit mechanism correctly.

Except when started with the **NO.LOG** option, a phantom process behaves as though it started with a **COMO ON** command, logging all output that would have been sent to the terminal if the same commands had been executed by a foreground process. For more details, see the **PHANTOM** command.
Example

    COMO ON LOG
    RUN END.OF.YEAR
    COMO OFF
    SPOOL $COMO END.OF.YEAR

The above command sequence turns on command output logging using a record named LOG, runs an end of year processing program, turns off logging and then sends this log file to the default printer (print unit 0).
4.37 COMPARE.FILES

The COMPARE.FILES command compares the content of two files.

Format

COMPARE.FILES {{DICT} file1} {{DICT} file2} [DETAIL]

where

file1, file2 are the names of the files to be compared. If omitted, the command prompts for the file names. Either or both file names may have the DICTIONARY prefix to reference the file's dictionary.

The COMPARE.FILES command compares the records in file1 with those in file2. If no differences are found, the value of @SYSTEM.RETURN.CODE on completion will be zero. If there are records with differences or records that appear in only one of the files, @SYSTEM.RETURN.CODE will be set to the number of failing records.

With the DETAIL option, the command will show details of the differences found.

Example

COMPARE.FILES CUSTOMERS CUSTOMERS.OLD DETAIL
Data mismatch in 10007
Field 2 differs
Records in file 1 but not in file 2:
  19403
  19766
Records in file 2 but not in file 1:
  04655
Differences in 4 records
4.38 COMPILE CONSTRAINTS

The COMPILE CONSTRAINTS command is used to compile data integrity constraints.

Format

```text
COMPILE CONSTRAINTS file.name
COMPILE CONSTRAINTS file.name, subfile.name
COMPILE CONSTRAINTS file.name,*
```

```text
COMPILE CONSTRAINTS file.name {CLEAR} {DISABLE | LOGGING}
```

where

- `file.name` is the name of the file for which data integrity constraints are to be compiled. This must correspond to a VOC F-type record.
- `subfile.name` is the name of the subfile if `file.name` references a multi-file. Use of an asterisk as the subfile name compiles the constraints into all subfiles.
- `CLEAR` causes the command to remove the compiled constraint definitions, leaving the source form in place. This qualifier can be used with all three of the file name formats.
- `DISABLE` causes validation at the point of writing to the file to be suppressed. Records that would have failed validation can be identified later using the VERIFY CONSTRAINTS command.
- `LOGGING` causes a validation error to be recorded in the error log but allows faulty data to be written.

The data integrity constraint rules are defined in field 12 of D, A or S-type records in the dictionary and must be compiled and stored in the data file before they take effect. This is done using the COMPILE CONSTRAINTS command.

For each field to which constraints are to be applied, the command shows the field number (0 for the record id), field name and constraints expression. If an error occurs, no updates are applied to the data files.

Where a field is defined from multiple dictionary records, only one may contain a constraints expression.

On completion of the command, `@SYSTEM RETURN CODE` will be zero if compilation was successful or a negative error number if it fails.

See also:
Data integrity constraints, VERIFY CONSTRAINTS
4.39 COMPILE.DICT

The **COMPILE.DICT** command (synonym **CD**) is used to compile A, C, I and S-type records in dictionaries.

**Format**

```
COMPILE.DICT {DATA} file.name {name} ... {NO.QUERY} {NO.PAGE} {BRIEF}
COMPILE.DICT ALL {NO.PAGE} {BRIEF}
COMPILE.DICT LOCAL {NO.PAGE} {BRIEF}
```

where

- `file.name` is the name of the file containing the I-types to be compiled
- `name` is the name of the record to be compiled. Multiple names may be given in a single use of the command. If omitted, all A, C, I and S-type records in the dictionary are compiled unless the default select list is active, in which case that list is used.

- **NO.QUERY** suppresses the confirmation prompt if a select list is used. The `NO.SEL.LIST.QUERY` mode of the **OPTION** command can be used to imply this option.

- **NO.PAGE** suppresses pagination of output to the screen.

- **BRIEF** suppresses all commentary during compilation except for error messages.

A, C, I and S-type records may also be compiled using **MODIFY** and are automatically compiled by all query processor commands if necessary.

The main need for the **COMPILE.DICT** command is where the expression in one dictionary item uses the value of another. Because nested expressions are handled by a compile time substitution rather than a run time call, a change to the second expression requires the dictionary item that uses it to be recompiled. The automatic compilation performed by the query processor will not detect this need. In general, it is recommended that all dictionary items are recompiled whenever a modification is made to an expression that may be used by another dictionary item.

The optional **DATA** prefix on the filename in the first syntax allows items in the data portion of the file to be compiled. This is mostly of use with the VOC or VOCLIB files.

The **COMPILE.DICT ALL** and **COMPILE LOCAL** formats provide an easy way to compile all items in multiple dictionaries. The **ALL** keyword processes the dictionaries of all files referenced by F-type VOC entries. The **LOCAL** keyword restricts this to files that do not have a directory separator in the dictionary pathname.
4.40 CONFIG

The CONFIG command reports your licence details and configuration parameters. It can also be used to modify the values of some parameters.

Format

CONFIG (LPTR)

CONFIG param new.value

The first format of the CONFIG command allows you to examine your system configuration. The report shows the licence details followed by the values of configurable parameters. The LPTR option directs the report to the default printer.

Each parameter name is be preceded by an asterisk if the parameter is set in the configuration file or a + sign if it is a private configuration parameter that has been modified in the current QM session. If neither marker is present, the parameter has its default value.

The second form of the CONFIG command can be used to set parameter param to new.value in the current process. Only private configuration parameters that are maintained on a per-process basis can be modified. To amend the value of a global configuration parameter, edit the qmconfig item in the QMSYS account directory using any text editor or the EDIT.CONFIG command.

Examples

CONFIG

Version number 3.0-6, 64-bit, ECS
Licence number 1961491396, System id LWWK-FTXT
Maximum users 50, available 43
Expiry date 31 December 2014
Licensed to Manor Developments Limited

+ CMDSTACK 40
* NUMFILES 60
* NUMLOCKS 70
  OBJECTS 0
  OBJMEM 0 kb
* SORTMEM 1024 kb
* SORTWORK c:\temp

The first section of output corresponds to the licence data entered when the system was installed or subsequently relicensed. The second section, shown abbreviated in this example, lists the values of configurable parameters.

Example

CONFIG MUSTLOCK 1

This command modifies the value of the MUSTLOCK parameter to be 1. Only the process in which the command is executed is affected.
4.41 CONFIGURE.FILE

The **CONFIGURE.FILE** command changes the configuration of a file.

**Format**

```
CONFIGURE.FILE {DICT} file.name parameters {REPORTING}
```

where

- **file.name** is the name of the file to be configured.
- **parameters** are the new settings of the file configuration parameters. The following parameters may be specified:
  - **DYNAMIC** converts file to dynamic hashed type. Ignored if file is already dynamic.
  - **DIRECTORY** converts file to directory type. Ignored if file is already a directory.
  - **DEFAULT** resets all parameters to their default values.
  - **CASE** converts the file to use case sensitive record ids.
  - **ECS {"map.name"}** sets ECS mode on file
  - **ENCODING name** sets encoding name for directory file
  - **GROUP gid** sets the group ownership of the file using either a group number or a group name (not Windows)
  - **GROUP.SIZE n** sets the group size in units of 1024 bytes. Values in the range 1 to 8 are permitted but for best performance should be a power of two.
  - **IMMEDIATE** causes an immediate file resize, if required, releasing unused primary subfile group space.
  - **LARGE.RECORD bytes** sets the large record size in bytes.
  - **MERGE.LOAD pct** sets the merge load factor for the file.
  - **MINIMUM.MODULUS n** sets the minimum modulus for the file. Any positive non-zero value may be used.
  - **MODE ddd{;sss}** sets the file permissions modes where **ddd** is desired mode settings for the directory that represents the file. This must be a
three digit octal value as used in the Linux chmod command. The optional \textit{sss} component is the mode settings to be applied to the items in this directory. If \textit{sss} is omitted, the \textit{ddd} value is used. (not Windows)

\textbf{NEXT} \textit{n} sets the next sequential record id to be used by the \textit{CREATINGSEQKEY} option of the \texttt{WRITE} statement.

\textbf{NEXT QUERY} displays the next sequential record id to be used by the \textit{CREATINGSEQKEY} option of the \texttt{WRITE} statement.

\textbf{NO.CASE} converts the file to use case insensitive record ids.

\textbf{NO.ECS} clears ECS mode on file.

\textbf{NO.MAP} suppresses mapping of characters in directory file record names that are normally restricted. This option affects only the data portion of the file and is therefore invalid when the \textit{DICT} keyword is present. Note that when this option is used the application is responsible for ensuring that record names comply with operating system file name rules. See \textit{directory files} for more information.

\textbf{NO.RESIZE} disables file resizing.

\textbf{RESIZE} enables file resizing.

\textbf{SPLIT.LOAD} \textit{pct} sets the split load factor for the file.

\textbf{USER} \textit{uid} sets the user ownership of the file using either a user number or a user name (not Windows)

\textbf{VERSION} \textit{vno} allows configuration of the file for compatibility with versions of QM prior to release 2.2-0.

Parameters which are not specified retain their existing values.

The \texttt{CONFIGURE.FILE} command adjusts the settings of one or more file parameters. Changes to the file type or group size result in immediate restructuring of the file and require exclusive access. The \texttt{REPORTING} option will show progress. Changes only affecting other \texttt{dynamic file parameters} will occur steadily as the file is updated unless the IMMEDIATE option is used.
Note that converting a file from case sensitive ids to case insensitive ids will result in loss of data if the file contains records that have ids with alternative casing of the same text.

The NO.RESIZE option disables the normal automatic split/merge operations that occur in dynamic files. The IMMEDIATE option can be used later to force the deferred splits/merges to be applied. See the description of dynamic files for more details on the use of this feature.

The resizing operations of the IMMEDIATE option are fully interruptible and can be performed while the file is in use.

Reconfiguring the VOC

The CONFIGURE.FILE command needs exclusive access to the file that is being reconfigured. Because the VOC file is permanently open, reconfiguration of this file can only be done from another account. Also, the file.name must reference an F-type VOC entry, not a Q-pointer. The process therefore becomes:
1. Log to another account (e.g. QMSYS)
2. Create an F-type VOC entry that points to the VOC of the original account
3. Do the CONFIGURE.FILE command
4. Delete the F-type VOC entry
5. Log to the original account

Examples

CONFIGURE.FILE STOCK MINIMUM.MODULUS 200 SPLIT.LOAD 75

This command changes the minimum modulus and split load percentage of the STOCK file. The actual change will take effect as the file is updated by future access.

CONFIGURE.FILE STOCK DIRECTORY

This command changes the file to be a directory file.

CONFIGURE.FILE STOCK DIRECTORY ENCODING UTF8

This is similar to the previous example but sets the file to use UTF-8 encoding.

CONFIGURE.FILE STOCK DYNAMIC ECS

This command converts the STOCK file to an ECS mode dynamic file.
The COPY command copies selected records from one file to another, or within the same file. COPYI is normally a synonym but see the discussion below.

**Format**

COPY FROM {DICT} src.file {TO {DICT} tgt.file} {src.rec,tgt.rec} {options}

where

- **src.file** is the file from which the records are to be copied.
- **tgt.file** is the file to which the records are to be copied. If omitted, records are copied within the **src.file**.
- **src.rec** is the name of the record to be copied.
- **tgt.rec** is the name of the record to which **src.rec** is to be copied. If omitted, the record is not renamed.
- **options** are taken from the following:
  
  - **ALL** Copy all records from **src.file** to **tgt.file**. This option cannot be used with named records or a select list.
  
  - **BINARY** Copy data in binary mode, suppressing translation between field marks and newlines when copying between a hashed file and a directory file in either direction. This mode is implied if both are directory files unless the **TEXT** option is used.
  
  - **DELETING** Delete the record(s) from **src.file** after copying.
  
  - **NO.QUERY** Suppresses confirmation prompt when using a select list. The NO.SEL.LIST.QUERY mode of the **OPTION** command can be used to imply this option.
  
  - **OVERWRITING** Overwrite existing record(s) of the same name in **tgt.file**. Without this option, records which already exist are not copied. This option may not be used with **UPDATING**.
  
  - **REPORTING** Display the id of each record as it is copied.
  
  - **TEXT** Copy data in text mode, translating between field marks and newlines when copying between a hashed file and a directory file in either direction. This option is also needed to override implied binary mode when
copying between directory files where the newline representation of the source and target are different (e.g. Windows to Linux).

**TO.LOWERCASE**  
Force target record ids to be lowercase.

**TO.UPPERCASE**  
Force target record ids to be uppercase.

**UPDATING**  
Only copy records if they already exist in the target file. This option may not be used with **OVERWRITING**.

Any number of source and target record pairs may be specified. If all records in the file are to be copied, the keyword **ALL** may be used in place of specified record names.

If no source records are specified and the default select list is active, this list is used to determine the records to be copied. A confirmation prompt is issued before copying commences unless the **NO.QUERY** keyword is used.

The **COPY** command does not normally overwrite existing records. The keyword **OVERWRITING** allows this operation. Thus a command of the form

```
COPY FROM src.file TO tgt.file ALL OVERWRITING
```

would only copy records that do not already exist in the target file.

Conversely, the keyword **UPDATING** copies records only if they already exist in the target file.

The **DELETING** keyword causes **COPY** to delete records from the source file after they have been successfully copied to the target file.

The **TO.LOWERCASE** and **TO.UPPERCASE** options can be useful when copying directory file items from a platform with a case insensitive file system (e.g. Windows) to a platform with a case sensitive file system (e.g. Linux), forcing the casing of the new record to be as specified.

The **REPORTING** keyword causes a message to be displayed for each record copied.

`@SYSTEM.RETURN.CODE` is returned as the number of records copied or a negative error code.

The **COPY** command can be used with data collections but if either of the source and target files is a data collections, both must be.

Users migrating from Pick style systems frequently use **ALIAS** to make **COPY** a synonym for **COPYP**, the Pick syntax copy command. QM also provides **COPYI** as a way to force use of the Information style syntax.

**Example**

```
COPY FROM NEWVOC TO VOC ALL OVERWRITING
```

This command copies all records from the NEWVOC file to the VOC, replacing existing records of the same name. This could be used, for example, if the VOC had been damaged by an accidental deletion of some standard records.

**See also:**  
**COPYP**
4.43 COPY.LIST

The **COPY.LIST** command copies a saved select list to another file or a different record in the same file. Alternatively, the list can be output to the display.

**Format**

```
COPY.LIST src.list {, tgt.list} {FROM src.file} {TO tgt.file} {options}
```

where

- **src.list** is the name of the saved select list that is to be copied. If `src.list` is given as `*`, all saved select lists in the source file are copied.

- **tgt.list** is the name to be used for the copied select list. If omitted, the list retains its original name. A `tgt.list` name cannot be specified when copying all saved lists from the source file.

- **src.file** is the name of the file holding the `src.list` to be copied. If omitted, the default saved lists file `$SAVEDLISTS` is used.

- **tgt.file** is the name of the file to receive the copied select list. If omitted, the default saved lists file `$SAVEDLISTS` is used.

- **options** may be any of the following

  - **CRT** Output the select list to the display. Neither `tgt.list` nor `tgt.file` may be specified with this option.

  - **DELETING** Delete `src.list` after copying.

  - **LPTR {n}** Output the select list to logical print unit `n`. If `n` is not specified, the default print unit is used.

  - **NO.PAGE** Used with the **CRT** option, this option suppresses the normal pause between successive pages of output.

  - **OVERWRITING** If `tgt.list` already exists in `tgt.file`, this option allows overwriting of the existing list. Without this option, a message is displayed and no copy occurs.

The **COPY.LIST** command is used to copy saved select lists. The **FROM** and **TO** options allow copying from and to files other than the default `$SAVEDLISTS` file. When the default file is used, it will be created if it does not already exist.

The **COPY.LIST** command does not affect any active select lists.

**Example**

```
COPY.LIST INVENTORY TO INVENT.LISTS OVERWRITING
```
This command copies the select list previously stored in $SAVEDLISTS as INVENTORY to a record of the same name in file INVENT.LISTS. Any existing record of the same name is overwritten.

See also:
DELETE.LIST, EDIT.LIST, GET.LIST, SAVE.LIST, SORT.LIST
4.44 COPY.LISTP

The COPY.LISTP command copies saved select list records from one file to another, or within the same file using Pick syntax.

Format

COPY.LISTP {{\{DICT\} src.file} {id.list | *} {options}}

where

* src.file is the file from which the records are to be copied. The optional DICT prefix indicates that the dictionary portion of the file is to be used. If omitted, the $SAVEDLISTS file is used.

* id.list is a list of ids of the records to be copied. If specified as an asterisk, all records in the source file are copied. If omitted, an active select list is used.

* options is a list of option codes. These must be prefixed by an open parenthesis. The available codes are:

  B Copy data in binary mode, suppressing translation between field marks and newlines when copying between a hashed file and a directory file in either direction. This mode is implied if both are directory files.

  D Deletes the source records after copying.

  I Suppresses display of record ids.

  N Suppresses pagination when displaying records on the terminal.

  O Overwrites existing records in the target file.

  P Sends the record data to a printer.

  S Suppresses field numbers with P or T.

  T Sends the record data to the terminal.

If the P or T options are used, the records identified by id.list are sent to the printer or the screen.

If neither the P nor T options are used, the command prompts for a space separated list of destination record ids. If there are more ids in id.list than in the destination list, the source id is used as the destination id for the extra items.

The destination list can begin with a file name prefixed by an open parenthesis to direct output to a different file. The name can optionally be followed by a close parenthesis.

@SYSTEM.RETURN.CODE is returned as the number of records copied or a negative error code.

The ALIAS command can be used to make COPY.LISTP the default for COPY.LIST without removing the ability for other users or software packages to access the original COPY.LIST command.

The COPY.LISTP command is effectively the same as the COPYP command except that the file name defaults to $SAVEDLISTS if omitted.

Examples

COPY.LISTP * (D
To: (COPIED.LISTS
17 record(s) copied and deleted.
This command copies all records from the $SAVEDLISTS file to the COPIED.LISTS file, deleting the originals.

COPY.LISTP COPIED.LISTS ORDERS
To: ORDERS2
1 record copied.

This command copies record ORDERS in the COPIED.LISTS file to a record named ORDERS2 in the same file.

See also:
COPY.LIST
The COPYP command copies selected records from one file to another, or within the same file, using Pick syntax.

**Format**

```
COPYP {DICT} src.file {id.list} {options}
```

where

- **src.file** is the file from which the records are to be copied. The optional DICT prefix indicates that the dictionary portion of the file is to be used.
- **id.list** is a list of ids of the records to be copied. If specified as an asterisk, all records in the source file are copied. If omitted, an active select list is used.
- **options** is a list of option codes. These must be prefixed by an open parenthesis. The available codes are:
  - A Copy data in text mode, translating between field marks and newlines when copying between a hashed file and a directory file in either direction. This option is also needed to override implied binary mode when copying between directory files where the newline representation of the source and target are different (e.g. Windows to Linux).
  - B Copy data in binary mode, suppressing translation between field marks and newlines when copying between a hashed file and a directory file in either direction. This mode is implied if both are directory files.
  - D Deletes the source records after copying.
  - I Suppresses display of record ids.
  - N Suppresses pagination when displaying records on the terminal.
  - O Overwrites existing records in the target file.
  - P Sends the record data to a printer.
  - S Suppresses field numbers with P or T.
  - T Sends the record data to the terminal.

If the P or T options are used, the records identified by id.list are sent to the printer or the screen.

If neither the P nor T options are used, the command prompts for a space separated list of destination record ids. If there are more ids in id.list than in the destination list, the source id is used as the destination id for the extra items.

The destination list can begin with a file name prefixed by an open parenthesis to direct output to a different file. The name can optionally be followed by a close parenthesis.

@SYSTEM.RETURN.CODE is returned as the number of records copied or a negative error code.

The ALIAS command can be used to make COPYP the default for COPY without removing the ability for other users or software packages to access the original COPY command.

The COPYP command can be used with data collections but if either of the source and target files is a data collections, both must be.

**Examples**
COPYP ACCOUNTS * (D
To: (SAVED.ACCOUNTS
17 record(s) copied and deleted.

This command copies all records from the ACCOUNTS file to the SAVED.ACCOUNTS file, deleting the originals.

COPYP BP PRT.INVOICE
To: PRT.INVOICE2
1 record copied.

This command copies record PRT.INVOICE in the BP file to a record named PRT.INVOICE2 in the same file.

See also:
COPY
4.46 CREATE.ACCOUNT

The CREATE.ACCOUNT command creates a new QM account.

Format

```
CREATE.ACCOUNT acc.name pathname {description} {NO.QUERY} {NO.CASE}
{MODE mmm}{sss} {USER uid} {GROUP gid}
```

where

- **acc.name** is the name to be given to the new account. This name must start with a letter and may not contain spaces. The name will be translated to uppercase and must not exceed 32 characters in length. The command will prompt for this if it is not given on the command line.

- **pathname** is the pathname of the operating system directory to hold the account. The pathname should be enclosed in quotes if it contains spaces. The directory will be created if it does not already exist but the parent directory must already exist. The command will prompt for this if it is not given on the command line.

- **description** is a text description of the account that will be displayed when listing the QM.ACCOUNTS file. It must be enclosed in quotes if it contains spaces.

- **NO.QUERY** suppresses the confirmation prompts when the pathname directory already contains a VOC file or when creating a new directory.

- **NO.CASE** creates the account with a case insensitive VOC file. This option is ignored if the VOC file is already present in the account directory.

The CREATE.ACCOUNT command creates a new account and populates the VOC file from the NEWVOC template and optional NEWVOC.MODS file stored in the QMSYS account. The new account is added to the register of accounts in the ACCOUNTS file in the QMSYS account.

Creation of accounts by specific users may be barred by the system administrator. See Application Level Security for more details.

If no **pathname** is specified on the command line, CREATE.ACCOUNT prompts for the pathname. The command will look in the VOC of the QMSYS account for an X-type record named SACCOUNT.ROOT.DIR and, if this is found, will use field 2 of this record to specify a directory name under which the account should be created by default. This default pathname can be selected by entering a null response to the pathname prompt.

The **MODE mmm** option sets file permission modes (not Windows). The **mmm** component is an octal permissions mask that will be applied to the directory that represents the account and all other items created in the account by this command. The optional **sss** component is the mode settings to be applied to the the subfiles in the newly created VOC file. If **sss** is omitted, the **ddd** value is used.

The **USER** and **GROUP** options set the owner and group for the account (not Windows). The **uid** and **gid** qualifiers may be specified either as numbers or as names. No error is reported if the user executing the command does not have sufficient access rights to apply these values. When either of these options is omitted, the user and group are inherited from the user executing the command.

Accounts may be installed anywhere within the operating system file hierarchy. It is often useful to distribute accounts across disk drives for improved load balancing. Alternatively, related information
can be clustered together in a single partition to simplify replication. It is recommended that new accounts are not created under the directories of existing accounts as this can lead to accidental deletion. For example, it is generally not a good idea to create application accounts as subdirectories of the QMSYS account.

Attempting to create an account in the root directory will ask for confirmation unless the **NO.QUERY** keyword has been used.

When QM is installed on a USB memory device, if *pathname* specifies a directory on the USB device, this will be transformed internally to a format that allows for the drive letter to be different when the stick is loaded in future.

**Example**

```plaintext
CREATE.ACCOUNT SALES D:\SALES
```

This command creates a new QM account named SALES in the D\SALES directory.

**See also:**
[DELETE.ACCOUNT](#), [UPDATE.ACCOUNT](#)
4.47 CREATE.FILE

The CREATE.FILE command is used to create a QM file.

Format

CREATE.FILE {portion} file.name {, subfile} {type} {configuration}
{USING DICT other.file} {ENCRYPT keyname} {NO.QUERY}
{MODE ddd{sss}} {USER uid} {GROUP gid} {"description"} where

- **portion** identifies the part of the file to create. This may be DATA to create just the data portion, DICT to create just the dictionary portion or omitted to create both.

- **file.name** is the name of the VOC record to be created to refer to the file. By default, the operating system pathname used for the directory representing the data file is the same as file.name and the directory for the dictionary component of a file is the same but with a .DIC suffix.

- **subfile** is the name of the subfile to be created in a multifile.

- **type** specifies the file type as DIRECTORY, DYNAMIC or COLLECTION. If omitted, a dynamic file is created by default. Dictionaries are always created as dynamic files regardless of any type argument.

- **description** is descriptive text to be included after the type code in the F-type VOC entry for the file.

The configuration options that apply to all file types are:

- **MULTIFILE** when creating the data portion of the file, causes the file to be created as a multifile even if no subfile is specified.

- **PATHNAME path** specifies the pathname of an existing operating system directory under which the file is to be created. If this is a relative pathname, it will be preserved in this form for the VOC entry of the new file.

- **IN name** As an alternative to the PATHNAME option, name specifies the name of an F-type VOC item that defines a directory file. The new file will be created within this directory. Although this option can be useful, the ability to create a new QM file inside the directory that represents a directory file should be used with caution as deleting name will also delete the operating system representation of any QM files within that directory, leaving their VOC entries in place.

- **NON.TRANSACTIONAL** The file is always to be opened in non-transactional mode. This option causes the N open mode flag to be included in field 6 of the F-type VOC entry.
The configuration options available when creating a dynamic file to specify the file’s configuration are:

**CASE**

creates a file where the record ids will be treated as case sensitive. This is the default action unless the CREATE.FILE.NO.CASE mode of the **OPTION** command is enabled.

**ECS {"map"}**

The file will be created in ECS mode and will be able to store records that include characters outside the 8-bit character set in their id or data. Files created in this mode cannot be opened by non-ECS mode versions of QM. The optional map name specifies the name of the ECS map to be used by the file to determine uppercase/lowercase relationships and the sort order for alternate key indices. If omitted, the system’s default map at the time of creating the file will be used. Note that only the map name, not the actual map, is stored in the file. Modifying the map may have undesired effects.

**MINIMUM.MODULUS n**

sets the minimum modulus for the file. Any positive non-zero value may be used. The default is 1.

**GROUP.SIZE size**

sets the group size as a multiple of 1024 bytes. This must be in the range 1 to 8 but for best performance should be a power of two. If omitted, the default group size is taken from the GRPSIZE configuration parameter.

**LARGE.RECORD bytes**

sets the large record size in bytes. The default is 80% of the group size.

**SPLIT.LOAD pct**

sets the split load factor for the file. The default is 80%.

**MERGE.LOAD pct**

sets the merge load factor for the file. The default is 50%.

**VERSION vno**

allows creation of files with internal formats compatible with releases of QM before version 2.2-0.

**NO.CASE {DATA | DICT}**

specifies case insensitivity as described below.

**NO.RESIZE**

creates the file with resizing disabled. See **CONFIGURE.FILE** and dynamic files for more information.

The configuration options available when creating a directory file are:

**ENCODING name**

specifies the default character encoding to be applied to this file.

**NO.MAP**

suppresses mapping of characters in directory file record names that are normally restricted. This option affects only the data portion of the file and is
therefore invalid when the DICT keyword is present. Note that when this option is used the application is responsible for ensuring that record names comply with operating system file name rules. See directory files for more information.

The USING DICT clause allows creation of a data file that is to share the dictionary of an existing file. The effect of this option is to copy the content of field 3 of the VOC entry for other file into field 3 of the newly created entry rather than setting up a new dictionary.

The ENCRYPT keyword enables record level data encryption and prefixes the name of the encryption key to be used. If omitted when creating a hashed file, the CREATE.FILE command looks for an X-type record named SENC in the VOC file. If this is present, field 2 specifies the name of an encryption key that will be applied automatically. To ignore this record, use ENCRYPT "". Automatic encryption does not apply to directory files.

The NO.CASE option specifies that record ids in a hashed file are to be treated as case insensitive. QM will write records preserving the casing specified by whatever performs the write. Reads will locate records regardless of casing. If the file name has been prefixed with DATA or DICT to create only the specified portion of the file, the NO.CASE keyword applies to that file portion. If the file name prefix is not present so that both portions will be created, the default action is for NO.CASE to apply to the data portion only. The optional DATA and DICT qualifiers to the NO.CASE option, both of which may be given, apply case insensitivity to the specified part of the file. The CREATE.FILE.NO.CASE and CREATE.DICT.NO.CASE modes of the OPTION command can be used to make case insensitivity the default.

Case sensitivity of record ids in directory files is dependent on the operating system.

The NO.QUERY option suppresses any confirmation prompts associated with the requested action.

The MODE ddd{;sss} option sets file permission modes (not Windows). The ddd component is an octal permissions mask that will be applied to the directory that represents the file. The sss component, applicable only to dynamic files, is an octal permissions mask that will be applied to the subfiles within the directory. If omitted, it defaults to ddd.

The USER and GROUP options set the owner and group for the file (not Windows). The uid and gid qualifiers may be specified either as numbers or as names. No error is reported if the user executing the command does not have sufficient access rights to apply these values. When either of these options is omitted, the behaviour of CREATE.FILE depends on the setting of the INHERIT.OWNERSHIP mode of the OPTION command. If this option is enabled, the user and group are inherited from the ownership of the account directory. If this option is not enabled, the user and group are inherited from the user executing the command.

To provide maximum flexibility, QM imposes no restrictions on the file name except that it may not contain the mark \ldots system file names are automatically transformed using the same translations as described for directory file record ids.

By default, the directories created at the operating system level to represent the QM file and its dictionary have their names converted to uppercase. The KEEP.FILENAME.CASE mode of the OPTION command can be used to suppress this conversion.

Data Collection Files
A data collection file is a special case of a dynamic hashed file in which the records represent data collections. All of the configuration options relating to dynamic files also apply to this file type.

**Multifiles**

A multifile is a collection of data files that share a common dictionary. Commands and application software refer to an individual subfile within the multifile by using a name that consists of the file name and subfile name separated by a comma. References to the file without specifying the subfile use the default subfile which has the same name as the file. For example a multifile named SALES might have subfiles named NORTH and SOUTH referenced as SALES,NORTH and SALES,SOUTH. Referring simply to SALES without a subfile name is taken as meaning SALES,SALES which does not have to exist. The subfile name will be treated as case insensitive if the VOC (and hence the file name) is case insensitive.

When creating a multifile element, the default action of CREATE.FILE is to create a subdirectory named file.name under the account and create the element within this directory as subfile. An alternative location can be specified using the PATHNAME or IN parameter. If neither of these parameters is present and the INHERIT.MFILE.PATH mode of the OPTION command is active, adding a new subfile to an existing file creates the subfile in the same directory as the default subfile.

The CREATE.FILE command can convert an existing simple file into a multifile. The existing data becomes a subfile with the same name as the file.

**File Permission Modes**

The MODE option can be used to set the file access permissions for the newly created file and its dictionary. This option is ignored on Windows. The ddd element is the octal value for the permission flags to be applied to the directory that represents the QM file. The first digit is the permissions for the owner of the file, the second digit is the permissions for other users in the group to which the file is assigned, and the third digit is permissions for all other users. Each digit is formed by adding 4 (write), 2 (read) and 1 (execute).

The sss element is the octal value for the permission flags to be applied to subfiles in the directory when creating a hashed file. If omitted, this defaults to the same value as ddd.

If the MODE option is not used, the file permissions are taken from the current operating system umask value.

**Examples**

```
CREATE.FILE STOCK MINIMUM.MODULUS 150 GROUP.SIZE 4
```

This statement creates a dynamic file named STOCK with minimum modulus of 150 and group size 4.

```
CREATE.FILE INVOICES MODE 775:660
```

This statement creates a dynamic file named INVOICES. The directory representing this file will have permission flags of 775 and the subfiles in the directory will have permission flags of 660.

```
CREATE.FILE SALES ENCRYPT SALESKEY
```
This statement creates a dynamic file named SALES and applies record level data encryption using the SALESKEY key.

```
CREATE.FILE DATA PROGRAMS DIRECTORY PATHNAME D:\APPS
```

This statement creates the data portion of a directory file named PROGRAMS. The full pathname for this directory file is specified as D:\APPS\PROGRAMS rather than using the default location.

```
CREATE.FILE ACCOUNTS,NORTH
```

This statement creates a multifile component named NORTH within the ACCOUNTS file.

See also:
CONFIGURE.FILE, DELETE.FILE, LISTF, LISTFL, LISTFR, data encryption, CREATE.KEY, ENCRYPT.FILE
4.48 CREATE.INDEX

The CREATE.INDEX command creates an alternate key index.

Format

CREATE.INDEX filename field(s) {options}

where

filename is the name of the file for which the index is to be built.

field(s) is one or more field names for which indices are to be created.

options are any combination of the following:

DELETING
ENCRYPT
NO.CASE
NO.NULLS
PATHNAME index.path

The CREATE.INDEX command creates the file structures to hold an alternate key index. The index must subsequently be populated using BUILD.INDEX before it can be used.

The field(s) referenced in the command must correspond to C, D, E, I, A or S-type dictionary items. The dictionary items can be deleted once the index has been constructed as all details of the indexed field are stored in the index file but this is not recommended. The value to be indexed must not exceed 255 characters. Values longer than this will not be included in the index.

Indices constructed on I or C-type dictionary items or on A or S-type items that use correlative expressions should be such that they always produce the same result when executed for the same data record. Examples of possibly invalid I-type expressions would be those that use the date or time and those that use the TRANS() function to access other files.

The DELETING option causes CREATE.INDEX to delete any previous version of the same index. It is useful when, for example, the index must be recreated because an I-type expression that derives the indexed value has been modified.

The NO.NULLS option specifies that no entry is to be added to the index for records where the indexed field is null.

The NO.CASE option specifies that the index is to be built using case insensitive key values. The internal sort order of the index is based on the uppercase form of the data being indexed. A case insensitive index can be used with case insensitive comparison operators in the query processor but not with case sensitive operations because of the sort order difference. Conversely, a case sensitive index can be used with case sensitive comparison operators in the query processor but not with case insensitive operations.

The ENCRYPT option creates an encrypted index. This is only possible when the data file being indexed uses record level encryption. There is a significant performance penalty in using encrypted indices with files that are frequently updated.

Normally, the indices are stored as subfiles in the directory that represents the data file. The PATHNAME option allows the indices to be stored in an alternative location. This might be useful,
for example, to balance loads across multiple disks or to exclude indices from backups as they can always be recreated.

All indices for a single data file must be stored together. The **PATHNAME** option can be used when creating the first index and specifies the pathname of a new directory that will be created at the same time as the index. If this option is included when creating subsequent indices the *index.path* must be the same as for the first index. It is suggested that the pathname should be based on the data file name for ease of recognition.

If the **PATHNAME** option is not included, the **CREATE.INDEX** command looks for an X-type VOC record named $INDEX.PATH in which the second field contains the pathname of a directory under which indices are to be created by default. If found, a subdirectory with the same name as the final element of the data file pathname is created under this location. For example, when creating an index for a file with pathname /hd1/sales/invoices, a $INDEX.PATH record that contains

```
1: X
2: /hd2/indices
```

would place the indices in /hd2/indices/invoices.

Use of **PATHNAME DEFAULT** will ignore the $INDEX.PATH record, placing the indices in the same directory as the data file elements.

Index subfiles can be moved using the operating system level *qmidx* utility.

**Data Encryption**

Alternate key indices may be applied to files that use record level data encryption and the index itself can also be encrypted as described above. Files using field level encryption cannot have indices on encrypted fields. Also, indices constructed from calculated values such as I-types that use encrypted fields will fail if the record is updated by a user that does not have access to the relevant encryption key.

**Example**

```
CREATE.INDEX ORDERS DATE
BUILD.INDEX ORDERS DATE
```

The above commands create and build an index on the DATE field of the ORDERS file.

**See also:**

**BUILD.INDEX, DELETE.INDEX, DISABLE.INDEX, LIST.INDEX, MAKE.INDEX, REBUILD.ALL.INDICES**
4.49 CREATE.KEY, CREATE.SECURE.KEY

The CREATE.KEY command creates a data encryption key. The CREATE.SECURE.KEY command is similar but creates a password protected key. These commands can only be executed by users with administrator rights in the QMSYS account.

Format

CREATE.KEY {keyname {algorithm {keystring}}}
CREATE.SECURE.KEY {keyname {algorithm {keystring {password}}}}

where

keyname is the name for the new encryption key.
algorithm is the encryption algorithm to be associated with the key.
keystring is the encryption key string.
password is the password to be used to protect the key.

The command prompts for items not supplied on the command line. For best security, the keystring and password should be entered via a prompt so that they will not appear on the command stack or in log files.

The CREATE.KEY command creates a new entry in the key vault defining the encryption algorithm and actual key string to be used. If the key vault does not already exist, this command will create it, prompting for the master key to be used to encrypt the key vault. If the key vault does exist, the user will be asked to enter the master key unless it has already been entered during this session.

The keyname may be any sequence of up to 64 letters, digits, periods and hyphens. It is case insensitive.

The algorithm may be any of AES128, AES192 and AES256. The algorithm name is case insensitive. These algorithms use a fixed initialisation vector that is the same for every use of the encryption algorithm. There are also extended forms of the algorithms named XAES128, XAES192 and XAES256 that use a random initialisation vector which is included in the encrypted form of the data. This gives greater security but increases the size of the encrypted data.

The keystring is up to 64 characters, is case sensitive and can contain any character. For best security, the length of the keystring should be close to the actual length needed by the selected algorithm. This is 16, 24 or 32 characters for the 128, 192 and 256 bit algorithms respectively. The CREATE.KEY command will automatically transform the supplied key to the required length if necessary.

Once a key has been defined, it may be referenced in commands that set up encryption without needing to enter the master key. The keyname does not need to be treated as a secure item. The keystring, on the other hand, must not be disclosed. It is strongly recommended that a copy of the keystring is maintained securely off-site in case it is ever necessary to rebuild the key vault.

Use of CREATE.SECURE.KEY applies password protection to the key such that the password must be supplied using the ENABLE.KEY command or corresponding ENABLE.KEY QMBasic statement in a QM session before the key can be used.
The **CREATE.KEY** and **CREATE SECURE.KEY** commands automatically grant access to the key to the user that created it. Other users can be granted access using the **GRANT.KEY** command.

**Example**

```plaintext
CREATE.KEY CARDNO AES256
```

The above command creates a 256 bit encryption key named CARDNO. The actual encryption string will be entered in response to a prompt.

**See also:**
- Data encryption
- CHANGE.KEY.PASSWORD
- CREATE.FILE
- DELETE.KEY
- DISABLE.KEY
- DISABLE.KEY (QMBasic)
- ENABLE.KEY
- ENABLE.KEY (QMBasic)
- ENCRYPT.FILE
- GRANT.KEY
- LIST.KEYS
- RESET.MASTER.KEY
- REVOKE.KEY
- SET.ENCRIPTION.KEY.NAME
- UNLOCK.KEY.VAULT
4.50 CREATE.USER

The CREATE.USER command creates a new user name in the register of users for security checks.

Format

```
CREATE.USER {username {account}}
```

where

- `username` is the name of the user to be created. User names may be up to 32 character in length. On Windows systems the name is case insensitive. If omitted, a prompt is displayed for the user name.

- `account` is the name of the account to be entered when this user logs in except QMClient processes. If not specified, an account name prompt will be issued when the user logs in. By default, the forced account also does not apply to phantoms but this can be altered using ADMIN.USER.

The new user will not have administrator rights and all options controlling user rights will be disabled. See the ADMIN.USER command for a more powerful method of managing user names.

On Windows USB and in user mode installations on Linux/Unix, the new QM user is created with no password. Either the PASSWORD command should be used to apply a password or the user should be encouraged to set a password on first login.

This command does not affect the underlying operating system user name database. If QM’s security system is enabled (see the SECURITY command), users attempting to enter QM who do not appear in the user register will be rejected.

See also: ADMIN.USER, DELETE.USER, LIST.USERS, PASSWORD, SECURITY, Application Level Security.
The **CT** (Copy to Terminal) command displays the content of record(s) from a file.

**Format**

```plaintext
CT {DICT} filename {record ... | *} {options}
```

where

- **filename** is the name of the file to be processed. The **DICT** keyword indicates that the dictionary portion of `filename` is to be used.
- **record** is the name of the record to be displayed. Multiple record names may be given in a single command. If the default select list (list 0) is active, this list is used as the source of record names. If the default select list (list 0) is active, this list is used as the source of record names. Specifying a record name of an asterisk (*) displays all records in the file. If no record name is given and the default select list is active, this list will be used to determine which records are reported.
- **options** are chosen from the following:
  - **BINARY** Display the record as a binary data item.
  - **HEX** Display the data in each field in hexadecimal format.
  - **ID.SUP** Suppress display of file and record name.
  - **LPTR n** Redirects the output to printer `n`. If `n` is omitted, the default printer is used.
  - **NO.QUERY** When using a select list, the confirmation prompt is omitted. The NO.SEL.LIST.QUERY mode of the **OPTION** command can be used to imply this option.
  - **NUM.SUP** Suppress display of line numbers.

The **CT** command displays the specified records from `file`. Each record is preceded by the file and record names.

When using the default select list as the source of record ids to be processed, a confirmation prompt is issued prior to commencing the display. This can be suppressed using the **NO.QUERY** option.

By default, the report shows each line (field) of the record on a separate line, prefixed with the line number. Lines that are wider than the output device are wrapped to the next line. If `filename` is a data collection file, the record is shown in JSON format.

The **HEX** option, produces a report in which the data is displayed in hexadecimal form, two hexadecimal digits per character with no intervening spaces.

The **BINARY** option treats the record as binary data in which field marks are simply part of the data. The record is shown in both hexadecimal and character format, 16 bytes per line. Non-printing characters are displayed as dots (.) except for the field mark, value mark and subvalue mark which as shown as ^, | and \. Each line is prefixed by the byte offset (from zero) of the first byte on the line.
On ECS mode systems, if the record contains characters outside the 8-bit set, the behaviour of the \texttt{HEX} and \texttt{BINARY} options is different. In this case, each character is shown as a four digit hexadecimal value, 8 characters per line for BINARY mode. The record id, if displayed, will be followed by "(ECS)".

**Example**

```
SELECT VOC WITH F1 LIKE X...
CT VOC
```

This command sequence would display each X type record from the VOC.

```
CT READERS 2
READERS 2
  1: Cartwright, D
  2: 7 Spring GroveyNottingham
  3: 1-ly3-l
```

The above command displays the record with id 2 from the READERS file. The $y$ in the final line is a terminal dependent representation of the value mark character.

See also: \texttt{DUMP}
4.52 CWD

The CWD command changes the current working directory.

Format

CWD {pathname}

where

pathname is the pathname of the desired new working directory. If omitted, the CWD command returns to the directory of the current QM account.

The CWD command allows an application to temporarily switch to an alternative working directory. It may be of use when using the SH command or the QMBasic OS.EXECUTE statement to perform operating system commands for which the working directory is must be set correctly.

It is important to revert back to the account directory as soon as possible because there is an assumption in many commands and user applications that the working directory is that of the current account.

Example

PA
CWD /home/appdev
SH gcc *.c
CWD

The above example paragraph uses CWD to switch to the /home/appdev directory and runs the gcc compiler. It then returns to the account directory.
4.53 DATA

The DATA command supplies data to be used by an associated verb or QMBasic program which would normally take input from the keyboard. It may only be used in paragraphs.

Format

\texttt{DATA \{text\}}

where

\texttt{text} is the data to be used by the verb or program.

The DATA command must immediately follow the verb to which it is to apply. Multiple DATA commands may be used to supply data to be processed consecutively by the associated verb or program. Any intervening blank lines or comments in a sequence of DATA commands will be ignored except for processing of inline prompts.

When the verb or program executes an INPUT statement, the data from the DATA command(s) will be used. If all stored data has been used, keyboard input proceeds as normal.

Data stored by the DATA command or the QMBasic DATA statement is cleared on return to the command prompt. Thus unused data where, for example, a program terminates at an error, will not be carried forward to a later command. The CLEAR.DATA command can be used to clear the data queue within a paragraph.

The DATA command cannot be used to provide text for inline prompts.

Example

\begin{verbatim}
PA
  * <<History>>
LOOP
    IF <<A,Record name>> = "" THEN GO DONE
    ED BP <<Record name>>
    DATA I * <<History>>
    DATA FI
  REPEAT
DONE:
\end{verbatim}

This paragraph inserts a history comment at the top of QMBasic programs. The editor commands to insert the text are provided using DATA commands. Note how the history text, which is only required once as it is common to all files edited, is obtained first using an inline prompt in a comment. The names of the records to be edited are then obtained in a loop which is terminated when a null name is entered.

See also:
QMBasic DATA statement
4.54 DATE

The DATE command displays the current date and time or translates a date between internal and external format.

Format

DATE {date}

DATE INTERNAL

where

date is an internal or external format date.

If no date is specified, the date and time are reported in the form

Tuesday, March 8, 1994 10:30 AM

This format is fixed, regardless of use of the DATE FORMAT command.

See TIME for an alternative format date and time report.

If date is specified, the converted form of this date (internal to external or vice versa) is reported.

The DATE INTERNAL form shows the internal day number for the current date.

Examples

DATE
Tuesday, February 15, 2000 00:12:50 PM

DATE 10012
Tuesday, May 30, 1995

DATE 1 Oct 99
11597

DATE INTERNAL
14447
The **DATE.FORMAT** command selects the date format to be used by default or displays this setting.

**Format**

**DATE.FORMAT OFF**

**DATE.FORMAT { ON }**

**DATE.FORMAT DISPLAY**

**DATE.FORMAT INFORM**

**DATE.FORMAT conv.code**

The **DATE.FORMAT** command controls the behaviour of date conversion operations that do not include a specific display format. Any conversion that has a fixed form such as use of the **DATE** or **TIME** commands is not affected. The initial state of the European date format setting is taken from the **EURODATE** configuration parameter, defaulting to off if this parameter is not present.

**DATE.FORMAT OFF** selects American date format (month, day, year) as the default for date conversions.

**DATE.FORMAT ON** or **DATE.FORMAT** with no qualifying information selects European date format (day, month, year) as the default for date conversions.

**DATE.FORMAT DISPLAY** displays the current setting of the date format mode. If a non-default conversion code has been set, this is also displayed. **@SYSTEM.RETURN.CODE** is set to 0 for American date format or 1 for European date format.

**DATE.FORMAT INFORM** sets **@SYSTEM.RETURN.CODE** as described above but does not display the date setting.

**DATE.FORMAT conv.code** sets the default conversion code that will be used for date conversions that specify a code of D with no qualifying options. The **conv.code** must be a valid date conversion code. Specifying **conv.code** as D reverts to the standard default setting.

**Examples**

**DATE.FORMAT ON**

**DATE.FORMAT DISPLAY**

 European date format is on

The above commands set European date format and then confirm this selection.
4.56 DEBUG

The DEBUG command enters the QMBasic program debugger.

Format

```
DEBUG {file.name} record.name {LPTR} {NO.PAGE}
```

where

- `file.name` is the name of the directory file holding the program to be run. If omitted, this defaults to BP. The .OUT suffix for the output file is supplied automatically.

- `record.name` is the name of the compiled program.

- `LPTR` causes output to logical print unit 0 to be directed to the printer. This is identical in effect to a PRINTER ON statement being performed within the program.

- `NO.PAGE` suppresses pagination of output to the terminal.

The DEBUG command enables detailed tracing of the operation of a QMBasic program to aid development and maintenance.
4.57 DECATALOG

The DECATALOG command removes an entry from the local catalogue and deletes the object code.

Format

```
DECATALOG {filename {progname...}}
```

where

- `filename` is the name of the file holding the program(s) to be deleted. If this is not specified, the command prompts for it. The .OUT suffix for the object code filename is added automatically.
- `progname` is the name of the program(s) to be deleted.

The DECATALOG command deletes the VOC entry for a locally catalogued program and also deletes the object code.

If the `progname` is not specified, the DECATALOG command looks for the default select list and, if active, uses this to determine the programs to be deleted. If the select list is not active, the command prompts for the program name. Specifying the program name as an asterisk deletes all locally catalogued programs from `filename`.

The DECATALOG command assumes that the program source code record name is the same as the catalogue entry name. If this is not the case, the DELETE.CATALOGUE command should be used and the object code deleted separately.

See also: BASIC, CATALOGUE, DELETE.CATALOGUE, MAP
4.58 DECRIPT.FILE

The `DECRIPT.FILE` command removes data encryption for specific fields or the entire record.

Format

```
DECRIPT.FILE filename
DECRIPT.FILE filename field ...
DECRIPT.FILE filename ALL
```

where

- `filename` is the name of the file in which encryption is to be cancelled.
- `field` is the name or field number of the field for which encryption is to be cancelled.

The first form of the `DECRIPT.FILE` command removes record level data encryption.

The second form of the `DECRIPT.FILE` command removes field level data encryption for the specified field. Multiple fields may be referenced in a single command.

The third form of the `DECRIPT.FILE` command removes field level data encryption for all fields.

If the file contains data records when this command is used, the file is processed to remove the encryption. A system failure or other process abort during this update will leave the file in a partially encrypted state and hence render it unusable. **Always back up a file before using this command if the file contains data.**

For security reasons, the user executing this command must have access to all data fields that will be affected. The command also prompts for the master key and is therefore usually restricted to use by the security administrator.

**Examples**

```
DECRIPT.FILE CUSTOMERS CCARD
```

The above command removes data encryption from the CCARD field of the CUSTOMERS file.

```
DECRIPT.FILE CUSTOMERS
```

The above command removes record level encryption from the CUSTOMERS.

**See also:**

Data encryption, CHANGE.KEY.PASSWORD, CREATE.FILE, CREATE.KEY, CREATE.SECURE.KEY, DECRYPT.FILE, DELETE.KEY, DISABLE.KEY, DISABLE.KEY (QMBasic), ENABLE.KEY, ENCRYPT.FILE, ENABLE.KEY (QMBasic), GRANT.KEY, LIST.KEYS, RESET.MASTER.KEY, REVOKE.KEY, SET.ENCRYPTION.KEY.NAME, UNLOCK.KEY.VAULT
4.59 DELETE

The DELETE command deletes specified records from a file.

Format

```
DELETE {DICT} file.name {record.name ...}
DELETE {DICT} file.name {NO.QUERY}     To use a select list
DELETE {DICT} file.name ALL {NO.QUERY}
```

where

- **DICT** indicates that the records are to be deleted from the dictionary portion of the named file.
- **file.name** is the name of the file holding the records to be deleted.
- **record.name** is the name of the record to be deleted. Multiple record names may be specified in a single DELETE command.
- **ALL** causes all records to be deleted.
- **NO.QUERY** suppresses the confirmation prompt when using a select list. The NO.SEL.LIST.QUERY mode of the OPTION command can be used to imply this option.

If no record names are specified and the default select list is active, this list is used to determine the names of the records to be deleted. A confirmation prompt is issued before deletion commences unless the NO.QUERY option is used.

If a record to be deleted is locked by another user, a warning message is displayed and the DELETE command continues without deleting the record.

The DELETE command reports the number of records deleted on completion.

@SYSTEM.RETURN.CODE is returned as the number of records deleted unless the delete fails in which case it contains the error code.

Examples

```
SELECT STOCK WITH PART.NO < 10000
DELETE STOCK
71 records deleted
```

This example selects all records from the STOCK file with PART.NO less than 10000 and deletes them.

```
DELETE PARTS.FILE A12745 A84543 C36590
Record A84543 not found
2 records deleted
```
This example attempts to delete three specific records from PARTS.FILE. One of the records does not exist.
4.60 DELETE.ACCOUNT

The DELETE.ACCOUNT command deletes a QM account.

Format

    DELETE.ACCOUNT acc.name {FORCE}

where

    acc.name

is the name of the account to be deleted. This name must be registered in
the ACCOUNTS file in the QMSYS account (visible to all accounts as
QM.ACCOUNTS).

The DELETE.ACCOUNT command deletes the named account and its entry in the accounts
register. The user will be prompted to confirm whether the account directory is to be deleted.

Deletion of accounts by specific users may be barred by the system administrator. See Application
Level Security for more details.

Before deleting the account, QM checks whether any files in the account are referenced from other
accounts and, if so, displays a list of these files. It also checks whether there are other accounts that
are stored as subdirectories of the account that is to be deleted. The FORCE option can be used to
suppress these checks.

Example

    DELETE.ACCOUNT SALES

This command deletes the account named SALES.

See also:

CREATE.ACCOUNT, UPDATE.ACCOUNT
4.61 DELETE.CATALOGUE

TheDELETE.CATALOGUE command (synonym DELETE.CATALOG) removes an entry from the system catalogue.

Format

DELETE.CATALOGUE {name...} {GLOBAL | PRIVATE | LOCAL}

where

name is a list of the catalogue call names of the programs or subroutines to be deleted. If the default select list is active, this will be used to identify the catalogue entries to be deleted and the name should be omitted.

GLOBAL indicates that a globally catalogued version of this subroutine is to be removed.

PRIVATE indicates that a private catalogued version of this subroutine is to be removed.

LOCAL indicates that a locally catalogued version of this subroutine is to be removed.

If none of the catalogue mode options is present, theDELETE.CATALOGUE command deletes entries from the private catalogue unless name has a prefix character that identifies a globally catalogued item. Use of the CATALOGUE.LOCAL setting of theOPTION command changes the default behaviour to delete the program from the local catalogue.

The private catalogue is normally a subdirectory, cat, under the account directory but can be moved by creating an X-type VOC entry named $PRIVATE.CATALOGUE in which field 2 contains the pathname of the alternative private catalogue directory. This only takes effect when QM is re-entered or on use of theLOGTO command. This feature is particularly useful where two or more accounts are to share a common private catalogue. The US spelling, $PRIVATE.CATALOG, may be used instead, if both are present, the British spelling takes priority.

See also:
BASIC, CATALOGUE, DECATALOG, MAP
4.62 DELETE.COMMON

The DELETE.COMMON command deletes named common blocks.

Format

```
DELETE.COMMON name
DELETE.COMMON ALL
DELETE.COMMON SAVING name1 {name2...}
```

where

```
name
```

is the name of the common block to be deleted. The keyword ALL causes all named common blocks to be deleted.

The DELETE.COMMON command deletes the named common block. It is particularly useful when debugging programs.

Common blocks can only be deleted if there is no active program referencing them. When the ALL keyword is used, blocks that cannot be deleted are ignored and no error is reported. When deleting a specific common block, a non-fatal error occurs if the block is in use.

Use of the SAVING keyword deletes all common blocks except for those named in the command.

Examples

```
DELETE.COMMON COM1
```

This command deletes common block COM1.

```
DELETE.COMMON ALL
```

This command deletes all named common blocks.

See also:

LIST.COMMON
**4.63 DELETE.DEMO**

The **DELETE.DEMO** command deletes the demonstration files.

**Format**

```
DELETE.DEMO
```

The **DELETE.DEMO** command deletes the demonstration files created by the **SETUP.DEMO** command. The command includes safety checks to ensure that it does not delete files of the same name that do not appear to belong to the demonstration database.

See also:

**SETUP.DEMO**
4.64 DELETE.FILE

The **DELETE.FILE** command deletes one or both portions of a file.

**Format**

```
DELETE.FILE {DATA | DICT} file.name {, subfile} {options}
```

Where

- **file.name** is the VOC name of the file to be deleted. The DATA prefix may be used to delete only the data portion of the file. The DICT prefix may be used to delete only the dictionary portion of the file.

- **subfile** is the subfile to be deleted from a multifile. If omitted and file.name refers to a multifile, the entire multifile will be deleted. Use of a subfile name implies use of the DATA keyword, leaving the dictionary in place.

- **options** are chosen from the following:
  - **FORCE** is used to delete files with non-default names.
  - **NO.QUERY** suppresses the confirmation prompt when using a select list or when deleting all elements of a multi-file. The NO.SEL.LIST.QUERY mode of the **OPTION** command can be used to imply this option.

If no file name is specified and the default select list is active, the **DELETE.FILE** command will use this list to determine the files to be deleted.

Deleting the data portion of a file deletes the associated operating system directory and clears field 2 of the VOC record describing the file. Deleting the dictionary portion of a file deletes the directory representing the dictionary and clears field 3 of the VOC record.

If the **DELETE.FILE** command results in a VOC record with fields 2 and 3 both null, the VOC record is also deleted. Thus deleting both portions of a file, the data portion of a file which had no dictionary or the dictionary portion of a file which had no data portion would also delete the VOC record.

Where the operating system name of the file recorded in the VOC entry is not the default name for the file (file.name for the data portion, file.name.DIC for the dictionary portion), the **DELETE.FILE** command prompts for confirmation unless the **FORCE** option is used. This traps accidental deletion of files which are remote to the account or for which file.name is not the primary VOC reference.

**Example**

```
DELETE.FILE DICT INVENTORY
```

This command deletes the dictionary part of the file named INVENTORY.

**See also:**  
**CREATE.FILE**, **LISTF**, **LISTFL**, **LISTFR**
4.65 DELETE.INDEX

The DELETE.INDEX command deletes an alternate key index.

Format

DELETE.INDEX file.name field(s) to delete specific indices
DELETE.INDEX file.name ALL to delete all indices

where

file.name is the VOC name of the file holding the indices.
field(s) are the names of indexed fields to be deleted. The casing of the name used in the command must match that of the actual index name.

The DELETE.INDEX command deletes the named indices. Once an index has been deleted, any queries against the named field(s) may require processing of the entire file to locate records.

Example

DELETE.INDEX ORDERS DATE

This command deletes the index on the DATE field of the ORDERS file.

See also:
BUILD.INDEX, CREATE.INDEX, DISABLE.INDEX, LIST.INDEX, MAKE.INDEX, REBUILD.ALL.INDICES
4.66 **DELETE.KEY**

The **DELETE.KEY** command deletes a data encryption key. This command can only be executed by users with administrator rights in the QMSYS account.

**Format**

```
DELETE.KEY {keyname}
```

where

`keyname` is the name of the encryption key to be deleted. This is case insensitive.

The command prompts for the key name if it is not supplied on the command line.

The **DELETE.KEY** command deletes an entry in the key vault defining the encryption algorithm and actual key string. The user will be asked to enter the master key unless it has already been entered during this session.

Any data in data files encrypted using this key will become inaccessible.

**Example**

```
DELETE.KEY CARDNO
```

The above command deletes the encryption key named CARDNO.

**See also:**
- Data encryption
- **CHANGE.KEY.PASSWORD**
- **CREATE.FILE**
- **CREATE.KEY**
- **CREATE.SECURE.KEY**
- **DISABLE.KEY**
- **DISABLE.KEY (QMBasic)**
- **ENABLE.KEY**
- **ENABLE.KEY (QMBasic)**
- **ENCRYPT.FILE**
- **GRANT.KEY**
- **LIST.KEYS**
- **RESET.MASTER.KEY**
- **REVOKE.KEY**
- **SET.ENCRYPTION.KEY.NAME**
- **UNLOCK.KEY.VAULT**
4.67 DELETE.LIST

The **DELETE.LIST** command deletes a previously saved select list.

**Format**

```plaintext
DELETE.LIST \{file.name\} list.name
```

where

- `file.name` is the name of the file from which the list is to be deleted. This defaults to `$_SAVEDLISTS` if omitted.
- `list.name` is the name of the record in `file.name` that holds the saved select list. This defaults to `"$temp.n"` if omitted, where `n` is the QM user number.

The **DELETE.LIST** command deletes a previously saved select list from the `$SAVEDLISTS` or alternative file. It has no effect on any active select lists.

**Example**

```plaintext
DELETE.LIST INVENTORY
Deleted saved list 'INVENTORY'.
```

This example deletes a select list saved as INVENTORY.

**See also:**

`COPY.LIST, EDIT.LIST, GET.LIST, SAVE.LIST, SORT.LIST`
4.68 **DELETE.SERVER**

The **DELETE.SERVER** command removes a QMNet or VFS server from QM’s list of public server definitions. The **DELETE.PRIVATE.SERVER** command deletes a private server definition that is local to the QM session.

**Format**

```
DELETE.SERVER name
DELETE.PRIVATE.SERVER name
```

where

`name` is the name previously used in a **SET.SERVER**, **SET.VFS.SERVER** or **SET.PRIVATE.SERVER** command to define the server name that is now to be deleted.

QMNet allows an application to access QM data files on other servers as though they were local file, with complete support for concurrency control via file and record locks. Servers are defined by the system administrator using the **SET.SERVER** or **SET.PRIVATE.SERVER** command and may subsequently be deleted with the **DELETE.SERVER** or **DELETE.PRIVATE.SERVER** command.

Virtual File Systems (VFS) servers created with the **SET.VFS.SERVER** command can also be deleted with **DELETE.SERVER**.

The **DELETE.SERVER** command to delete a public server definition requires administrator rights within QM and can only be executed from the QMSYS account.

Any QMNet or VFS connections currently open to the server that is being deleted will not be affected.

**Example**

```
DELETE.SERVER ADMIN
```

This example will delete the server known within QM as ADMIN.

**See also:**
The [virtual file system](#), QMNet, [ADMIN.SERVER](#), [LIST.SERVERS](#), [SET.SERVER](#)
4.69 DELETE.USER

The **DELETE.USER** command deletes a user name from the register of users for security checks.

**Format**

```
DELETE.USER {username}
```

where

`username` is the name of the user to be deleted. If omitted, a prompt is displayed for the user name.

The named user is deleted from the user name register. It is possible to delete a user who is logged in.

See the **ADMIN.USER** command for a more powerful method of managing user names.

On Windows NT onwards and on all non-Windows platforms, this command does not affect the underlying operating system user name database, however, users who do not appear in the register will not be able to enter QM if the internal security system is enabled.

See also: **ADMIN.USER**, **CREATE.USER**, **LIST.USERS**, **PASSWORD**, **SECURITY**, Application Level Security
4.70 **DISABLE.INDEX**

The **DISABLE.INDEX** command disables update and use of one or more alternate key indices.

**Format**

```
DISABLE.INDEX filename field(s)
DISABLE.INDEX filename ALL
```

where

- `filename` is the name of the file for which the index is to be built.
- `field(s)` is one or more field names for which indices have been created.

The **DISABLE.INDEX** command disables updates and use of the index data for the named field(s). The **ALL** keyword can be used to disable all indices that have been created for the file.

To re-enable the index, use the **BUILD.INDEX** command.

This command is of use when importing a large volume of data into a file that uses indices or when performing batch processing that may update a large proportion of the records stored in the file. The performance gain of not applying effectively randomly sequenced index updates may be significantly greater than the cost of using **BUILD.INDEX** to rebuild the index after the data import or processing is complete because this command can optimise the sequence in which it constructs the index.

**See also:**

**BUILD.INDEX, CREATE.INDEX, DELETE.INDEX, LIST.INDEX, MAKE.INDEX, REBUILD.ALL.INDICES**
4.71 DISABLE.KEY

The **DISABLE.KEY** command is used to disable access to a secure encryption key previously enabled with **ENABLE.KEY**.

**Format**

```
DISABLE.KEY {keyname}
```

where

- **keyname** is the name of the encryption key to be disabled.

The command prompts for the *keyname* if it is not included on the command line.

An encryption key created using the **CREATE.SECURE.KEY** command can only be accessed in a QM session after it has been enabled by use of the **ENABLE.KEY** command or the corresponding **ENABLE.KEY** QMBasic statement. Access to the key can later be denied by use of **DISABLE.KEY**.

**See also:** Data encryption, **CHANGE.KEY.PASSWORD**, **CREATE.FILE**, **CREATE.KEY**, **CREATE.SECURE.KEY**, **DELETE.KEY**, **DISABLE.KEY (QMBasic)**, **ENABLE.KEY**, **ENABLE.KEY (QMBasic)**, **ENCRYPT.FILE**, **GRANT.KEY**, **LIST.KEYS**, **RESET.MASTER.KEY**, **REVOKE.KEY**, **SET.ENCRYPTION.KEY.NAME**, **UNLOCK.KEY.VAULT**
4.72 DISABLE.PUBLISHING

The DISABLE.PUBLISHING command disables publishing to all replication subscribers.

Format

DISABLE.PUBLISHING

The DISABLE.PUBLISHING command, present only in the QMSYS account and restricted to users with administrator rights, suspends recording of updates to published files in the replication log area. Any updates applied to published files while publishing is disabled will not be replicated on the subscriber systems. Account replication is also disabled by this command.

The command is intended for use in synchronising the publisher and subscriber files either while setting up replication or when recovering from a failure of the publisher system.

Publishing can be re-enabled with the ENABLE.PUBLISHING command. It is automatically re-enabled if QM is restarted.

See also: Replication, ENABLE.PUBLISHING, PUBLISH, PUBLISH.ACCOUNT, SUBSCRIBE
4.73 DISPLAY

The DISPLAY command displays text at the user's terminal.

Format

\[
\text{DISPLAY } text \{;\} \\
\text{DISPLAY @(col{,row})text } \{;\}
\]

The optional colon at the end of the line suppresses the normal newline after the \textit{text} is displayed.

The second format positions the cursor to the given column and row (both numbered from zero) before displaying the \textit{text}. If the \textit{row} component and the preceding comma are omitted, the cursor is positioned to the specified column on the current line. There must be no spaces in the cursor position element. Any spaces following the close bracket are treated as part of the \textit{text}. The QMBasic @(1) function variants with a negative value for the first (or only) argument are also supported.

The \texttt{SYSTEM.RETURN.CODE} variable is not affected by this command.

Example

A paragraph containing

\[
\text{DISPLAY Hello :} \\
\text{DISPLAY world}
\]

would display

Hello world
4.74 DO.REMOTE

The DO.REMOTE command executes a command on a remote QM system.

Format

```
DO.REMOTE server account command
```

where

- `server` is the remote server name.
- `account` is the QM account on the remote server in which the command is to be executed.
- `command` is the command to be executed.

The DO.REMOTE command uses the QMClient API to execute a command on a remote server. Output from the command will be displayed on the local system. If the command interacts with the user, input data may be entered from the keyboard. Because QMClient command execution does not support use of the QMBasic KEYIN() and related functions, commands that interact at a character level such as the SED editor cannot be executed with this command. Because QMClient buffers all output from a command until the command terminates or it requests input, execution of commands that produce large volumes of output data is likely to fail. The DO.REMOTE command is intended for simple commands only.

The `server` name provided on the command line must correspond to a server previously defined using `SET.SERVER` or `SET.PRIVATE.SERVER`. The QMCLIENT configuration parameter on the remote system but be set to zero to allow command execution.

The `@SYSTEM.RETURN.CODE` variable will be returned as set by the executed command.

Example

```
DO.REMOTE QMSERVER SALES PHANTOM RUN OVERNIGHT
```

The above example starts a phantom in the SALES account on the server named QMSERVER to execute a program named OVERNIGHT.
4.75 DUMP

The DUMP command displays the content of record(s) from a file in hexadecimal and character format.

Format

DUMP {DICT} filename {record ... | *} {options}

where

filename is the name of the file to be processed. The DICT keyword indicates that the dictionary portion of filename is to be used.

record is the name of the record to be displayed. Multiple record names may be given in a single command. If the default select list (list 0) is active, this list is used as the source of record names. If the default select list (list 0) is active, this list is used as the source of record names. Specifying a record name of an asterisk (*) displays all records in the file. If no record name is given and the default select list is active, this list will be used to determine which records are reported.

options are chosen from the following:

- COL.HDG Show column headings on each page.
- LPTR n Redirects the output to printer n. If n is omitted, the default printer is used.
- NO.QUERY When using a select list, the confirmation prompt is omitted.

The DUMP command displays the specified records from file. Each record is preceded by the file and record names.

The record is treated as binary data in which field marks are simply part of the data. It is shown in both hexadecimal and character format, 16 bytes per line. Non-printing characters are displayed as dots (.) except for the field mark, value mark and subvalue mark which as shown as ^, ] and \. Each line is prefixed by the byte offset (from zero) of the first byte on the line.

When using the default select list as the source of record ids to be processed, a confirmation prompt is issued prior to commencing the display. This can be suppressed using the NO.QUERY option.

The DUMP command is equivalent to use of CT with the BINARY option.

Example

DUMP READERS 2
READERS 2
00000000: 43 61 72 74 77 72 69 67 68 74 2C 20 44 FE 37 | Cartwright, D^7
00000010: 53 70 72 69 6E 67 20 47 72 6F 76 65 FD 4E 6F | Spring Grove]Not
00000020: 74 69 6E 67 68 61 6D FE 31 2D 31 FD 33 2D 31 | tingham^1-1]3-1

The above command dumps the record with id 2 from the READERS file.

See also: CT
4.76 ECHO

The ECHO command suspends or enables echoing of keyboard input.

Format

- **ECHO OFF**: Disable echoing of keyboard input
- **ECHO ON**: Enable echoing of keyboard input
- **ECHO**: Toggle echo status

The ECHO command allows temporary suspension of keyboard input echo. Echoing is automatically resumed in the event of an abort.

The [@SYSTEM.RETURN.CODE](#) variable is not affected by this command.
4.77 ECS.MAP

The ECS.MAP command specifies the character map to be used by the QM session.

Format

ECS.MAP \{ name \}

where

name is the name of the map to be used.

This command, available only on an ECS mode system, allows a QM session to change the character map used to control sort ordering, upper/lower case pairing and character types.

When a QM session is started (login, phantom creation, etc), the first map defined in the QM configuration parameters is selected as the default map for the session or, if no map was specified in the configuration parameters, a system supplied default map is used. The ECS.MAP command can be used to select an alternative map within a QM process, for example, to meet local language conventions for character sort order.

If the map identified in the ECS.MAP command was specified in the configuration parameters to be loaded into shared memory when QM was started, the command links to this map data. If the map is not in shared memory, it will be loaded into the process private memory. Where possible, shared memory maps should be used if they apply to multiple processes to minimise memory usage.

Use of the ECS.MAP command without a map name displays the name of the currently selected map.

Note that alternate key indices are stored internally in sorted order based on a map specified when the file was created. The ECS.MAP command sets the name of the base map and does not affect index operations. The map associated with the file will be selected when an index related operation commences and the system will revert to the base map when the operation is complete. Actions within the dictionary expression of the index will be performed in the context of the file’s map.

See also:

Extended Character Set Support, SET.ECS.MAP()
4.78 ED

The ED command enters the QM line editor. The synonym EDIT can be used.

Format

ED {DICT} file.name {record.id | CREATING.SEQKEY} {NO.QUERY}

where

DICT indicates that records from the dictionary portion of the file are to be edited.

file.name is the name of the file holding the record(s) to be edited.

record.id is the name of the record to be edited. Multiple record ids may be given in which case each record is edited in turn. Use of CREATING.SEQKEY instead of a record id automatically generates the next sequential key for the file.

NO.QUERY suppresses the confirmation prompt if a select list is used. The NO.SEL.LIST.QUERY mode of the OPTION command can be used to imply this option.

If no record.id is specified and the default select list is active, this list is used to identify the records to be edited. If no record.id is specified and the default select list is not active, the ED command prompts for the record.id.

A record.id of * either on the command line or as the first record.id entered in response to the prompt will cause ED to select all records of the file and edit each in turn.

The editor maintains an update lock on the record that is being edited.

When editing a compiled dictionary item (C-type, I-type, or A/S type with a correlative), ED removes the compiled code thus forcing recompilation when the modified item is first referenced in a query.

Overview

The editor takes its commands from the keyboard or the DATA queue. Each command line contains one editor command. Commands are retained in a stack similar to the command processor stack and can be repeated without complete retyping. Commands that take arguments specifying their exact function can be repeated by entering just the command name.

ED normally rejects commands and input text that contain non-printing characters. Where a non-printing character is to be entered, it can be typed as ^nnn where nnn is the decimal value of the character, or ^Xnnnn where nnnn is the hexadecimal value of the character. Alternatively, the NPRINT command can be used to enable entry of non-printing characters.

The editor operates in two modes; edit and input. In edit mode, commands affect the current line (field). In input mode, new data is entered into the record. The editor numbers lines from one and the line number is displayed as a four digit number followed by a colon whenever lines are displayed or during input. The editor command prompt is four hyphens followed by a colon.
Positioning Commands

The commands listed below alter the position of the current line. In addition, entering a blank line at the command prompt advances the current position by one line, displaying the newly selected line.

\( n \)

Entering a number at the command prompt positions the current line to line \( n \).

\(+n\)

Moves the current line position forward by \( n \) lines.

\(-n\)

Moves the current line position backward by \( n \) lines.

\( B \)

The \( B \) (bottom) command moves to the last line of the record.

\( F{n} \ {string} \)

The \( F \) (find) command moves forward to the next line containing \( string \) starting at column position \( n \) (from one). If \( n \) is omitted, \( string \) must occur at the start of the line.

The \( string \) must be preceded by a single space or a delimiter chosen from

\[ ! " # $ % & ( ) * + , . / : = @ [ \] \_ ` ' { } | \]

All characters after the delimiter will be treated as part of the string to be located. The string is case sensitive by default but the \( CASE \) command can be used to select case insensitive searches. See the \( W \) command for a description of wildcards.

If \( string \) is omitted, the string used by the most recent \( F \) command is used. If no \( F \) command has been executed, the editor moves forward by one line.

\( G{n} \)

The \( Gn \) (go to) command moves to line \( n \). This is identical to the \( n \) command described above.

\( G< \)

The \( G< \) command moves to the first line of the currently defined block.

\( G> \)

The \( G> \) command moves to the last line of the currently defined block.

\( L{n} \ {string} \)

The \( L \) (locate) command moves forward to the next line containing \( string \) which must be preceded by a single space or a delimiter chosen from

\[ ! " # $ % & ( ) * + , . / : = @ [ \] \_ ` ' { } | \]

All characters after the delimiter will be treated as part of the string to be located. The string is case sensitive by default but the \( CASE \) command can be used to select case insensitive searches. See the \( W \) command for a description of wildcards.

The optional line count, \( n \), limits the search to \( n \) lines from the current position. If \( n \) is present, all occurrences of \( string \) in the region to be searched are displayed and the current position is left at the end of the search region.

If \( string \) is omitted, the string used by the most recent \( L \) command is used. If no \( L \) command has been executed, the editor moves forward by one line.

\( M \ {pattern} \)
The **M** (match) command moves forward to the next line matching *pattern*. The *pattern* argument is any valid pattern as used for the query processor **LIKE** operator. There must be a space before *pattern*.

If *pattern* is omitted, the string used by the most recent **M** command is used. If no **M** command has been executed, the editor moves forward by one line.

**POn**
The **POn** (position) command moves to line *n*. This is identical to the **n** command described above.

**T**
The **T** (top) command moves to before line 1. There is no current line after this action.

### Displaying Text

**Ln**
The **Ln** (list) command displays *n* lines, moving the current line forward to the final displayed line. It is similar to the **P** command except that *n* must be included. Omitting *n* results in execution of a locate command as described above.

**P[*n]***
The **P** (print) command displays *n* lines starting at the current line, moving the current line forward to the final displayed line. The value of *n* defaults to 23 on first use of the **P** command and to the value of *n* for the most recently executed **P** command thereafter. There must be no space between **P** and *n*.

**PL{<-}n**
The **PL** (print lines) command displays *n* lines relative to the current line position. Negative values of *n* print lines before the current line. The value of *n* defaults to 23 on first use of the **PL** command and to the value of *n* for the most recently executed **PL** command thereafter. There must be no space between **PL** and *n*. The current line position is not changed by this command.

**PP[*n]***
The **PP** (print position) command displays *n* lines surrounding the current line position. The value of *n* defaults to 23 on first use of the **PP** command and to the value of *n* for the most recently executed **PP** command thereafter. There must be no space between **PP** and *n*.

### Inserting Text

**I {text}**
The **I** (insert) command inserts *text* after the current line, making this the current line. There must be a single space before *text*. Any additional spaces are treated as part of the inserted text. To insert a blank line type **I** followed by a single space.

If the **I** command is entered with no *text* and no space after the **I**, the editor enters input mode. It will prompt for successive lines until a blank line is entered at which point it returns to edit mode. Entering a line containing just a single space inserts a blank line.

**IB {text}**
The **IB** (insert before) command is similar to the **I** command described above except that *text* is inserted before the current line.

**IX[c]**
The IX command is similar to the I command described above except that blank lines in the inserted data are not treated as a special case. Input is terminated by entry of a line containing just the single character c. If c is omitted it defaults to a backslash.

LOAD {(filename) record.id}
The LOAD command inserts part or all of the specified record into the record being edited after the current line position. If filename is omitted, it defaults to the file associated with the current record. After the operation is complete, the current line is the first line of the newly inserted text.

The editor prompts for the start and end line numbers to be inserted. These default to the first and last lines respectively.

Deleting Lines

D{n}
The Dn (delete) command deletes n lines starting at the current line position. If n is not specified, only the current line is deleted. The line after the last line deleted becomes current.

Dn/string/
Deletes all lines containing string within the next n lines starting at the current line position.

DE{n}
Same as Dn described above.

Commands that Edit the Current Line

A {string}
The A (append) command appends string to the current line. A single space must separate string from the command. Any further spaces are treated as part of the inserted text.

If string is omitted, the most recent A command is repeated.

B string
The B (break) command splits the current line into two after string. The string argument must be present and is preceded by a single space. Any additional spaces are treated as part of string.

C/old.string/new.string/{n} {G} {B}
The C (change) command changes old.string to new.string in the current line. The delimiter around the strings may be any of

! " # $ % & ( ) * + , - . / : = @ [ ] \ _ ` ' { } |

The optional n component specifies that n lines starting at the current line are to be changed.

G causes all occurrences of old.string to be replaced. Without G only the first occurrence on the line is changed.

B applies the change to the currently defined block rather than the current line. B and n may not be used together.

The n, G, and B, qualifiers can be placed before the first string delimiter as an alternative to the syntax shown above.

Entering C with no strings repeats the last substitution.
Searches for *old.string* are case sensitive by default. See the **CASE** command for a way to select case insensitive searches. See the **W** command for a description of wildcards.

**CAT {string}**
The **CAT** command concatenates the current line, *string* and the following line to form a single line. Omitting *string* merges the lines with no intervening characters. There must be a single space between the command and the *string*. Any additional spaces are treated as part of *string*.

**DUP {n}**
The **DUP** command duplicates the current line *n* times. The value of *n* defaults to one. The first line added by **DUP** becomes the current line.

**R/old.string/new.string/{n} {G} {B}**
Identical to **C** described above.

**R {text}**
The **R** (replace) command replaces the current line with the specified *text*. There must be a single space before *text*. Any additional spaces are treated as part of the replacement text. To replace a line by a blank line, type **R** with no *text*. The space may be omitted in this case.

### Working with Multivalued Data

**EV**
The **EV** (edit values) command enters a mode where each value of the current line becomes a line of its own in a new editable item. To edit subvalues, use the **EV** command when already in EV mode. Used with a dictionary I-type entry, the **EV** command breaks compound I-types into separate lines to simplify editing.

**QV**
Exits from EV mode and returns to the previous edit text, discarding any changes made while in EV mode.

**SV**
Exits from EV mode and returns to the previous edit text, saving any changes made while in EV mode.

### Block Edit Commands

Blocks are defined by two pointers; the start and end line. Block operations enable the entire block to be deleted, moved or copied.

<
Sets the current line to be the start line of the block. When used at the top of the record (above the first line), the < command clears the block pointers.

>
Sets the current line to be the end line of the block.

<>
Sets the block to be just the current line.

**BLOCK**
Toggles block verification mode. When enabled, **COPY**, **DROP** and **MOVE** commands cause a prompt for confirmation prior to performing the operation. Block verification mode is enabled by default.
COPY
Copies the currently defined block to immediately after the current line position without affecting the original block.

DROP
Deletes the currently defined block.

MOVE
Copies the currently defined block to immediately after the current line position and deletes the original block.

PB
The PB (print block) command displays the currently defined block.

File Handling Commands and Leaving the Editor

DELETE
Prompts for confirmation and then deletes the entire record from the file.

FD

After the record has been deleted, the editor either terminates, continues with the next record from a select list or prompts for a new record id depending on the way in which it was entered.

The confirmation prompt can be suppressed using the ED.NO.QUERY.FD mode of the OPTION command.

FILE {{DICT} {filename} record.id}
If no arguments are included, the FILE command (which may be abbreviated to FI) writes the record being edited back to its file.

If record.id is specified, the modified record is saved under the new name. A confirmation prompt will be issued if a record of this name already exists.

If both filename and record.id are given, the record is saved to the specified file and record. Again, a confirmation prompt will be issued if a record of this name already exists.

After the record has been saved, the editor either terminates, continues with the next record from a select list or prompts for a new record id depending on the way in which it was entered.

Two extended forms of the FI command are available for use when editing QMBasic programs:

FIB {{filename} record.id} Files the record and then runs the QMBasic compiler.

FIBR {{filename} record.id} Files the record, runs the QMBasic compiler and, if the compilation is successful, runs the compiled program.

N
When using a select list, the N command moves to the next record in the list. A confirmation prompt is issued if there are unsaved changes.

QUIT
The QUIT command (which may be abbreviated to Q) and its synonym EX terminates editing of the current record. A confirmation prompt is issued if there are unsaved changes.

If a select list is in use, the editor will move on to the next record. Use the X command described below to terminate the entire edit in this case.
SAVE \{ [DICT] \{filename\} record.id \}

If no arguments are included, the SAVE command writes the record being edited back to its file.

If \textit{record.id} is specified, the modified record is saved under the new name. A confirmation prompt will be issued if a record of this name already exists.

If both \textit{filename} and \textit{record.id} are given, the record is saved to the specified file and record. Again, a confirmation prompt will be issued if a record of this name already exists.

Unlike the FILE command, editing continues after saving the record. The SAVE command does not change the names associated with the record being edited. A subsequent SAVE or FILE with the file and record names omitted will use the original names, not those of an intermediate SAVE command.

UNLOAD \{ [DICT] \{filename\} record.id \}

The UNLOAD command saves part or all of the record being edited into the named file and record. If \textit{filename} is omitted, it defaults to the file associated with the current record.

The editor prompts for the start and end line numbers to be saved. These default to the first and last lines respectively.

\textbf{X}

The \textbf{X} command aborts an edit when a select list is in use without saving any changes made to the record. A confirmation prompt is issued if there are unsaved changes. Any further entries in the select list are discarded and the editor terminates.

\textbf{Miscellaneous Commands}

\textbf{?}

The \textbf{?} command displays status information about the editor and the record being edited. This includes
- The file name and record id
- The current line number
- The item size (lines and bytes)
- Block start and end line positions
- The command, if any, which will be reverted by OOPS.
- Non-printing character expansion mode status (\textasciitilde{)}
- Non-printing character entry mode status (NPRINT)
- Block verify mode status (BLOCK)
- The setting of search case sensitivity (CASE)

\textasciitilde

Toggles non-printing character expansion mode. When this mode is enabled, non-printing characters are displayed as \textasciitilde{nnn} where \textit{nnn} is the decimal character number. On ECS mode systems, characters outside the 8 bit set will be displayed as \textasciitilde{Xnnnn} where \textit{nnnn} is the hexadecimal character number.

\textbf{CASE OFF}

Sets case insensitive mode for the \textbf{C}, \textbf{F} and \textbf{L} commands.

\textbf{CASE ON}

Sets case sensitive mode for the \textbf{C}, \textbf{F} and \textbf{L} commands. This is the default mode of the editor.

\textbf{COL}

Displays a column number ruler to aid alignment of inserted text
HELP \{topic\}
The HELP command displays a short description associated with the command identified by topic. If topic is omitted, this command enters the full help system. If topic is present but not recognised, ED tries to find a help page on this topic from the full help system.

NPRINT
Toggles non-printing character entry mode. When this mode is enabled, non-printing characters may be included in commands and input text. When disabled, non-printing characters are rejected but may still be entered using the ^nnn notation where nnn is the decimal character number. This command is not available on ECS mode systems.

OOPS
The OOPS command undoes the most recent function that modified the record. It cannot be used to forget positioning functions.

STAMP
The STAMP command inserts a single comment line below the current line indicating the account name, user name, date and time of the modification.

SPOOL \{lines\}
The SPOOL command prints a copy of the record on the default printer. If changes have been made but not yet written to the file, the printed version includes these changes. The optional lines qualifier specifies the number of lines to be printed starting at the current line. If this is omitted, the entire record is printed.

W\{char\}
Specifies a wildcard character that may be used in the C and R text replacement commands or the F and L search commands. Use of char within the search string of these commands will match against any single character. The wildcard character may not be a letter or the caret symbol (^). Use of the W command with no char qualifier turns off the wildcard.

XEQ \{command\}
The XEQ command executes the specified command which may be any command valid at the command prompt. The command may include any of the following items to substitute text into the command:
- @FILE The file name
- @ID The record id
- @LINE The text of the current line
- @FM A field mark (to separate multiple commands)
- @VM A value mark
- @SM A subvalue mark

Pre-Stored Edit Commands
Frequently used sequences of editor commands may be saved in a file and subsequently executed by entering just one command. Pre-stored command sequences can also include loops to repeat a series of commands.

Command sequences are saved using the .S command. This has several different syntaxes:

- .S item Save the most recent command
- .Sn item Save command n
- .S item n,m Save commands n to m
- .S item m Save commands n to m
In each case, *item* may be given as either a record id to be created in the default $ED file or as a file name and record id separated by a space. The values *n* and *m* may be given in either order. The commands are always saved in the same sequence as they appear on the editor command stack. The first line in the saved item has a type code of E as its first character followed by text describing when it was created.

The $ED file will be created automatically first time that the .S command is used to save commands in this file.

Multi-line inserts cannot be repeated from a saved command sequence, will cause the .S command to fail and, if edited into the pre-stored sequence manually, will be cause a warning message to be displayed when the command sequence is executed.

Alternatively, a user can create a $ED pre-stored item by using the editor:

```
ED $ED item.name
```

The first field of the item must contain E as its first character. Subsequent fields each contain one editor instruction in the sequence that the user requires the operations to be performed.

A saved command sequence is executed by a command of the form

```
.X item
```

where *item* is either a record id in $ED or a file name and record id separated by a space. Unlike other command stack operations, .X *item* command is added to the command stack so that .X *n* can be used to repeat the pre-stored sequence.

Other useful command stack extensions for use with stored edit commands are:

```
.D item Delete the specified item
.L item List the item. If the record id is given as an asterisk, a list of stored edit sequences in the file is displayed.
.R item Recall a previously saved set of commands to the stack.
```

A stored edit sequence may include the PAUSE command. Execution stops and the user may decide whether to continue by entering .XR or to terminate the sequence by entering .XK. Other editing commands may be executed before either of these responses is entered.

The LOOP command can be used to repeat a series of steps in the stored commands. The format is

```
LOOP lineno count
```

The edit continues with the command on *lineno*. The LOOP will be jump back to the specified line *count* times before dropping through to the next command. Both *lineno* and *count* default to 1 if omitted.

Note that a sequence such as

```
001 1 xyz
002 LOOP 1 3
```

executes the insert command four times because the LOOP jumps back three times. Note also that although the first edit command is on line 2 of the pre-stored sequence, the first line (type code and description) is discarded by the editor and the edit commands are numbered from one when setting *lineno*.

A loop may validly include use of the FI command to file the record. When processing records from a select list, the pre-stored sequence continues execution from one record to the next.
**Editor Command Stack**

The following commands manipulate the editor command stack. Unless otherwise stated, the stack position argument, \( n \), defaults to one.

- \( \text{.A}[n]\) *string*
  
  Append *string* to entry \( n \) of the editor command stack.

- \( \text{.C}[n]/\text{old.string/new.string}/\) Change *old.string* to *new.string* in line \( n \) of the editor command stack.

- \( \text{.D}[n] \)
  
  Delete line \( n \) of the editor command stack.

- \( \text{.I}[n]\) *string*
  
  Insert *string* as entry \( n \) of the editor command stack.

- \( \text{.L}[n] \)
  
  List \( n \) lines of the editor command stack. The value of \( n \) defaults to nine.

- \( \text{.R}[n] \)
  
  Recall line \( n \) of the editor command stack to the top of the stack.

- \( \text{.X}[n] \)
  
  Execute line \( n \) of the editor command stack.

See also **ED Pre-Stored Commands** above for additional command stack features.

**External Commands**

The editor includes a facility whereby users can implement additional commands using catalogued subroutines. External commands are identified by use of pattern matching, comparing the command line with patterns specified in the \$ED.OPTIONS record (see below) in a case insensitive manner. Commands defined in this way can override standard ED commands, allowing alternative or extended functionality to be added very easily.

When an external command is recognised, the editor calls a user written subroutine with two arguments, INFO and ERROR. The INFO argument is an 8 element dimensioned array containing:

- \( \text{INFO}(1) = \) full file name, including possible DICT prefix
- \( \text{INFO}(2) = \) record id
- \( \text{INFO}(3) = \) data being edited
- \( \text{INFO}(4) = \) current line number, zero if at the top
- \( \text{INFO}(5) = \) block start line number, zero if not set
- \( \text{INFO}(6) = \) block end line number, zero if not set
- \( \text{INFO}(7) = \) command line in casing as typed
- \( \text{INFO}(8) = \) edit value depth (0 = normal, 1 = editing values, 2 = editing subvalues)

The ERROR argument is used to return an error status to the editor. It should be set to True if processing of the command fails.

If ERROR is not set True, the editor will examine the INFO array on return from the subroutine, updating its own status variables to correspond with changes made by the subroutine to the data in \( \text{INFO}(3) \) or the position information in INFO elements 4, 5 and 6. Positional data is subject to validation.
Setting Default Modes

On entry, ED looks for an X-type VOC record named $ED.OPTIONS and, if this exists, examines fields 2 onwards of this record for options that set the default modes for the editor. These may be:

- **BLOCK {ON | OFF}**
  - Turn on/off prompting for confirmation on COPY, DROP and MOVE. Default is ON.

- **CASE {ON | OFF}**
  - Turn on/off case sensitivity for searches (C, F, L). Default is ON.

- **EXTERNAL name cmd**
  - Defines catalogued subroutine name as the handler for external command cmd where cmd is a pattern match template or simple constant value.

- **FIT.SCREEN {ON | OFF}**
  - If on, the default display size of the P, PL and PP commands is chosen to fit the screen size. Default is OFF.

- **NO.EV.POSITION**
  - Suppress display of higher level line numbers when in edit values mode.

- **NPRINT {ON | OFF}**
  - Turn on/off acceptance of non-printing characters on input. Default is OFF. This option is ignored on ECS mode systems.

Unrecognised options or qualifiers are ignored.
4.79 EDIT.CONFIG

The EDIT.CONFIG command edits the QM configuration parameters.

Format

EDIT.CONFIG

The EDIT.CONFIG command provides an easy way to modify the QM configuration parameters. It is available only in the QMSYS account and requires the user to have administrator rights.

A typical screen display from this command on a system with many configuration parameters defined is as shown below. Parameters that are omitted from this file take default values.

```
ERRLOG=20
EXCLREM=0
ILERULE=7
FIXUSERS=21,10
FSYNC=0
GDI=0
GRPSIZE=1
INTPREC=13
JNLDIR=\qmjnl
JNLMODE=3
LPTRHIGH=66
LPTRWIDTH=80
MAXIDLEN=127
MUSTLOCK=0
NETFILES=2
NUMFILES=500

Additive value controlling extended file syntax.
1 = Allow account:file,
2 = Allow server:account:file,
4 = Allow PATH: prefix.
```

The upper portion of this display shows the configuration data with the cursor positioned on the currently selected parameter. The lower portion shows descriptive text for the selected parameter. A message may also appear if the parameter value fails to meet validation rules.

To move through the parameters, use any of the following keys:

- Ctrl-N: Cursor down
- Ctrl-P: Cursor up
- Ctrl-Z: Page up
- Ctrl-A: Home
- Ctrl-B: Cursor left

The amend a line, simply type new data or use any of the standard editing keys:
<table>
<thead>
<tr>
<th>Key Combination</th>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl-D</td>
<td>Delete</td>
<td>Delete character under cursor</td>
</tr>
<tr>
<td>Ctrl-E</td>
<td>End</td>
<td>Position the cursor at the end of the input data</td>
</tr>
<tr>
<td>Ctrl-F</td>
<td>Cursor right</td>
<td>Move the cursor right one character</td>
</tr>
<tr>
<td>Ctrl-H</td>
<td>Backspace</td>
<td>Backspace one character</td>
</tr>
<tr>
<td>Ctrl-K</td>
<td></td>
<td>Delete all characters after the cursor</td>
</tr>
</tbody>
</table>

Insert

Toggle insert/overlay mode. When overlay mode is enabled, data entered by the user replaces the character under the cursor rather than being inserted before this character.

Clearing a line deletes the parameter.

To insert a new parameter, press the Return key on the line under which the insertion is to occur.

To terminate the editor, optionally saving changes, press the Esc key.
4.80  EDIT.LIST

The **EDIT.LIST** command invokes the **ED** line editor to edit a saved select list in the $SAVEDLISTS file.

**Format**

```
EDIT.LIST list.name
```

where

```
list.name
```

is the name of the saved select list to be edited. If omitted, a prompt is output.

The **EDIT.LIST** command enters **ED** to edit the named saved select list. All editing functions are available and care should be taken to ensure that the record remains a valid select list.

**See also:**

**COPY.LIST, DELETE.LIST, GET.LIST, SAVE.LIST, SORT.LIST**
4.81 EDIT.MAP

The EDIT.MAP command modifies a character map.

**Format**

```
EDIT.MAP name
```

where

*name* is the name of the map to be modified.

This command performs a closely related function on ECS and non-ECS systems but the latter has reduced capability.

**EDIT.MAP on ECS Mode Systems**

This command allows an application developer to modify an ECS character map. The standard BASE map cannot be modified.

If the named map does not exist, the EDIT.MAP command will prompt to confirm that a new map is to be created and then ask for the name of the map that is to be used as the starting point, defaulting to the BASE map.

The format of data displayed by this command is as below:

```
Codept Char A N G S Z Uppercase Lowercase Sort Trans...
>U+004C L Y . Y ... U+004C L U+006C l 004C 00 00 ...
U+004D M Y . Y ... U+004D M U+006D m 004D 00 00 ...
U+004E N Y . Y ... U+004E N U+006E n 004E 00 00 ...
U+004F O Y . Y ... U+004F O U+006F o 004F 00 00 ...
U+0050 P Y . Y ... U+0050 P U+0070 p 0050 00 00 ...
U+0051 Q Y . Y ... U+0051 Q U+0071 q 0051 00 00 ...
U+0052 R Y . Y ... U+0052 R U+0072 r 0052 00 00 ...
U+0053 S Y . Y ... U+0053 S U+0073 s 0053 00 00 ...
U+0054 T Y . Y ... U+0054 T U+0074 t 0054 00 00 ...
...etc...
```

**EDIT.MAP on Non-ECS Mode Systems**

This command allows an application developer to create or modify a character map that can be used to replace the default mapping.

If the named map does not exist, the EDIT.MAP command will prompt to confirm that a new map is to be created and then ask for the name of the map that is to be used as the starting point.

The format of data displayed by this command is as below:

```
Codept Char A N G S Z Uppercase Lowercase Sort
>U+4C L Y . Y ... U+4C L U+6C l 4C
U+4D M Y . Y ... U+4D M U+6D m 4D
U+4E N Y . Y ... U+4E N U+6E n 4E
U+4F O Y . Y ... U+4F O U+6F o 4F
U+50 P Y . Y ... U+50 P U+70 p 50
U+51 Q Y . Y ... U+51 Q U+71 q 51
```
Using EDIT.MAP (All Systems)

The columns of this display are:

<table>
<thead>
<tr>
<th>Codept</th>
<th>The code point value for the character described by this line. This is shown as a hexadecimal value prefixed with &quot;U+&quot;.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Char</td>
<td>The actual character. This will only be displayed correctly if the terminal device supports the character.</td>
</tr>
<tr>
<td>A</td>
<td>Set to Y if this character is alphabetic.</td>
</tr>
<tr>
<td>N</td>
<td>Contains the numeric value for characters that represent digits.</td>
</tr>
<tr>
<td>G</td>
<td>Set to Y if this character is a graphic (printing) character.</td>
</tr>
<tr>
<td>S</td>
<td>Set to Y if this character is whitespace (space, tab, etc).</td>
</tr>
<tr>
<td>W</td>
<td>Set to Y if this is a double width character (ECS mode systems only).</td>
</tr>
<tr>
<td>Z</td>
<td>Set to Y if this character has the user defined attribute set. See IS.USER.CHAR().</td>
</tr>
<tr>
<td>Uppercase</td>
<td>The code point value and character that should be returned by the QMBasic UPCASE() function.</td>
</tr>
<tr>
<td>Lowercase</td>
<td>The code point value and character that should be returned by the QMBasic DOWNCASE() function.</td>
</tr>
<tr>
<td>Sort</td>
<td>The sort weight of this character as a hexadecimal value.</td>
</tr>
<tr>
<td>Trans</td>
<td>Transliteration character(s) for 8-bit representation of data (ECS mode systems only).</td>
</tr>
</tbody>
</table>

The currently selected character is shown by a > marker at the left of the line.

There is a command prompt line under the displayed data. The commands and special keys available are:

- **CLEAR.ALL x**: Clear character type attribute $x$ from all characters
- **EXPORT path**: Export map to $path$
- **HELP**: Display help text
- **IMPORT path**: Import map changes from $path$
- **QUIT or Q**: Exist from the character map editor.
- **SAVE**: Save changes. Users of the map being edited will see the changes immediately.
- **SEQ F4**: Toggle sort order. The characters are shown initially in code point value order. This command toggles an alternative display in which the characters are shown in sort order.
- **SORT**: Amend sort order (see below).
- **U+nnnn**: Position on the character with the given code point value. On non-ECS systems, this is two digits.
- **blank line**: Edit the attributes of the selected character (see below).
- **Cursor down**: Select the next character on the page.
Cursor up  Select the previous character on the page.
Page down  Move to the next page.
Page up  Move to the previous page.

**Editing Character Attributes**

The character attributes of the selected character can be edited by entering a blank line at the command prompt to highlight the character. Then use any of the following to amend the attributes:

- **A** Toggle alphabetic attribute.
- **0 - 9** Set digit attribute with the given numeric value assigned to the character.
- **N** Clear numeric attribute.
- **G** Toggle graphic attribute.
- **L** Set lowercase character pairing. The L key should be followed by the character to be assigned as the lower case equivalent of the selected character.
- **S** Toggle space attribute.
- **T** Set transliteration character (ECS mode systems only). The T key should be followed by one or two 8-bit character(s) to be used for transliteration and the Return key. Return alone removes transliteration.
- **U** Set uppercase character pairing. The U key should be followed by the character to be assigned as the upper case equivalent of the selected character.
- **W** Toggle wide character attribute (ECS mode systems only).
- **|Z** Toggle user defined character attribute (ECS mode systems only).

When all attributes that are to be changed have been amended, press the return key to exit from this mode.

For operations that require entry of a character (e.g. case pairing), this may be entered by use of the escape key followed by its four (ECS) or two (non-ECS) digit hexadecimal codepoint value. This is particularly useful when a character is required that is not present on the keyboard in use.

**Amending the Sort Order**

The sort order is amended by entering SORT at the ```EDIT.MAP``` command prompt. Then enter a series of characters in the desired sort order. The first character in this list acts as an index point in the sort sequence. The remaining characters will be moved from their previous positions to immediately follow the index point.

**Importing Map Changes**

The IMPORT command reads a text file from the given pathname and applies changes to the map. Each line of this file represents changes to a single codepoint in the form

```
0054  A=1,  LC=0074
```

where the first item is the hexadecimal codepoint value of the entry to be modified followed by at least one space and a series of comma separated items that modify character attributes or case pairings. Items not specified retain their existing values. The items allowed are:

- **A=n** Set or clear alphabetic status (n = 0 to clear, 1 to set).
Blank lines and lines beginning with a # symbol are ignored. The IMPORT command does not provide a way to modify the sort order. In all cases, codepoint values are four digits on an ECS mode system and two digits on a non-ECS mode system.

Exporting Map Data

The EXPORT command writes a text file containing details of each character in the form used by IMPORT. This file is typically around 3.75Mb on an ECS mode system and is intended only as a starting point for creating an import file.

See also:
Extended character set, ECS.MAP, CHAR.MAP
4.82 ENABLE.KEY

The **ENABLE.KEY** command is used to enter the password for a secure encryption key.

**Format**

```plaintext
ENABLE.KEY {keyname {password}}
```

where

- **keyname** is the name of the encryption key to be enabled.
- **password** is the password protecting this key.

The command prompts for items not included on the command line. For best security, the **password** should be entered via a prompt so that it will not appear on the command stack or in log files.

An encryption key created using the **CREATE.SECURE.KEY** command can only be accessed in a QM session after it has been enabled by use of the **ENABLE.KEY** command or the corresponding QMBasic statement.

**See also:**
- Data encryption, **CHANGE.KEY.PASSWORD**, **CREATE.FILE**, **CREATE.KEY**, **CREATE.SECURE.KEY**, **DELETE.KEY**, **DISABLE.KEY**, **DISABLE.KEY (QMBasic)**, **ENABLE.KEY (QMBasic)**, **ENCRYPT.FILE**, **GRANT.KEY**, **LIST.KEYS**, **RESET.MASTER.KEY**, **REVOKE.KEY**, **SET.ENCRYPTION.KEY.NAME**, **UNLOCK.KEY.VAULT**
4.83 **ENABLE.PUBLISHING**

The **ENABLE.PUBLISHING** command re-enables publishing to all replication subscribers.

**Format**

**ENABLE.PUBLISHING**

The **ENABLE.PUBLISHING** command, present only in the QMSYS account and restricted to users with administrator rights, resumes recording of updates to published files in the replication log area following earlier use of **DISABLE.PUBLISHING**. Any updates applied to published files while publishing was disabled will not be replicated on the subscriber systems.

Publishing is automatically re-enabled if QM is restarted.

**See also:**

[Replication](#), **DISABLE.PUBLISHING**, **PUBLISH**, **PUBLISH.ACCOUNT**, **SUBSCRIBE**
4.84 **ENCRYPT.FILE**

The **ENCRYPT.FILE** command sets the data encryption key for specific fields or the entire record.

**Format**

```
ENCRYPT.FILE filename field, keyname ...
ENCRYPT.FILE filename keyname
ENCRYPT ALL keyname [NO.QUERY]
```

where

- `filename` is the name of the file to which encryption is to be applied.
- `field` is the name or field number of the field to which encryption is to be applied.
- `keyname` is the name of the encryption key to be used. This is case insensitive.

The first form of the **ENCRYPT.FILE** command sets the encryption key for one or more fields within a file that uses field level encryption. Encryption cannot be applied to a field that is used for an alternate key index.

The second form of the **ENCRYPT.FILE** command sets the encryption key for record level encryption. Alternate key indices can be defined in files that use record level encryption but, because the index itself is not encrypted by default, the indexed fields have reduced security.

The third form of the **ENCRYPT.FILE** command encrypts all hashed files defined by F-type VOC entries that are not already using encryption. Standard system files such as the VOC are omitted. The default behaviour is to build and display a list of files that are eligible for encryption. The list can then be modified to remove unwanted files. The **NO.QUERY** keyword omits this file selection step but does not disable other confirmation prompts. The displayed list shows an alphabetically sorted numbered list of files that can be scrolled from page to page using the cursor up/down or page up/down keys. Files selected for encryption are marked with an asterisk. Entry of a line number (e.g. “12”) or a range of line numbers (e.g. “12-18”) toggles the selection state of the related files. Once the list is correct, entering "A" applies the encryption. Use of "Q" or "QUIT" will abort the action.

If the file contains data records when this command is used, the file is processed to apply the encryption. A system failure or other process abort during this update will leave the file in a partially encrypted state and hence render it unusable. **Always back up a file before using this command if the file contains data.**

**Examples**

```
ENCRYPT.FILE CUSTOMERS CCARD,CARDNO
```

The above command encrypts the CCARD field of the CUSTOMERS file using the CARDNO encryption key.

```
ENCRYPT.FILE CUSTOMERS CKEY
```
The above command encrypts the CUSTOMERS file using the CKEY encryption key for record level encryption.

See also:
Data encryption, CHANGE.KEY.PASSWORD, CREATE.FILE, CREATE.KEY, CREATE.SECURE.KEY, DECRYPT.FILE, DELETE.KEY, DISABLE.KEY, DISABLE.KEY (QMBasic), ENABLE.KEY, ENABLE.KEY (QMBasic), GRANT.KEY, LIST.KEYS, RESET.MASTER.KEY, REVOKE.KEY, SET.ENCRYPTION.KEY.NAME, UNLOCK.KEY.VAULT
4.85 EXECUTE.LIST

The **EXECUTE.LIST** command executes one or more commands for each entry in a select list.

**Format**

```
EXECUTE.LIST command1; command2; command3...
```

where

```
command1, etc
```

are the commands to be executed.

The **EXECUTE.LIST** command executes the specified commands for each entry in the default select list (list 0). The value extracted from the list for each iteration of the command processing can be substituted into the command by use of a pair of empty round brackets.

Where a command displays output, use of **PAUSE** as one of the commands in the list may be useful to allow the user to indicate that the output has been seen before continuing with the next command.

**Example**

```
SELECT VOC WITH TYPE = 'F'
EXECUTE.LIST ANALYSE.FILE (); PAUSE
```

The above sequence executes the **ANALYSE.FILE** command for each file defined by an **F-type** VOC entry. The () construct is replaced by the filename for each file processed.
4.86 FILE.SAVE

The FILE.SAVE command creates a Pick style FILE-SAVE tape.

Format

```
FILE.SAVE {account.list} {options}
```

where

- `account.list` is a list of the names of the accounts to be saved. If omitted and the default select list is active, this list is used to determine the accounts to be saved. Otherwise, all accounts referenced in the QMSYS ACCOUNTS file are saved.
- `options` specifies options processing features:
  - **BINARY** suppresses translation of newlines to field marks when saving directory files. Use this option when saving binary data.
  - **DET.SUP** suppresses display of the names of files saved.
  - **EXCLUDE.REMOTE** causes remote files to be omitted as described below.
  - **INCLUDE.REMOTE** causes remote files to be saved as described below.
  - **NO.QUERY** suppresses the confirmation prompt when using a select list. The NO.SEL.LIST.QUERY mode of the **OPTION** command can be used to imply this option.

The FILE.SAVE command creates a Pick style "compatible mode" tape and saves one or more QM accounts to it.

Note carefully: Although it would be possible to use this command to create a backup of a QM account, it is recommended that operating system level tools are used for this purpose. The media format used by FILE.SAVE cannot save ECS mode files or information related to triggers, alternate key indices, replication or encryption (see below). For this reason, FILE.SAVE is not considered suitable for routine system backups. It is provided primarily as a way to return data to a Pick style system during migration of an application.

The media format also has no way to represent a distributed file. These will be omitted and a warning message will be displayed. The data elements of the distributed file will be saved in the usual way so no data should be lost, however, it will be necessary to recreate the distributed file itself if the data is restored.

The tape to be created must first be opened to the process using the SET.DEVICE command.
The command reports its progress by displaying the name of each file as it is saved unless the DET.SUP option is used.

**FILE.SAVE** normally saves all files referenced by F-type records in the VOC of the account being saved. There is a three level mechanism by which files can be excluded:

1. Field 5 of the F-type VOC entry can contain
   - D Save the dictionary but omit the data element
   - E Exclude this file from an ACCOUNT.SAVE or FILE.SAVE
   - I Include this file in an ACCOUNT.SAVE or FILE.SAVE

2. If field 5 of the VOC record does not specify any of the above flags, the EXCLUDE.REMOTE and INCLUDE.REMOTE options are used to determine whether remote files (those with a directory delimiter in their pathnames) are to be saved.

3. If neither of the above methods of file selection is used, the value of the EXCLREM configuration parameter is used to determine whether remote files are to be saved.

By use of a combination of the above methods, it should be possible to achieve total control of what is included in a save.

**Encryption**

**FILE.SAVE** operates within the security rules imposed by use of QM’s encryption features. Files that use record level encryption will not be saved if the user performing the save does not have access to the encryption key. The data saved for files that use field level encryption will have all fields for which the user is denied access set to null strings. All saved data is recorded in decrypted form and hence storage of **FILE.SAVE** media may reduce system security. The media format used by **FILE.SAVE** is an industry standard that does not provide a way to record details of data encryption. Restoring a save in which encrypted fields have been omitted is unlikely to yield a usable file. Backup of accounts that include encrypted data should be performed using operating system level tools.

**See also:**
ACCOUNT.RESTORE, ACCOUNT.SAVE, FIND.ACCOUNT, QMSAVE, RESTORE.ACCOUNTS, SEL.RESTORE, SET.DEVICE, T.ATT, T.DUMP, T.LOAD, T.xxx
4.87 FILE.STAT

The FILE.STAT command reports details of all files in selected accounts.

Format

FILE.STAT {DICT} {account.list | ALL} {LPTR}
FILE.STAT {{DICT} {file.list}} {LPTR}

where

account.list is a list of the names of the accounts to be reported. The ALL keyword may be used to process all accounts. If neither ALL nor a list of accounts is given, a report is produced for the current account.

file.list is a list of files to be reported. If no file list is given and the default select list is active, this list is used to provide a list of files to be processed.

DICT includes dictionaries in the report.

LPTR directs the report to the default printer (print unit 0) instead of the screen.

The FILE.STAT command can be used to report a summary of the files present in one of more accounts. The report that it produces requires a minimum output width of 132 characters.

The mode of operation is determined by the first name in the command line. If this is an account name, the command reports files in one or more accounts, otherwise the command reports details of one or more files in the current account.

The columns of the report are:

File name  The name of the file. For a multi-file, each element is reported separately.
Typ  The file type. This may be DYN for a dynamic hashed file, DIR for a directory file, VFS for a virtual file or COL for a data collection file.
Cur Mod  The current modulus value (number of groups).
Min Mod  The minimum modulus setting of the file.
Grp  The group size in multiples of 1024 bytes.
Record Cnt  The count of records in the file. For a hashed file, this is an approximate count but should be close to the correct value. For a directory file, the FILE.STAT command counts the records.
Primary  The size in bytes of the primary subfile of a dynamic hashed file, including any space allocated to groups that have been discarded as a result of a merge.
Overflow  The size in bytes of the overflow subfile of a dynamic hashed file, including any space allocated to overflow blocks that have been discarded and are available for reallocation.
Total Size  The total size of the file.
AK  Does the file use alternate key indices (Y/N).
TR  Does the file have a trigger function defined (Y/N).
<table>
<thead>
<tr>
<th>Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
<td>Is statistics monitoring enabled on this file (Y/N).</td>
</tr>
<tr>
<td>NC</td>
<td>Does the file use case insensitive record ids (Y/N).</td>
</tr>
<tr>
<td>NR</td>
<td>Is the NO.RESIZE option active for this file (Y/N).</td>
</tr>
<tr>
<td>EN</td>
<td>Does the file use encryption (Y/N).</td>
</tr>
<tr>
<td>RP</td>
<td>Is replication enabled on this file (Y/N).</td>
</tr>
<tr>
<td>EC</td>
<td>Does this file support ECS (Y/N). Shown on ECS mode systems only.</td>
</tr>
<tr>
<td>JN</td>
<td>Is journalling enabled on this file (Y/N).</td>
</tr>
</tbody>
</table>

Columns that are not applicable to directory files will be left blank for such files.

See also: [ANALYSE.FILE](#)
4.88 FIND.ACCOUNT

The FIND.ACCOUNT command locates an account on a Pick style FILE.SAVE tape.

Format

FIND.ACCOUNT account.name

where

account.name is the name of the account to be located.

The FIND.ACCOUNT command can be used to position a multi-account FILE.SAVE tape to a specified account. The account can then be restored using the ACCOUNT.RESTORE command with the POSITIONED option.

The tape to be restored must first be opened to the process using the SET.DEVICE command.

Use of FIND.ACCOUNT with an account name that is not present on the tape can be used to display a list of accounts that are on the tape.

See also:
ACCOUNT.RESTORE, ACCOUNT.SAVE, FILE.SAVE, QMSAVE, RESTORE.ACCOUNTS, SEL.RESTORE, SET.DEVICE, T.ATT, T.DUMP, T.LOAD, T.xxx
4.89 **FIND.FILE**

The **FIND.FILE** command reports summary details of a file.

**Format**

```
FIND.FILE filename
```

where

`filename` is the name of the file to be located.

The **FIND.FILE** command attempts to open the name file. If successful, it reports information about that file such as pathname, file type, configuration, use of indices, triggers, encryption, etc.

If the file cannot be opened, the command displays details of the error that occurred.

**Examples**

```
FIND.FILE CUSTOMERS
Dynamic hashed file, group size 4, modulus 89
Pathname: C:\SALES\CUSTOMERS
File uses encryption
```

```
FIND.FILE BP
Directory file
Pathname: C:\DEV\BP
File uses case insensitive record ids
```
4.90 FIND.PROGRAM

The FIND.PROGRAM command reports the catalogue mode of a catalogued program.

Format

FIND.PROGRAM name [DETAIL]

where

name is the name of the catalogued program to be located.

The FIND.PROGRAM command examines the local, private and global catalogue areas to locate the program catalogued as name. If found, it reports the catalogue mode, compilation date/time and program type. If the program is in more than one catalogue area, they are each reported separately in the same search order as is used when loading programs into memory.

The FIND.PROGRAM command can be useful in diagnosing problems where the system appears to be running the wrong version of a program as a result of it being catalogued in multiple locations.

The DETAIL option extends the output to show the compiler version, object code size, source pathname (where available) and embedded comments.

Example

:FIND.PROGRAM DTMOD
Program is in private catalogue
    Compiled 20 Jan 09  16:24:16
    Subroutine with 3 arguments

:FIND.PROGRAM MYTRIG DETAIL
Program is in global catalogue
    Compiled 07 Nov 11 10:36:50 using 2.12-6
    Subroutine with 5 arguments, 492 bytes
    Source pathname: /home/sales/BP/mytrig
    Comments:
        Trigger function for sales data files
        Copyright 2011

See also:
CATALOGUE, !FINDPROG()
4.91 FORMAT

The FORMAT command reformat QMBasic source programs to aid readability.

Format

```
FORMAT {file.name} {record.name} {CASE}
```

where

- `file.name` is the name of the directory file holding the QMBasic source program. If omitted, the `filename` defaults to BP.
- `record.name` is the name of the record within the file.

If the `record.name` is omitted and select list 0 is active, this is used as the source of record names to be formatted. Entries in this select list with a .H suffix are ignored. Thus the command

```
SELECT BP
```

is adequate to construct the select list.

The FORMAT command reformats a QMBasic program to comply with a conventional indentation standard. Programs may be entered with no regard to indentation and subsequently tidied up using FORMAT.

FORMAT will not move line breaks. Statements must be correctly delimited. The format actions performed are:

- The PROGRAM, FUNCTION or SUBROUTINE statement and compiler directives are adjusted to start at the leftmost column.
- Statements inside conditional blocks (THEN, ELSE, ON ERROR, LOCKED, CASE, etc.) are indented by three columns.
- WHILE and UNTIL statements are aligned with their corresponding LOOP or FOR statement.
- Multiple spaces between language elements are reduced to a single space.
- Spaces before commas in statements or argument lists are removed. A single space will follow such a comma.
- Labels are aligned to the left margin and any further text except comments is moved to the next line.
- A comment on the same line as a statement is not moved unless not to do so would place it over the statement or with less than one space before the semicolon.
- Lines holding left aligned comments are not changed.
- Comment lines which commence with spaces are moved to the alignment of the surrounding code except where the preceding line was a comment (excluding trailing comments) in which case the line is aligned with the preceding line.
EQUate and $DEFINE lines are unchanged to preserve possible user defined column alignment.

The CASE option converts all language elements, labels and data names to lowercase. Names corresponding to EQUate or $DEFINEd tokens retain the casing of the definition.

Where a $INCLUDE directive is encountered, the include record is read to establish any EQUATE or $DEFINE tokens in it. All references to these tokens in the record being formatted are converted to upper case. The include record itself is not changed.

If FORMAT fails because of faulty syntax such as unmatched THEN and END statements, the source record remains unchanged. Diagnostic messages to aid location of such errors are displayed.

Example

```
1     2     3     4     5
1234567890123456789012345678901234567890123456789
SUBROUTINE GET.DATE(PROMPT.TEXT, VALUE)
LOOP
  DISPLAY PROMPT.TEXT: ;* Prompt for date
  INPUT NEW.DATE
  VALUE = ICONV(NEW.DATE, "DDMY");* Convert the date
  WHILE STATUS()
  REPEAT
END
```

A program initially entered as above, after formatting becomes

```
1     2     3     4     5
1234567890123456789012345678901234567890123456789
SUBROUTINE GET.DATE(PROMPT.TEXT, VALUE)
LOOP
  DISPLAY PROMPT.TEXT: ;* Prompt for date
  INPUT NEW.DATE
  VALUE = ICONV(NEW.DATE, "DDMY");* Convert the date
  WHILE STATUS()
  REPEAT
END
```

With the CASE option this becomes

```
1     2     3     4     5
1234567890123456789012345678901234567890123456789
SUBROUTINE GET.DATE(PROMPT.TEXT, VALUE)
LOOP
  DISPLAY PROMPT.TEXT: ;* Prompt for date
  INPUT NEW.DATE
  VALUE = ICONV(NEW.DATE, "DDMY");* Convert the date
  WHILE STATUS()
  REPEAT
END
```
4.92 FORM.LIST

The FORM.LIST command creates an active select list from a list of record keys in a file.

Format

```
FORM.LIST [DICT] file.name record.id [ TO list.no ]
```

where

- `file.name` is the name of the file holding the list of record keys to be used to form the select list. The **DICT** qualifier specifies that the dictionary of the file is to be processed.
- `record.id` is the name of the record in `file.name` holding the list of record keys to be used to form the select list.
- `list.no` is the select list number in the range 0 to 10 to which `list.name` is to be restored. If omitted, select list zero is used.

The FORM.LIST command reads the named record and uses it to form an active select list. Typically, the list of record keys has been generated by a user written program.

Example

```
FORM.LIST INVENT.LISTS INVENTORY
92 records selected.
```

In this example, a record named INVENTORY in file INVENT.LISTS is restored to become active select list zero.

See also:

- **GET.LIST**
- **SAVE.LIST**
4.93 FSTAT

The FSTAT command collects and report file access statistics.

Format

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FSTAT ON file.name</td>
<td>Enable statistic collection</td>
</tr>
<tr>
<td>FSTAT file.name... {LPTR}</td>
<td>Report statistics</td>
</tr>
<tr>
<td>FSTAT OFF file.name</td>
<td>Disable statistics collection</td>
</tr>
<tr>
<td>FSTAT GLOBAL {LPTR}</td>
<td>Report global system statistics</td>
</tr>
<tr>
<td>FSTAT RESET</td>
<td>Clear global statistics counters</td>
</tr>
<tr>
<td>FSTAT</td>
<td>Periodic global statistics display</td>
</tr>
</tbody>
</table>

where

- **file.name** is the name of the file to be processed. Multiple file names may be included in a single command. Alternatively, if *no file.name* is specified and the default select list is active, this list will be used to determine the files to be processed.
- **LPTR** directs the report to the default printer. When reporting for multiple files, each file’s data appears on a separate page.

The FSTAT command controls collection and reporting of file access statistics for dynamic files and any associated alternate key indices. Directory files cannot be used with this command but are included in the global statistics.

Use of FSTAT with the ON keyword clears the statistics counters associated with the file and enables collection of statistics. The overhead for data collection is extremely low except that a file that is opened to read only a few records will require a write to update the counters on disk when the file is closed. The LISTF, LISTFL and LISTFR commands can be used to determine which files have file statistics enabled.

The second form of FSTAT displays or prints a report of the file access statistics. Data collection must be enabled when this mode is used. When using a select list, entries that do not correspond to dynamic files for which statistics are enabled will be ignored.

Use of FSTAT with the OFF keyword disables collection of statistics.

The GLOBAL keyword displays a report of statistics accumulated across all files regardless of whether statistics collection is enabled for the individual files. The global counters are reset when QM is started. On Windows NT and later, this occurs automatically when the last user leaves QM unless QMSvc has been configured to maintain a persistent shared memory image. The counters can be reset manually by use of the RESET keyword.

Use of the FSTAT command with no arguments displays a periodic display of the global statistics, updated once per second. This shows four columns; the overall data since the counts were last reset, the data since FSTAT was entered, the data for the last second and average per-second values since FSTAT was entered.

The figures displayed by FSTAT are:

- **Opens** The number of times the file has been opened.
- **Reads** The number of read operations performed. This covers all types of read action (e.g. READ, READU, READL, MATREAD, MATREADU).
MATREADL, etc). Reads that fail because a lock is held by another user are not counted.

Writes
The number of write operations performed.

Deletes
The number of delete operations performed. This includes deletes attempted for records that did not exist.

Clears
The number of times the file was cleared.

Selects
The number of QMBasic style SELECT operations performed.

Splits
The number of dynamic file split actions.

Merges
The number of dynamic file merge actions.

AK Reads
The number of records read from alternate key indices. This includes both application level reads (e.g. SELECTINDEX) and internal reads performed when updating an AK.

AK Writes
The number of records written to alternate key indices.

AK Deletes
The number of records deleted from alternate key indices.

Cache Hits
The number of records found in the record cache. See the RECCACHE configuration parameter.

Example

GLOBAL FILE STATISTICS

12:02:33

........System .........Total .........Average .........Total .......this run .......Per sec .......Per sec
Period 00:17:05 00:00:34
Opens 124 0 0 0
Reads 273161 143505 4228 4220.7
Writes 258086 134876 3958 3966.9
Deletes 13901 8628 269 253.8
Clears 0 0 0 0
Selects 0 0 0 0
Splits 4937 10 0 0.3
Merges 0 0 0 0
AK Reads 501091 269179 7924 7917.0
AK Writes 344094 178509 5221 5250.3
AK Deletes 156997 90670 2705 2666.8
4.94 GENERATE

The **GENERATE** command generates a QMBasic include record from a dictionary.

**Format**

```
GENERATE file.name
```

where

- **file.name** is the name of the file for which the dictionary is to be processed.

Well structured QMBasic programs should not reference fields by field number but should instead use names defined using **EQUATE** tokens. The **GENERATE** command processes the dictionary of a named file and constructs an include record with an entry for each field. Optionally, it can also produce tokens for conversion codes associated with fields.

The generation process is controlled by an X-type record named **$INCLUDE** in the dictionary. The fields of this record are:

1. **X**
2. Target file name for include record. Defaults to BP.
3. Record name to be produced for dynamic array style tokens. Defaults to `file.name` with .H suffix.
4. Token prefix for dynamic array style tokens. Each token produced is constructed from the field name with this prefix. The prefix is separated from the field name by a dot.
5. Text to be inserted into copyright line.
6. "S" if only a single entry is to be included for any field. This is the default. "M" if multiple D-type records for the same field location should produce separate include tokens. If duplicates are not enabled, a warning message will be displayed if multiple dictionary entries are found defining the same field.
7. Include conversion code tokens? "N" omits conversion tokens. "Y" generates tokens for fields that have conversion codes. "A" generates tokens for all fields including those with a null conversion code.
8. Type to create: D for dynamic array tokens (default), M for matrix tokens. Both may be used together in either order.
9. Record name to be produced for matrix style tokens. Defaults to `file.name` with .MAT.H suffix.
10. Matrix name for matrix style tokens. Defaults to `file.name`.
11. Token prefix for matrix style tokens. Each token produced is constructed from the field name with this prefix. The prefix is separated from the field name by a dot.

If the **$INCLUDE** record does not exist, it will be created when **GENERATE** is first run for the file. A prompt will be issued for the type of tokens to be generated (field 8) and the prefix character to be inserted into fields 4 and/or 11. All other fields will be left empty except for field 1 (X).

When creating the matrix style include record for use with **MATREAD**, the matrix is dimensioned to have one more element than the highest field number referenced in the dictionary. This allows for the different ways in which normal and Pick style matrices handle unexpected fields.
4.95 GET.LICENCE.UPDATE

The GET.LICENCE.UPDATE command (or the US spelling GET.LICENSE.UPDATE) checks for licence updates.

Format

GET.LICENCE.UPDATE

The GET.LICENCE.UPDATE command makes a network connection to the QM licensing server to check for an updated licence. This might be as a result of changes to the user limit, expiry date or optional features or it might be as a result of renewing the maintenance agreement.

If a licence update is available, it is automatically applied to the qmconfig file in the QMSYS account.

It is also possible to automate this process by setting the AUTOLIC configuration parameter to a non-zero value. When this option is enabled, QM will check for updates a few seconds after starting the QMSvc service or qmlnxd process and then again at intervals of the AUTOLIC value days.

Note for security auditing purposes that no application data is transmitted by this command.
4.96 GET.LIST

The GET.LIST command is used to restore a previously saved select list.

Format

```
GET.LIST {{file.name} list.name} {TO list.no} { COUNT.SUP } {DELETING}
```

where

- `file.name` is the name of the file from which the list is to be retrieved. This defaults to $SAVEDLISTS if omitted.
- `list.name` is the name of the record in `file.name` that holds the saved select list. This defaults to "$temp.n" if omitted, where `n` is the QM user number.
- `list.no` is the select list number in the range 0 to 10 to which `list.name` is to be restored. If omitted, select list zero is used.
- `COUNT.SUP` suppresses display of the number items in the restored list.
- `DELETING` deletes the list from disk.

The GET.LIST command retrieves a previously saved select list from $SAVEDLISTS or an alternative file. If the target list `list.no` was already active, the retrieved list replaces the previous list.

Both @SYSTEM.RETURN.CODE and @SELECTED are set the number of items in the list.

See the SELECT command for a discussion of chained selection where a command can be executed automatically if the GET.LIST is successful.

Examples

```
GET.LIST OVERDUE.INVOICES
57 records selected.
```

This example restores the default select list from a list named OVERDUE.INVOICES in the $SAVEDLISTS file.

```
GET.LIST INVENTORY TO 3
91 records selected.
```

This example restores a select list saved as INVENTORY to select list 3.

See also: COPY.LIST, DELETE.LIST, EDIT.LIST, FORM.LIST, SAVE.LIST, SORT.LIST
4.97 GET.STACK

The GET.STACK command restores a previously saved command stack.

Format

```
GET.STACK {file.name} stack.name
```

where

- `file.name` is the name of the file from which the stack is to be restored. If omitted but the `stack.name` is present, the $SAVEDLISTS file is used.
- `stack.name` is the name of the saved command stack.

The GET.STACK command replaces the current command stack with the record named `stack.name` from the $SAVEDLISTS file. The previous content of the command stack is discarded.

If no command line options are given, the user is prompted to enter the `stack.name` and the `file.name` defaults to $SAVEDLISTS.

Example

```
GET.STACK <<@LOGNAME>>
Command stack restored from 'jsmith'
```

This command restores the command stack from a record with id as the user's login name in the $SAVEDLISTS file.

See also:

CLEAR.STACK, SAVE.STACK
4.98 GO

The GO command is used within paragraphs to jump to a labelled line.

**Format**

```
GO label[;]
```

Any number of lines in a paragraph may be labelled. A label name consists of any sequence of characters except for spaces and mark characters. The label must be terminated with a colon and, if there is a command on the same line as the label, there must be at least one space after the colon.

The label name in the GO command may be followed by an optional colon with no intervening spaces.

The command processor scans forwards through the current paragraph for a line with the given label. An error is reported if the label is not found and the paragraph is aborted. It is valid for a paragraph to contain multiple instances of the same label name though this is not recommended as it can make maintenance more difficult.

It is not possible to jump backwards within a paragraph or from a GO command in one paragraph to a label in another paragraph.

**Example**

A paragraph containing the sequence

```
DISPLAY Line 1
GO SKIP
DISPLAY Line 2
DISPLAY Line 3
SKIP: DISPLAY Line 4
DISPLAY Line 5
```

would display

```
Line 1
Line 4
Line 5
```
4.99 **GRANT.KEY**

The **GRANT.KEY** command grants access to a specific data encryption key. This command can only be executed by users with administrator rights in the QMSYS account.

**Format**

```
GRANT.KEY keyname {GROUP} name ...
```

where

- `keyname` is the name of the encryption key. This is case insensitive
- `name ...` is a list of user names for users to be granted access. If prefixed by the `GROUP` keyword, this is a list of user groups. On Windows systems, user and group names are stored as entered but treated as case insensitive internally. On other platforms, user and group names are case sensitive.

The **GRANT.KEY** command grants access to an encryption key to one or more users or user groups. The user will be asked to enter the master key unless it has already been entered during this session.

No error occurs if the user or group specified already has access to the key.

**Example**

```
GRANT.KEY CARDNO jsmith bjones
```

The above command grants access to the encryption key named CARDNO for users jsmith and bjones.

**See also:**
- Data encryption, CHANGE.KEY.PASSWORD, CREATE.FILE, CREATE.KEY, CREATE.SECURE.KEY, DELETE.KEY, DISABLE.KEY, DISABLE.KEY (QMBasic), ENABLE.KEY, ENABLE.KEY (QMBasic), ENCRYPT.FILE, LIST.KEYS, RESET.MASTER.KEY, REVOKE.KEY, SET.ENCRYPTION.KEY.NAME, UNLOCK.KEY.VAULT
4.100 HELP

The HELP command provides help on a wide variety of topics.

Format

HELP

The HELP command invokes the Windows help system in the same way as selection of the help option from the QM program group. The help system is not available for users connecting from non-Windows clients though there is a browser based HTML help package available by download.

The operation of this system depends on the way in which the user has entered QM:

- For QM Console users on a Windows server, the help system is invoked on the server for the specified topic.

- For QMTerm users, a command is sent to the QMTerm session to cause it to open a help window on the client system. The qm.chm file must be installed in the QMSYS directory of the client system.

- For other network users, QM checks the definition of the terminal type in use and, if the u8 (asynchronous command) entry is defined, executes the hh.exe program on the client system. The qm.hlp file must be installed in the default location (c:\qmsys\qm.hlp).

- For terminals where the u8 code is not defined, the help system must be run from the QM program group.
4.101 HSM

The HSM command controls the Hot Spot Monitor performance monitoring tool.

Format

- `HSM ON {USER n}`: Start monitoring, clearing the counters
- `HSM OFF`: Stop monitoring
- `HSM DISPLAY {USER n} {LPTR u}`: Display performance data

The Hot Spot Monitor records the number of times each program module is called, the processor time in seconds spent in that module and i/o counters for opens, reads, writes and deletes.

The `HSM ON` command clears any old data and starts monitoring.

The `HSM OFF` command terminates monitoring.

The `HSM DISPLAY` command displays the collected data. It may be used while monitoring is active or after it has been switched off. The `DISPLAY` keyword can be omitted. If the `LPTR` keyword is present, the report is sent to print unit `u`, defaulting to printer 0 if the unit number is omitted.

The `USER n` option causes the command to affect the specified user and is useful in monitoring processes that do not display a command prompt.

Note that although QM maintains processor timings internally to microsecond precision and reports the figures to millisecond precision, some platforms do not provide the raw data to this level. For example, some Linux systems only report processor usage in units of one hundredth of a second.

Example

```
HSM ON
... processing ...
HSM DISPLAY

Calls    CP time   Opens   Reads   Writes   Deletes   Program
1        0.000     0       0       0        0        !SCREEN
13       0.060     2       7       0        0        $PROC
18       0.000     0       49      0        0        !PARSER
1        0.000     1       61      0        1
PRINT.DEFERED
4        0.010     0       0       0        0        $SETPTR
1        0.000     6       8       2        0        CHARGE.TOTAL
19       0.034     38      58      0        0
CALC.INVOICE.VALUE
24       0.000     0       0       0        0        ADD.MONTH
3        0.000     0       3       0        0        REVERSE
1        0.000     1       2       0        0        C:
\QM\BP.OUT\PROC
6        17.040    31      141     0        0        C:
\QM\BP.OUT\FINANCE
19       14.010    19      19      19       0        C:
\QM\BP.OUT\INVOICE
```
1 0.800 1 1 0 0 "C:\QM\BP.OUT\CHECK_EX"
1 0.000 0 0 0 0 "$HSM"

Program names beginning with a $ sign are internal components of QM. For example, $CPROC included in the report above is the command processor.
4.102 HUSH

The HUSH command suspends or enables output to the display.

Format

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HUSH ON</td>
<td>Suspend display output</td>
</tr>
<tr>
<td>HUSH OFF</td>
<td>Resume display output</td>
</tr>
<tr>
<td>HUSH</td>
<td>Toggle hush status</td>
</tr>
</tbody>
</table>

The HUSH command allows temporary suspension of display output. Output is automatically resumed in the event of an abort.

Example

```
HUSH ON
SELECT STOCK WITH QTY < REORDER.LEVEL
HUSH OFF
```

This sequence selects records from the STOCK file but suppresses any terminal output from the SELECT command. Use of the COUNTSUP option to SELECT might be a better alternative in this example.
4.103 IF

The IF command allows conditional execution of sentences within paragraphs.

Format

IF value.1 rel.op {NO.CASE} value.2 THEN sentence
IF EXISTS filename id THEN sentence
IF NOT.EXISTS filename id THEN sentence
IF value.1 IN {NO.CASE} value.2, value.3... THEN sentence
IF value.1 NOT.IN {NO.CASE} value.2, value.3... THEN sentence

where

value.1, value.2 are two items to be compared. These may be inline prompts, constants or @-variables as described below.
rel.op is the relational operator to be applied to the two values.
sentence is the sentence to be executed if the condition is True.
filename is the name of a file in which existence of a record is to be checked.
id is the name of a record for which existence is to be checked.

In its first form, the IF command compares two values using a specified relational operator. The values may be inline prompts, constants or @-variables. Null strings or string constants which include spaces should be enclosed in single or double quotation marks. The value.1 and value.2 items need to be quoted if they may evaluate to strings with embedded spaces or to reserved words such as the relational operators.

Note that because inline prompts are evaluated as the first stage of processing a command, an inline prompt in the sentence component of an IF statement will be evaluated before determining whether the condition is True. To avoid this problem, a statement such as

IF @SYSTEM.RETURN.CODE = 1 THEN LIST <<Filename>>

must be written as

IF @SYSTEM.RETURN.CODE # 1 THEN GO SKIP   LIST <<Filename>>SKIP:

The relational operator may be any of:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>LT</td>
</tr>
<tr>
<td>&lt;=</td>
<td>LE</td>
</tr>
<tr>
<td>=</td>
<td>EQ</td>
</tr>
<tr>
<td>&gt;=</td>
<td>GE</td>
</tr>
<tr>
<td>&lt;=</td>
<td>&lt;=</td>
</tr>
<tr>
<td>=</td>
<td>EQUAL</td>
</tr>
<tr>
<td>=&gt;</td>
<td>=&gt;</td>
</tr>
</tbody>
</table>
The function of each relational operator as applied to values of different types is the same as its QM Basic equivalent.

The forms of **IF** using **EXISTS** or **NOT.EXISTS** check the existence or non-existence of a record. The filename and id values are identify the file and record id. If the record id contains spaces or other special characters, it must be enclosed in quotes.

The forms of **IF** using **IN** or **NOT.IN** test whether value.1 is in or not in the list of literal values.

The optional **NO.CASE** qualifier performs a case insensitive comparison.

Multiple conditions may be linked by the keywords **AND** and **OR**. These operators are of equal priority and are evaluated left to right. Use of brackets to alter the order of interpretation is not supported in the **IF** command.

**Examples**

```qmb
PA
BASIC <<Program name>>
IF @SYSTEM.RETURN.CODE = 1 THEN CATALOGUE <<Program name>>
```

This paragraph compiles a QM Basic program (the record name of which it obtains using an inline prompt) and, if successful, adds the program to the system catalogue. In this example, use of the inline prompt in the conditioned statement is not a problem as the prompt was displayed as part of processing of the previous line.

```qmb
IF EXISTS VOC <<@LOGNAME>> THEN <<@LOGNAME>>
```

This example shows how it is possible to produce the effect found in some other multivalue products where each user may have a separate initialisation script instead of the single LOGIN paragraph used in QM. Inserting this line in the LOGIN paragraph would check if there was a paragraph with the same name as the user’s login name and, if so, execute it.

```qmb
IF @TTY IN 'phantom', 'startup', 'vbsrvr' THEN STOP
```

The above command used in the LOGIN paragraph would terminate the paragraph if the value of the @TTY variable was any of the listed values.
4.104 **INHIBIT.LOGIN**

The **INHIBIT.LOGIN** command prevents users logging in new QM sessions.

**Format**

- **INHIBIT.LOGIN ON**: Prevent further users logging in
- **INHIBIT.LOGIN OFF**: Allow users to login
- **INHIBIT.LOGIN**: Display state of login inhibit

The **INHIBIT.LOGIN** command can be used to prevent users logging in while system maintenance tasks are in progress. The command can only be executed by users with administrator rights.

With the **ON** option, this command disables further logins. The action will fail if the login inhibit is already active from some other user.

When executed with the **OFF** option by the user that set the login inhibit, logins are re-enabled.

With neither option, the command displays the current state of the login inhibit mechanism.

In all cases, `@SYSTEM.RETURN.CODE` is set to the user number of the user that has set the login inhibit or to zero if logins are allowed. Success is indicated by the value of `@SYSTEM.RETURN.CODE` being equal to `@USERNO` after use of **ON**, or zero after use of **OFF**.

While logins are inhibited, no new QM processes can be started except for phantom processes started by the user that set the inhibit. Setting a login inhibit does not logout other users who are already active.

The login inhibit is automatically released if the user that set it logs out.

**Example**

```plaintext
INHIBIT.LOGIN ON
RUN SYSTEM.BACKUP
INHIBIT.LOGIN OFF
```

This sequence prevents new users logging in while the **SYSTEM.BACKUP** program is executed.
4.105 LIST.COMMON

The LIST.COMMON command lists named common blocks.

Format

    LIST.COMMON {DETAIL}

The LIST.COMMON command displays a list of all currently created QMBasic named common blocks for the process in which the command is executed.

The DETAIL option extends the display to show where the block was created.

Examples

    LIST.COMMON
    SYS.FILES
    SCREEN.DATA

    LIST.COMMON DETAIL
    SYS.FILES      MYPROG
    SCREEN.DATA    SHOW.SCREEN

See also:

DELETE.COMMON
4.106 LIST.DF

The LIST.DF command lists the part files that form a distributed file.

**Format**

```
LIST.DF dist.file
```

where

```
dist.file
```

is the name of the distributed file.

The LIST.DF command produces a report that shows the partitioning algorithm name and source code followed by a list of the part file numbers and their associated pathnames in a distributed file.

**Example**

```
LIST.DF ORDERS
Partitioning algorithm: PART.ALC OCONV(@ID['-','1,1'],'DY')

Part No. Pathname
8 C:\SALES\ORDERS-08
9 C:\SALES\ORDERS-09
```

**See also:**

Distributed files, ADD.DF, REMOVE.DF
4.107 LIST.DIFF

The LIST.DIFF command creates a new named select list from the entries that appear in one named list but not in another named list.

Format

```
LIST.DIFF list1 {list2 {tgt.list}} {COUNT.SUP}
```

where

- `list1, list2` identify the select lists to be merged. These must correspond to the names of records in the $SAVEDLISTS file. If `list2` is omitted, a prompt is displayed for the name.
- `tgt.list` is the name of the new list to be created in $SAVEDLISTS. It is valid for `tgt.list` to be the same as one of the source lists. If `tgt.list` is omitted, a prompt is displayed for this name.
- `COUNT.SUP` indicates that display of the record count in the merged list is to be suppressed.

The LIST.DIFF command allows construction of one select list from two others. The resultant list will contain all of the items that are in `list1` but not in `list2`.

The result list will replace any existing list with the name `tgt.list`. The ordering of `tgt.list` is undefined.

`@SYSTEM.RETURN.CODE` is set to the number of items in the new list or a negative error code.

Example

```
LIST.DIFF FRANCE.CUSTOMERS MAJOR.CUSTOMERS MERGED.CUSTOMERS
```

This example merges two previously saved select lists, one holding keys for customers in France, the other for major customers to form a new list containing non-major French customers.

See also:

LIST.INTER, LIST.UNION, MERGE.LIST
4.108 **LIST.FILES**

The **LIST.FILES** command displays details of open files.

**Format**

```
LIST.FILES
LIST.FILES BRIEF
LIST.FILES DETAIL [{DICT } filename}
LIST.FILES USER {n}
```

The **LIST.FILES** command displays the number of files currently open, the peak number of files open and the limit on the number of open files as set by the **NUMFILES** configuration parameter.

The **BRIEF** option restricts the output to show only the first line. Without this option, the pathnames of all open files are shown.

The **DETAIL** option extends the report to show the user numbers and user names for each open file. Used with a file name, it shows only users who have the named file open.

The **USER** option shows the pathnames of all files open to the specified QM user number. If the number is omitted, the command shows all files open to the user executing the command.

The **NUMFILES** configuration parameter is a hard limit on the number of files that may be open at one time by all QM users. A file opened by multiple users only counts as one file. Attempting to exceed this limit will cause the application to fail. The displayed peak open count is useful in determining whether the value of **NUMFILES** is large enough.

**Example**

```
LIST.FILES
Number of files open = 4. Peak = 16. Limit (NUMFILES) = 40.
C:\QM\VOC
C:\QM\$IPC
C:\QM\BP
C:\QM\&SED.EXTENSIONS&
```

The example above shows the default output from **LIST.FILES** with the counts and the file pathnames. Adding the **BRIEF** option, as below, shows only the counts.

```
LIST.FILES BRIEF
Number of files open = 4. Peak = 16. Limit (NUMFILES) = 40.
```

Use of the **DETAIL** option shows the user numbers and user names of the QM users that have the files open.

```
LIST.FILES DETAIL
Number of files open = 4. Peak = 16. Limit (NUMFILES) = 40.
C:\QM\VOC
  2 smithm, 7 sales
C:\QM\$IPC
  2 smithm, 7 sales
```
C:\QM\BP
  7 sales
C:\QM\&SED.EXTENSIONS&
  7 sales
4.109 LIST.INDEX

The LIST.INDEX command reports details of one or more alternate key indices.

Format

```
LIST.INDEX filename field(s) {STATISTICS | DETAIL} {LPTR {n}} {NO.PAGE}
LIST.INDEX ALL {LOCAL} {STATISTICS | DETAIL} {LPTR {n}} {NO.PAGE}
```

where

- **filename** is the name of the file to be processed
- **field(s)** are the names of the fields to be reported. The keyword ALL can be used instead of a list of fields to report all indices on the file.
- **STATISTICS** reports additional statistical data.
- **DETAIL** reports statistical data and key reference detail information.
- **LPTR** directs the report to the default printer (printer 0). The optional n qualifier directs the report to an alternative printer.
- **NO.PAGE** suppresses pagination for a report directed to the user’s screen.

The first form of the LIST.INDEX command reports information about alternate key indices in the **filename** for the named **field(s)**. If no **field(s)** are specified on the command line, the user is prompted to enter the indexed field name.

The second form with the **ALL** keyword in place of the **filename** produces a composite report of all files in the account that have alternate key indices. The **LOCAL** qualifier omits files referenced via Q-pointers.

The basic report appears as below.

```
Alternate key indices for file ORDERS
Number of indices = 1

Index name....... En Tp Nulls SM Fmt NC Field/Expression
DATE            Y  D   Yes   S  R  N  17
```

The columns following the index name show:

- **En** Y if the index is enabled (built and active). N if it requires building. B if the index is being built.
- **Tp** A, C, D, I or S corresponding to the dictionary entry type used to define the index.
- **Nulls** Yes or No depending whether records with null values of the indexed field are included in the index.
- **SM** S or M corresponding to the single/multivalued nature of the indexed data.
Fmt: L or R corresponding to left or right alignment of the indexed data.
NC: Y if index is case insensitive, otherwise N.
Field/Expression: The field number or evaluated expression. Where an index is built on an A or S type dictionary item that has a conversion code in field 8, the expression shown by LIST.INDEX appears like an A-correlative conversion expression. Thus, a conversion code of MCU applied to field 1 of the data record would be shown as 1(MCU).

Use of the **STATISTICS** (short form **STATS**) keyword extends the report to include statistical information about the index showing the number of index entries (different field values) and the minimum, average and maximum number of records per index entry.

Use of the **DETAIL** keyword shows the statistics and also shows up to 63 characters of each key value and the number of records for that key value.

**See also:**
- BUILD.INDEX
- CREATE.INDEX
- DELETE.INDEX
- DISABLE.INDEX
- MAKE.INDEX
- REBUILD.ALL.INDICES
4.110 LIST.INTER

The **LIST.INTER** command creates a new named select list from the entries that appear in both of two other named lists.

**Format**

```
LIST.INTER list1 {list2 {tgt.list}} {COUNT.SUP}
```

where

- `list1, list2` identify the select lists to be merged. These must correspond to the names of records in the $SAVEDLISTS file. If `list2` is omitted, a prompt is displayed for the name.

- `tgt.list` is the name of the new list to be created in $SAVEDLISTS. It is valid for `tgt.list` to be the same as one of the source lists. If `tgt.list` is omitted, a prompt is displayed for this name.

- `COUNT.SUP` indicates that display of the record count in the merged list is to be suppressed.

The **LIST.INTER** command allows construction of one select list from two others. The resultant list will contain all of the items that are in both `list1` and `list2`.

The result list will replace any existing list with the name `tgt.list`. The ordering of `tgt.list` is undefined.

[@SYSTEM.RETURN.CODE](#) is set to the number of items in the new list or a negative error code.

**Example**

```
LIST.INTER FRANCE.CUSTOMERS MAJOR.CUSTOMERS MERGED.CUSTOMERS
```

41 records selected.

This example merges two previously saved select lists, one holding keys for customers in France, the other for major customers to form a new list containing the major customers in France.

**See also:**

- LIST.DIFF
- LIST.UNION
- MERGE.LIST
4.111 LIST.KEYS

The LIST.KEYS command lists details of encryption keys.

**Format**

```
LIST.KEYS \{LPTR \{unit\}\}
LIST.KEYS filename \{LPTR \{unit\}\}
```

where

`filename` is the name of the file to be reported.

The first form of the LIST.KEYS command is available only to users with administrator rights in the QMSYS account. It produces a report of the encryption key names defined in the key vault, showing the encryption algorithm name and the users who have access to the key. The actual encryption key is not reported. The user will be asked to enter the master key unless it has already been entered during this session. The key name is shown with an asterisk suffix if it is a secure password protected key (see CREATE.SECURE.KEY).

The second form of the LIST.KEYS command is available to all users and produces a report of the encryption keys used by the named file.

In either form, the LPTR keyword can be used to direct the output to a printer. If the print `unit` number is omitted, the default printer (unit 0) is used.

**Examples**

```
LIST.KEYS
Key.............. Algorithm Users..............
Groups............
CARDNO* AES128 jsmith
RHKEY AES256 jsmart
```

The above example shows the report from the first format of the LIST.KEYS command. There are two encryption keys defined on this system. The CARDNO key is password protected.

```
LIST.KEYS CLIENTS
Filename: CLIENTS
Pathname: /usr/sales/CLIENTS
Field 7, CARDNO
Field 22, RHKEY
```

The above example shows the report from the second format of the LIST.KEYS command. The CLIENTS file uses field level encryption with a different key for each encrypted field.

See also:

Data encryption, CHANGE.KEY.PASSWORD, CREATE.FILE, CREATE.KEY, CREATE.SECURE.KEY, DELETE.KEY, DISABLE.KEY, DISABLE.KEY (QMBasic), ENABLE.KEY, ENABLE.KEY (QMBasic), ENCRYPT.FILE, GRANT.KEY.
RESET.MASTER.KEY, REVOKE.KEY, SET.ENCRYPTION.KEY.NAME, UNLOCK.KEY.VAULT
4.112 LIST.LOCKS

The LIST.LOCKS command reports the state of the 64 system wide task locks.

Format

LIST.LOCKS

The LIST.LOCKS command displays a table of locks showing the user number of the process holding each of the 64 task locks where appropriate. The displayed user number is followed by an asterisk if it is the process from which the LIST.LOCKS command was executed.

Examples

|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|-----|

This example shows the display when task locks 7 and 19 are in use. Lock 7 is held by user 1 who also issued the LIST.LOCKS command. Lock 19 is held by user 3.

LIST.LOCKS
No task locks reserved by any user

In this example, there are no task locks in use.

See also:
CLEAR.LOCKS, LIST.LOCKS, LOCK (command), LOCK (QMBasic), UNLOCK
4.113 LIST.PHANTOMS

The LIST.PHANTOMS command shows a list of currently running phantom processes.

Format

LIST.PHANTOMS {USER username} {POOL poolname}

The LIST.PHANTOMS command displays a summary of currently active phantom processes. It is effectively a subset of the information shown by the LISTU command. For each phantom process, it shows the user number, operating system process id, login name, login time and account name. The login time is shown in the local time zone of the user executing the command.

The optional USER qualifier causes the command to show only phantoms for the specified username. Except on Windows, the user name is case sensitive.

The optional POOL qualifier causes the command to show only phantoms for the specified connection pool name.

Phantom processes that are running as replication subscribers also show the name of the publisher system from which they are receiving data.

Example

<table>
<thead>
<tr>
<th>User</th>
<th>Pid</th>
<th>Username</th>
<th>Login time</th>
<th>Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5001</td>
<td>root</td>
<td>28 May 07:02</td>
<td>QMSYS</td>
</tr>
<tr>
<td>34</td>
<td>5220</td>
<td>jenny</td>
<td>28 May 11:24</td>
<td>SALES</td>
</tr>
</tbody>
</table>

LIST.PHANTOMS USER jenny

<table>
<thead>
<tr>
<th>User</th>
<th>Pid</th>
<th>Username</th>
<th>Login time</th>
<th>Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>34</td>
<td>5220</td>
<td>jenny</td>
<td>28 May 11:24</td>
<td>SALES</td>
</tr>
</tbody>
</table>

See also: CHILD(), PHANTOM, !PHLOG(), STATUS
4.114 LIST.PRINTERS

The LIST.PRINTERS command shows a list of printers known to the operating system. This is currently supported only on Windows systems.

Format

LIST.PRINTERS

The LIST.PRINTERS command displays an alphabetically sorted list of printer names known to the operating system. If a printer is known but cannot be accessed the operating system error code is shown after the name.

The default printer and network printers are marked as in the example below.

Example

LIST.PRINTERS
Dell-c3760dn  (Default)
Dell-c3760dn-PS
Fax
Fax  -  HP Officejet 7500 E910  (Network)

See also:
SETPTR
4.115 LIST.READU

The LIST.READU command displays details of file and record locks.

Format

```
LIST.READU {user} {DETAIL} {NO.PAGE} {LPTR {n}} {WAIT}
```

where

- **user** is the user number for which the locks are to be reported. If omitted, all locks are displayed.
- **DETAIL** includes the limit, current count and peak number of record locks and the date/time at which the lock was acquired or a blocked process started waiting.
- **NO.PAGE** suppresses display pagination.
- **LPTR {n}** directs output to logical print unit n. If n is omitted, it defaults to zero, the default print unit.
- **WAIT** includes details of users waiting for locks held by other users.

The LIST.READU command displays or prints details of file, read and update locks held by one or all users.

Example

```
User File Path......................... Type Id......
1  2 D:\SALES\STOCK                RU   P-174-43
1  2 D:\SALES\STOCK                RU   P-967-47
5  2 D:\SALES\STOCK                RU   P-954-55
2  4 D:\SALES\INVOICES             FX
3  4 D:\SALES\INVOICES             WAIT 17565
```

In the above report, users 1 and 5 hold record update locks in file 2 (D:\SALES\STOCK) and user 2 has a file lock on file 4 (D:\SALES\INVOICES). The file number is an internal reference to the file and is also needed for the UNLOCK command.

The lock type is shown as RL for shareable record locks, RU for record update locks and FX for file locks. Files opened for exclusive access by internal components of QM will appear as a pseudo-lock type of EXCL.

Details of users waiting for locks are only shown if the WAIT keyword is used. In this example, user 3 is waiting to lock record 17565 in file 4 but is blocked by user 2.

Adding the DETAIL keyword to the command produces an extended report:

```
Record lock limit = 400, Current = 3, Peak = 73
User File Path......................... Type Id......
1  2 D:\SALES\STOCK                RU   P-174-43
```
The first line of the above report shows statistical information for active record locks. The file lock and the user waiting for a lock in this example do not contribute to these numbers. The peak number of locks is useful in determining a good value for the `NUMLOCKS` configuration parameter.

The time at which the lock was acquired is shown in the timezone of the user executing the command.
4.116 LIST.REPLICATED

The LIST.REPLICATED command displays a summary of all replicated files within an account.

Format

LIST.REPLICATED {NO.PAGE} {LPTR \{n\}} {LOCAL} \{server ...\}

where

LPTR \{n\} directs output to print unit \( n \). If \( n \) is omitted, it defaults to zero, the default print unit.

NO.PAGE suppresses display pagination.

LOCAL omits files referenced via Q-pointers.

server is a list of one or more server names. Only files replicated to the named server(s) will be reported.

The LIST.REPLICATED command displays a summary of all replicated files that are referenced by F or Q type VOC entries unless the LOCAL option is used to omit Q-pointer files. Dictionaries are included for F-type VOC entries.

For each file that has a replication target defined, the command shows the file name and a list of the replication targets.

Example

<table>
<thead>
<tr>
<th>File Name</th>
<th>Targets</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORDERS</td>
<td>REPORTING:SALES:ORDERS</td>
</tr>
<tr>
<td>INVOICES</td>
<td>REPORTING:SALES:INVOICES</td>
</tr>
</tbody>
</table>

In the above report, the ORDERS and INVOICES files are replicated to files of the same names in the SALES account on the REPORTING server.

See also:

Replication
4.117 LIST.SERVERS

The LIST.SERVERS command shows a list of QMNet and VFS servers known to the system.

Format

LIST.SERVERS \{ALL\} \{DETAIL\} \{LPTR \{n\}\}

QMNet allows an application to access QM data files on other servers as though they were local files, with complete support for concurrency control via file and record locks. Public servers are visible to all users and are defined by the system administrator using the SET.SERVER command. Private servers are defined using SET.PRIVATE.SERVER and are local to the QM session in which they are defined.

The Virtual File System (VFS) allows an application to access other file systems as though they were local files. VFS servers are visible to all users and are defined by the system administrator using the SET.VFS.SERVER command.

The LIST.SERVERS command displays a list of the server names that have been defined and to which the user has access. It shows the QM name for the server, the IP address or network name, the port number, the security status, and the user name that will be used to connect. For a VFS server, the VFS handler name is also shown.

The security status column, applicable only to public servers, indicates whether specific security rules have been set for this server. See the ADMIN.SERVER command for more details.

The ALL option, available only to users with administrative rights, includes servers to which the user has no access.

The DETAIL keyword, available only to users with administrative rights, extends the report to show the users/groups that have access for each remote user name. This report needs a screen width of at least 120 characters.

The LPTR keyword directs the report to a printer. If the unit number \(n\) is not given, the default printer (unit 0) is used.

Example

```
LIST.SERVERS
Public Servers.  IP address.          Port...  Sec  User
name............
SALES            193.100.13.11       Default  No
martin
ADMIN            193.100.13.18       4000    Yes  root
Private Servers. IP address.          Port...  Sec  User
name............
HR               193.100.13.41       Default  tony
```

See also:
The Virtual File System, QMNet, ADMIN.SERVER, DELETE.SERVER, SET.SERVER
4.118 LIST.TERM.TYPES

The LIST.TERM.TYPES command displays a list of terminal types.

Format

LIST.TERM.TYPES

The LIST.TERM.TYPES command scans the terminfo database and displays an alphabetically sorted list of available terminal types and their descriptions.

Example

LIST.TERM.TYPES
  ANSI             ANSI/pc-term compatible with color
  ANSI-GENERIC    Generic ANSI standard terminal
  DEC-VT100       VT100 emulation
  DEC-VT220       DOS tnvt200 terminal emulator
  DEC-VT330       DEC vt340 graphics terminal with 24 line page
  DEC-VT340       DEC vt340 graphics terminal with 24 line page
  DEC-VT400       DEC vt400 24x80 column autowrap
  etc....

See also:
Terminfo, TERM
4.119 LIST.TRIGGERS

The LIST.TRIGGERS command displays a summary of all trigger subroutines used within an account.

Format

LIST.TRIGGERS {NO.PAGE} {LPTR {n}} {LOCAL}

where

LPTR {n} directs output to print unit n. If n is omitted, it defaults to zero, the default print unit.

NO.PAGE suppresses display pagination.

LOCAL omits files referenced via Q-pointers.

The LIST.TRIGGERS command displays a summary of all trigger functions used by files that are referenced by F or Q type VOC entries unless the LOCAL option is used to omit Q-pointer files. Dictionaries are included for F-type VOC entries.

For each file that has a trigger defined, the command shows the file name, the trigger function name and the events that activate the trigger. The events are shown in two columns, one for pre-event and one for post-event, using letters W (write), R (read), D (delete) and C (clear).

Example

<table>
<thead>
<tr>
<th>File Name</th>
<th>Function</th>
<th>Pre</th>
<th>Post</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORDERS</td>
<td>STOCK.CHECK</td>
<td>W</td>
<td>D</td>
</tr>
<tr>
<td>STAFF</td>
<td>STF.VALIDATE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In the above report, the ORDERS file has a trigger function named STOCK.CHECK that is activated for post-write and post-delete events. The STAFF file has a pre-write trigger named STF.VALIDATE.

See also: SET.TRIGGER
The `LIST.UNION` command creates a new named select list from the entries that appear in either of two other named lists. Items that appear in both lists are not duplicated.

**Format**

```
LIST.UNION list1 {list2 {tgt.list}} {COUNT.SUP}
```

where

- `list1`, `list2` identify the select lists to be merged. These must correspond to the names of records in the $SAVEDLISTS file. If `list2` is omitted, a prompt is displayed for the name.
- `tgt.list` is the name of the new list to be created in $SAVEDLISTS. It is valid for `tgt.list` to be the same as one of the source lists. If `tgt.list` is omitted, a prompt is displayed for this name.
- `COUNT.SUP` indicates that display of the record count in the merged list is to be suppressed.

The `LIST.UNION` command allows construction of one select list from two others. The resultant list will contain all of the items that are in `list1` plus all of the items in `list2` that are not also in `list1`.

The result list will replace any existing list with the name `tgt.list`. The ordering of `tgt.list` is undefined.

`@SYSTEM.RETURN.CODE` is set to the number of items in the new list or a negative error code.

**Example**

```
LIST.INTER FRANCE.CUSTOMERS MAJOR.CUSTOMERS MERGED.CUSTOMERS
41 records selected.
```

This example merges two previously saved select lists, one holding keys for customers in France, the other for major customers to form a new list containing the major customers in France.

**See also:**

`LIST.DIFF`, `LIST.INTER`, `MERGE.LIST`
4.121 LIST.USERS

The LIST.USERS command lists users from the register of users for security checks. The command is restricted to users with administrator rights.

Format

LIST.USERS

The LIST.USERS command lists the names of all entries in the user name register.

The report shows the user name, the login account (if set), the date and time of last login, whether the user has administrator rights and whether there are account restrictions on this user. For more details regarding account restrictions, see Application Level Security.

If QM security is not enabled, the output from the LIST.USERS command will include a line

WARNING: System is running in insecure mode

Example

<table>
<thead>
<tr>
<th>User name</th>
<th>Login account</th>
<th>Last login</th>
<th>Adm</th>
<th>Restr</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADMINISTRATOR</td>
<td></td>
<td>22 Oct 06 17:27</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>BERT</td>
<td>QMTEST</td>
<td>04 Nov 02 10:40</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>GEORGE</td>
<td>SALES</td>
<td>03 Nov 06 09:03</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>MARTIN</td>
<td></td>
<td>01 Nov 06 17:23</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

See also: ADMIN.USER, CREATE.USER, DELETE.USER, PASSWORD, SECURITY, Application Level Security
4.122 LIST.VARS

The LIST.VARS command displays user @-variables.

Format

LIST.VARS {ALL}

The LIST.VARS command displays the values of user defined @-variables. The optional ALL keyword extend the display to include the standard @USER.RETURN.CODE and @USER0 to @USER4 variables.

Example

LIST.VARS
 LOOP.CT : 7
 LAST.LOG: 41926

See also:
SET
4.123 LISTDICT

LISTDICT is a paragraph that lists a dictionary in a format similar to that used by Pick style systems.

Format

LISTDICT filename {LPTR}

where

filename is the name of the file for which the dictionary is to be listed.

The LISTDICT paragraph reports A and S-type dictionary items in a format similar to that used by Pick style systems. D and I-type items are also included though the column headings may not truly reflect the role of the item. The column widths in the report are different depending on the width of the terminal display.

The LPTR keyword will direct the output to the printer.

LISTDICT shows only A, D, I and S-type records. The entire content of a dictionary can be viewed using LIST.

Examples

LISTDICT ORDERS

The above command produces a Pick style representation of the records in the dictionary of the ORDERS file.

LIST DICT ORDERS

The above command produces a report of all records in the dictionary of the ORDERS file in the default QM dictionary list format.
4.124 LISTF

The LISTF command lists all files defined in the VOC.

Format

    LISTF \{modes\} \{LPTR\}

The LISTF command is a sentence to list all F type VOC entries. The listing shows the VOC name of the file, the file type, the description (field 1 of the VOC entry), the data portion pathname (field 2) and the dictionary pathname (field 3). The report is sorted by file name.

The file type column shows:

- **DH** for a dynamic hash file. This type code may be followed by
  - C the file is a data collection
  - E the file uses encryption
  - I the file has alternate key indices
  - R the file is exported for replication
  - S file statistics are enabled
  - T the file has a trigger function defined
  - X the file is in ECS mode
- **Dir** for a directory file. This type code may be followed by
  - E the file uses encryption
  - R the file is exported for replication
  - T the file has a trigger function defined
- **Mult** for a multifile. This entry will be followed by a line for each subfile.
- **Err** if the file cannot be opened.

The column is blank for a VOC entry with no data pathname.

The optional *modes* element of this command is a query processor WITH or WITHOUT clause using ENCRYPTION, REPLICATION or TRIGGER. For example,

    LISTF WITH REPLICAATION

would show only files that use replication.

The LPTR keyword will direct the output to the printer. Other query processor keywords can be appended to the command.

See also:
CREATE.FILE, DELETE.FILE, LISTFL, LISTFR, LIST.REPLICATED
4.125 LISTFL

The **LISTFL** command lists all files defined in the VOC that are local to the account.

**Format**

```
LISTFL {modes} {LPTR}
```

The **LISTFL** command is a sentence to list all F type VOC entries referencing files that are local to the account. Selection is performed by reporting only those records for which the data portion pathname does not contain a directory separator character. It is thus possible to defeat this selection process by using absolute pathnames for local files. The **CREATE.FILE** command always uses the file name only for a local file.

The listing shows the VOC name of the file, the file type, the description (field 1 of the VOC entry), the data portion pathname (field 2) and the dictionary pathname (field 3). The report is sorted by file name.

The file type column shows:

- **DH** for a dynamic hash file. This type code may be followed by:
  - **C** the file is a data collection
  - **E** the file uses encryption
  - **I** the file has alternate key indices
  - **R** the file is exported for replication
  - **S** file statistics are enabled
  - **T** the file has a trigger function defined
  - **X** the file is in ECS mode

- **Dir** for a directory file. This type code may be followed by:
  - **E** the file uses encryption
  - **R** the file is exported for replication
  - **T** the file has a trigger function defined

- **Mult** for a multifile. This entry will be followed by a line for each subfile.

- **Err** if the file cannot be opened.

The column is blank for a VOC entry with no data pathname.

The optional *modes* element of this command is a query processor **WITH** or **WITHOUT** clause using ENCRYPTION, REPLICATION or TRIGGER. For example,

```
LISTFL WITH REPLICATION
```

would show only files that use replication.

The **LPTR** keyword will direct the output to the printer. Other query processor keywords can be appended to the command.

**See also:**

**CREATE.FILE, DELETE.FILE, LISTF, LISTFR, LIST.REPLICATED**
4.126 LISTFR

The **LISTFR** command lists all files defined in the VOC that are remote to the account.

**Format**

```
LISTFR {modes} {LPTR}
```

The **LISTFR** command is a sentence to list all F type VOC entries referencing files that are remote to the account. Selection is performed by reporting only those records for which the data portion pathname contains a directory separator character. It is thus possible to defeat this selection process by using absolute pathnames for local files. The **CREATE.FILE** command always uses the file name only for a local file.

The listing shows the VOC name of the file, the file type, the description (field 1 of the VOC entry), the data portion pathname (field 2) and the dictionary pathname (field 3). The report is sorted by file name.

The file type column shows:

- **DH** for a dynamic hash file. This type code may be followed by:
  - C the file is a data collection
  - E the file uses encryption
  - I the file has alternate key indices
  - R the file is exported for replication
  - S file statistics are enabled
  - T the file has a trigger function defined
  - X the file is in ECS mode

- **Dir** for a directory file. This type code may be followed by:
  - E the file uses encryption
  - R the file is exported for replication
  - T the file has a trigger function defined

- **Mult** for a multifile. This entry will be followed by a line for each subfile.

- **Err** if the file cannot be opened.

The column is blank for a VOC entry with no data pathname.

The optional **modes** element of this command is a query processor **WITH** or **WITHOUT** clause using ENCRYPTION, REPLICATION or TRIGGER. For example,

```
LISTFR WITH REPLICATION
```

would show only files that use replication.

The **LPTR** keyword will direct the output to the printer. Other query processor keywords can be appended to the command.

**See also:**

**CREATE.FILE, DELETE.FILE, LISTF, LISTFL, LIST.REPLICATED**
4.127 LISTK

The LISTK command lists all keywords defined in the VOC.

Format

LISTK {LPTR}

The LISTK command is a sentence to list all K type VOC entries defining keywords.

The listing shows the keyword, the description (field 1 of the VOC entry) and the keyword value. The report is sorted by keyword value.

The LPTR keyword will direct the output to the printer. Other query processor keywords can be appended to the command.
4.128 LISTM

The **LISTM** command lists all menus defined in the VOC.

**Format**

```
LISTM {LPTR}
```

The **LISTM** command is a sentence to list all **M type VOC entries**.

The listing shows the VOC name of the menu and its title line.

The **LPTR** keyword will direct the output to the printer. Other query processor keywords can be appended to the command.
4.129 LISTPA

The **LISTPA** command lists all paragraphs defined in the VOC.

**Format**

```
LISTPA {LPTR}
```

The **LISTPA** command is a sentence to list all PA type VOC entries.

The listing shows the VOC name of the paragraph, the description (field 1 of the VOC entry) and the first command in the paragraph (field 2).

The **LPTR** keyword will direct the output to the printer. Other query processor keywords can be appended to the command.
4.130 LISTPH

The **LISTPH** command lists all phrases defined in the VOC.

**Format**

```
LISTPH {LPTR}
```

The **LISTPH** command is a sentence to list all PH type VOC entries.

The listing shows the VOC name of the phrase, the description (field 1 of the VOC entry) and the phrase expansion (field 2).

The **LPTR** keyword will direct the output to the printer. Other query processor keywords can be appended to the command.
4.131 LISTPQ

The LISTPQ command lists all PROCs defined in the VOC.

Format

LISTPQ {LPTR}

The LISTPQ command is a sentence to list all PQ type VOC entries.

The listing shows the VOC name of the PROC, the description (field 1 of the VOC entry) and the first command in the PROC (field 2).

The LPTR keyword will direct the output to the printer. Other query processor keywords can be appended to the command.
4.132 LISTQ

The LISTQ command lists all indirect file references defined in the VOC.

Format

```
LISTQ {LPTR}
```

The LISTQ command is a sentence to list all Q type VOC entries.

The listing shows the VOC name of the item, the description (field 1 of the VOC entry), target account (field 2), target file (field 3) and server name (field 4).

The LPTR keyword will direct the output to the printer. Other query processor keywords can be appended to the command.
4.133 **LISTR**

The **LISTR** command lists all remote items defined in the VOC.

**Format**

```
LISTR {LPTR}
```

The **LISTR** command is a sentence to list all R type VOC entries.

The listing shows the VOC name of the item, the description (field 1 of the VOC entry), the target file (field 2) and the target record id (field 3).

The **LPTR** keyword will direct the output to the printer. Other query processor keywords can be appended to the command.
4.134 LISTS

The LISTS command lists all sentences defined in the VOC.

Format

LISTS {L PTR}

The LISTS command is a sentence to list all S type VOC entries.

The listing shows the VOC name of the sentence, the description (field 1 of the VOC entry) and the sentence expansion (field 2).

The L PTR keyword will direct the output to the printer. Other query processor keywords can be appended to the command.
4.135 LISTU

The LISTU command lists the users currently in QM sorted by user number. The SORTU command is identical except that it sorts by user name.

Format

LISTU [DETAIL]

The LISTU command displays a list of users in QM. For each user, it shows their QM user number, the corresponding operating system process id and their network address or device name, user name and account name. The network address is shown as "Console" for console users (sessions running QM directly rather than a network connection) or "Phantom" for a background process. The login time is shown in the local time zone of the user executing the command.

The user executing the LISTU command is marked by an asterisk at the left edge of the displayed report. For phantom users, the parent user number is shown unless the original parent process has logged out. The parent user number is followed by "G" if the phantom process is grouped to logout with its parent.

Use of the DETAIL keyword adds a line at the start of the report that shows:

- the current number of interactive (terminal user) processes
- the current number of QMClient processes
- the current number of interactive phantom processes that consume a QM licence
- the current number of other phantom processes that do not consume a QM licence
- the peak number of processes that consumed a licence since QM was started. Note that on a system with device licensing, this figure can be lower than the current total process count.

The user count detail information can be retrieved in a QMBasic program using SYSTEM(1056).

Example

LISTU

<table>
<thead>
<tr>
<th>User</th>
<th>Pid</th>
<th>Puid</th>
<th>Login time</th>
<th>Origin : User name, Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>156</td>
<td></td>
<td>15 Apr 11:48</td>
<td>Console: ADMINISTRATOR, QMSYS</td>
</tr>
<tr>
<td>2</td>
<td>186</td>
<td></td>
<td>15 Apr 11:21</td>
<td>193.118.13.10: JSMITH, SALES</td>
</tr>
<tr>
<td>4</td>
<td>196</td>
<td></td>
<td>15 Apr 12:02</td>
<td>Phantom: ADMINISTRATOR, QMSYS</td>
</tr>
</tbody>
</table>

The example report above shows three processes logged on. The user running this command is marked with an asterisk.

Use of the DETAIL keyword might produce a report such as that below, showing the current count of each process type and the peak number of processes that were included in the licensed user count.

LISTU DETAIL

2 interactive, 0 QMClient, 0 iPhantom, 1 phantom, 6 peak

<table>
<thead>
<tr>
<th>User</th>
<th>Pid</th>
<th>Puid</th>
<th>Login time</th>
<th>Origin : User name, Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>156</td>
<td></td>
<td>15 Apr 11:48</td>
<td>Console: ADMINISTRATOR, QMSYS</td>
</tr>
<tr>
<td>2</td>
<td>186</td>
<td></td>
<td>15 Apr 11:21</td>
<td>193.118.13.10: JSMITH, SALES</td>
</tr>
<tr>
<td>4</td>
<td>196</td>
<td></td>
<td>15 Apr 12:02</td>
<td>Phantom: ADMINISTRATOR, QMSYS</td>
</tr>
</tbody>
</table>

See also:
LIST.PHANTOMS, STATUS, !LISTU(), LGNWAIT configuration parameter
4.136 LISTV

The LISTV command lists all verbs defined in the VOC.

Format

    LISTV [LPTR]

The LISTV command is a sentence to list all V type VOC entries.

The listing shows the VOC name of the verb, the description (field 1 of the VOC entry), the verb type (field 2) and the processing function or internal dispatch code (field 3).

The LPTR keyword will direct the output to the printer. Other query processor keywords can be appended to the command.
4.137 **LOAD.LANGUAGE**

The **LOAD.LANGUAGE** command loads a new set of message definitions into the QM message library.

**Format**

`LOAD.LANGUAGE pathname`

where

`pathname` identifies the message text file to be loaded.

QM includes support for text message output in multiple languages. The standard message library installed with QM contains English messages texts. Additional language libraries may be available from Zumayas or from QM resellers. Users can also perform their own translations of the source text downloaded from the product web site.

The default message text source file is available for download from the openqm.com web site and includes comment lines detailing the rules for successful translation. It is strongly recommended that users should use the two letter international country code as the language identifier in the PREFIX line (e.g. FR for French) but variations are permitted such as, perhaps, CHF for Swiss French.

A set of message texts obtained by download of a pre-translated file or by translation of the default English messages can be installed by executing

`LOAD.LANGUAGE pathname`

in the QMSYS account where `pathname` points to the message text file to be installed. A single system may have any number of languages installed and the non-English messages will be preserved at an upgrade.

The message text source file uses UTF-8 encoding for any characters with codepoint values over 127. The **LOAD.LANGUAGE** command will read the named file using this encoding. On non-ECS mode systems, all characters in the source text must, after decoding, lie in the 8-bit character set. Any characters outside the 8-bit character set will be written to the message library incorrectly.

Message text are stored in a file named MESSAGES in the QMSYS account. On first installation of QM, this file will be created in either ECS or non-ECS mode depending on the mode of the underlying QM system. Installing an ECS mode version of QM over a non-ECS mode version will automatically reconfigure the messages file to ECS mode. The reverse action, installing a non-ECS mode version of QM over an ECS mode version will delete and recreate the file, populating it only with the standard English messages. Any locally applied alternative messages would need to be reloaded.

**See also:**

Multi-language applications, **SET.LANGUAGE**
4.138 LOCK

The **LOCK** command sets a task lock.

**Format**

```
LOCK lock.number [NO.WAIT]
```

where

- `lock.number` is the number of the task lock (0 to 63) to be set.
- `NO.WAIT` specifies that the process is not to wait if the lock is not available.

The **LOCK** command sets one of the 64 system wide task locks. Four situations exist:

If the lock is available, the process acquires the lock and continues. `@SYSTEM.RETURN.CODE` will be set to `lock.number`.

If the lock is already owned by this process, a warning message is displayed and execution continues. `@SYSTEM.RETURN.CODE` will be set to `lock.number`.

If the lock is owned by another process and the `NO.WAIT` option has been used, a message is displayed indicating the unavailability of the lock and the process continues. The value of `@SYSTEM.RETURN.CODE` will be negative (-ER$LCK) and can be tested to check for this situation in paragraphs.

If the lock is owned by another process and the `NO.WAIT` option has not been used, a message is displayed indicating that the process is waiting and execution is suspended until the lock becomes available. The break key may be used to interrupt this wait.

**Examples**

```
LOCK 5
Set task lock 5
```

In this example, task lock 5 was available when the **LOCK** command was executed.

```
LOCK 5
Waiting for task lock to become available
```

In this example, task lock 5 held by another process when the **LOCK** command was executed. The process waits for the lock to become available.

```
LOCK 5 NO.WAIT
Task lock is already in use
```

As in the previous example, task lock 5 was held by another process when the **LOCK** command was executed. In this case, the `NO.WAIT` option causes the process to continue without waiting for the lock to become available.
See also:
C\textsc{lear} \textsc{locks}, \textsc{l}\textsc{i}st \textsc{l}\textsc{o}cks, \textsc{l}\textsc{o}ck (command), \textsc{l}\textsc{o}ck \textsc{(qmbasic)}, \textsc{t}\textsc{estlock()}, \textsc{u}n\textsc{l}ock
4.139 LOCK.TRACER

The LOCK.TRACER command enables or disables lock tracing and provides a tool to display the logged data.

Format

LOCK.TRACER ON            Start logging
LOCK.TRACER OFF            Stop logging
LOCK.TRACER {USER \textit{n}} {LPTR \textit{n}}    Display or print logged data

The LOCK.TRACER command requires administrative rights and is normally only present in the VOC of the QMSYS account. The V-type VOC entry can be copied to other accounts if required.

Enabling lock tracing logs locking related actions for all users. The volume of data recorded and the impact on performance are likely to be significant so it is best to enable logging only for short periods. The USER option to the command filters data on display to show only actions relating to the specified QM user number.

The logged data is written to a file named locktrace in the TEMP subdirectory of the QMSYS account. On ECS mode systems, this file will be recorded with UTF-8 encoding. Re-enabling the lock tracer will overwrite any existing locktrace file.

The report displayed by the third form of the LOCK.TRACER command requires a terminal width of 132 characters. When using a QMConsole session on Windows, the window will automatically adjust size. For all other terminal types, the width must be configured in the terminal emulator.

Users may wish to write their own log analysis tool to perform more advanced filtering. Each line of the log is a value mark delimited dynamic array where the values are:

1. Timestamp as an epoch value
2. QM user number
3. File pathname
4. Record id
5. Action code
6. Program name
7. Program execution offset as a hexadecimal value
8. Program line number, -1 if not available

The action code is a numeric value. The list below shows the actions for each value, and short text used in the default report.

1  Get RU  Get record update lock
2  Get RL  Get shareable read lock
3  RU->RL  Downgrade record update lock
4  RL->RU  Upgrade shareable record lock
5  Get FX  Get file exclusive lock
6  Rel RX  Release record lock
7  Rel FX  Release file lock

See also:
Application Profiling
4.140 LOGIN.PORT

The `LOGIN.PORT` command logs in a serial port from within another QM session. This command is currently only available on Windows systems.

**Format**

```
LOGIN.PORT port {account} {params}
```

where

- `port` is the serial port name (e.g. COM1).
- `account` is the QM account in which the user is to run. If omitted, the new process runs in the same account as the user issuing the command.
- `params` is any combination of the following serial communications parameters:
  - `BAUD rate` sets the data rate (default 9600).
  - `BITS n` sets the number of bits per character (default 8).
  - `PARITY mode` sets the parity mode: NONE (default), ODD or EVEN.
  - `STOP.BITS n` sets the number of stop bits (default 1).

The `LOGIN.PORT` command creates a new QM process that uses the named serial port as its terminal device. This process will run with the same user name as the process performing the command.

The new process will execute the `MASTER.LOGIN` and `LOGIN` paragraphs in the same way as any other QM process. The `LOGIN` paragraph could be used, for example, to start a program that monitors the named port for activity.
4.141 LOGMSG

The LOGMSG command adds a line to the system error log.

Format

   LOGMSG text

where

   text    is the message to be logged.

This command is identical in effect to use of the LOGMSG statement in a QMBasic program.

QM includes the option to maintain a log of system error messages in a file named errlog in the QMSYS account. The LOGMSG command can be used to write messages into this file. If the error log is disabled by setting the ERRLOG configuration parameter to zero, the LOGMSG command will be ignored.
4.142 LOGOUT

The LOGOUT command terminates a QM process.

Format

    LOGOUT {{GROUP } user}
    LOGOUT ALL

where

    user          is the user number of the process to terminate. Multiple user numbers may be
given.

The LOGOUT command aborts the specified process. A user who does not have administrator
rights can only log out processes running under the same user name.

If user is omitted, the LOGOUT command terminates the user's own session.

Use of the GROUP option, also logs out phantom processes started by the specified user. Where
multiple user numbers are specified, this option applies to all user numbers following the GROUP
keyword.

The LOGOUT ALL form of the command, available only in the QMSYS account to users with
administrator rights, logs out all QM processes except the one from which it is issued.

The process is terminated without execution of the ON.EXIT paragraph.

Example

    LOGOUT 2
    Phantom 2 : forced logout

This example shows forced termination of phantom user 2.

See also:
QUIT
4.143 LOGTO

The LOGTO command moves to an alternative account directory without leaving QM.

Format

```
LOGTO name [RESET {ALL}] }
```

where

- `name` is the name of the target account. This may be an account name as in the ACCOUNTS register of the QMSYS account or the pathname of the new account directory.
- `RESET` causes the command processor to discard all active paragraphs, menus, etc.

If the user has account restrictions imposed on them (see Application Level Security), `name` must be an account name, not be a pathname.

Multiple accounts are useful where there are several distinct projects. They can also be used to separate development and production versions of an application.

The LOGTO command closes the current VOC, moves to the account directory specified by `name` and opens the VOC of the new directory.

If the `RESET` keyword is present, any active programs, menus, etc at the current command processor level are discarded. This is particularly useful when using LOGTO in a menu.

Use of `RESET ALL` discards all active programs, menus, etc and returns the bottom level command processor. If a higher level command processor had been started from a program that used the TRAPPING ABORTS option of the QMBasic EXECUTE statement, the program will not trap this action. The `RESET ALL` mode of LOGTO is not available in QMClient sessions.

If the VOC of the current account contains an executable item named `ON.LOGTO`, usually a paragraph, this will be executed before moving to the new account.

If the VOC of the new account contains an executable item named `LOGIN`, this will be executed on arrival in the new account.

If the LOGTO action is successful, the account name as reported by the WHO command or returned as the value of the @WHO system variable is set to the new account.

LOGTO will fail if `name` cannot be found or is not a valid account.

The QUIT command to leave QM will return to the original account directory before exiting.
4.144 LOOP / REPEAT

The LOOP and REPEAT commands define the top and bottom of a group of sentences to be repeated within a paragraph.

Format

```
LOOP
  sentence(s)
REPEAT
```

where

```
sentence(s)
```

are the sentence(s) to be executed within the loop.

The LOOP and REPEAT statements surround one or more sentences to be repeated. The loop continues until the paragraph terminates by use of STOP or ABORT, something run by the paragraph causes an abort event, or a GO statement is used to leave the loop.

Loops may be nested to any depth. Each REPEAT statement is paired with a corresponding LOOP statement and is equivalent to a GO statement that allows backward jumps to a label at the position of the LOOP statement. The behaviour of paragraphs that branch into or out of loops can be determined by this label and GO equivalent.

The keywords LOOP and REPEAT are recognised by the paragraph pre-processor and not via the VOC and hence cannot be replaced by alternative words.

Example

```
PA
LOOP
  IF <<A,File to delete>> = "" THEN GO DONE
  DELETE.FILE <<File to delete>>
REPEAT
DONE:
```

This paragraph prompts for names of files to delete until a blank line is entered. Note the need for the A control option on the first inline prompt in the loop so that the prompt is repeated on each cycle.

Because the DONE label is at the end of the paragraph, the above example could alternatively be written as

```
PA
LOOP
  IF <<A,File to delete>> = "" THEN STOP
  DELETE.FILE <<File to delete>>
REPEAT
```
4.145 MAKE.INDEX

The MAKE.INDEX command creates and builds an alternate key index. It is equivalent to use of CREATE.INDEX followed by BUILD.INDEX.

Format

MAKE.INDEX filename field(s) {options}

where

filename is the name of the file for which the index is to be built.
field(s) is one or more field names for which indices are to be created.
options are any combination of the following:
CONCURRENT
DELETING
ENCRYPT
NO.CASE
NO.NULLS
PATHNAME index.path

The MAKE.INDEX command creates the file structures to hold an alternate key index and then builds the index.

The field(s) referenced in the command must correspond to C, D, E, I, A or S-type dictionary items. The dictionary items can be deleted once the index has been constructed as all details of the indexed field are stored in the index file but this is not recommended. The value to be indexed must not exceed 255 characters. Values longer than this will not be included in the index.

Indices constructed on I or C-type dictionary items or on A or S-type items that use correlative expressions should be such that they always produce the same result when executed for the same data record. Examples of possibly invalid I-type expressions would be those that use the date or time and those that use the TRANS() function to access other files.

The NO.NULLS specifies that no entry is to be added to the index for records where the indexed field is null.

The NO.CASE option specifies that the index is to be built using case insensitive key values. The internal sort order of the index is based on the uppercase form of the data being indexed. A case insensitive index can be used with case insensitive comparison operators in the query processor but not with case sensitive operations because of the sort order difference. Conversely, a case sensitive index can be used with case sensitive comparison operators in the query processor but not with case insensitive operations.

The ENCRYPT option creates an encrypted index. This is only possible when the data file being indexed uses record level encryption. There is a significant performance penalty in using encrypted indices.

Normally, the indices are stored as subfiles in the directory that represents the data file. The PATHNAME option allows the indices to be stored in an alternative location. This might be useful, for example, to balance loads across multiple disks or to exclude indices from backups as they can always be recreated.
All indices for a single data file must be stored together. The **PATHNAME** option can be used when creating the first index and specifies the pathname of a new directory that will be created at the same time as the index. If this option is included when creating subsequent indices the `index.path` must be the same as for the first index. It is suggested that the pathname should be based on the data file name for ease of recognition.

If the **PATHNAME** option is not included, the **MAKE.INDEX** command looks for an X-type VOC record named `$INDEX.PATH` in which the second field contains the pathname of a directory under which indices are to be created by default. If found, a subdirectory with the same name as the final element of the data file pathname is created under this location. For example, when creating an index for a file with pathname `/hd1/sales/invoices`, a `$INDEX.PATH` record that contains

```
1: X
2: /hd2/indices
```

would place the indices in `/hd2/indices/invoices`.

Use of **PATHNAME DEFAULT** will ignore the `$INDEX.PATH` record, placing the indices in the same directory as the data file elements.

Index subfiles can be moved using the operating system level `qmidx` program.

Except when used with the **CONCURRENT** keyword, the **MAKE.INDEX** command requires exclusive access to the file and may take some time to complete for a very large file. It is therefore best executed at quiet times. The **CONCURRENT** keyword allows an index to be built while the file is in use. There is a performance overhead for this and the QMBasic `SELECTINDEX`, `SELECTLEFT` and `SELECTRIGHT` statements may stall waiting for the build to complete.

**Data Encryption**

Alternate key indices may be applied to files that use record level data encryption but developers should be aware that the index itself is not encrypted and hence weakens the security of the indexed fields.

Files using field level encryption cannot have indices on encrypted fields. Also, indices constructed from calculated values such as I-types that use encrypted fields will fail if the record is updated by a user that does not have access to the relevant encryption key.

**Example**

```
MAKE.INDEX ORDERS DATE
```

The above command creates and builds an index on the DATE field of the ORDERS file.

**See also:** `BUILD.INDEX`, `CREATE.INDEX`, `DELETE.INDEX`, `DISABLE.INDEX`, `LIST.INDEX`, `MAKE.INDEX`, `REBUILD.ALL.INDICES`
The MAP command produces a map of the system catalogue.

Format

```
MAP {mode} {ALL} {LPTR {n}} {FILE {file.name}} {DETAIL} {NO.PAGE}
```

where

- **mode** is one of **GLOBAL**, **PRIVATE** or **LOCAL** to indicate which catalogue is to be mapped. If no **mode** is specified, the report shows the content of the private and global catalogues.
- **ALL** indicates that system entries are to be included in the map.
- **LPTR** specifies that the output is to be sent to a print unit. The print unit number, **n**, defaults to 0 if omitted.
- **FILE** specifies that the output is to be sent to a file. If **file.name** is omitted, the $MAP file is used by default.
- **DETAIL** shows the source pathname, if available, for each reported program.
- **NO.PAGE** suppresses pagination of reports sent to the terminal.

The MAP command produces a combined list of the contents of the global and private catalogues. Without the **FILE** keyword, the map shows the name of each catalogued item with its date and time of compilation and its size, separating the object code and cross-reference tables. Items from the global catalogue have an asterisk in the leftmost column of the report. The report ends with a line giving the total size of all reported items.

The file produced with the **FILE** keyword can be listed using the query processor.

The private catalogue is normally a subdirectory, cat, under the account directory but can be moved by creating an X-type VOC entry named $PRIVATE.CATALOGUE in which field 2 contains the pathname of the alternative private catalogue directory. This only takes effect when QM is re-entered or on use of the **LOGTO** command. This feature is particularly useful where two or more accounts are to share a common private catalogue. The US spelling, $PRIVATE.CATALOG, maybe used instead. If both are present, the British spelling takes priority.

See **FIND.PROGRAM** for another method of locating the source version of a catalogued program.

Example

```
MAP ALL LPTR
```

The above command prints a map of the catalogue, including system items.

See also: **BASIC**, **CATALOGUE**, **DELETE.CATALOGUE**, **FIND.PROGRAM**
The **MED** command creates or modifies a menu definition.

**Format**

```
MED {file.name} {menu.name}
```

where

- **file.name** identifies the file holding the menu.
- **menu.name** is the name of the menu record to be processed. Menu names are case sensitive.

If the file or menu names are omitted from the command line, **MED** prompts for these. When prompting for menu names, **MED** will edit the given menu and then prompt for a further name. This is repeated until a null menu name is entered.

Menus are normally stored in the VOC file but can be stored in any file of the application designer’s choice if an R-type (remote) VOC record is used to point to the menu record in the other file.

If the menu does not exist, **MED** prompts for confirmation that it is to be created.

The display of a typical menu might appear as shown below:

```
Title : ORDER PROCESSING SYSTEM : MAIN MENU
Subr : OPS.MENU.VALIDATE
Prompt:
Exits :
Stops :

----------------------------------------
Text 1: Customer Maintenance
Action: CUST.MENU
Help : Enter customer maintenance menu
Access: CUST
Hide :
----------------------------------------
Text 2: Order Entry
Action: RUN BP ORDER.ENTRY
Help : Runs the order entry system
Access: ORDERS
Hide :
----------------------------------------
Text 3: Invoice Management
Action: INV.MENU
Help : Enter invoice management menu
Access: INVOICES
Hide :
VOC ORDERS
Access control subroutine key for this option

F1=Help
```
**MED** uses a subset of the **SED** full screen editor default key bindings to allow the user to move around within the display and enter or modify text. These are:

<table>
<thead>
<tr>
<th>Key Code</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl-A</td>
<td>Home</td>
</tr>
<tr>
<td>Ctrl-B</td>
<td>Cursor left</td>
</tr>
<tr>
<td>Ctrl-D</td>
<td>Del char</td>
</tr>
<tr>
<td>Ctrl-E</td>
<td>End</td>
</tr>
<tr>
<td>Ctrl-F</td>
<td>Cursor right</td>
</tr>
<tr>
<td>Ctrl-G</td>
<td>Cancel partially entered key sequence</td>
</tr>
<tr>
<td>Ctrl-K</td>
<td>Kill line</td>
</tr>
<tr>
<td>Ctrl-L</td>
<td>Refresh screen</td>
</tr>
<tr>
<td>Ctrl-N</td>
<td>Cursor down</td>
</tr>
<tr>
<td>Ctrl-O</td>
<td>Insert</td>
</tr>
<tr>
<td>Ctrl-P</td>
<td>Cursor up</td>
</tr>
<tr>
<td>Ctrl-V</td>
<td>Page down</td>
</tr>
<tr>
<td>Ctrl-X</td>
<td>Ctrl-X Ctrl-C</td>
</tr>
<tr>
<td>Ctrl-X</td>
<td>Ctrl-X Ctrl-S</td>
</tr>
<tr>
<td>Ctrl-Y</td>
<td>Esc-Y</td>
</tr>
<tr>
<td>Esc-V</td>
<td>Page up</td>
</tr>
<tr>
<td>Backspace</td>
<td>Backspace</td>
</tr>
<tr>
<td>Esc-H</td>
<td>F1</td>
</tr>
<tr>
<td>Ctrl-X V</td>
<td>F4</td>
</tr>
</tbody>
</table>

Because of key binding conflicts on some terminal emulations, key bindings in the second column may not be available on all terminal types.

The leftmost few characters of each line display a line type key. The remainder of the line is editable and the line on which the cursor is positioned will pan if required to allow text data wider than the display device.

The bottom line of the screen displays a single line help prompt at all times. Pressing F1 will display a help page appropriate to the line on which the cursor is positioned. Pressing F1 again while this help text is displayed will display help on the key bindings.

The first section of the menu data displayed by **MED** contains information that relates to the entire menu:

- The menu title line
- The access control subroutine name (see [VOC M-type records](#))
- The prompt text
- Exit codes
- Stop codes

The remainder of the displayed data is a set of sections corresponding to the menu options in the order in which they will appear on the menu. Each section contains:

- The option text
- The action sentence. Terminate with a semicolon for an automatic "Press return to continue" prompt
- Help text for the option
- The access subroutine key
- A flag indicating whether the option is to be hidden if it is unavailable
The separator line between each section has some special features. Pressing the return key while on this line inserts a new menu option entry under the line. Use of the kill line function (Ctrl-K) on this line deletes the menu option under the line, placing it in the kill buffer. Repeated use of the kill line function places all of the deleted menu options in the kill buffer.

When the kill buffer contains one or more complete menu options, use of the paste function (Ctrl-Y) inserts the kill buffer content under the current option. Use of kill and paste can be used to move options within the menu.

When on a text line, the kill line function deletes all text after the cursor, placing it in the kill buffer. The paste function can be used to paste this text into another line.

See VOC M-type records for more details of menu definitions.
4.148 MERGE.LIST

The **MERGE.LIST** command creates a new active select list by merging two other lists according to one of three relational operators.

**Format**

```
MERGE.LIST list1 rel.op list2 {TO tgt.list} {COUNT.SUP}
```

where

- **list1, list2** identify the select lists to be merged. These must be select list numbers in the range 0 to 10. They may not reference the same list.
- **rel.op** is the relational operator and is one of
  - **INTERSECTION** Create a new list containing only those record keys that appear in both list1 and list2. The keyword may be abbreviated to **INTERSECT**.
  - **UNION** Create a new list containing all record keys from both list1 and list2. Keys appearing in both lists appear only once in the resultant list.
  - **DIFFERENCE** Create a new list containing all record keys from list1 except those that are also in list2. The keyword may be abbreviated to **DIFF**.
- **tgt.list** is the number of the select list (0 to 10) to receive the result. If omitted, select list zero is used. It is valid for tgt.list to reference one of the source lists.
- **COUNT.SUP** indicates that display of the record count in the merged list is to be suppressed.

The **MERGE.LIST** command allows construction of one select list from two others. Use of **MERGE.LIST** can be significantly faster than a full select of the file to create the new list.

If either source list has already been partially processed before the **MERGE.LIST** command is executed, only the remaining unprocessed items are used. The resultant list will replace any already active tgt.list. The source lists are cleared after the new list has been set up. The ordering of tgt.list is undefined.

**@SYSTEM.RETURN.CODE** is set to the number of items in the new list or a negative error code.

Example
GET.LIST FRANCE.CUSTOMERS TO 1
27 records selected.
GET.LIST GERMANY.CUSTOMERS TO 2
31 records selected.
MERGE.LIST 1 UNION 2 TO 3
58 records selected.

This example restores two saved select lists, one holding keys for customers in France, the other for customers in Germany and merges these to form select list 3 as a list of customers in either of these countries.

See also:
LIST.DIFF, LIST.INTER, LIST.UNION
4.149 MESSAGE

The MESSAGE command sends a message to selected other users.

Format

MESSAGE user  {IMMEDIATE}  {message.text}
MESSAGE OFF
MESSAGE ON

where

user  identifies the user(s) to receive the message. This may be a user number, a case insensitive user name, or the keyword ALL. Messages cannot be sent to phantom or QMClient processes.

IMMEDIATE  causes immediate display of the message as described below.

message.text  is the text of the message to be sent. If omitted from the command line, the user is prompted to enter the text. With the IMMEDIATE keyword, messages that are wider than the screen are truncated on terminals that support immediate message display.

Where a specific user number is given, the MESSAGE command checks that this user is logged in and is not a phantom process. The ALL keyword sends the message to all non-phantom users (except the user sending the message).

Messages are normally displayed when the user next arrives at the command prompt. If the IMMEDIATE keyword is used and the destination process is using a terminal that supports screen save and restore (QMConsole on Windows, QMTerm, AccuTerm), the message is displayed immediately and the screen is restored when the user acknowledges the message.

The MESSAGE OFF command disables receipt of messages. Messages sent while message reception is disabled are not queued for later display and will never be seen. Use of the MESSAGE command with a user number will report an error if the target user has message reception disabled.

The MESSAGE ON command enables receipt of messages. This is the default state.

@SYSTEM.RETURN.CODE is set to zero for success or to a negative error code.
4.150 MODIFY

The MODIFY command enters the QM record modification processor.

Format

\[
\text{MODIFY \{DICT\} file.name \{ field.list \} \{ id.list \ | \text{CREATING.SEQKEY} \}}
\]

where

DICT indicates that the dictionary portion of the file is to be modified.

file.name is the name of the file to be modified.

field.list is the list of field(s) to be modified. Each entry must correspond to a D-type dictionary entry. These name may alternatively be a PH (phrase) type entry which will be expanded and all fields referenced by the phrase will be modified. If no fields are specified on the command line, MODIFY looks for a phrase named @MODIFY and, if found, uses this as the source of field names. If no @MODIFY phrase exists, MODIFY will use the @ phrase or, if this also does not exist, a default list of fields is constructed from the dictionary.

id.list is the list of records to be modified. An item is assumed to be a record id if it is not a field name defined in either the dictionary or the VOC, or if it is enclosed in quotes. If no id.list is specified, MODIFY uses the default select list or, if that is inactive, prompts for record ids. Use of CREATING.SEQKEY instead of a record id automatically generates the next sequential key for the file.

The MODIFY command provides a data editor which uses the dictionary associated with a file to determine the format in which data is displayed or entered and to provide prompts in terms which relate to the data. It is useful for making changes to existing records or entering new data. MODIFY is particularly suited to entry of dictionary records where the prompts remove the need to remember the meaning of each field.

The displayed fields are sequentially numbered for reference but these numbers are not necessarily the actual field numbers.

MODIFY prompts for a record id or uses the next item from id.list or the default select list. Entry of a question mark (?) at the id prompt will display a pick list of record ids.

If the record already exists, a list of modifiable fields is displayed. This list contains one entry for each single valued field followed by an entry for each multivalued field or associated set of fields.

The prompt displayed with the list allows the following responses:
Entry of an item number from the list selects that field or association for modification. Data may be entered or modified in a panning input area at the bottom of the screen. The edit keys available are:

- **Ctrl-A or Home**: Position cursor at the start of the data
- **Ctrl-E or End**: Position cursor at the start of the data
- **Ctrl-B or Left**: Move the cursor left one character
- **Ctrl-F or Right**: Move the cursor right one character
- **Ctrl-D or Del**: Delete the character under the cursor
- **Backspace**: Delete the character to the left of the cursor
- **Ctrl-K**: Delete all characters from the cursor position onwards
- **Ctrl-O or Ins**: Toggle overlay / insertion mode
- **Ctrl-Q**: Quote character. The next character is inserted without interpretation as a command. If the character is V, S or T, a value, subvalue or text mark is inserted.
- **Return**: Accept the entered data
- **Ctrl-X**: Abort entry, returning to the field list
- **F1**: Display help screen

Non-printing characters can be inserted using the Ctrl-Q prefix shown above or by typing ^nnn where nnn is the ASCII character number of the character to be inserted. On an ECS mode system, use of ^Xnnnn allows entry of a four hexadecimal digit character value.

As fields are modified, their values are inserted into the displayed list of fields.

**FI**  
Writes the modified record to the file and prompts for a new record id. If the record was a dictionary I-type or C-type, **MODIFY** will compile it.

**Q**  
Quits from the record, discarding any changes and prompts for a new record id.

**X**  
Quits from the record, discarding any changes and exits without moving to the next item in a select list.

**N**  
Displays the next page of fields available for modification.

**P**  
Displays the previous page of fields available for modification.

?  
Displays a brief expansion of the available options.

Selecting a multivalued field or an association enters a separate display screen showing one line for each value in the field(s).

The prompt displayed with the list allows the following responses:

**line no**  
Entry of a line number from the list selects that value set for modification. Data may be entered or modified for each field in the association in turn in a panning input area at the bottom of the screen. The edit keys available are as above.
Dn Deletes the value set at line n.
In Inserts a new value set at line n.
E Extends the values, repeatedly accepting new data until either a null entry is made in the first field of the association or the exit key (Ctrl-X) is used.
N Displays the next page of values available for modification.
P Displays the previous page of values available for modification.
? Displays a brief expansion of the available options.

Where the record does not already exist, MODIFY prompts for data for each field in turn and then allows changes as for an existing record.

When editing a dictionary, MODIFY automatically chooses the editable fields based on the record type.
### 4.151 NLS

The **NLS** command sets or reports national language support settings.

**Format**

- `NLS` Report all settings
- `NLS { DEFAULT }` Set defaults
- `NLS { key }` Report setting for given parameter
- `NLS { key value }` Set value for given parameter

where

- `key` identifies the parameter to be set or reported.
- `value` is the new value for the parameter.

The **NLS** command sets or reports national language parameter values. The available parameters and their default values are:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENCY</td>
<td>$</td>
<td>Maximum 8 characters</td>
</tr>
<tr>
<td>THOUSANDS</td>
<td>,</td>
<td>Thousands separator character</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>.</td>
<td>Decimal separator character</td>
</tr>
<tr>
<td>IMPLICIT.DECIMAL</td>
<td>.</td>
<td>Decimal separator character for implicit data conversions. See <a href="#">SETNLS</a> for details.</td>
</tr>
</tbody>
</table>

Setting two or more of the national language support parameters to the same character will have undefined effects.

Setting the DECIMAL character also sets the IMPLICIT.DECIMAL character to the same value. When using both, the DECIMAL character must be set first.

**See also:**

[SETNLS](#)
4.152 NSELECT

The **NSELECT** command refines a select list by removing items that are in a named file.

**Format**

```
NSELECT { DICT } file { FROM from.list } { TO to.list }
```

where

- **DICT** indicates that the dictionary portion of the file is to be used.
- **file** identifies the file to be processed.
- **from.list** is the select list to be used as the source of record ids to be checked against **file**. If omitted, the default list (list 0) is used.
- **to.list** is the select list to be receive the modified list. If omitted, the default list (list 0) is used.

The **NSELECT** command refines the source select list by removing from it any items that correspond to record ids present in the named file.

**Example**

```
SELECT VOC
NSELECT NEWVOC
LIST VOC
```

The above sequence of commands builds a list of all records in the VOC file, removes from this list all items present in NEWVOC and then lists the remaining VOC records. The effect is to show those VOC records added since the account was created.
4.153 OPTION

The OPTION command sets, clears or displays configurable options.

To ease application portability, options that are not meaningful on a particular QM platform are ignored.

Format

```
OPTION {option.name... {ON | OFF | DISPLAY | LPTR {unit}}} }
OPTION Xhex.string
OPTION ALL OFF
```

The OPTION command, normally only used in the LOGIN or MASTER.LOGIN paragraphs, sets options that determine how the system behaves for that user session. Beware that setting options in the MASTER.LOGIN paragraph will affect users of the QMSYS account and may cause some administrative commands to fail.

The ON keyword is used to set an option and is the default action if no keyword is present. The OFF keyword is used to clear an option. The DISPLAY keyword is used to display the current setting of an option.

The OPTION command with no qualifying information displays the settings of all options. The LPTR keyword directs this report to the specified print unit, printer zero if unit is omitted. When displaying all options, the final line is of the form

```
Encoded: X1141E0EOEO4
```

where the hexadecimal value can be used as the hex.string element in an OPTION command to set the options to the same state as at the time of display. This can be of use in the LOGIN paragraph to set all options to the required state in a single operation.

A single command may contain multiple option names. The ON, OFF or DISPLAY keyword, if present, may appear at any position within the list of option names.

The OPTION ALL OFF syntax turns off all options. It is useful in LOGIN paragraphs to ensure that all options are off before turning on those that are required in applications that may use LOGTO to move between accounts.

Option settings are not automatically inherited by phantom processes unless the INHERIT option described below is active.

The available options are:

- AMPM.UPCASE: Causes the am/pm suffix displayed by some time conversions to appear in uppercase instead of the default lowercase.
- ASSOC.ALL: Causes the query processor to ignore any definitions of associations in the dictionary, treating all fields as being in the same association. Use of this option is not recommended but it may be of use when migrating applications in which the associations are not properly defined. The preferred action in this case is to correct the dictionaries.
<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSOC.UNASSOC.MV</td>
<td>Treats all multivalued fields for which no association is defined in the dictionary as being associated together. This provides close compatibility with Pick style systems but may lead to unintentional association of unrelated fields.</td>
</tr>
<tr>
<td>BACKSLASH.NOT.QUOTE</td>
<td>Causes the command processor to treat backslashes as data characters instead of being recognised as string quotes.</td>
</tr>
<tr>
<td>CATALOGUE.LOCAL</td>
<td>Use local mode cataloguing by default.</td>
</tr>
<tr>
<td>CHAIN.KEEP.COMMON</td>
<td>Retains the unnamed common block and command processor level on use of <code>CHAIN</code>.</td>
</tr>
<tr>
<td>CHAINED.SELECT</td>
<td>If set, a query processor select operation that selects at least one record will execute any command in the <code>DATA</code> queue on completion. If no records are selected, the data queue is cleared.</td>
</tr>
<tr>
<td>CLEAR.SELECT</td>
<td>Clear the default select list on return to the command processor except for commands that intentionally leave an active select list. See the QMBasic <code>KEEP.SELECT</code> statement.</td>
</tr>
<tr>
<td>COMMAND.LEVEL.LOCKS</td>
<td>Associates record and file locks with the command processor level, releasing the locks at that level on completion of a command.</td>
</tr>
<tr>
<td>CORRELATIVE.NOCASE</td>
<td>Causes string operations in correlative expressions to be performed in a case insensitive manner. This option is applied when the correlative is executed, not when it is compiled.</td>
</tr>
<tr>
<td>CORRELATIVE.REUSE</td>
<td>Causes all operators in correlative expressions to behave as though the reuse (R) flag was present. This option is applied when the correlative is executed, not when it is compiled.</td>
</tr>
<tr>
<td>CRDB.UPCASE</td>
<td>Causes the cr/db suffix displayed by some decimal conversions to appear in uppercase instead of the default lowercase.</td>
</tr>
<tr>
<td>CREATE.DICT.NO.CASE</td>
<td>Causes <code>CREATE.FILE</code> to create file dictionaries with case insensitive record ids by default.</td>
</tr>
<tr>
<td>CREATE.FILE.NO.CASE</td>
<td>Causes <code>CREATE.FILE</code> to create files with case insensitive record ids by default.</td>
</tr>
<tr>
<td>DEBUG.REBIND.KEYS</td>
<td>Causes the QMBasic debugger to rebind the function keys on entry, replacing any user defined bindings with those specified in the terminfo entry for the current terminal type.</td>
</tr>
<tr>
<td>DEFAULT.MV</td>
<td>Causes C, D or I-type dictionary items in which the single/multivalue flag is blank to be treated as multivalued instead of defaulting to single valued.</td>
</tr>
<tr>
<td>DIR.DTM</td>
<td>If enabled, writes to a directory file or closing a sequential file that has been written will update the date/time modified of the directory. (Not Windows)</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>DIR.SEL.OMIT.HIDDEN</td>
<td>Omit hidden files when building a select list of items in a directory file.</td>
</tr>
<tr>
<td>DIV.ZERO.WARNING</td>
<td>Attempts to divide by zero in QMBasic programs should report a warning rather than a fatal error. The division will return a zero result. This option should only be used during application development as it may cause faulty applications to appear to work correctly.</td>
</tr>
<tr>
<td>DUMP.ON.ERROR</td>
<td>Causes generation of a process dump file at a process abort such as a run time fatal error.</td>
</tr>
<tr>
<td>ED.NO.QUERY.FD</td>
<td>Suppresses the confirmation prompt in the ED editor when using the FD command or its synonym DELETE.</td>
</tr>
<tr>
<td>FORCE.RELOAD</td>
<td>Force reload of QMBasic object code at compilation or catalogue. See the QMBasic CALL statement for more information.</td>
</tr>
<tr>
<td>INHERIT</td>
<td>Phantom processes will inherit the option settings of the parent process. Use of an OPTION command in the MASTER.LOGIN or LOGIN paragraphs of the phantom process may modify these settings.</td>
</tr>
<tr>
<td>INHERIT.MFILE.PATH</td>
<td>Causes CREATE.FILE to create new multifile subfiles in the same directory as the default subfile unless the PATHNAME or IN option is used.</td>
</tr>
<tr>
<td>INHERIT.OWNERSHIP</td>
<td>Causes CREATE.FILE to inherit ownership (user id and group id) from the account unless overridden by command options. This option has no effect on Windows systems.</td>
</tr>
<tr>
<td>KEEP.FILENAME.CASE</td>
<td>Causes CREATE.FILE to preserve the casing of the QM name of the file when creating the operating system directories that represent this file.</td>
</tr>
<tr>
<td>KEEP.OLD.OBJECT</td>
<td>Causes the QMBasic compiler to retain any previous version of the compiled object code if a compilation fails.</td>
</tr>
<tr>
<td>LOCAL.CAT.RELATIVE</td>
<td>Changes the default behaviour of local cataloguing to store the relative pathname of the program in the VOC entry.</td>
</tr>
<tr>
<td>LOCK.BEEP</td>
<td>Emits a beep at the terminal once per second while waiting for a record or file lock.</td>
</tr>
<tr>
<td>NO.DATE.WRAPPING</td>
<td>Suppresses rolling of dates with over-large day numbers into the following month on input conversion.</td>
</tr>
<tr>
<td>NO.ECHO.DATA</td>
<td>Suppresses echoing of data read by applications from the DATA queue. Equivalent to compiling all programs with the NO.ECHO.DATA option of the $MODE compiler directive.</td>
</tr>
<tr>
<td>NO.PHANTOM.NOTIFY</td>
<td>Used in a phantom process, this option suppresses the message normally sent to the parent process when the phantom terminates.</td>
</tr>
<tr>
<td>Feature</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>NO.SEL.LIST.QUERY</td>
<td>Suppresses display of the confirmation prompt in commands that take an optional select list of records to process. This is equivalent to use of the NO.QUERY option to those commands.</td>
</tr>
<tr>
<td>NO.SEL.WHENFILTER</td>
<td>Omit WHEN clause filtering in SELECT for compatibility with releases prior to 3.3-0.</td>
</tr>
<tr>
<td>NO.USER.ABORTS</td>
<td>Suppresses all options that allow a user to generate an abort event. These are: the &quot;Press return to continue&quot; prompt, the pagination prompt when using the SCROLL keyword of the query processor, and the break key options.</td>
</tr>
<tr>
<td>NON.NUMERIC.WARNING</td>
<td>QMBasic programs attempting to use a non-numeric value where a number is required should use zero and report a warning rather than a fatal error. The operation will return a zero result. This option should only be used during application development as it may cause faulty applications to appear to work correctly.</td>
</tr>
<tr>
<td>NUMERIC.EXPONENT</td>
<td>When enabled, data of the form 123.45E-2 will be accepted as numeric.</td>
</tr>
<tr>
<td>PAGINATE.ON.HEADING</td>
<td>When used with the RUN.NO.PAGE option, setting a page heading or page footing will cause output pagination to be enabled. The NO.PAGE option to the RUN command can be used to override this.</td>
</tr>
<tr>
<td>PICK.BREAKPOINT</td>
<td>Causes the query processor to recognize Pick style syntax for the BREAK.ON and BREAK.SUP keywords where the optional text qualifier follows the field name rather than appearing before it.</td>
</tr>
<tr>
<td>PICK.BREAKPOINT.U</td>
<td>Causes the query processor to handle the U breakpoint option differently. See the BREAK.ON and BREAK.SUP keywords for further details.</td>
</tr>
<tr>
<td>PICK.EXPLODE</td>
<td>When using BY.EXP, if an associated field has only one value, do not explode this field.</td>
</tr>
<tr>
<td>PICK.GRAND.TOTAL</td>
<td>Causes the query processor to display the text of the GRAND.TOTAL keyword on the same line as the total values.</td>
</tr>
<tr>
<td>PICK.IMPLIED.EQ</td>
<td>Causes the query processor to handle a selection clause that has no operator between the field name and a literal value enclosed in double quotes as though there was an EQ operator.</td>
</tr>
<tr>
<td>PICK.ML.CONV.MASK</td>
<td>Allows parenthesised format masks in the ML and MR conversion codes, disabling recognition of a left parenthesis as requesting negative values to be output in round brackets.</td>
</tr>
<tr>
<td>PICK.NULL</td>
<td>Causes the ML and MR conversion codes and format expressions that use options applicable to numeric data to return a null string for null data instead of zero.</td>
</tr>
<tr>
<td>PICK.PROC</td>
<td>D3 compatibility for PQ style Procs. Use of this option implies the PROC.A option. Note that the</td>
</tr>
</tbody>
</table>
features covered by this option may change in future releases.

**PICK.WILDCARD**
Causes the query processor to recognise Pick style wildcards in equality tests as an alternative to the **LIKE** operator.

**PROC.A**
Causes the Proc $A(n,m)$ command not to terminate copying data at the end of the field.

**PROC.IF.NO.CASE**
Makes the Proc IF command case insensitive.

**QUALIFIED.DISPLAY**
Causes the query processor to recognise Pick style qualified display clauses.

**QUERY.FOOTER.PAUSE**
Pause for user confirmation before continuing at the end of a query that uses a page footer. This option is ignored in query commands that use the **NO PAGE** qualifier.

**QUERY.MERGE.PRINT**
Causes a query report directed to a printer to merge the output as continuation of the active print job if the printer is already active, leaving it active on completion of the report. Without this option, the printer is closed at the end of the report.

**QUERY.NO.CASE**
Causes the query processor to perform selection operations and sorting in a case insensitive manner.

**QUERY.PRIORITY.AND**
Causes the **AND** operator to take priority over the **OR** operator in query processor commands. This does not affect the behaviour of these operators in QMBasic programs.

**QUERY.SORT.NO.CASE**
Causes all sort operations in the query processor to be case insensitive.

**QUERY STRING .COMP**
Forces all EQ and NE operator comparisons in query processor selection clauses to be performed as string comparisons regardless of the data type. See also the QUERY STRING .COMP .ALL option.

**QUERY .STR .COMP .ALL**
Forces all relational operator comparisons in query processor selection clauses to be performed as string comparisons regardless of the data type. See also the QUERY STRING .COMP option.

**RUN.NO.PAGE**
Causes the **RUN** and **DEBUG** commands to start the program with screen pagination disabled. This option also affects user catalogued programs. If the **PAGINATE.ON**.HEADING option is set, pagination will be enabled if the application sets a page heading or page footing unless the **NO.PAGE** option has been used when starting the program.

**SELECT.KEEP.CASE**
Causes QM to preserve the case of record ids when building a select list from a directory file on an operating system that uses case insensitive file names. This currently only affects Windows systems.
SHOW.STACK.ON.ERROR Displays the call stack at a fatal program error, showing the program name, line number (where available), and object code address.

SILENT.TANDEM Allow TANDEM monitoring without a confirmation prompt.

SPACE.MCT Modifies the behaviour of the MCT conversion code such that only the first character and letters immediately after a space are converted to uppercase.

SPOOL.COMMAND Displays the operating system command used to perform printing (Diagnostic aid, not Windows).

STACKED.ACCOUNT Causes a QMBasic EXECUTE statement that switches accounts to revert to the previous account on return to the calling program.

SUPPRESS.ABORT.MSG Suppresses display of program location diagnostic information when a QMBasic ABORT statement is executed.

UNASS.WARNING Unassigned variables in QMBasic programs should report a warning rather than a fatal error. This option should only be used during application development as it may cause faulty applications to appear to work correctly.

WITH.IMPLIES.OR In a query containing multiple WITH clauses, there is an implied OR rather than the default implied AND between these clauses.

Special short form options

The OPTION command allows use of shortcut names to enable multiple options in a single step. Other options that were already set when these are used will not be cleared. These names cannot be used with ON, OFF or DISPLAY. The shortcuts that give closest compatibility when migrating an application to QM are described under "Migrating to QM". The four shortcut names below are retained from QM version 3.4-17 and earlier.

D3 This option gives closer compatibility with D3. The options included are ASSOC, UNASSOC, MV, BACKSLASH, NOT, QUOTE, CATALOGUE, LOCAL, CHAINED, SELECT, DEFAULT, DIV, ZERO, WARNING, KEEP, OLD, OBJECT, LOCK, BEEP, NO, DATE, WRAPPING, NON, NUMERIC, WARNING, PAGINATE, ON, HEADING, PICK, BREAKPOINT, PICK, BREAKPOINT, U, PICK, EXPLODE, PICK, GRAND, TOTAL, PICK, IMPLIED, EQ, PICK, ML, CONV, MASK, PICK, NULL, PICK, PROC, PICK, WILDCARD, PROC, A, QUALIFIED, DISPLAY, QUERY, NO, CASE, QUERY, PRIORITY, AND, RUN, NO, PAGE, UNASS, WARNING and WITH, IMPLIES, OR but may change in future releases.
QM Commands

MVBASE

This option gives closer compatibility with mvBase. The options included are AMPM.UPCASE, ASSOC.UNASSOC.MV, BACKSLASH.NOT.QUOTE, CATALOGUE.LOCAL, CHAINED.SELECT, CRDB.UPCASE, DEFAULT.MV, KEEP.OLD.OBJECT, LOCK.BEEP, NO.DATE.WRAPPER, PAGINATE.ON.HEADER, PICK.BREAKPOINT, PICK.EXPLODE, PICK.GRAND.TOTAL, PICK.IMPLIED.EQ, PICK.ML.CONV.MASK, PICK.NULL, PICK.PROC, PICK.WILDCARD, PROC.A, QUALIFIED.DISPLAY, QUERY.PRIORITY.AND, RUN.NO.PAGE and SELECT.KEEP.CASE but may change in future releases.

PICK

This option gives closer compatibility with generic Pick systems. The options included are ASSOC.UNASSOC.MV, PICK.BREAKPOINT, PICK.BREAKPOINT.U, PICK.EXPLODE, PICK.GRAND.TOTAL, PICK.NULL, PICK.WILDCARD, QUALIFIED.DISPLAY and WITH.IMPLIES.OR and these will not change in future releases.

QMBASIC.WARNINGS

This option sets DIV.ZERO.WARNING, NON.NUMERIC.WARNING and UNASS.WARNING options.

See also:
OPTION()
4.154 OSCOPY

The OSCOPY command copies an operating system level file or directory to a specified location.

**Format**

```
OSCOPY src.path TO tgt.path {APPENDING} {NO.QUERY}
```

where

- `src.path` is the pathname of the file or directory to be copied.
- `tgt.path` is the pathname of destination.

The OSCOPY command provides an easy way to copy an operating system file or directory.

In all cases, `tgt.path` is the pathname of the new file or directory, not the parent directory into which items will be copied. Any existing item will be overwritten.

The **APPENDING** option applies only when `src.path` is a file and causes the data to be appended to `tgt.path` if the output file already exists.

The **NO.QUERY** option suppresses the confirmation prompt asking whether the target directory should be created if it does not already exist.

The OSCOPY command locks each item as it progresses. When copying the directory that represents a QM hashed file, it is essential that the file is not open in any QM process as this may result in incorrect data being copied.

Note that failure may occur at any stage of copying a directory structure and updates performed before the error will not be undone.

**Example**

```
OSCOPY C:\SALES TO C:\BACKUPS\SALES
```

The above command copies `C:\SALES` (perhaps an operating system directory representing a QM hashed file) to a subdirectory of the same name under `C:\BACKUPS`.

See also

![OSCOPY()](opensm480)
4.155 PACKAGE.LICENCE

The PACKAGE.LICENCE command, available only in the QMSYS account, applies new licence details for a registered package.

**Format**

```bash
PACKAGE.LICENCE
PACKAGE.LICENCE pkg.name auth.code
PACKAGE.LICENCE pkg.name CANCEL [NO.QUERY]
```

where

- `pkg.name` is the name of a registered package.
- `auth.code` is a licence authorisation code for the named package.

The PACKAGE.LICENCE command allows a user to inspect, install or remove package licences.

With no command line options, the PACKAGE.LICENCE displays a list of currently installed package licences and allows the user to modify, delete, or add licence details. The cursor up and down keys are used to position on the line to be amended. Ctrl-P or ctrl-Z can also be used to move up and ctrl-N can be used to move down. The return key is then used to move between fields.

Care should be taken to enter the information exactly as it appears on the licence information sheet.

To remove a package licence, clear the package name field.

With the package name and authorisation code on the command line, the PACKAGE.LICENCE command applies this licence, possibly replacing a previous licence for the same package.

With the CANCEL keyword in place of an authorisation code, the PACKAGE.LICENCE command deletes an existing licence. The NO.QUERY option suppresses the confirmation prompt.

**Examples**

```bash
PACKAGE.LICENCE SJD JJHH-YJBF-MMND-HDKS
```

The above command would add a licence for the package named SJD using the authorisation code provided on the command line.

Use of interactive mode after this licence has been applied would initially show the screen below.

```
PACKAGE LICENCE ADMINISTRATION
Package name...  Authorisation code.  Licence
  01>SJD            JJHH-YJBF-MMND-HDKS  5 users, expires 12 Dec 13
  02
```
See also:
Package licensing, ENTER.PACKAGE(), EXIT.PACKAGE(), !PACKAGE()
4.156 PASSWORD

The PASSWORD command changes the password for a QM user on systems that use QM's user security features.

Format

    PASSWORD {username}

The PASSWORD command prompts for the existing password of the user and then prompts twice for the new password. Terminal echo is suppressed during password entry.

The username argument may only be used by users with administrator rights and allows changes to the password of a user other than that executing the command.

See also:
ADMIN.USER, CREATE.USER, DELETE.USER, LIST.USERS, SECURITY, Application Level Security
4.157 PAUSE

The PAUSE command displays a "Press return to continue" prompt.

Format

    PAUSE

The PAUSE command, intended for use in paragraphs, pauses processing and displays a prompt for user input before continuing. By default, this prompt offers two special responses; A to abort and Q to quit. If the NO.USER.ABORTS mode of the OPTION command is active, the A option is not offered.
4.158 PDEBUG

The PDEBUG command runs the phantom debugger.

**Format**

```
PDEBUG {command}
PDEBUG USER userno
```

where

- `command` is the command to be executed by the phantom process.
- `userno` is the QM user number of the phantom process to be debugged.

The PDEBUG command allows debugging of a QMBasic program in a phantom or QMClient process using the same debugger interface as for foreground processes.

The first form of the PDEBUG command waits for a phantom or QMClient process running in the same account and as the same user name to attempt to enter the debugger. At that point, the process executing the PDEBUG command will enter the QMBasic debugger and can use this in the usual way except that it is not possible to view the application screen because a phantom process is not associated with a terminal device.

The phantom process to be debugged may be started separately or by use of the `command` option to the PDEBUG command.

The second form of the PDEBUG command forces the phantom process with the specified user number into debug mode. It is equivalent to the application executing a QMBasic `DEBUG` statement. Actual entry to the phantom debugger will occur when the process next executes a statement in a program compiled in debug mode.

If a phantom process attempts to enter the debugger but there is no PDEBUG command waiting to trap this situation, the phantom process will normally be terminated. Alternatively, this situation can be trapped as a SYS.PROGRAM.NO_PDEBUG exception in the phantom process. This exception is only thrown if there is a handler to catch it.
4.159 PDUMP

The **PDUMP** command generates a process dump file for a specified QM process.

**Format**

```
PDUMP userno
```

where

```
userno
```

is the QM user number of the process to dump.

The **PDUMP** command forces generation of a [process dump file](#) for the user identified by `userno`.

Because use of the **PDUMP** command can weaken system security by allowing a user to see data inside another user's program, the **PDUMP** configuration parameter may be used select a mode where only system administrators can dump processes running under other user names. For alternative security rules, add a [security subroutine](#) to the relevant VOC entry.
The **PHANTOM** command starts execution of a verb, sentence or paragraph as a background process.

**Format**

```
PHANTOM {options} command
```

where

- **options** are selected from the following items described in detail below:
  - **ACCOUNT name** Requests that the phantom process runs in a specific account.
  - **DATA** Passes the data queue of the parent process to the phantom.
  - **GROUP** Specifies that the phantom process is to be grouped with its parent.
  - **LOGGING name** Specifies a non-default name for the log file.
  - **NO.LOG** No log file is to be produced.
  - **NO.MSG** No start or termination message to be displayed in parent process.
  - **POOL name** Makes the phantom process a member of the named connection pool.
  - **USER n** Requests that the phantom runs as a specific user number.

- **command** is the command to be executed by the phantom process.

A new background process is started to execute the command which must be a valid verb, sentence or paragraph. The process from which the **PHANTOM** command was performed continues without waiting for command to be completed. A message is displayed indicating the user number associated with the phantom. The user number is also returned in `@SYSTEM.RETURN.CODE`. If the phantom process cannot be started, `@SYSTEM.RETURN.CODE` holds the negative error code.

When using the **POOL** option, the system will first look for an idle process in that pool that can be awakened. If no such process is found, a new phantom is started. The pool name may be followed by the keyword **NEW.PROCESS** to force creation of a new process even if an idle process exists. This allows an application to start a number of phantom processes that perform initialisation and then go into the idle pool waiting for work to do.

When the background process terminates a message is queued for display immediately before the next command prompt. This message is

```
Phantom n : Normal termination.
```

where n is the user number if the process completed successfully or

```
Phantom n : Abnormal termination.
```

if the process aborted.
The default behaviour of the phantom process is to create a record in the $COMO file named $PHn_date_time exactly as though it had commenced with the command

`COMO ON PHn_date_time`

This record will contain all output from the phantom process that would be directed to the terminal in an interactive process. The logging file may be switched off or redirected as desired from within the phantom process.

Use of the **NO.LOG** keyword causes the phantom process to run with no log file. Alternatively, the **LOGGING** keyword may be used to specify the name of the log file record to be created. If name is a quoted string containing a space, the portion before then space must correspond to a VOC file definition record for a directory file and the portion after the space will be used as the record name within that file.

Except when the **NO.LOG** or **LOGGING** option is used, the **PHANTOM** command checks for the presence of a catalogued subroutine named PHLOGNAME. If found, it is called with two arguments: the first through which it return an alternative log location, the second passed in as the command to be executed by the phantom process. The return value must be a file name and record name, separated by a single space where the file name must correspond to an already existing directory file on the QM server. This feature allows automatic selection of alternative log file locations and provides the opportunity for compatibility with log file naming conventions used by other systems.

For example, the phantom log record strategy used by UniVerse can be achieved with a PHLOGNAME subroutine as below.

```qmb
subroutine phlogname(result, command)$catalogue   result = '&PH& ' : upcase(field(command, ' ',1)):'_':time():'_':date()   returnend
```

In all cases, a QMBasic program can determine the log record location for an active phantom by use of the **!PHLOG()** subroutine.

The **ACCOUNT** option specifies the name of the account in which the phantom process is to start execution, defaulting to the current account if omitted. The user must not be barred from access to this account or forced into some other account by the application security controls.

The **DATA** keyword causes the **DATA queue** of the process in which the **PHANTOM** command is executed to be passed into the phantom process, clearing it in the parent process.

The **GROUP** keyword causes the phantom process to be grouped with its parent such that, if the parent process terminates, the phantom is automatically logged out.

The **USER** option allows the phantom process to be created as a specific user number within a range of numbers reserved by the **PHANTOMS** configuration parameter. An error will occur if there is already a phantom running as the specified user.

Any attempt to read data from the keyboard will cause the process to abort. **DATA** statements may be used in the phantom to supply input that would normally be read from the keyboard.

Phantom processes may not be started within a transaction.

All QM processes, including phantoms, execute the VOC LOGIN paragraph, if it exists. To exit from the LOGIN paragraph for a phantom process, insert a line

```qmb
IF @TTY = 'phantom' THEN STOP
```

```
at the relevant point in the paragraph. See \texttt{@TTY} for more details.

Any \texttt{secure encryption keys} enabled in the parent process are inherited by the phantom process.

\textbf{Example}

\begin{verbatim}
PHANTOM BASIC BP INVOICE
\end{verbatim}

This command starts a phantom process to compile the QMBasic program INVOICE in the BP file.

\textbf{See also:}
\begin{verbatim}
CHILD(), LIST.PHANTOMS, !PHLOG(), STATUS
\end{verbatim}
4.161 PRINTER

The PRINTER command provides control for print units.

Format

PRINTER \{print.unit\} action

The print.unit argument identifies the print unit on which the action is to occur and must be in the range -1 to 255. If omitted, print unit zero is used.

The action may be any of the following. Multiple actions may be specified in a single PRINTER command and will be performed in the order in which they occur on the command line.

- **AT printer.name**  
  Output is directed to the named printer

- **BOTTOM.MARGIN n**  
  Sets the bottom margin size.

- **CLOSE**  
  The print unit is closed.

- **FILE filename recordname**  
  Selects the destination for printed output.

- **LEFT.MARGIN n**  
  Sets the left margin size.

- **LINES n**  
  Sets the number of lines per page.

- **KEEP.OPEN**  
  Keeps the printer open to merge successive printer output.

- **QUERY**  
  Reports the current settings

- **RESET**  
  Resets to the default parameter values.

- **TOP.MARGIN n**  
  Sets the top margin size.

- **WIDTH n**  
  Sets the number of characters per line.

The PRINTER command sets or reports the settings of printer control parameters. The action of each keyword is described in detail below.

**PRINTER print.unit AT printer.name**  
Subsequent output is directed to the named printer. The printer.name must be a printer defined in Windows.

**PRINTER print.unit BOTTOM.MARGIN n**  
Sets the bottom margin size. On reaching the foot of the page, $n$ blank lines will be output to reach the start of the next page. This value defaults to 0 and is reset on closing a print unit.

**PRINTER print.unit CLOSE**  
The print unit is closed. This action overrides any previous use of the KEEP.OPEN option. Note that the QMBasic PRINTER CLOSE statement does not cancel the KEEP.OPEN mode as this is the purpose of KEEP.OPEN.
If this print unit was directed to a spool file, the data will be printed. Any heading and footing text or file name associated with the printer is discarded and further use of this print unit by a program or command will start a new file.

**PRINTER** `print.unit FILE filename recordname`

Selects the destination for printed output. Output to print units 1 to 255 is normally directed to a hold file. This command associates a record of a directory file with the print unit. The record is not created until the first output is directed to the print unit.

**PRINTER** `print.unit KEEP.OPEN`

Allows merging of successive printer output into a single print job. Any request from a program to close the print unit clears the heading and footing but leaves the print job open to receive further output. The print unit is finally closed, and the job printed, by using the `CLOSE` option to this command. On Windows systems it may be important that the Print Manager option to start printing while a print job is being created is disabled as this could result in the printer being assigned to an incomplete job.

**PRINTER** `print.unit LEFT.MARGIN n`

Sets the left margin size. Each line will be indented by `n` spaces. This value defaults to 0 and is reset on closing a print unit.

**PRINTER** `print.unit LINES n`

Sets the number of lines per page. No validation of the value of `n` is performed. The effect of specifying a number of lines per page greater than that of the physical device on which the data is subsequently printed is undefined. This value defaults to 66 and is reset on closing a print unit.

**PRINTER** `print.unit QUERY`

Reports the current settings of the width, lines per page, top margin, bottom margin and left margin.

**PRINTER** `print.unit RESET`

Resets to the default values for width, lines per page, top margin, bottom margin and left margin. This function does not affect any file association.

**PRINTER** `print.unit TOP.MARGIN n`

Sets the top margin size. Each page of output will commence with `n` blank lines. This value defaults to 0 and is reset on closing a print unit.

**PRINTER** `print.unit WIDTH n`

Sets the number of characters per line. No validation of the value of `n` is performed. The effect of specifying a width greater than that of the physical device on which the data is subsequently printed is undefined. This value defaults to 80 and is reset on closing a print unit.

See also:
**PRINTER** (QMBasic). **SETPTR**
4.162 PROFILER

The PROFILER command handles all actions relating to the application event profiler.

Format

PROFILER ON {USER n} {mode...} Start logging
PROFILER OFF {USER n} Stop logging and dump data to disk
PROFILER {DISPLAY}{USER n} Display logged data. The DISPLAY keyword is optional.

When using the ON or OFF keywords, if the USER qualifier is absent, it defaults to the user performing the command. For the DISPLAY mode, if no user number is specified in the command, a list of available profiles is displayed.

The PROFILER ON command starts collection of event log data. The optional mode qualifier specifies the event type(s) to be recorded. It may be any combination of the following and defaults to recording all events:

- CALL Calls to QMBasic subroutines
- OPEN QMBasic OPEN and OPENPATH statements
- READ QMBasic READ statements, including locking variants and MATREAD
- WRITE QMBasic WRITE statements
- DELETE QMBasic DELETE statements
- OPENSEQ QMBasic OPENSEQ statements
- READSEQ QMBasic READSEQ statements
- WRITESEQ QMBasic WRITESEQ statements
- EXECUTE QMBasic EXECUTE statements. Only the first 32 characters of the command are logged.
- OBJECT Calls to public subroutines/functions in a class module.

The PROFILER OFF command terminates logging and writes the data to a file named profile.uuu in the QM temporary directory where uuu is the user number. Note that although this appears to be a text file, it may contain both field marks and newline sequences and hence may not behave as expected in user written analysis tools. No assumptions should be made regarding the content of the log file as it may change between QM releases.

The PROFILER DISPLAY command displays the collected data. The initial display shows the event types that appear in the logged data together with the number of occurrences of each event type. The cursor down/up keys may then be used to select an event type. The return key will drill down into the data, displaying the qualifying data associated with each logged event of the selected type. In the same way, it is then possible to select and drill down into a list of programs that logged the selected event type and qualifying data combination. A final step, only available if the program was compiled in debug mode, allows the user to drill down to a list of line numbers within the selected program at which the logged event occurred. For non-debug mode programs, the line number will appear as zero. If there are no line numbers recorded for any displayed program, a
message is displayed instead of the final screen. At each level, the backspace key can be used to return to the previous level.

See also:
Application Profiling
4.163 PSTAT

The PSTAT command displays the status of one or all QM processes.

Format

\[
PSTAT \{ \{USER\} user \} \{ LEVEL level \}
\]

where

- **user** is the QM user number or user login name of the process to report. Multiple user numbers or user names may follow a single USER keyword or there may be multiple USER keywords in the command line. If the command starts with numeric data that is not preceded by USER or LEVEL, the data is assumed to be a user number.

- **level** is the reporting level.

The PSTAT command displays diagnostic status information about the processes identified by the USER option or, if the USER option is omitted, all QM processes.

For each process reported, PSTAT shows the account name, user name, the last command executed and the current execution point (program name, line number and execution address). Note that the command shown may have completed and returned to the program from which it was executed. If in a transaction, the transaction id, command processor level and start time are shown, repeating for parent transactions if appropriate.

The level parameter specifies extended report features and is formed by adding together the following components:

1. Shows each program and subroutine in the call stack. If not included, only the currently active program is shown.
2. Shows internal subroutine calls within each reported program and subroutine. If not included, only external subroutine calls are shown.
3. Shows details of QMNet connections to remote servers.

Adding Application Data

A QM application can set the @PSTAT variable to contain text that will be shown by the PSTAT command. This provides an easy way to display diagnostic data or other application status information.

Examples

```
PSTAT USER 2 LEVEL 3
User Detail
  2 Account: SALES, Username: Joe, Pid: 1383
  Command: RUN INVOICES
    !SCREEN 953 (14E6, KEYIN)
    750 (118E)
    450 (08F3)
    323 (061B)
```
In this example, the most recent command executed by user 2 was RUN INVOICES. It is currently executing the !SCREEN subroutine at line 953, address 14E6 which is a KEYIN operation. Because the LEVEL parameter includes level 2, internal subroutine calls are also shown. The !SCREEN subroutine was called from The INVOICES program at line 105 (address 01BE). This program was started from the command processor which was itself started from line 104 of program PROC which was itself started from the command processor.

The same process could be reported in less detail using other values of the LEVEL option as shown below:

Level 2 (Internal subroutine stack within current program)

PSTAT USER 2 LEVEL 2
User Detail
  2 Account: SALES, Username: Joe, Pid: 1383
  Command: RUN INVOICES
  !SCREEN 953 (14E6, KEYIN)
    750 (118E)
    450 (08F3)
    323 (061B)

Level 1 (External subroutine stack but exclude internal calls)

PSTAT USER 2 LEVEL 1
User Detail
  2 Account: SALES, Username: Joe, Pid: 1383
  Command: RUN INVOICES
  !SCREEN 953 (14E6, KEYIN)
  D:\LBS\QM\BP.OUT\INVOICES 105 (01BE)
  Command processor
  D:\LBS\QM\BP.OUT\PROC 104 (05B8)
  Command processor

Level 0 (Current location only)

PSTAT USER 2
User Detail
  2 Account: SALES, Username: Joe, Pid: 1383
  Command: RUN INVOICES
  !SCREEN 953 (14E6, KEYIN)
4.164 PTERM

The PTERM command sets or displays terminal characteristics.

Format

PTERM BINARY { ON | OFF }
PTERM BREAK { ON | OFF }
PTERM BREAK { n | ^c }
PTERM CASE { INVERT | NOINVERT }
PTERM CODEPAGE { page }
PTERM CTRL.FILTER { ON | OFF }
PTERM ENCODING name
PTERM ERASE { ON | OFF | n | ^c }
PTERM MARK { ON | OFF }
PTERM NEWLINE { CR | LF | CRLF }
PTERM PROMPT "string1" { "string2" }
PTERM QUERY
PTERM RESET string
PTERM RETURN { CR | LF }
PTERM DISPLAY
PTERM LPTR

Multiple options from the above may be included in a single command.

The PTERM BINARY ON or OFF command determines whether terminal input/output is processed by QM to handle special character transformation rules appropriate to a telnet connection. When binary mode is enabled, all data is passed in/out without any modification.

The PTERM BREAK ON or OFF command determines whether use of the break key is considered to be a break or a data character. If set on, the break key will interrupt processing. If set off, the break key is treated as a normal data character. The setting of this mode does not affect interpretation of the telnet break command.

The PTERM BREAK n or ^c command sets the character to be used as the break key. The first form takes the character number (1 - 31); the second form takes the printable character associated with the control key (A - Z, [, ], ^, _). The default break character is ctrl-C (character 3). Note that some terminal emulators send a telnet negotiation parameter instead of the break character itself and may require changes to the emulator configuration to use an alternative character.

The PTERM CASE command determines whether the case of alphabetic characters is inverted on entry at the keyboard. Running with case inversion enabled may be useful in a development environment as, for historic reasons, command lines tend to be entered in upper case. In QMBasic programs, case inversion affects INPUT statements, the KEYCODE() and KEYINC() functions but not the KEYIN() function. The initial state of case inversion is set by the INVCASE configuration parameter.
The **PTERM CODEPAGE** command, available only in Windows console sessions, displays or sets the code page used to determine how single byte characters are displayed. The code page setting is irrelevant to ECS mode systems.

The **PTERM CTRL.FILTER** command enables or disables filtering of control characters in **INPUT** and **INPUT@** other than those that are defined as edit keys. This setting is maintained separately for each command processor level.

The **PTERM ENCODING** command specifies the character encoding used by the terminal device. The only encoding names allowed at UTF8 to set UTF-8 multi-byte character encoding and 8BIT to set the default unencoded 8 bit character set. The encoding is ignored for Windows QMConsole sessions. See also the `-utf8` command line option.

The **PTERM ERASE** command sets the erase (backspace) key for the QMBasic **INPUT** statement. The initial value for this key is taken from kbs element of the terminfo definition for the terminal type in use. The **OFF** qualifier to **PTERM ERASE** disables recognition of the erase key. The **ON** qualifier sets the erase key to character 8 (ctrl-H) which corresponds to the backspace key on most terminal types. Use of **n** or **^c** sets the erase character to a user specified value. The first form takes the character number (1 - 31 or 127); the second form takes the printable character associated with the control key (A - Z, [, \], ^, _). Note that setting the erase character to 127 (the Delete key) also requires that the kdc1 element of the relevant terminfo definition is removed or modified so that this key does not clash with its internal processing as a forward delete. Note that the **INPUT@** statement which supports extended editing capabilities always uses the terminfo definition for the erase key.

The **PTERM MARK** command can be used to enable a mark character translation feature whereby entry of characters 28 - 30 (ctrl-\, ctrl-[, ^] at the keyboard gets translated to subvalue, value and field marks respectively. Used without the **ON** or **OFF** qualifier, this command shows the current state. Translation is off by default.

The **PTERM NEWLINE** command determines whether QM sends CR, LF or a CR/LF pair as the newline sequence on terminal output. The default mode is **CRLF**.

The **PTERM PROMPT** command changes the command prompt from the default colon to **string1**. The optional **string2** changes the alternative prompt displayed when the default select list is active. The prompt strings must be quoted and may be from 1 to 10 characters in length.

The **PTERM QUERY** command can be used to determine what codes specific keys send. It is a valuable tool when constructing terminfo definitions. The command waits for a key to be pressed and then displays the characters received by the QM system in hexadecimal form and as a character if printable. If the key sequence includes characters that suggest this might be a key that could be bound in terminfo, the corresponding terminfo definition string is also displayed. To exit from this **PTERM** mode, enter a space.

The **PTERM RESET** command sets a control string to be sent to the terminal device on return to the command prompt. This can be used, for example, to ensure that the terminal reverts to a chosen foreground/background colour scheme regardless of how the application left it set. The **string** may include use of the QMBasic style `@()` function to insert device dependent control codes or any of the following special codes:

- \B Backspace
- \E Escape
- \F Form feed
- \N Newline
- \R Carriage return
- \T Tab
- ^
- \"
The `PTERM RETURN` command determines whether `KEYIN()` and related QMBasic functions return 10 (LF) or 13 (CR) when the return key is pressed. The actual effect of this mode setting is to replace incoming carriage returns with the given character unless the session is operating over a binary mode telnet connection. The default mode is **CR**.

The `PTERM DISPLAY` command reports the current settings of the terminal. `PTERM LPTR` directs the same report to the default printer.

**See also:**

`PTERM()`, `TERM`
4.165 PUBLISH

The **PUBLISH** command sets up a file in the current account for replication.

**Format**

```
PUBLISH [DICT] src.file [DICT] server:account:tgt.file ...
PUBLISH [DICT] src.file CANCEL server
PUBLISH [DICT] src.file CANCEL ALL

PUBLISH [DICT] src.file QUERY
PUBLISH [DICT] ALL QUERY
```

where

- **src.file** is the file to be published. This must be a reference to a VOC F-type record that defines a hashed file.
- **server** is the name of the server to which the file is to be exported. This must have been defined using **SET.SERVER** so that **PUBLISH** can validate the target file reference.
- **account** is the name of the account on **server** to which the file is to be exported.
- **tgt.file** is the name of the target file within **account** to which the file is to be exported.

The first form of the **PUBLISH** command marks the specified file as being exported to the specified target. Multiple targets may be specified in a single command or each target can be specified separately. Following successful execution of this command, all updates to the file will be logged for export.

The two forms with the **CANCEL** keyword terminate publication of the specified file. With a server name, publication to the specified **server** is terminated. With the **ALL** keyword, all publication of this file is terminated.

The two forms with the **QUERY** keyword show a list of replication targets, either for the specified file or for all files.

Adding or cancelling replication targets is restricted to users with administrator rights.

**See also:** Replication, **DISABLE.PUBLISHING**, **ENABLE.PUBLISHING**, **PUBLISH.ACCOUNT**, **SUBSCRIBE**
4.166 PUBLISH.ACCOUNT

The PUBLISH.ACCOUNT command sets up files in the current account for replication.

Format

PUBLISH.ACCOUNT {DICT} serveraccount
PUBLISH.ACCOUNT CANCEL serveraccount

where

serveraccount identifies the replication target

The first form of the PUBLISH.ACCOUNT command displays a list of files that are candidates for export to the specified serveraccount. The display is divided into pages with each file numbered. Initially, no files are marked for publishing. The following commands, based on those used by the SHOW command can be entered to modify the publication list:

Sn Sets the publish flag for file n.
Sm-n Sets the publish flag for files m to n.
S ALL Sets the publish flag for all files.
S VISIBLE Sets the publish flag for all files on the displayed page.
Cn Clears the publish flag for file n.
Cm-n Clears the publish flag for files m to n.
C ALL Clears the publish flag for all files.
C VISIBLE Clears the publish flag for all files on the displayed page.
T Displays the top page.
N Displays the next page.
P Displays the previous page.
A Applies replication to the selected files.
Q Quits without publishing.

The ALL and VISIBLE keywords can be abbreviated and do not need a preceding space. For example, SA is equivalent to S ALL,

By default, the PUBLISH.ACCOUNT command does not show or publish dictionaries. Including the DICT keyword extends the list of displayed files to include dictionaries, allowing these to be included in the list of items selected for publication.

The second form of the PUBLISH.ACCOUNT command with the CANCEL keyword displays a list of files that are published to the specified serveraccount. Files for which publication is to be cancelled can be selected with the keyboard actions listed above.

Adding or cancelling replication targets is restricted to users with administrator rights.
See also:
Replication, DISABLE.PUBLISHING, ENABLE.PUBLISHING, PUBLISH, SUBSCRIBE
4.167 QSELECT

The QSELECT command constructs a select list from the content of selected records.

Format

QSELECT {DICT} file.name {id... | * | FROM list} {TO list} {SAVING field}

where

id... is a list of records to be processed.

* specifies that all records are to be processed.

FROM list specifies a select list of records to be processed.

TO list specifies the list to be created. If omitted, the default list (list 0) is created.

SAVING field identifies the field from which items are to be taken. If omitted, all fields in the record are processed. The field item may be a field number, a field name or a data collection element reference. This clause of the command is equivalent to (fieldno in the Pick version of this command.

The QSELECT command reads selected records from the given file and constructs a select list from the content of the named field or all fields. Multivalued fields are expanded to give a separate list entry for each value or subvalue.

If the default select list is active and there are no record ids or FROM clause on the command line, this list is used to control processing.

Both @SELECTED and @SYSTEM.RETURN.CODE will be set to the number of item selected.

See the SELECT command for a discussion of chained selection where a command can be executed automatically if the QSELECT is successful.
4.168 QUIT

The **QUIT** command terminates the current QM session. The synonym **OFF** can be used.

**Format**

```plaintext
QUIT
```

The **QUIT** command terminates the QM session. If the account has the command stack recording option active (see [Command Stack](#)), the current command stack is written to the VOC.

The **ON.EXIT** paragraph, if present, is executed before final return to the operating system.

See also:

[LOGOUT](#)
4.169 REBUILD.ALL.INDICES

The REBUILD.ALL.INDICES command rebuilds the alternate key index on all files in the account.

Format

    REBUILD.ALL.INDICES {CONCURRENT}

The REBUILD.ALL.INDICES command scans the vocabulary file of the account in which it is executed and identifies all of the F-type items that reference files that have alternate key indices. These indices are then rebuilt. If a file is referenced by more than one VOC entry, the indices are only built once.

Except when executed in the QMSYS account, the command ignores files in the QMSYS account (e.g. system files).

Under normal circumstances, it should not be necessary to rebuild indices as they should be correctly maintained in step with updates applied to the associated data files. REBUILD.ALL.INDICES may be useful after restoring a backup for which indices were omitted or to ensure system integrity after a power failure.

Except when used with the CONCURRENT keyword, the REBUILD.ALL.INDICES command requires exclusive access to each file that it processes. It is therefore best executed at quiet times.

The CONCURRENT keyword allows indices to be built while the files are in use. There is a small performance overhead for this and the QMBasic SELECTINDEX, SELECTLEFT and SELECTRIGHT statements may stall waiting for the build to complete.

See also:
BUILD.INDEX, CREATE.INDEX, DELETE.INDEX, DISABLE.INDEX, LIST.INDEX, MAKE.INDEX
4.170 REDO

The REDO command repeats a command at specified intervals.

**Format**

```
REDO {interval {count}} command
```

where

- **interval** is the delay in seconds between successive executions of `command`. If omitted or specified as zero, there is no delay.
- **count** is the number of times `command` is to be executed. If omitted or specified as zero, the command is repeated forever.
- **command** is the command to be executed.

The REDO command repeats the specified `command` every `interval` seconds.

The REDO can be terminated before the `count` has expired by pressing any key on the keyboard.
4.171 RELEASE

The RELEASE command releases record or file locks.

Format

RELEASE filename id...

RELEASE FILELOCK filename

In the first form, RELEASE releases locks on the specified record id(s) in the named file. The second form releases the file lock on the named file.

RELEASE can only release locks held by the process in which the command is issued. System administrators can use the UNLOCK command to release locks held by other users.

Locks are released automatically when a file is closed. Applications can store file variables in a named common block so that the file remains open when the program terminates. In this case, locks left in place when the program ends will not be released automatically.
4.172 REMOVE.DF

The REMOVE.DF command removes a part file from a distributed file.

Format

\texttt{REMOVE.DF \textit{dist.file} \textit{part.file}}

\texttt{REMOVE.DF \textit{dist.file} \texttt{ALL}}

where

- \textit{dist.file} is the name of the distributed file.
- \textit{part.file} is the name of the part file to be removed. The part number may be specified instead.

The REMOVE.DF command will remove the named part file from the distributed file but does not delete the part file. If this is the only part file, the distributed file is also deleted.

Use of the \texttt{ALL} keyword in place of the part file name deletes the entire distributed file. The part files are not deleted by this operation.

Example

\texttt{REMOVE.DF ORDERS ORDERS-08}

This command removes ORDERS-08 as a part file within the ORDERS distributed file.

See also:

Distributed files, \texttt{ADD.DF}, \texttt{LIST.DF}
### 4.173 REPORT.SRC

The **REPORT.SRC** command turns on or off display of the [@SYSTEM.RETURN.CODE](#) variable on return to the command prompt. It is particularly useful when testing applications.

**Format**

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPORT.SRC OFF</td>
<td>To turn off display of @SYSTEM.RETURN.CODE</td>
</tr>
<tr>
<td>REPORT.SRC ON</td>
<td>To turn on display of @SYSTEM.RETURN.CODE</td>
</tr>
<tr>
<td>REPORT.SRC</td>
<td>To toggle display of @SYSTEM.RETURN.CODE</td>
</tr>
</tbody>
</table>

When reporting is enabled, the value of [@SYSTEM.RETURN.CODE](#) is displayed on return to the command prompt.
4.174 REPORT.STYLE

The REPORT.STYLE command sets the default query processor report style.

Format

   REPORT.STYLE name          To set the default report style
   REPORT.STYLE               To display the current setting
   REPORT.STYLE OFF           To disable the default report style

The query processor can highlight selected components of a report using colour on a displayed report or font weights on a report directed to a PCL printer. The REPORT.STYLE command sets the default style to be used for all reports unless overridden by an alternative definition.

In the first form of this command, the name element is the id of an X-type VOC record containing the style definition.

See the query processor STYLE option for full details of report styles.
4.175 **RESET.MASTER.KEY**

The **RESET.MASTER.KEY** command resets the encryption master key. This command can only be executed by users with administrator rights in the QMSYS account.

**Format**

```
RESET.MASTER.KEY
```

The command prompts for the master key string and whether this must be entered every time QM is started.

The **RESET.MASTER.KEY** command is intended for use after moving the encryption key vault from another system or after application of a new licence. It can also be used to enable or disable the need to enter the master key using **UNLOCK.KEY.VAULT** every time that QM is started.

The key string entered must be the same as when the key vault was created. The master key cannot be changed unless the key vault is cleared and rebuilt.

**See also:**

Data encryption, CHANGE.KEY.PASSWORD, CREATE.FILE, CREATE.KEY, CREATE.SECURE.KEY, DELETE.KEY, DISABLE.KEY, DISABLE.KEY (QMBasic), ENABLE.KEY, ENABLE.KEY (QMBasic), ENCRYPT.FILE, GRANT.KEY, LIST.KEYS, REVOKE.KEY, SET.ENCRYPTION.KEY.NAME, UNLOCK.KEY.VAULT
4.176 RESTORE.ACCOUNTS

The RESTORE.ACCOUNTS command restores all accounts from a Pick style FILE.SAVE tape.

Format

RESTORE.ACCOUNTS target {options}

where

target is the parent directory under which the restored accounts are to be placed. If omitted, the pathname specified in the $ACCOUNT.ROOT.DIR.VOC entry is used or, if this record does not exist, the user is prompted for the directory pathname.

options is any combination of the following:

- **BINARY** Suppresses translation of field marks to newlines when restoring directory files. Use this option when restoring binary data.
- **DET.SUP** Suppresses display of the name of each file as it is restored.
- **NO.CASE** Causes new files to be created with case insensitive record ids. Existing files are not reconfigured.
- **NO.INDEX** Do not create alternate key indices.
- **NO.OBJECT** Omits restore of object code. This is particularly useful when migrating to QM from other environments.
- **POSITIONED** Assumes that the tape is already positioned at the start of the data to be restored.

The RESTORE.ACCOUNTS command processes a Pick style FILE.SAVE tape or pseudo tape and restores data from it into a QM system. It can also restore from a tape containing multiple ACCOUNT.SAVEs.

The tape to be restored must first be opened to the process using the SET.DEVICE command.

All accounts found on the tape are restored unless there is already an account of the same name or the target account directory already exists. In these cases, the account is skipped.

For more details of the tape processing applied during restore, see the ACCOUNT.RESTORE command.

See also:
ACCOUNT.RESTORE, ACCOUNT.SAVE, FILE.SAVE, FIND.ACCOUNT, QMSAVE, SEL.RESTORE, SET.DEVICE, T.ATT, T.DUMP, T.LOAD, T.xxx
4.177 REVOKE.KEY

The REVOKE.KEY command removes access to a specific data encryption key. This command can only be executed by users with administrator rights in the QMSYS account.

Format

```
REVOKE.KEY keyname \{GROUP\} name ...
```

where

- `keyname` is the name of the encryption key. This is case insensitive
- `name ...` is a list of user names for users to be denied access. If prefixed by the `GROUP` keyword, this is a list of user groups. On Windows systems, user and group names are treated as case insensitive.

The REVOKE.KEY command removes access to an encryption key to one or more users or user groups. The user will be asked to enter the master key unless it has already been entered during this session.

No error occurs if the user or group specified did not have access to the key.

Example

```
REVOKE.KEY CARDNO jsmith bjones
```

The above command removes access to the encryption key named CARDNO for users jsmith and bjones.

See also: Data encryption, CHANGE.KEY.PASSWORD, CREATE.FILE, CREATE.KEY, CREATE.SECURE.KEY, DELETE.KEY, DISABLE.KEY, DISABLE.KEY (QMBasic), ENABLE.KEY, ENABLE.KEY (QMBasic), ENCRYPT.FILE, GRANT.KEY, LIST.KEYS, RESET.MASTER.KEY, SET.ENCRYPTION.KEY.NAME, UNLOCK.KEY.VAULT
4.178 RPL.STATUS

The RPL.STATUS command displays a summary of the replication system status.

Format

RPL.STATUS

The RPL.STATUS command is normally only present in the VOC of the QMSYS account but it can be copied to other accounts if needed.

The report produced by this command has two sections:

The first section shows information relating to the system as a publisher. This is the settings of the REPLDIR and REPLPORT configuration parameters followed by a list of the servers to which data is being published. A system is shown if there is a replication log directory for that server. Where a system is no longer used as a replication target, it will continue to be shown until the replication log directory is deleted. For each system, the report shows it as active (subscriber phantom on the remote system is running) or inactive (subscriber phantom not running). It will also show a warning if there is a backlog of replication updates.

The second section shows information relating to the system as a subscriber. This is the setting of the REPLSRVR configuration parameter followed by a list of the systems from which the local server is subscribing updates. This shows the remote server name and the QM user number of the subscriber phantom.

Example

Information about this system as a publisher
---------------------------------------------

Replication log location (REPLDIR): /media/ssd/repldir
Listening port (REPLPORT): 4244

Publishing to:
   LBS1 (inactive)
   LBS21 (active)
   ZSRVR (active)

Information about this system as a subscriber
---------------------------------------------

Subscriber server name (REPLSRVR): LBS20

Subscribing from:
   LBS1 (local user 3)
   LBS21 (local user 393)

See also:

Replication
4.179 RUN

The **RUN** command initiates execution of a compiled QMBasic program. It can also be used to execute VOC style items that are stored in alternative files.

**Format**

```plaintext
RUN {file.name} record.name [LPTR] [NO.PAGE]
```

where

- **file.name** is the name of the directory file holding the program or VOC style item to be run. If omitted, this defaults to BP. The .OUT suffix for the compiler output file is supplied automatically when using the command to execute a QMBasic program.

- **record.name** is the name of the compiled program or VOC entry.

- **LPTR** causes output to logical print unit 0 to be directed to the printer. This is identical in effect to a **PRINTER ON** statement being performed within the program.

- **NO.PAGE** suppresses pagination of output to the terminal.

The rules regarding location of the item to be executed are:

1. If only one name is provided, BP is assumed as the file name.

2. If a file with the .OUT suffix added to the name is defined in the VOC and can be opened, **record.name** is assumed to be the name of a compiled QMBasic program. The **RUN** command can also be used to run an object instantiated from a class module that contains a public subroutine named MAIN.

3. If the file is not defined in the VOC or cannot be opened for any reason, **record.name** is assumed to be the name of a VOC style item (sentence, paragraph, menu, etc) in the named file without the .OUT suffix.

4. If the item identified by the above steps cannot be found, an error is reported.

By default, output directed to the screen from a program will pause at the end of each page waiting for user confirmation. For compatibility with some other systems, the **RUN NO.PAGE** mode of the **OPTION** command can be enabled such that the pause will not occur. If the **PAGINATE.ON.HEADING** mode of the **OPTION** command in also enabled, setting a page heading or footing in the application will cause the page end pause to be enabled. The **NO.PAGE** keyword to the **RUN** command will override this such that pagination never occurs. This keyword can also be used when running a catalogued program by entering its name.
4.180 SAVE.LIST

The **SAVE.LIST** command is used to save an active select list for future use.

**Format**

```
SAVE.LIST  [{file.name} list.name] {FROM list.no} {COUNT.SUP} {KEEP}
```

where

- **file.name** is the name of the file in which the list is to be saved. This defaults to $SAVEDLISTS if omitted.
- **list.name** is the name of the record to be created in `file.name` to hold the saved select list. This defaults to "$temp.n" if omitted, where `n` is the QM user number.
- **list.no** identifies the select list (0 to 10) to be saved. If omitted, select list zero is used.
- **COUNT.SUP** suppresses display of the number items in the saved list.
- **KEEP** Retains that active list, saving a copy of it.

The **SAVE.LIST** command copies an active select list to the $SAVEDLISTS or alternative file. When using $SAVEDLISTS, the file will be created if it does not already exist.

If the active list has already been partially processed, only the remaining items are saved. The active select list is cleared after it has been saved unless the **KEEP** option is used.

@SYSTEM.RETURN.CODE is set to the number of items in the saved list. In the event of an error, the value is a negative error code.

**Example**

```
SAVE.LIST INVENTORY FROM 3
Saved list 'INVENTORY' in $SAVEDLISTS.
```

This example saves active select list 3 as INVENTORY.

**See also:**

COPY.LIST, DELETE.LIST, EDIT.LIST, FORM.LIST, GET.LIST, SORT.LIST
4.181 SAVE_STACK

The `SAVE_STACK` command saves the current command stack.

**Format**

```
SAVE_STACK {file.name} stack.name
```

where

- `file.name` is the name of the file in which the stack is to be saved. If omitted but the `stack.name` is present, the `$SAVEDLISTS` file is used.
- `stack.name` is the name to be given to the saved command stack. A prompt is issued if this name is omitted.

The `SAVE_STACK` command copies the current command stack to `file.name` as a record named `stack.name`. Any existing record of the same name will be overwritten.

If no command line options are given, the user is prompted to enter the `stack.name` and the `file.name` defaults to `$SAVEDLISTS`.

**Example**

```
SAVE_STACK <<@LOGNAME>>
Command stack 'jsmith' saved in $SAVEDLISTS
```

This command saves the current command stack to a record with id as the user's login name in the `$SAVEDLISTS` file.

**See also:**

`CLEAR_STACK`, `GET_STACK`
4.182 SCRB

The SCRB command runs the screen builder to create or modify a screen definition for use by the QMBasic SCREEN() subroutine.

Format

```
SCRB {screen.file} {screen.name}
```

where

- **screen.file** is the name of the file holding the screen definition.
- **screen.name** is the screen definition record name. If neither a screen.file nor a screen.name is given, SCRB will check for an active select list before prompting for a screen name.

The QMBasic SCREEN() subroutine uses screen definitions which are created and maintained using SCRB. The name of the account’s default screen definition file is stored in field 2 of an X-type VOC record named $SCRB.FILE and will be used if no file name is given in the SCRB command line. If this record does not exist, SCRB uses the $SCREENS file which is common to all accounts. The $SCREENS file may also contain screen definitions which are part of the QM product. These have a dollar sign in their name and should not be modified or removed.

A screen consists of a number of steps, each of which may have a fixed text display, output of data from a data record, and input of data into a data record. Data may be converted or formatted on output and input. Steps that include data input may perform validation within the screen driver subroutine and may also have user defined help and error messages. Programs can undertake all aspects of managing the flow from one step to another themselves, pass this task to the screen driver using a variety of conditional flow control options or use a mixture of the two modes.

The screen builder can automatically generate an include record identifying the screen steps by name for use in programs that use the screen. This include record will be placed in a nominated file with the same name as the screen with a .SCR suffix.

On entry to SCRB, the user is invited to enter a screen name unless this has been provided on the command line. SCRB will look for this in the form entered and, if not found, in uppercase. If neither exists, a new screen is created.

The user can then select from the following options:

- **D** Delete the screen definition
- **F** File the screen definition
- **H** Amend the screen header line
- **I** Set the file to store the generated include record
- **L** List the screen steps
- **P** Paint the screen
- **LPTR** Print the screen definition
- **Sn** Show / edit step n
- **X** Exit without filing
Amending the Screen Header Line

Selecting the H option allows entry/modification of the screen header line. It is useful to include the screen name in the header, perhaps at the left margin. All the normal input editing keys are available. Pressing the return key will end the header update.

Listing Screen Steps

The L option displays a list of screen steps. The information displayed is the step number and the fixed text (if any) associated with the step. If the step has no fixed text, the display shows <<step name>>. The escape key can be used to terminate the list before the last page is displayed.

Painting the Screen

The P option paints an image of the screen, showing the fixed text associated with steps except those that are tagged as not to be included in a full screen paint. A prompt is issued allowing selection of a step number to be executed. This allows easy debugging of the screen from within the screen builder. Entering X at the action prompt exits from paint mode.

Printing the Screen

The LPTR option send the screen definition to the printer. Details of each step are printed followed by a representation of the screen as it would be appear after the screen paint action of the !SCREEN() subroutine.

Show / Edit a Screen Step

The Sn option displays the definition of screen step n. A further prompt allows selection of the action to be taken

- Cn: Copy step n over the currently displayed step
- Cscrn,n: Copy step n of screen scrn over the currently displayed step
- D: Delete this step
- I: Insert a new step before this step
- Mn: Move currently displayed step to become step n
- N: Advance to the next step
- P: Move to the previous step
- R: Return to the top level screen
- Sn: Move to step n
- n: Edit step definition starting at item n

The items that make up a screen step are described below.

Name
Steps may optionally be given names. These must be unique within the screen definition and are used in generating an include record of step names for use in QMBasic programs. The include record tokens are formed by adding a prefix of SS. to the step name.

**Type**
The step type determines the way in which the screen driver will handle the step. The type may be any of the following:

- **D** A display step. Any data item referenced by the step will be displayed but no input is permitted.
- **I** An input step. The data is displayed as for a display step but input may also be performed.
- **N** A control step. Any data item referenced by the step is not displayed but conditional flow control elements based on this data item may still be included.
- **Gn** A step group to be repeated \( n \) times. The display step field described below contains a list of steps to be repeated. The repeat will terminate if the next step determination of any repeated step evaluates to anything other than the default. Step groups cannot be nested.

The step type may also include the following qualifying codes:

- **B** Sounds the terminal "bell".
- **C** Clears the text and data of this step after the step is completed. Usually used with the **X** display mode options described later, this feature is particularly useful for clearing temporary prompt fields from the screen.
- **H** Causes the screen header to be displayed as part of this step.
- **Rn** This step is to be repeated \( n \) times. The repeat will terminate if the next step evaluates to anything other than the default. Repeated steps may not appear inside step groups.
- **X** Excludes this step from the initial screen painting process. This should normally be included on the elements of a repeating group (not a repeated step).

**Clear**
The clear item contains **Y** if the screen is to be cleared prior to displaying this step.

**Display step** (multivalued)
This field lists steps by name or number which are to be displayed prior to displaying this step. For repeating step group definitions it holds the steps to be repeated as described above.

A list may be entered in a multivalued screen definition field by pressing F2.

**Text**
- **Text row**
- **Text col**
- **Text mode**
The text item contains a fixed text string which will be displayed on the screen at the position given by the text row and text col values using the text mode display style. The mode may contain any combination of **H** (to display in half intensity), **R** to set reverse video (interchange foreground and background colours) and **X** (omit from full screen paint).
Value
Subvalue
These items determine the position of the data item to be displayed or input. The field may be specified as a numeric position or as a filename and field name separated by a space. In the latter case, the screen builder will look up the actual field position in the dictionary of the specified file when the screen definition is being entered. Later changes to the dictionary will not cause the screen definition to change. The value and subvalue fields may be left blank where the entire field or value is to be displayed.

A value of zero for the field uses an internal temporary variable. This is of particular use in confirmation prompts, for example, where the data input is used to determine flow through the screen steps but is not part of the record being amended.

For a repeated step or the elements of a repeating group, the field, value or subvalue may be specified as ‘1+’, for example, which will cause the screen to use successive items starting at the given position.

Prompt char
This item may be used to specify a character to appear immediately to the left of the data field. It will be ignored if the data is displayed at the left margin.

Fill char
The fill character is used to pad out short data items on the display. It is not entered into the stored data.

Data row
Data col
Data mode
These items operate in the same way as their equivalents for the text area.

For a repeated step or the elements of a repeating group, the row may be specified as \( y+i \) where \( y \) is the line on which the first item is to appear and \( i \) indicates the number of lines by which the position is to advance for successive elements. The value of \( i \) defaults to 1.

Output len
The output length item specifies the length of the data item on the display. If the actual data is longer than this, the display is truncated.

Output conv (multivalued)
This item holds the conversion to be performed on the data prior to display. The same conversion will be applied to redisplay input values when the step is completed. The conversion may be multivalued to perform successive conversions. Output conversions are:

- \( \texttt{Ffmt} \) Apply \texttt{FMT()} using \( \texttt{fmt} \) as the format specifier.
- \( \texttt{Ifile,rec} \) Execute the I-type named \( \texttt{rec} \) in the dictionary of \( \texttt{file} \) against the data record.
- \( \texttt{Sname} \) Execute catalogued subroutine \( \texttt{name} \), passing in the field value as argument 2 and replacing it with the valued returned through argument 1.
- \( \texttt{Tfile,fld,code} \) Apply \texttt{TRANS()} using the data item as the record id to access field \( \texttt{fld} \) of \( \texttt{file} \). The code determines the action if the record is not found. \( \texttt{C} \) returns the record id, \( \texttt{X} \) returns a null string.
- \( \langle f,v,s \rangle \) Extract the given field, value or subvalue.
Other codes are passed to **OCONV**().

**Justify**
This item contains **L** for left justification or **R** for right justification.

**End mark**
This item may be used to specify a character to appear immediately to the right of the data field. It will be ignored if the data field extends to the right margin.

**Input len**
The input length item specifies the permissible length of the data. If this value exceeds the output length, the field is panned to allow entry of long data. Many of the screen definition items are themselves panned in this way by **SCRB**.

**Required**
This item indicates whether the field may be left blank. It may contain:

- **Y** The field may not be left blank.
- **N** The field may be left blank.
- **F** For a repeated step or element of a repeating group, the field must not be blank for the first iteration of the repeat but may be blank for subsequent iterations.
- \(<f>\)cond The field is required if field \(<n>\) (or \(<f,v>\) or \(<f,v,s>\)) meets the supplied condition. The condition code is as described for the next step determinations described below.

**Input val 1** (multivalued)
The first input validation is performed on the input data prior to input conversion. This item may be multivalued to perform multiple validations. The data is deemed acceptable if any of the validation criteria are satisfied. Validation codes are:

- **m** Numeric value \(m\). This is a numeric comparison, leading zeros will be ignored.
- **m-n** A numeric value in the range \(m\) to \(n\).
- **=x** String equality with \(x\). \(x\) may be a null string.
- **C'abc',65,68-70** Character validation. The data must only contain characters from the set listed. These may be a literal string enclosed in single or double quotes, a character number or a range of character numbers. Multiple character sets can be given with a comma separator as in this example which allows "abcADEF".
- **D** A valid date.
- **Ffilename** A record named as the input data exists in **filename**.
- **Ffilename,n** Similar to the simple **F** validation but, if the record is found, the input data is replaced by the contents of field \(n\) of the record.
- **Mtemplate** The input data matches the specified **template**.
- **Rfile,rec,fld,case,subst** Record \(rec\) is read from **file**. Field **fld** of this record is scanned for a match with the input data. If **case** is **X**, this scan is case insensitive. If
subst is specified, the input data is replaced with the content of the corresponding value of field subst of the record. Field fld may be broken down into subvalues to specify alternative strings all of which are replaced by the value (not subvalue) in subst.

@subrname Calls the named user supplied validation subroutine. This subroutine takes three arguments; the return status (1 if ok, 0 if error), the data record being processed and the input data field to be validated.

XXX Inverts the condition xxx. For example, XFINDEX would check that there is no record named as the input data in file INDEX.

Input conv (multivalued)
This item holds the conversion to be performed on the input data. The conversion may be multivalued to perform successive conversions. Input conversions are:

- **Ef**mt: Apply FMT() using fmt as the format specifier.
- **N**n: If the data is numeric, extend it to be right justified in n digits.
- **S**name: Execute catalogued subroutine name, passing in the field value as argument 2 and replacing it with the valued returned through argument 1.
- **T**file,fld,code: Apply TRANS() using the data item as the record id to access field fld of file. The code determines the action if the record is not found. C returns the record id, X returns a null string.
- **<f,v,s>**: Extract the given field, value or subvalue.
- **Other**: All other codes are passed to ICONV().

Input val 2 (multivalued)
The second input validation is performed on the input data after input conversion. This item may be multivalued to perform multiple validations. The data is deemed acceptable if any of the validation criteria are satisfied. Validation codes are as for the first input validation.

Back step
The back step item determines whether the back-tab key is allowed and may contain Y, N or blank which is taken as N. If back-tab is allowed, the screen driver will perform the action internally if it has a step history. If there is no step history, the screen driver returns to the calling program with a status indicating that the backstep key was used. The back step will correctly back-track through a repeated step or repeating group.

Next step
The next step item defines the action to be taken after the step is completed. It may be multivalued with conditional elements. A null action simply increments the step number. All next step action lists effectively end with a null element which would be executed if all previous elements were conditional and not satisfied.

An action item comprises three parts; a field reference, a condition and an action. The action is preceded by a colon.

The field reference identifies the field within the screen data which is to be used in the following condition. This may be specified as <field>, <field,value> or <field,value,subvalue>. If omitted entirely (as is usual), the data from the current step is used. If a field reference is included, the condition must also be present.
The condition compares the selected data with a fixed string or numeric value. The generic form of
this is EQ'string', EQ'number or EQ<file,fieldname>. The operator may be any of EQ, NE, LT, GT, LE or
GE. If the condition is omitted, the action is performed unconditionally.

The action may be null to advance to the next step, a step number, a step name or X to exit to the
calling program. Within a repeated step or a repeating group any non-null action terminates the
repeat.

Example: EQ'':X would exit if the field is empty.

Help msg
The help message item defines a message to be displayed if the F1 key is pressed. The message may
be split over multiple lines by using the F2 multivalue entry feature of SCRB.

Error msg
The error message item defines a message to be displayed if input validation fails in a similar manner
to the help message.

Exit key
The exit key item defines the action to be taken if the escape key is pressed. The format and
processing of this item is as for the next step item except that a null value causes an error message
indicating that the exit key is not allowed.

F2 action
The F2 key is used for extended pick-list based help. If this item is blank, F2 is treated the same as
other function keys as described below. There are three formats to this item:

filename, selection/sort clauses, field names
This format causes the screen driver to select records from the specified file using the selection
and sort clauses (as for the query processor). A pick list is displayed based on the specified field
names (which are space separated and may include I-types). The first field in this list is used as
the returned value from the selection. The pick list short cut system described below uses the
field name by default. An alternative field may be used by prefixing its name with a & character. The short cut will only work correctly if the list is sorted in ascending order by the
short cut field.

#filename, record.name, sort.field, field.list
This format builds a pick list from given file, record, field. The sort.field and field.list
reference fields in the named record by number. Dictionary names cannot be used in this
action.

The data is sorted based on the value in the sort.field. The sort.field number may be followed
by R to perform a right justified sort. The field.list is a space separated list of fields to be
displayed. The sort.field must be explicitly referenced if it is to be displayed. The first field in
field.list contains the value to be returned by the selection process.

@subr
@subr(arglist)
Calls a user written subroutine subr to generate the list of items to display. This subroutine
returns the data to be displayed via its first argument, one field per pick list line each containing
a value mark delimited set of data values. The second argument should return the pick list
column number (from one) of the column to be used by the short cut system described below.
Up to four additional arguments may be passed into the subroutine from arglist which contains
literal values separated by commas. No string delimiters are required.

The pick list is displayed as a rolling window. The cursor up/down, page up/down, home and end
keys may be used to explore this window. Keys corresponding to printable characters cause a short
cut jump to a page displaying items starting at the first that commences with the entered character.
The return key will place the data displayed in the first column of the selected item into the screen field. The escape key will exit from the pick list processing without entering data into the screen field.

**Func keys**
The screen driver may accept or reject function keys. Entering Y in this item causes the screen driver to return function keys to the calling program. Entering N or leaving it blank causes an error message if function keys are used.

**Key val**
This field may contain the name of a validation subroutine that will be called after each input keystroke in the field. The subroutine name is followed by a comma and the error message to be displayed if validation fails.

The subroutine takes three arguments; the returned status (1 if ok, 0 if error), the data record being processed and the input data for the field (not just the last keystroke).

**Special keys**
The screen driver uses the following keys for special purposes. The control key bindings shown after some entries are provided for compatibility with other parts of QM.

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>Help</td>
</tr>
<tr>
<td>F2</td>
<td>Pick list help</td>
</tr>
<tr>
<td>F3</td>
<td>Delete the contents of the current field</td>
</tr>
<tr>
<td>F4</td>
<td>Restore the contents of the current field after incorrect entry</td>
</tr>
<tr>
<td>Return</td>
<td>Execute the next step action</td>
</tr>
<tr>
<td>Tab</td>
<td>Treated identically to the return key</td>
</tr>
<tr>
<td>Ctrl-P</td>
<td>Execute the back step action</td>
</tr>
<tr>
<td>Exit</td>
<td>Execute the exit key action (Ctrl-X)</td>
</tr>
<tr>
<td>Home</td>
<td>Mode to start of field (Ctrl-A)</td>
</tr>
<tr>
<td>End</td>
<td>Move to end of field (Ctrl-E)</td>
</tr>
<tr>
<td>Cursor left</td>
<td>Move left one character (Ctrl-B)</td>
</tr>
<tr>
<td>Cursor right</td>
<td>Move right one character (Ctrl-F)</td>
</tr>
<tr>
<td>Delete</td>
<td>Delete character under the cursor (Ctrl-D)</td>
</tr>
<tr>
<td>Ctrl-K</td>
<td>Delete all to right of the cursor</td>
</tr>
<tr>
<td>Backspace</td>
<td>Delete character before cursor</td>
</tr>
<tr>
<td>Insert</td>
<td>Toggle overlay mode</td>
</tr>
<tr>
<td>Page up</td>
<td>Scroll up in pick list display (Esc-V)</td>
</tr>
<tr>
<td>Page down</td>
<td>Scroll down in pick list display (Ctrl-V)</td>
</tr>
</tbody>
</table>

Within SCRB itself, the F2 key is used to enter multivalued items such as validation criteria. Input prompts for successive items appear at the bottom of the screen until a null value is entered.
4.183 SECURITY

The SECURITY command enables, disables or reports the state of QM’s internal security system.

Format

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SECURITY ON</td>
<td>Enable security</td>
</tr>
<tr>
<td>SECURITY OFF</td>
<td>Disable security</td>
</tr>
<tr>
<td>SECURITY</td>
<td>Report the current security setting</td>
</tr>
</tbody>
</table>

The SECURITY command controls access to QM.

If security is enabled, users can only enter QM if their user name appears in the user register. Users who are system administrators within the operating system are not restricted.

If security is disabled, all users who have valid authentication details for access at the operating system level can use QM.

Use of the SECURITY command is restricted to QMConsole users with administrator rights. If security is disabled, all users are considered as administrators.

See Application Level Security for more information.

See also:
ADMIN.USER, CREATE.USER, DELETE.USER, LIST.USERS, PASSWORD. Application Level Security
4.184 SED

The SED command is a screen based editor based on the principles of the EMACS editor, widely used on Unix/Linux systems. It is particularly useful for editing QMBasic source programs where many lines (fields) can be seen at once.

Format

```
SED {{DICT} file.name {record.id ... | CREATING.SEQKEY}}
```

where

- **DICT** indicates that records from the dictionary portion of the file are to be edited.
- **file.name** is the name of the file holding the record(s) to be edited.
- **record.id** is the name of the record to be edited. Multiple record names may be specified.

If no **file.name** is specified, SED will prompt for this name. The response may include a prefix of **DICT** to select the dictionary portion of the file.

If no **record.id** is specified and the default select list is active, this list is used to identify the records to be edited. If no **record.id** is specified and the default select list is not active, SED prompts for the **record.id**. Use of **CREATING.SEQKEY** instead of a record id automatically generates the next sequential key for the file.

An asterisk (*) either on the command line or as the first **record.id** entered in response to the prompt will cause SED to select all records of the file and edit each in turn.

A question mark (?) as the first **record.id** entered in response to the prompt (not on the command line) causes direct entry in explore mode, displaying a list of records in the file. The ? character may be followed by a single space and a selection template as described under the list records function to produce a filtered list of records.

The editor maintains an update lock on the record(s) being edited.

When editing a compiled dictionary item (C-type, I-type, or A/S type with a correlative), SED removes the compiled code thus forcing recompilation when the modified item is first referenced in a query.

To allow SED to be used internally from within applications without reducing system security, if the editor is started from a V-type VOC item that has E in field 4, the user is restricted to editing records specified on the command line and also cannot execute QM commands from within the editor.

SED Topics

- [Records, Buffers and Windows](#)
- [Standard key bindings](#)
- [Cursor movement and search functions](#)
Data insertion
Copying, deleting and restoring data
Working with multivalued data
Functions that operate on a block of data
Changing text
Macros
File Handling
Repeating functions
Miscellaneous functions
Commands
Setting up default modes
Source control
Dynamic key bindings
Extension programming
SED - Records, buffers and windows

A record to be edited is held in a buffer. Buffers may also hold records being created but not yet written to disk or other data such as lists of records in a file. SED allows use of up to 20 buffers.

The editor displays a window in which a number of lines of the record being edited can be seen at any one time. Where a line is wider than the display, the entire window pans from side to side to maintain the cursor within the display area.

The bottom two lines of the screen display status information. The upper status line shows the file and record names of the data being edited. If the data has been changed and hence does not match what is stored in the file, an asterisk is displayed at the start of this line.

The lower status line displays several status fields. From left to right these show:
- the number of lines in the record
- the current cursor position (line and column, both numbered from one)
- the status of macro collection
- the state of insertion overlay mode
- the state of indentation mode
- the search mode
- the count for repeated functions and operations with a numeric prefix

The lower status line is also used by some editor functions to request qualifying information. A limited set of editing functions can be used within this prompt area. These are forward char, back char, start line, end line, delete char, backspace, kill line and insert kill buffer. The kill line function used in this area deletes all characters after the cursor without affecting the kill buffer. The insert kill buffer function will insert the first line from the kill buffer at the current cursor position.

The display may optionally include line numbering. See the LNUM command for further details.
SED - Standard key bindings quick reference

**Simple Cursor Movements**

- Ctrl-P (previous)
- Ctrl-B (backwards)
- Ctrl-N (next)
- Ctrl-F (forwards)
- Ctrl-A (end)
- Ctrl-K
- Ctrl-V
- Esc-V
- Esc->

**"All the way" movements**

- Esc-<
- ^
- v

**Delete char**
- Ctrl-D

**Backspace**
- Backspace key

**Delete line**
- Ctrl-K

**Forward one screen**
- Ctrl-V

**Backward one screen**
- Esc-V

**Other Important Keys**

- Cancel: Ctrl-G (Terminates partly entered codes)
- Save changes: Ctrl-X S
- Close editor: Ctrl-X C

**Searching and Replacing**

- Forward search: Esc-S (Prompts for search string, defaulting to last)
- Reverse search: Esc-R (Prompts for search string, defaulting to last)
- Replace: Ctrl-X R (Prompts for search string and replacement)
- Query replace: Ctrl-X Q (Like Replace but queries each change)
- View: Ctrl-X V (View all lines containing specified text)

**Other Movements**

- Forward word: Esc-F
- Backward word: Esc-B
Cut and Paste

<table>
<thead>
<tr>
<th>Function</th>
<th>Key bindings</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set mark</td>
<td>Esc-</td>
<td>Marks current cursor position</td>
</tr>
<tr>
<td>Copy region</td>
<td>Esc-W</td>
<td>Saves a copy of the text between the cursor and the mark</td>
</tr>
<tr>
<td>Delete region</td>
<td>Ctrl-W</td>
<td>Deletes the text between the cursor and the mark, saving it</td>
</tr>
<tr>
<td>Paste</td>
<td>Ctrl-Y</td>
<td>Inserts text saved by Esc-W, Ctrl-W or Ctrl-K</td>
</tr>
</tbody>
</table>

SED - Standard key bindings

SED commands in the default key bindings consist of keystrokes which are

- Control shift + key
- ESCape followed by another key
- Ctrl-X followed by another key
- Ctrl-X followed by control shift + key

The table below summarises the standard editor function key bindings but these can be changed. All other keystrokes except for unused control shift codes cause the character to be inserted into the record text at the current cursor position.

<table>
<thead>
<tr>
<th>Ctrl-</th>
<th>Esc-</th>
<th>Ctrl-X</th>
<th>Ctrl-X Ctrl-</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Start line</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Back char</td>
<td>Back word</td>
<td>Goto buffer</td>
</tr>
<tr>
<td>C</td>
<td>Repeat</td>
<td>Capital init</td>
<td>*Quit</td>
</tr>
<tr>
<td>D</td>
<td>Delete char</td>
<td>Delete word</td>
<td>*List records</td>
</tr>
<tr>
<td>E</td>
<td>End line</td>
<td>Run extension</td>
<td>*Execute macro</td>
</tr>
<tr>
<td>F</td>
<td>Forward char</td>
<td>Forward word</td>
<td>*Find record</td>
</tr>
<tr>
<td>G</td>
<td>Cancel</td>
<td>Goto line</td>
<td></td>
</tr>
<tr>
<td>H</td>
<td>Backspace</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>Tab</td>
<td>Align text</td>
<td>Import</td>
</tr>
<tr>
<td>J</td>
<td>Newline</td>
<td></td>
<td></td>
</tr>
<tr>
<td>K</td>
<td>Kill line</td>
<td></td>
<td>Delete buffer</td>
</tr>
<tr>
<td>L</td>
<td>Refresh</td>
<td>Lowercase</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>Newline</td>
<td>Compile</td>
<td>Compile and run</td>
</tr>
<tr>
<td>N</td>
<td>Down line</td>
<td>Next buffer</td>
<td>Split down</td>
</tr>
<tr>
<td>O</td>
<td>Overlay</td>
<td></td>
<td>Toggle window</td>
</tr>
<tr>
<td>P</td>
<td>Up line</td>
<td>Previous buffer</td>
<td>Split up</td>
</tr>
<tr>
<td>Key</td>
<td>Function 1</td>
<td>Function 2</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>---------------------</td>
<td>---------------------</td>
<td>----------------------</td>
</tr>
<tr>
<td>Q</td>
<td>Quote char</td>
<td>Quote char</td>
<td>Query replace</td>
</tr>
<tr>
<td>R</td>
<td>Reverse search</td>
<td>Reverse search</td>
<td>Replace</td>
</tr>
<tr>
<td>S</td>
<td>Forward search</td>
<td>Forward search</td>
<td>Save record</td>
</tr>
<tr>
<td>T</td>
<td>Toggle chars</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>Repeat</td>
<td>Uppercase</td>
<td>Up to parent</td>
</tr>
<tr>
<td>V</td>
<td>Forward screen</td>
<td>Back screen</td>
<td>View</td>
</tr>
<tr>
<td>W</td>
<td>Delete region</td>
<td>Copy region</td>
<td>*Write record</td>
</tr>
<tr>
<td>X</td>
<td>Ctrl-X prefix</td>
<td>*Command</td>
<td>*Export</td>
</tr>
<tr>
<td>Y</td>
<td>Insert kill buffer</td>
<td>Insert kill buffer</td>
<td></td>
</tr>
<tr>
<td>Z</td>
<td>Up line</td>
<td>Split up</td>
<td>Nudge up</td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>Unsplit window</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>Split window</td>
</tr>
<tr>
<td>(</td>
<td></td>
<td></td>
<td>*Start macro</td>
</tr>
<tr>
<td>)</td>
<td></td>
<td></td>
<td>*End macro</td>
</tr>
<tr>
<td>=</td>
<td></td>
<td></td>
<td>*Expand char</td>
</tr>
<tr>
<td>.</td>
<td></td>
<td></td>
<td>Set mark</td>
</tr>
<tr>
<td>&lt;</td>
<td></td>
<td></td>
<td>Top</td>
</tr>
<tr>
<td>&gt;</td>
<td></td>
<td></td>
<td>Bottom</td>
</tr>
<tr>
<td>Bkspc</td>
<td>Backspace</td>
<td></td>
<td>Back del word</td>
</tr>
<tr>
<td>Del</td>
<td>Delete char</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Space</td>
<td></td>
<td></td>
<td>Close spaces</td>
</tr>
</tbody>
</table>

Functions marked with an asterisk cannot be included in a macro and cannot be repeated using the `repeat` or `repeat count` functions.

Some functions are available using alternative key sequences. Such alternatives are shown in the descriptions.

Most functions can be repeated multiple times by use of the repeat count prefix. Unless otherwise specified, the repeat count defaults to one if not explicitly set. Some functions use the repeat counter for different purposes.

When learning SED, the [Quick Reference Chart](#) summarises the more important commands.
SED - Cursor movement and search functions

Start line (Ctrl-A or Home)
Moves the cursor to the start of the current line (column 1).

End line (Ctrl-E or End)
Moves the cursor to the position following the last character in the current line.

Top (Esc-<)
Moves to the start of line 1.

Bottom (Esc- >)
Moves to the start of the line immediately after the last line in the record.

Down line (Ctrl-N or Cursor Down)
Moves the cursor vertically down one line. If this position is beyond the end of the data in the new line, the cursor is displayed immediately to the right of the final character. The editor remembers the column position from which the cursor was moved so that a further vertical movement will continue to place the cursor at the lesser of its original column position and the end of the current line. The default behaviour if the cursor is on the bottom line of the displayed data is to scroll down such that the cursor line is in the centre of the displayed data. If the SCROLL1 option is set in the &SED.OPTIONS& record, the screen scrolls by just one line.

Up line (Ctrl-P or Ctrl-Z or Cursor Up)
Moves the cursor vertically up one line. The same process is used for determining the column position as for the down line operation described above. The default behaviour if the cursor is on the top line of the displayed data is to scroll up such that the cursor line is in the centre of the displayed data. If the SCROLL1 option is set in the &SED.OPTIONS& record, the screen scrolls by just one line.

Forward char (Ctrl-F or Cursor Right)
Moves the cursor right. Moving beyond the end of a line positions the cursor at the start of the following line.

Back char (Ctrl-B or Cursor Left)
Moves the cursor left. Moving beyond the start of a line positions the cursor at the end of the previous line.

Forward word (Esc-F)
Moves the cursor to the first character after the next word. A word is defined as a continuous sequence of letters or digits.

Back word (Esc-B)
Moves the cursor to the first character of the previous word. A word is defined as a continuous sequence of letters or digits.

Forward screen (Ctrl-V or Page Down)
Moves the cursor down by one screen or to the end of the buffer.

Back screen (Esc-V or Page Up)
Moves the cursor up by one screen or to the start of the buffer.

Goto line (Esc-G)
Moves the cursor to the line number specified by the repeat counter. If no count has been entered, a prompt for the line number is issued.
Tab (Tab or Ctrl-I)
Advances the cursor to the next horizontal tabulation position (columns 11, 21, 31, etc. by default. See the TABS command). If this is beyond the end of the data in the current line, spaces are inserted to extend the line to the required position. The tab function does not insert a tab character. Use the quote character prefix to do this.

Forward search (Ctrl-S or Esc-S)
The forward search function prompts for a search string and advances to the next occurrence of this string within the record. The prompt defaults to the same string as the previous search function (if any). Searches may be performed in any of four modes:

- **Case sensitive:** Data will only be found if it matches the search string exactly. This is the default search mode.
- **Case insensitive:** The case of both the search string and the data is ignored.
- **Word:** Data is only considered to match where it is a whole word. A word is a sequence of letters preceded and followed by a line break or a character other than a letter. Searches performed in this mode are case insensitive.
- **Basic word:** Similar to word search mode but the characters valid in the “word” are extended to cover acceptable syntax for a Basic program variable name.

The default search mode may changed during initialisation or by commands entered at the SED command prompt. Use of the up line and down line functions whilst entering the search string can also be used to change the mode except when collecting functions for a macro.

When a search is included in a macro, the prompt for the search string only occurs when collecting the macro. Subsequent executions use the same search string. Multiple search functions in a macro may have different search strings.

Reverse search (Ctrl-R or Esc-R)
The reverse search function prompts for a search string and moves backwards to the previous occurrence of this string within the record. The prompt defaults to the same string as the previous search function (if any). Search modes are as described above.

View (Ctrl-X V)
The view function prompts for a search string displays a new buffer containing all lines beyond the current cursor position that include this string, showing the line number and the text of the line with leading spaces removed. Use any of the standard SED cursor movement functions to position on a line and press the return key to go to this line of the full record text. The view buffer is automatically deleted at any editing function that invalidates its content (e.g. entering new text).

Nudge down (Ctrl-X Ctrl-N)
This function moves the displayed window down the record by one line. The cursor remains in the same position within the data unless it is on the top line of the screen in which case it will move down by one line.

Nudge up (Ctrl-X Ctrl-P or Ctrl-X Ctrl-Z)
This function moves the displayed window up the record by one line. The cursor remains in the same position within the data unless it is on the last line of the screen in which case it will move up by one line.

Align text (Esc-tab)
This function aligns the data on the current line to align text with the preceding line.
SED - Data insertion

Data is inserted at the current cursor position. If overlay mode is set the new data overwrites any existing data at this position, otherwise it is inserted before the character under the cursor. Overlay mode may be toggled using the `overlay` function (Ctrl-O) or the `OVERLAY` command.

The return key inserts a newline. If indent mode is active (see the `INDENT` command), the cursor is indented to line up with the previous line.

Any character other than a field mark or item mark may be inserted. The `quote character` function (Ctrl-Q or Esc-Q) allows insertion of non-printing characters. It may be used in four ways:

- Followed by a number of up to three digits, it inserts the character with that decimal ASCII sequence (0 to 253 only).
- Followed by V, S or T, it inserts a value mark, subvalue mark or text mark respectively.
- Followed by X and two (non-ECS mode) or four (ECS mode) hexadecimal digits, it inserts the character with that character code point value.
- Followed by K, it waits for a key to be pressed and then inserts the key binding code required for this key as described under Dynamic Key Bindings.

Followed by any other character, usually a non-printing character, it will insert that character.
SED - Copying, deleting and restoring data

Delete char (Ctrl-D or Delete)
The character at the current cursor position is deleted unless the cursor is positioned at the end of the line in which case the following line is appended to the current line.

Backspace (Ctrl-H or Backspace)
The backspace function removes the character to the left of the cursor position unless the cursor is already at the start of the line in which case the current line is appended to the previous line.

Kill line (Ctrl-K)
The kill line function behaves in one of two ways depending on the cursor position.

If there is data at or beyond the cursor position on the current line, the line is truncated at the cursor position.

If the cursor is beyond the last data character on the current line, the line break is removed, bringing data up from the next line.

The data removed by consecutive uses of the kill line function is placed in the kill buffer (see Functions that Operate on a Block of Data). Any other function will cause a later use of the kill line function to reset the kill buffer.

The kill line function may also be used in an explore buffer to delete the record named on the current line.

Delete word (Esc-D)
Text is deleted from the current cursor position to the end of the next word.

Back del word (Esc-Backspace)
Text is deleted backwards from the current cursor position to the start of the previous word.
SED - Working with multivalued data

When in a normal data window, some edit functions normally associated with file handling actions have special usage and allow entry to and exit from value edit mode.

Value edit mode takes the contents of a multivalued field and displays each value as a separate line in much the same way as the EV command of the ED line editor. While value edit mode is active, the original parent buffer becomes read only.

Used with a dictionary I-type entry, this mode breaks compound I-types into separate lines to simplify editing.

**Up to parent** (Ctrl-X U)
Used in a value edit buffer, this function returns to the parent buffer, saving any changes into the main buffer.

**Dive** (Ctrl-X Ctrl-D)
When not positioned on a $INCLUDE statement, this function enters edit value mode.

**Delete buffer** (Ctrl-X K)
When in a value edit buffer, this function returns to the parent buffer, discarding any changes.
SED - Functions that operate of a block of data

The editor maintains reference to two positions within the record; the cursor position and the **mark**.

**Set mark** (Esc-.)
The mark is set at the current cursor position.

**Swap mark** (Ctrl-X Ctrl-X)
The swap mark function interchanges the positions of the cursor and the mark. It has no effect if the mark has not been set.

**Copy region** (Esc-W)
The text within the region bounded by the mark and the current cursor position (which may be either way around) is copied to the **kill buffer**.

**Delete region** (Ctrl-W)
The text within the region bounded by the mark and the current cursor position (which may be either way around) is copied to the kill buffer and deleted from the record.

**Insert kill buffer** (Ctrl-Y or Esc-Y)
The contents of the kill buffer are inserted at the current cursor position. The kill buffer retains a copy of this text thus allowing multiple insertions.

**Lowercase region** (Ctrl-X Ctrl-L)
All words in the region between the mark and the cursor are converted to lowercase.

**Uppercase region** (Ctrl-X Ctrl-U)
All words in the region between the mark and the cursor are converted to uppercase.
SED - Changing text

Replace (Ctrl-X R)
The replace function prompts for a search string and a replacement string. All occurrences of the search string from the current cursor position to the end of the record are replaced by the replacement string. Search modes are applied as described for the forward search function.

When a replace is included in a macro, the prompt for the search and replacement strings only occurs when collecting the macro. Subsequent executions use the same strings. Multiple replace functions in a macro may have different strings.

Query replace (Ctrl-X Q)
This function is like the replace function except that a prompt is issued for each possible replacement. Entering a space causes replacement to occur, the return key causes no replacement and Ctrl-G causes the function to be aborted. Search modes are applied as described for the forward search function.

Capital init (Esc-C)
This function locates the next word in the record and converts it so that the first letter is in uppercase and the remainder is in lowercase.

Lowercase (Esc-L)
This function locates the next word in the record and converts it to lowercase.

Uppercase (Esc-U)
This function locates the next word in the record and converts it to uppercase.

Toggle chars (Ctrl-T)
The character at the cursor position and the preceding character are interchanged. This function has no effect if the cursor is at the start of the line or beyond the end of the line.

Close spaces (Esc-space)
All spaces surrounding the current cursor position are replaced by a single space.
SED - Macros

The editor allows multiple command sequences to be collected as a macro for subsequent re-execution.

The functions marked with an asterisk in the table at the start of this description cannot be included within a macro and will be rejected during collection of a macro.

**Start macro** (Ctrl-X open bracket)
Subsequent keystrokes are collected to form the macro. Each function is executed as it is collected.

**End macro** (Ctrl-X close bracket)
Terminates collection of a macro.

**Execute macro** (Ctrl-X E)
Causes execution of the macro. The repeat counter may be used to execute the macro multiple times.

Use of the repeat function after an execute macro function will repeat the macro. If the repeat count is set for the repeat function, the macro is executed that number of times. Otherwise the repeat count applied to the previous execution of the macro is used.
SED - File handling

**Save record** (Ctrl-X S or Ctrl-X Ctrl-S)
The current record is saved, overwriting the previous version (if any).

**Write record** (Ctrl-X W or Ctrl-X Ctrl-W)
A prompt is issued for the file and record names under which the data is to be saved. The file name may be prefixed by DICT to indicate that the dictionary portion is required.

After saving the data, the selected file and record names become current. A later use of the **save record** function would replace this record not the record that was specified when editing began.

The update lock is transferred to the new record by this function.

**Import** (Ctrl-X I)
A prompt is issued for the file and record names for the data to be imported. The data is then inserted at the current cursor position.

**Export** (Ctrl-X X)
The **export** function saves the contents of the region between the cursor and the mark to a file. A prompt is issued for the file and record names under which the data is to be saved. The file name may be prefixed by DICT to indicate that the dictionary portion is required.

**Find record** (Ctrl-X Ctrl-F)
The editor can handle multiple records simultaneously. The **find record** function prompts for file and record names and reads the record for editing. The file name defaults to that of the record in the current buffer. If the named record does not exist, a prompt is issued to confirm that it is to be created. Multiple records are of particular use when copying data between records.

The **find record** function will accept the record name on the same response line as the file name as an alternative to entering the file and record names in response to separate prompts. This is achieved by separating the file and record names by a single space. SED still attempts to open a file with a name corresponding to the complete prompt response first to allow for the unlikely situation of a file name which includes a space.

The **find record** function maintains a stack of the last 10 files referenced by this command. The **up line** and **down line** functions can be used to restore previous file names from the stack when the file name prompt is displayed.

**List buffers** (Ctrl-X Ctrl-B)
The **list buffers** function displays a list of all currently defined buffers, showing the buffer number, and its corresponding file and record names. It is of use when multiple records have been read. The colon normally following the buffer number is replaced by an asterisk if the buffer has been modified or a hyphen if it is a read-only buffer.

A marker is displayed to the left of the buffer number of the current buffer. This marker can be moved up and down using the **up line** and **down line** functions. Pressing **return** selects the marked buffer. The **cancel** function will revert to the previous buffer.

**Next buffer** (Esc-N)
When multiple records are in use, this function selects the next buffer as the current buffer for display and editing.

**Previous buffer** (Esc-P)
When multiple records are in use, this function selects the previous buffer as the current buffer for display and editing.

**Goto buffer (Ctrl-X B)**
This function selects the buffer identified by the repeat counter value.

**Delete buffer (Ctrl-X K)**
The current buffer is deleted, freeing it for use by other records. A confirmation prompt is displayed if the buffer has been modified.

**List records (Ctrl-X D)**
Displays a list of the records in a file allowing the user to explore the content of the file. The editor prompts for the file name which defaults to that of the record in the current buffer.

The file name may be followed by a selection template separated from the file name by a single space. If this template begins with the word **LIKE** or **WITH** the entire template is taken as a selection clause for the internally executed SELECT command. If the template does not begin with either of these words, it is assumed to be a pattern for matching with the SELECT statement LIKE clause. Thus a template of

```
PRT...
```

is equivalent to

```
WITH @ID LIKE PRT...
```

Single or double quotes may be used as required to ensure correct parsing of the template. Multiple conditions may be included exactly as in a SELECT statement. A template not starting with **LIKE** or **WITH** may not include both single and double quotes.

The list of records is displayed in a buffer which is tagged with a pseudo-record name of -Explore. If an explore list already exists for this file, the list is rebuilt by this function, thus showing any changes to the file content.

The normal editor functions may be used to move around this buffer but it cannot be updated. The **return** key or the **dive** function (Ctrl-X Ctrl-D) will cause the editor to read and display the record identified by the line of the explore list on which the cursor lies. If this record is already loaded into a buffer, the existing buffer is selected.

The **kill line** function executed in an explore buffer deletes the record named on the current line. A confirmation prompt is issued before the record is deleted. If this record is currently loaded into a buffer, the buffer is also deleted. A second confirmation prompt is issued to confirm this action.

**Up to parent (Ctrl-X U)**
Moves 'up' from the displayed record to an explore list for the file holding the record. If the explore list already exists, the list is not rebuilt by this function.

Used in an explore buffer, this function moves up to a display of all files in the VOC. Diving into a file from this list shows a list of records in the file. It does not dive into the VOC record itself.

Used in a value edit buffer, this function returns to the parent buffer, saving any changes into the main buffer.

**Dive (Ctrl-X Ctrl-D)**
When positioned on a $INCLUDE statement of a QMBasic program, this function loads the associated include record into a new buffer.

When not positioned on a $INCLUDE statement, this function enters edit value mode.
SED - Repeating functions

A function may be repeated multiple times by use of the repeat count prefix. This is performed by use of the ESCape key followed by the number of times the command is to be repeated. The repeat count is displayed on the status line. Functions for which a repeat count is irrelevant ignore it.

Functions within macro definitions may be repeated in this way and the execution of the macro itself may also be repeated.

The repeat function (Ctrl-C or Ctrl-U) repeats the previously executed function. The repeat count prefix can also be used with this function.
**SED - Miscellaneous functions**

**Cancel** (Ctrl-G)
The **cancel** function aborts partially entered commands such as searches or repeat counts.

**Refresh** (Ctrl-L)
The **refresh** function rebuilds the screen display if it should be corrupted in any way. The current line is placed at the centre of the screen if possible. Alternatively, the repeat counter may be used to specify the screen line number on which the current line is to be placed.

**Expand character** (Ctrl-X =)
Certain control characters (e.g. tab, form feed) are represented on the screen by question marks. The **expand character** function displays the character sequence number for the character at the cursor position. It also shows the cursor position in terms of field, value and subvalue which is useful when editing data files.

**Split window** (Ctrl-X 2)
The **split window** function divides the screen into two separate windows. These initially hold two views of the same buffer but can be used to show different buffers.

**Unsplit window** (Ctrl-X 1)
The **unsplit window** function returns to single window mode from a split window view.

**Split up** (Ctrl-X P or Ctrl-X Z)
When using a split window display, this moves the split point up, enlarging the lower part of the window.

**Split down** (Ctrl-X N)
When using a split window display, this moves the split point down, enlarging the upper part of the window.

**Toggle window** (Ctrl-X O)
The **toggle window** function moves between the two windows of a split window display.

**Quit** (Ctrl-X Ctrl-C or Ctrl-X C)
The **quit** function terminates an editing session. If any records have been modified but not written back to disk, **SED** will prompt for confirmation that this action is intended.

Where a select list is in use, **SED** will move to the next item from this list.

Where only a file name was specified on the command line, **SED** will prompt for a further record name.

**Run extension** (Esc-E)
The **run extension** function prompts for an extension program name and executes that extension.

**CompRun** (Ctrl-X M)
The **CompRun** function compiles the QMBasic program in the current window and, if the compilation is successful, runs the program.

**Command** (Esc-X)
The **command** function is used to alter various long term states of the editor and to perform other actions.

[Click here for a list of commands](#).
SED - Commands

The command function is used to alter various long term states of the editor and to perform other actions. After entering the command function (Esc-X), a prompt for the command name is issued. The following commands are available:

**BASIC** Saves the current record and runs the Basic compiler. This command is similar to COMPILE (described below) but it inserts marker lines into the source program where error or warning messages have been produced by the compiler.

**BWORD** Sets Basic Word mode for search and replace functions as described for the forward search function.

**CASE OFF** Sets Case Insensitive mode for search and replace functions.

**CASE ON** Sets Case Sensitive mode for search and replace functions.

**COMPILE** Saves the current record and runs the Basic compiler. For compatibility with SED on other platforms, if this record includes a line

*CATALOG catname

the compiled program will be catalogued automatically after successful compilation. The actual cataloguing command issued is

CATALOG filename catname recordname

It is thus possible to perform either private or global cataloguing. The same action can be performed on QM using the $CATALOG compiler directive.

**EXPAND.TABS** Expands tab characters in the record being edited to align data on the columns determined by the current setting of the tab interval. The default tab columns are 11, 21, 31, etc.

**FORMAT** Applies standard format rules to the layout of a QMBasic program.

**FUNDAMENTAL** Reverts to the default (fundamental mode) key bindings.

**INDENT** Toggles indent mode.

**KEYS** Displays the name of the active key binding record.

**LNUM** The LNUM command controls display of line numbering. Used alone, it toggles the current state of numbering on the displayed buffer. It may also be used with the following qualifiers:

- **OFF** Turn off line numbering in the current buffer.
- **ON** Turn on line numbering in the current buffer.
- **ALL** Turn on line numbering in all buffers.
- **OFF ALL** Turn off line numbering in all buffers.
- **ON ALL** Synonym for **ALL**.

**LOAD.KEYS** Loads a named key binding record.

**OVERLAY** Toggles overlay mode. The OVERLAY function (Ctrl-O) has the same effect.

**QUIT** Ends editing of the current record in a similar way to the quit key sequence but also aborts any select list.

**READ.ONLY** Makes the current buffer read-only, helping to avoid accidental updates.
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RELEASE</strong></td>
<td>Releases the update lock on the current record.</td>
</tr>
<tr>
<td><strong>RUN</strong></td>
<td>Entering <strong>RUN</strong> with no qualifying details runs the program in the current buffer (which must have been compiled). If <strong>RUN</strong> is followed by any qualifying details, the command is passed to the command processor for execution.</td>
</tr>
<tr>
<td><strong>SAVE.KEYS</strong></td>
<td>Saves the key bindings as described later.</td>
</tr>
<tr>
<td><strong>SPOOL</strong></td>
<td>Spools the contents of the current buffer to print unit zero.</td>
</tr>
<tr>
<td></td>
<td>The <strong>SPOOL</strong> command has optional qualifiers which may be used together if required. <strong>LNUM</strong> adds a line number prefix to each line printed. <strong>REGION</strong> prints only the lines between the mark and the cursor (which may be in either order). <strong>AT</strong> followed by a printer name selects the destination printer. <strong>NHEAD</strong> or <strong>NOHEAD</strong> suppresses the banner page.</td>
</tr>
<tr>
<td><strong>TABS</strong></td>
<td>Sets the tab interval to be used by the <strong>tab</strong> function and the <strong>EXPAND.TABS</strong> command. The value given after <strong>TABS</strong> must be in the range 1 to 99.</td>
</tr>
<tr>
<td><strong>TRIM</strong></td>
<td>Removes trailing spaces from all lines of the current buffer.</td>
</tr>
<tr>
<td><strong>WORD</strong></td>
<td>Sets Word mode for <strong>search</strong> and <strong>replace</strong> commands.</td>
</tr>
<tr>
<td><strong>XEQ {cmd}</strong></td>
<td>Executes QM command <strong>cmd</strong>. The <strong>XEQ</strong> command prefix is only required where <strong>cmd</strong> is also an internal <strong>SED</strong> command. All commands not recognised by <strong>SED</strong> are passed to the command processor for execution.</td>
</tr>
</tbody>
</table>

The **command** function maintains a stack of the last 100 commands executed. The **up line** and **down line** functions can be used to restore commands from the stack when the command prompt is displayed.
SED - Setting up default modes

On entry, SED looks for a record named &SED.OPTIONS& in the VOC file. This record may be used to set up default configuration data.

Field 1 of the record should contain X, the record type code.

Field 2 may contain the following keywords separated by spaces:

- BWORD Turn on Basic word search mode.
- CASE.OFF Turn off search case sensitivity.
- CASE.ON Turn on search case sensitivity.
- INDENT Turn on indentation mode.
- LNUM Line numbering is to be on in all windows by default.
- NO.CASE.EXPLORE Ignore casing when building an explore buffer.
- NO.INVERT Turn of case inversion during edit.
- OVERLAY Turn on overlay mode.
- SCROLL1 Scroll by just on line for cursor up/down that moves out of displayed text.
- TABS $n$ Sets the default tab interval to $n$ columns.
- WORD Turn on word search mode.

Fields 3 and 4 may contain the name of a key binding record.

Field 5 may be used to modify the default tab interval used by the tab function and by the EXPAND.TABS command. The value in this field must be between 1 and 99.

The record should contain no other data. Other fields may be used by later revisions of SED.
SED - Source control

SED includes a mechanism that may be used to implement a source control system or other special processing when updated records are written to disk.

Whenever a write is attempted using the save record or write record functions, SED checks for a catalogued subroutine named SOURCE.CONTROL. If this is present, it is called to validate whether data may be written to the file. The subroutine is defined as:

```
SUBROUTINE SOURCE.CONTROL(dict.flag, file.name, record.name, rec, caller, write.allowed, updated)
```

where

- `dict.flag` is "DICT" if attempting to write to a dictionary, a null string otherwise.
- `file.name` is the name of the file to be written.
- `record.name` is the name of the record to be written.
- `rec` is the record data.
- `caller` is 1 to indicate a call from SED, 2 for a call from ED.
- `write.allowed` should be returned by the subroutine as True if the write may be performed, False if not. This argument is 1 on calling the subroutine.
- `updated` should be set True if the subroutine has made any changes to the data in `rec`. This argument is 0 on calling the subroutine.

The source control subroutine may be used in any way you wish. Typical uses are simple validation of whether the record may be written or addition of edit history information prior to writing the data. In the latter case, where changes are made to the data passed via the `rec` argument, the `updated` flag should be set True so that SED rebuilds its working copy of the data on return.

The following simple example appends a history entry to the end of any record edited in BP or a file with a name ending .BP but ignores dictionaries.

```
SUBROUTINE SOURCE.CONTROL(DICT.FLAG, FILE.NAME,                          RECORD.NAME, REC,                          FULL.SCREEN, WRITE.ALLOWED,                          UPDATED)

IF LEN(DICT.FLAG) THEN RETURN ;* No interest in dictionary
IF FILE.NAME # 'BP' AND FILE.NAME[3] # '.BP' THEN RETURN

DISPLAY @(-1):
DISPLAY SPACE(27) : "SOURCE CONTROL INFORMATION" : @(-4)
DISPLAY "Change description:"
PROMPT ""
HDR = ""
TAG = OCONV(DATE(), "D2EL")
LOOP
  DISPLAY @67) : "<" : @0:
  INPUT S, 66

```
WHILE LEN(S)
    HDR<-1> = "* " : TAG : " " : S[1, 66]
    TAG = SPACE(9)
    REPEAT
    REC<-1> = HDR
    UPDATED = @TRUE
    RETURN
END
SED - Dynamic key bindings

The key bindings used in the function descriptions in this manual are the defaults used by SED, known as the fundamental mode bindings. The dynamic key binding system allows these to be changed if desired.

On entry, SED looks for a file named &SED.BINDINGS&. In most cases where this exists, it would be set up as a remote file pointer to a single file shared by all accounts. This file contains records defining named sets of key bindings.

SED works through a sequence of steps in locating the binding record to be used. This sequence, described below, is designed to allow almost any user or terminal specific precedence rules to be created.

Try the (optional) name in field 3 of the &SED.OPTIONS& VOC record.

Try a name constructed from loginname-termtype where both the login name and the terminal type are mapped to upper case.

Try USER.loginname where the login name is mapped to upper case.

Try termtype where the terminal type is mapped to upper case.

Try the (optional) name in field 4 of the &SED.OPTIONS& VOC record.

Try DEFAULT.

Revert to the fundamental mode bindings.

If there is no &SED.BINDINGS& file, SED uses the fundamental mode key bindings.

Key bindings can be changed at run time by use of the LOAD.KEYS command. This takes the name of a key binding record as its argument and the bindings defined in that record will be loaded.

Fundamental mode can be restored by use of the FUNDAMENTAL command.

The SAVE.KEYS command saves the current key bindings into a record named in the command argument (e.g. SAVE.KEYS DEFAULT to generate the DEFAULT bindings record).

Key binding records consist of a series of lines (fields), each of which defines one or more bindings for the function associated with that line separated by value marks. Control characters are represented by @A, @B, etc. for Ctrl-A, Ctrl-B, etc. Thus escape is @[]. The @ character can be included in a binding as @@. All characters before the first @ are ignored thus allowing comments to be included. The SAVE.KEYS command inserts the function name as a comment in each binding.

One way to create new bindings is to use SAVE.KEYS to create a template which is then edited to make any desired modifications. The quote char function followed by K inserts the key binding expansion of the next key pressed and can be used while editing a key binding record. This function simply inserts the characters sent by the next key pressed. It does not validate the modified binding record for duplicate or ambiguous key sequences.

Used in this way, quote char waits for the first character to arrive and then sleeps for one second. It then inserts the codes for all characters waiting to be processed. This ensures that the entire sequence sent by, for example, a function key is processed. You should wait until the character sequence appears on the screen before typing any further characters.
SED - Extension Programming

The extension programming feature allows users to add new functions to the editor. These may range from simple insertion or modification of text with a single keystroke to complex programs that manipulate the data to achieve tasks that are specific to your own editor usage.

Extensions may be stored in any QM file. Ideally, this should be a dynamic hashed file and SED uses &SED.EXTENSIONS& by default. A list of alternative locations to look for extensions can be defined using the $SEXTENSIONS variable described in SED Extensions - Variables, Constants and Functions. For all except the START.UP extension described below, if an extension is not found in any of these locations, a final check is made in the &SED.EXTENSIONS& file in the QMSYS account.

The editor looks for and executes an extension named START.UP on loading the first data record. Extension names must be upper case and consist only of letters, digits, periods (.) and dollar signs.

Extensions must be compiled before use. This is performed using SED’s COMPILe command which recognises extension programs as distinct from QMBasic programs. The compiled version is stored in the same file as the source but with a suffix of -EXT added to the record name.

An extension is executed by the SED run extension function which is normally bound as Esc-E. Extensions may also be bound directly to user defined key sequences or made available via the command function. Typically this would be performed by the optional START.UP extension.

All those Brackets...

The extension programming language is based on the LISP language. This yields programs with very simple, though somewhat strange looking, structure.

Extension programs come in two types; procedures perform some operation whereas functions also return a value. The outermost structure of a procedure is

PROC
  {
    ...operations...
  }

and for a function it is

FUNC
  {
    ...operations...
  }

where ...operations... is a sequence of steps that makes up the program.

A function returns a value using the return operation at any point in its execution. There is an implicit return of a zero value at the end of the function text.

Each of these operations is also a procedure or a function in that they perform some operation on the editing environment and/or they return information that can be used by other operations.

Each complete operation and any data items on which it works are enclosed in a further layer of brackets. Since the language allows functions to be nested to a high degree, a typical program at first appears to contain a large number of brackets. By applying some thought to the layout of the
program, the actual structure can be made very clear to the reader. The language has no built-in format rules except that no token (individual word, constant, variable name, etc.) can span lines.

For example, a simple program to provide the equivalent of the STAMP command of ED could be written as:

```plaintext
PROC
  (goto.col 1)
  (insert "*Last updated by ' @who ' (' @logname ')'
    ' at ' (oconv @time 'MTS')
    ' on ' (oconv @date 'D4/'))
  (newline 1)
)
```

The screen is not updated during execution of an extension program except by functions that are documented as doing so. This allows the extension program to perform complex data movements without the screen continually tracking the internal workings of the extension. The screen is updated when the extension terminates.

For detailed information follow the links below:

Variables, constants and functions
Standard variables and functions
Argument passing
Local procedures and functions
An example of a complex extension

SED Extensions - Variables, constants and functions

All extension program source text is case insensitive except for literal strings.

Local Variables

These are names commencing with a letter and containing only letters, digits, periods (.) and dollar signs. A procedure or function can use at most 250 local variables. Local variables are private to the procedure or function and the same name used in another procedure refers to a different local variable.

Global Variables

These are names commencing with a dollar sign and containing only letters, digits, periods (.) and dollar signs. Global variables are common to all extensions and retain their values until the user leaves SED.

Global variables beginning with two dollar signs are reserved. Current usage of these names is:

```
$$EXTENSION.FILES
```
Contains a space separated list of the files to be searched for extension programs. By default, this variable contains &SED.EXTENSIONS& but it may be modified at any time by user written extension programs. File names are used left to right. If the requested extension is not found in any of these files, a final attempt is made using the
&SED.EXTENSIONS& file in the QMSYS account. Once an extension program has been loaded on first use, it remains loaded unless it is specifically unloaded by use of `unload` or by recompilation by the user who has it loaded.

**Constants**

Numeric constants are written as a sequence of digits, optionally prefixed by a sign or including a decimal point. A numeric constant with an absolute value of less than one must be written with a leading zero (e.g. 0.5).

A string constant may be enclosed in either single or double quotes. It may contain any character except the mark characters (which are available using the tokens shown below) and ASCII character 0 (nul).

The logical values are represented as 0 for False and 1 for True. In general, use of any value other than zero is treated as True by operations that expect logical values as their arguments.

**Key Tokens**

The `get.key` function returns a code that relates to an internal function number. Each function has a corresponding symbolic name. These all begin with a period (.) and are the same as the comment inserted at the start of each line of a key binding record with spaces replaced by period. The names are:

```
newline .start.line .end.line .back.char
.fwd.char .up.line .down.line .top
.bottom .page.up .page.down .del.char
.backspace .kill.line .save.record .quit
.overlay .tab .goto.line .toggle.chars
.fwd.search .replace .query.replace .swap.mark
.execute.macro .nudge.down .nudge.up .set.mark
.delete.region .copy.region .insert.killed .forward.word
.delete.word .import .reverse.srch .lowercase
.uppercase .capital.init .back.word .del.back.word
.close.spaces .next.buffer .prev.buffer .goto.buffer
.delete.buffer .up.to.list .repeat .refresh
.quote.char .list.buffers .find.record .write.record
.start.macro .end.macro .expand.char .list.records
.export .command .cancel .run
.insert .align.text
```

**System Variables**

Extension programs may examine the current state of many editor features by use of system variables. These all begin with a percent sign (%) and are read only (i.e. they cannot be used in a set function).
The following system variables may also be referenced in extensions:

@IM    @FM    @VM    @SM
@TM    @LOGNAME    @CRTHIGH    @CRTWIDE
@DATE    @TIME    @PATH    @SENTENCE
@WHO    @TTY    @USERN0

Comments

A comment is introduced by an asterisk (*) and extends to the end of the line.

Erroneous Programs

Extension programs may test whether the buffer being processed is read only. Attempts to change such a buffer are ignored. No error is displayed.

Variables are type variant in the same way as for QMBasic programs and they follow the same rules. In most cases, attempts to use a non-numeric value where a number is required result in use of a default.

SED Extensions - Standard variables and functions

Variables are type variant in the same way as for QMBasic programs and they follow the same rules. In most cases, attempts to use a non-numeric value where a number is required result in use of a default.

Buffer Information

%buffer.no Returns the current buffer number.
%buffer.type Returns a type code for the current buffer:
    1 : data buffer
    2 : explore buffer
    3 : file list buffer
%changed Returns a logical value indicating whether the current buffer is marked as having been changed since it was last saved.
%col Returns current column number.
%current.char Returns character under cursor.
%current.line Returns text of current line.
%file Returns file name for current buffer. This is prefixed by “DICT “ if the buffer is from the dictionary part of the file.
%height Returns the number of lines on the screen display excluding the two editor status lines.
%id Returns the record id for the current buffer. For an explore buffer or a file list buffer it returns a pseudo id value.
%kill.buffer Returns the content of the kill buffer.
%line Returns current line number.
%line.len Returns length of current line.
%lines Returns number of lines in current record.
%mark.col Returns the column number of the mark position, zero if no mark set.
%mark.line  Returns the line number of the mark position, zero if no mark set.
%overlay   Returns a logical value indicating whether the current buffer is operating in overlay mode.
%pan       Returns the current pan position. This is the column number (from one) of the leftmost displayed column of the current buffer.
%read.only Returns a logical value indicating whether the current buffer is read only.
%scroll    Returns current scroll position. This is the line number of the data displayed on the first line of the screen.
%tab.interval Returns the current setting of the tab interval.
%width     Returns the width of the screen display.
(delete.buffer) Deletes the current buffer. Unlike the SED delete.buffer keyboard function, this operation allows deletion of modified buffers without any confirmation checks.
(find.buffer file rec) Returns the internal number of the buffer holding data from the given file and record. Use of a null string as file allows location of a scratch buffer. This function returns zero if no such buffer exists.
(find.record file rec) Reads record rec from file into a newly created buffer. If the record is already loaded, it simply switches to that buffer. This function returns True if successful, False if not.
(goto.buffer n)  Select buffer number n.
(make.buffer name) Makes a new scratch buffer named name. Scratch buffers may be used for internal purposes of the extension program or as the place to create data which will later be written to disk. Use of %file with a scratch buffer returns a null string. %id returns name. This function returns True if successful, False if not. If a scratch buffer of the given name already exists, it makes that buffer current and returns True.
(next.buffer n)  Select the buffer with buffer number n greater than the current one. This operation cycles to the lowest numbered buffer if necessary.
(prev.buffer n)  Select the buffer with buffer number n less than the current one. This operation cycles to the highest numbered buffer if necessary.
(save.record)    Saves the current buffer. It has no effect on a scratch buffer or if the current buffer is read only. The normal source control actions are performed if this editor feature is in use.
(set.changed n)  Sets the status of the buffer changed flag. If n is zero, the buffer is marked as unchanged, otherwise it is marked as changed. Use this operation with care as it can result in data not being saved on leaving the editor. It is of particular use with scratch buffers.
(set.overlay n)  Sets the overlay status of the current buffer. If n is zero, overlay is turned off, otherwise it is turned on.
(set.pan n)      Set the current pan position to n. The value of n must be greater than zero. The next screen update (paint or implicit from some other function) will move the pan position if the current cursor position is not in the displayed region of the buffer.
(set.read.only n) Sets the read only status of the current buffer. If n is non-zero, the buffer is marked as read only, otherwise modifications are allowed. This function is of particular use with scratch buffers. It is ignored for buffers containing explore record lists or file lists.
(set.scroll n) Sets the current scroll position to n. The value of n must be greater than zero and must not be greater than the number of lines in the current buffer. The next screen update (paint or implicit from some other function) will move the scroll position if the current line is not in the displayed region of the buffer.

(set.tab.interval n) Sets the tab interval to n. The value of n must be in the range 1 to 99.

(write.record file rec) Writes the current buffer to record rec in file. This operation may result in the current buffer being renamed. The normal source control actions are performed if this editor feature is in use. The write.record operation is ignored if the target record is locked by another user.

Moving the Cursor, Insertion and Deletion

(backspace n) Backspace n characters.

(back.char n) Move cursor left n columns.

(back.word n) Moves the cursor backward by n words.

(bottom) Move to after the final line of the buffer.

(copy.region) Copies the region between the mark and the cursor, which may be in either order, to the kill buffer.

(delete.region) Copies the region between the mark and the cursor, which may be in either order, to the kill buffer and deletes the text from the buffer.

(del.back.word n) Deletes n words backwards from the current cursor position.

(del.char n) Delete n characters.

(del.word n) Deletes n words from the current cursor position.

(down n) Synonym for down.line.

(down.line n) Move cursor down n lines.

(end) Synonym for end.line.

(end.line) Move to the end of the current line.

(fsearch mode str) Search forwards for string str. The mode indicates the style of search:
    0 : Use the current search mode setting
    1 : Case sensitive
    2 : Case insensitive
    3 : Word search, case insensitive
This function returns a logical value indicating whether the search was successful. An unsuccessful search does not move the current position.

(fwd.char n) Move cursor right n columns.

(fwd.word n) Moves the cursor forward by n words.

(left n) Synonym for back.char.

(goto.col a) Go to column a of current line. The line is extended if necessary by adding trailing spaces.

(goto.line a) Go to column 1 of line a.

(home) Synonym for start.line.

(insert s) Insert string s at current cursor position. Any field marks in the text are treated as newlines. The insert function can take more than one argument, each of which is inserted in turn.
(newline rpt) Insert \textit{rpt} newlines at the current cursor position.
(page.down \textit{n}) Move cursor down by \textit{n} display pages.
(page.up \textit{n}) Move cursor up by \textit{n} display pages.
(retype \textit{s}) Replace the current line by string \textit{s}.
(right \textit{n}) Synonym for \texttt{fwd.char}.
(rsearch \textit{mode} \textit{str}) Search backwards for string \textit{str}. The \textit{mode} indicates the style of search:
\begin{itemize}
  \item 0 : Use the current search mode setting
  \item 1 : Case sensitive
  \item 2 : Case insensitive
  \item 3 : Word search, case insensitive
\end{itemize}
This function returns a logical value indicating whether the search was successful. An unsuccessful search does not move the current position.
(set.case \textit{mode} \textit{rpt}) Changes the case of the next \textit{rpt} words in the current buffer. The \textit{mode} argument is:
\begin{itemize}
  \item 0 : Set lower case
  \item 1 : Set upper case
  \item 2 : Set capital initial casing
\end{itemize}
(set.mark) Set the mark at the current cursor position.
(start.line) Move to the start of the current line.
(swap.mark) Interchange the cursor and mark positions. Has no effect if there is no mark position defined.
(tab \textit{rpt}) Advances the cursor to the next tab position as determined by the current setting of the tab interval. Additional spaces are appended to the current line if necessary. The \textit{rpt} argument specifies how many tab positions the cursor is to be advanced.
(toggle.chars) Interchanges the character at the cursor position with that in the preceding column.
(top) Move to the start of line 1.
(up \textit{n}) Synonym for \texttt{up.line}.
(up.line \textit{n}) Move cursor up \textit{n} lines.

\textbf{Arithmetic, String and Logical Functions}

(add \textit{a} \textit{b}) Returns \textit{a} + \textit{b}.
(and \textit{a} \textit{b}) Forms logical relationship \textit{a} and \textit{b}. The \texttt{and} function can take more than two arguments, each of which is and'ed in turn.
(cat \textit{a} \textit{b}) Returns concatenation of strings \textit{a} and \textit{b}. The \texttt{cat} function can take more than two arguments, each of which is concatenated in turn.
(char \textit{n}) Returns the character with ASCII character value \textit{n}.
(del \texttt{str} \textit{f} \textit{v} \textit{s}) Returns a string formed from \texttt{str} with field \textit{f}, value \textit{v}, subvalue \textit{s} deleted. Specify \textit{v} and \textit{s} as zero to delete field \textit{f}, \textit{s} as zero to delete field \textit{f}, value \textit{v}.
(div \textit{a} \textit{b}) Returns \textit{a} / \textit{b}.
(downcase \textit{s}) Returns string \textit{s} converted to lower case.
(eq \textit{a} \textit{b}) Test \textit{a} equal to \textit{b}. 
(extract str f v s) Returns field f, value v, subvalue s of string str. Specify v and s as zero to extract field f, s as zero to extract field f, value v.

(field str d n count) Returns a portion of str starting at the n’th substring delimited by d and extending for count such substrings.

(ge a b) Test a greater than or equal to b.

(gt a b) Test a greater than b.

(ins str f v s new) Returns a string formed from str with new inserted before field f, value v, subvalue s. Specify v and s as zero to insert before field f, value v.

(int a) Returns the integer portion of value a by truncating any fractional part.

(le a b) Test a less than or equal to b.

(len s) Returns the length of string s.

(lt a b) Test a less than b.

(max a b) Returns the maximum of a and b. If either value is not numeric, this function returns the item that appears last in collating sequence order.

(min a b) Returns the minimum of a and b. If either value is not numeric, this function returns the item that appears first in collating sequence order.

(mul a b) Returns a * b.

(ne a b) Test a not equal to b.

(not a) Returns logical inverse of a.

(or a b) Forms logical relationship a or b. The or function can take more than two arguments, each of which is or’ed in turn.

(pad s n) Pad string s with spaces to be n characters long. If string s is already at least n characters long, this function returns the original string.

(rem a b) Returns the remainder of dividing a by b.

(rep str f v s new) Returns a string formed from str with field f, value v, subvalue s replaced by new. Specify v and s as zero to replace field f, s as zero to replace field f, value v.

(seq c) Returns the ASCII character sequence number of the first character of c.

(sub a b) Returns a – b.

(substr s a b) Returns a substring from s starting at character a, b characters long.

(trim s) Returns string s with all leading and trailing spaces removed and all multiple embedded spaces replaced by a single space.

(trimb s) Returns string s with all trailing spaces removed.

(trimf s) Returns string s with all leading spaces removed.

(upcase s) Returns string s converted to upper case.

User Input and Screen Display

%key.char Returns the character from the last get.key if it was an insert action. For other actions, it returns a null string.

%key.ready Returns a logical value indicating whether there is data waiting from a user key depression.
%prefix.count

Returns the prefix count for the last use of get.key. Returns 1 if no prefix count was entered.

(get.char)

Waits for and returns the character sent by the next key pressed by the user. This function does not position the cursor. Precede it with paint if the cursor should be refreshed at the current position.

%prefix.set

Returns a logical value indicating whether a prefix count was entered for the last use of get.key.

(bexp)

Sounds the terminal bell.

(get.key)

Waits for user input of a bound key and returns the internal code for this key. These codes are shown under the heading key tokens above. If the key is a data character, the key type code is .insert and the associated data character can be retrieved using %key.char. See also %prefix.count and %prefix.set.

(paint mode)

Refreshes the screen display and positions the cursor. The mode argument is:

- 0 : Updates screen for any changes
- 1 : Clears the screen and repaints all displayed data

Repainting of the screen terminates on detection of type-ahead.

(prompt prompt dftr)

Displays prompt prompt text on the upper status line and invites input. Dflt is the default input if the return key is pressed without entering any response. On Initial entry to a procedure started from the keyboard, this function returns any prefix count associated with the keyboard action.

(status.msg str)

Displays message str on the upper status line. The displayed text may be cleared by using the status.msg operation with a null str.

(wait.input)

Wait for the user to press a key. The actual key pressed remains available for subsequent processing.

Miscellaneous File Handling

(delete rec)

Deletes rec from file. If the file cannot be opened, this function has no effect

(exists file rec)

Returns a logical value indicating whether record rec exists in file.

(read file rec)

Returns record rec from file. If the record cannot be read, this function returns a null string.

(write file rec str)

Writes str to record rec of file. If the file cannot be opened, this function has no effect

Conditional Execution and Loops

(exit)

Exits from the innermost active loop.

(if a proc1 else proc2)

Executes procedure proc1 if logical item a is True, proc2 if it is False. The else component is optional. Both proc1 and proc2 may be one or more procedures.

(loop proc
)

Executes proc repeatedly. proc may be one or more procedures. A loop is terminated by use of exit.
(quit \( n \)) Terminates current edit. If \( n \) is zero, SED continues processing a select list (equivalent to the `quit` editor function). For non-zero values of \( n \), SED terminates the entire sequence (equivalent to the QUIT command).

(return) Returns from the current PROC.

(return \( n \)) Returns \( n \) as the value of the current FUNC.

(stop) Terminates extension program and returns to SED edit mode.

(switch \( \text{val} \)
  case \( a \) proc1
  case \( b \) proc2
  ...
  else proc)
Executes one of several procedures depending on the value of \( \text{val} \). Items \( a \), \( b \) and \( c \) (etc.) are values which are compared with \( \text{val} \). The `else` component is optional and is executed only if none of the preceding conditions is met.

Setting Variables

(set \( \text{var} \) \( \text{val} \)) Set local or global variable \( \text{var} \) to \( \text{val} \).

Extension Control

%key.bindings Returns a dynamic array, each field of which corresponds to a bindable internal function and consists of one or more values which hold the actual character sequence. Note that this character sequence is the actual characters, not the encoded form used in the key bindings records to avoid use of control characters. The field number of any particular key can be identified using the key tokens described earlier.

%macro.state Returns state of editor macro system:
  0 : Not collecting or executing
  1 : Collecting macro
  2 : Executing macro

(bind.command \( \text{ext name} \)) Binds extension procedure \( \text{ext} \) as command \( \text{name} \). The extension name is automatically converted to upper case.

(bind.key \( \text{ext keyseq} \)) Binds extension procedure \( \text{ext} \) as key sequence \( \text{keyseq} \). The extension name is automatically converted to upper case. This function returns a logical value indicating whether it was successful.

(unload) Unloads all extensions on exit from outermost extension program.

(xeq \( \text{cmd} \)) Executes \( \text{cmd} \) as though it were entered using the editor `command` function. Commands which are themselves bound to extensions cannot be executed in this way. SED checks for extensions bound as command names before internal commands. It is therefore possible to replace a built-in command. Furthermore, since extensions cannot execute extension commands, the extension can be used to provide a prelude to a built-in command.

**Basic Functions** (Operations that mimic QMBasic programming functions)

(alpha \( \text{str} \)) Equivalent to `ALPHA(str)`
Returns a logical value indicating whether \( str \) is an entirely alphabetic string.

\[(\text{change } str \ old \ new)\]
Equivalent to \( \text{CHANGE}(str, old, new) \)
Replace each occurrence of \( old \) with \( new \) in \( str \).

\[(\text{convert } old \ new \ str)\]
Equivalent to \( \text{CONVERT}(old, new, str) \)
For each character in \( old \), this function replaces all occurrences of that character in \( str \) by the character in the corresponding position in \( new \). If \( new \) is shorter than \( old \), characters in \( old \) for which there is no replacement in \( new \) are deleted from the returned version of \( str \).

\[(\text{count } str \ substr)\]
Equivalent to \( \text{COUNT}(str, substr) \)
Returns a count of the number of occurrences of \( substr \) in \( str \).

\[(\text{dcount } str \ delim)\]
Equivalent to \( \text{DCOUNT}(str, delim) \)
Returns a count of the number of substrings in \( str \) delimited by \( delim \). The delimiter must be a single character.

\[(\text{iconv } str \ conv)\]
Equivalent to \( \text{ICONV}(str, conv) \)
Returns the result of applying input conversion \( conv \) to \( str \).

\[(\text{index } str \ substr \ occ)\]
Equivalent to \( \text{INDEX}(str, substr, occ) \)
Returns the character position (from one) of the \( occ \)'th occurrence of \( substr \) in \( str \). This function returns zero if the specified occurrence is not found.

\[(\text{matches } str \ pattern)\]
Equivalent to \( str \ MATCHES \ pattern \)
Tests whether \( str \) matches the given \( pattern \).

\[(\text{matchfield } str \ pattern \ n)\]
Equivalent to \( \text{MATCHFIELD}(str, pattern, n) \)
Returns the \( n \)'th component of \( str \) when matched against \( pattern \).

\[(\text{num } str)\]
Equivalent to \( \text{NUM}(str) \)
Returns a logical value indicating whether \( str \) can be interpreted as a number.

\[(\text{oconv } str \ conv)\]
Equivalent to \( \text{OCONV}(str, conv) \)
Returns the result of applying output conversion \( conv \) to \( str \).

\[(\text{subr } func \ arg1 \ arg2\ldots)\]
Equivalent to \( \text{SUBR}(func, arg1, arg2\ldots) \)
Call QMBasic function.

**Keyboard Functions Not Available as Extension Functions**

The following editor keyboard functions are not available directly as extension operations. They can all be achieved by use of other operations.

<table>
<thead>
<tr>
<th>Editor Function</th>
<th>Equivalent Extension Coding</th>
</tr>
</thead>
<tbody>
<tr>
<td>export</td>
<td>( \text{write file rec %kill.buffer} )</td>
</tr>
<tr>
<td>import</td>
<td>( \text{insert (read file rec)} )</td>
</tr>
<tr>
<td>insert killed</td>
<td>( \text{insert %kill.buffer} )</td>
</tr>
</tbody>
</table>

**SED Extensions - Argument Passing**

Any user written PROC or FUNC may take arguments. The actual number the compiler expects to pass is determined by the first reference to that PROC or FUNC and is checked at run time against the number that are expected by the called item.

To declare arguments in a PROC, it is written as

```
PROC
ARGS arg1, arg2, arg3...
```
The \textit{arg1}, \textit{arg2}, etc. items are local variables into which the argument values are to be transferred when the procedure is called.

\textbf{SED Extensions - An example of a complex extension}

The following extension implements a “walk” function which is provided as a standard part of QM in the \&SED.EXTENSIONS\& file of the QMSYS account.

This function allows you to define a rectangular block of text and use the cursor keys to “walk” it left, right, up or down. As the text block “runs over” other text, this reappears on the opposite side of the block being moved. The walk function is very useful in rearranging tabular data.

To use this extension, first position the cursor at the top left of the block. Execute the extension and move the cursor to the bottom right of the block by using the up, down, left and right functions then press the return key. Further use of the up, down, left and right functions will move the defined block until the return key is pressed again.

\begin{verbatim}
PROC()
(if %read.only
 (status.msg 'Read only buffer')
 (beep)
 (wait.input)
 (return)
)

* Step 1 - Get block coordinates
(set top %line)
(set left %col)
(status.msg 'Set block limits')
(loop
 (switch (get.key)
   case .up.line
     (set ct %prefix.count)
     (loop
      (if (le %line top) (exit))
      (retype (trimb %current.line)) * Remove trailing spaces
      (up.line 1)
      (if (lt %line.len %col) * Must extend line
        (retype (pad %current.line %col))
      )
      (set ct (sub ct 1))
      (if (eq ct 0) (exit))
    )
  case .down.line
   (set ct %prefix.count)
   (loop
    (if (gt %line %lines) (exit))
    (down.line 1)
    (if (lt %line.len %col) * Must extend line
        (retype (pad %current.line %col))
    )
  )
  ...operations...
)
\end{verbatim}
(set ct (sub ct 1))
(if (eq ct 0) (exit))
)

case .fwd.char
(set x (add %col %prefix.count))
(if (gt x %line.len) (retype (pad %current.line x)))
(goto.col x)

case .back.char
(set x (sub %col %prefix.count))
(if (le x left) (set x left))
(goto.col x)

case .newline
(set height (add (sub %line top) 1))
(set width (add (sub %col left) 1))
(exit)

case .cancel
(status.msg '')
(stop)

else
(beep)
)

* Step 2 - Move the block

(goto.line top)
(goto.col left)

{status.msg 'Move block'}
(loop
  (switch (get.key)
      case .up.line * Move block up
        (set rpt ... (add left width)) * Col of bit to right of block
        (set rc (sub right 1)) * Righthmost column of block
        (if (le %line 1) (exit))
        (up.line 1)
        (if (lt %line.len rc) * Must extend line
          (retype (pad %current.line rc))
        }
        (set wrapped.bit (substr %current.line left width))
        (set ct height)
        (loop
          (down.line 1)
          (set moving.bit (substr %current.line left width))
          (up.line 1)
          (retype (cat (substr %current.line 1 lw) moving.bit
                      (substr %current.line right 999999)))
          (down.line 1)
          (set ct (sub ct 1))
          (if (eq ct 0) (exit))
        )
  )
)
(retype (trimb (cat (substr %current.line 1 lw) wrapped.bit
    (substr %current.line right 999999))))
(set top (sub top 1))
goto.line top
(goto.col left)

(set rpt (sub rpt 1))
(if (eq rpt 0) (exit))
)

case .down.line  * Move block down
(set rpt %prefix.count)
(loop
  (set lw (sub left 1))       * Width of bit to left of block
  (set right (add left width))  * Col of bit to right of block
  (set rc (sub right 1))      * Rightmost column of block
  (if (gt (add %line.height) %lines) (exit))
  (down.line height)
  (if (lt %line.len rc)        * Must extend line
    (retype (pad %current.line rc))
  )
  (set wrapped.bit (substr %current.line left width))
  (set ct height)
  (loop
    (up.line 1)
    (set moving.bit (substr %current.line left width))
    (down.line 1)
    (retype (cat (substr %current.line 1 lw) moving.bit
        (substr %current.line right 999999)))
    (up.line 1)
    (set ct (sub ct 1))
    (if (eq ct 0) (exit))
  )
  (retype (trimb (cat (substr %current.line 1 lw) wrapped.bit
    (substr %current.line right 999999))))
  (set top (add top 1))
goto.line top
(goto.col left)

(set rpt (sub rpt 1))
(if (eq rpt 0) (exit))
)

case .fwd.char   * Move block to the right
(set rpt %prefix.count)
(loop
  (set lw (sub left 1))       * Width of bit to left of block
  (set right (add left width))  * Col of bit to right of block
  (set x (add right 1))
  (set ct height)
  (loop
    (if (lt %line.len right)        * Must extend line
      (retype (pad %current.line right))
    )
    (set wrapped.bit (substr %current.line right 1))
    (set moving.bit (substr %current.line left width))
    (retype (cat (substr %current.line 1 lw) wrapped.bit
        moving.bit
        (substr %current.line x 999999)))
  )
(down.line 1)
(set ct (sub ct 1))
(if (eq ct 0) (exit))
)
(set left (add left 1))
(goto.line top)
(goto.col left)
(set rpt (sub rpt 1))
(if (eq rpt 0) (exit))
)

case .back.char
(set rpt %prefix.count)
(loop
  (set lw (sub left 1))       * Width of bit to left of block
  (set right (add left width)) * Col of bit to right of block
  (set rc (sub right 1))       * Righmost column of block
  (if (le left 1) (exit))
  (set x (sub lw 1))
  (set ct height)
  (loop
    (if (lt %line.len rc)  * Must extend line
      (retype (pad %current.line rc))
    )
    (set wrapped.bit (substr %current.line lw 1))
    (set moving.bit (substr %current.line left width))
    (retype (cat (substr %current.line 1 x) moving.bit
                  wrapped.bit
                  (substr %current.line right 999999)))
  )
  (down.line 1)
  (set ct (sub ct 1))
  (if (eq ct 0) (exit))
)
(set left (sub left 1))
(goto.line top)
(goto.col left)

(set rpt (sub rpt 1))
(if (eq rpt 0) (exit))
)

case .newline
  (exit)

case .cancel
  (status.msg '')
  (stop)

else
  (beep)
)

* Step 3 - Tidy up by trimming trailing spaces from lines in block

(status.msg '')
(set ct height)
(loop
  (retype (trimb %current.line))
  (down.line 1)
  (set ct (sub ct 1))
  (if (eq ct 0) (exit))
)
SED Extensions - Local procedures and functions

A single extension source record may contain local procedures and functions that are only accessible to the other components of that extension. These must follow the main procedure or function and take the form

```
LPROC name
ARGS arg1, arg2, arg3...
{
    ...operations...
}
```

or

```
LFUNC name
ARGS arg1, arg2, arg3...
{
    ...operations...
    (return result)
}
```

There is no concept of local variable scope. Variable names used within local PROCs and FUNCs refer to the same set of variables as in the main PROC or FUNC. In particular, note that the argument variables simply provide an easy way to transfer information into the local PROC or FUNC. The two alternatives below are exact equivalents.

```
PROC
{
    ...
    (MYPROC 12 A)
    ...
}

LPROC MYPROC
ARGS X, Y
{
    ...
}
```

```
PROC
{
    ...
    (SET X 12)
    (SET Y A)
    (MYPROC)
    ...
}

LPROC MYPROC
{
    ...
}
```


Although local procedures may recurse (that is call themselves) it is likely that the lack of scoped variables makes this of limited use.
The **SEL.RESTORE** command restores a single file from a Pick style ACCOUNT.SAVE or FILE.SAVE tape.

**Format**

```
SEL.RESTORE {DICT} target.file.name {item.list} {options}
```

where

- **target.file.name** is the name of the file into which data is to be restored.
- **item.list** is a list of records to be restored. The default select list may be used instead. If omitted, and no list is active, all records are restored.
- **options** is any combination of the following:
  - **BINARY** Suppresses translation of field marks to newlines when restoring directory files. Use this option when restoring binary data.
  - **DET.SUP** Suppresses display of the name of each file as it is restored.
  - **NO.CASE** Causes new files to be created with case insensitive record ids. Existing files are not reconfigured.
  - **NO.INDEX** Do not create alternate key indices.
  - **NO.OBJECT** Omits restore of object code. This is particularly useful when migrating to QM from other environments.
  - **POSITIONED** Assumes that the tape is already positioned at the start of the data to be restored.
  - **NO.QUERY** suppresses the confirmation prompt when restoring using a select list. The NO.SEL.LIST.QUERY mode of the **OPTION** command can be used to imply this option.

Unless the POSITIONED option is used, the **SEL.RESTORE** command prompts for the name of the account and the file within that account. It then restores the data for this file from the tape into the specified target file which must already exist.

The tape to be restored must first be opened to the process using the **SET.DEVICE** command.

**See also:**
4.186 SELECT.DIR

The SELECT.DIR command constructs a select list from the contents of an operating system directory.

Format

SELECT.DIR pathname {FILE | DIRECTORY} {TO list} {COUNT.SUP} {PATHNAME}

where

pathname is the pathname of the directory to be processed.

FILE specifies that a list of files in the directory is to be created. This is the default action of the command.

DIRECTORY specifies that a list of subdirectories in the directory is to be created.

TO list specifies the list to be created. If omitted, the default list (list 0) is created.

COUNT.SUP suppresses display of the number of items in the select list.

PATHNAME causes the command to return a list in which each entry is a full pathname. Without this option, the list contains the names without the pathname prefix.

The SELECT.DIR command scans the directory identified by pathname and constructs a select list containing the names of either the files or subdirectories in this directory. The casing of the directory items is maintained.

SELECT.DIR does not recurse down into sub-directories.

If successful, both @SELECTED and @SYSTEM.RETURN.CODE will be set to the number of item selected. If the command fails, @SYSTEM.RETURN.CODE will contain the negative value of the error number.

Example

SELECT.DIR C:\MYAPP\EMAILS TO 1

The above command would build select list 1 as a list of the files in the C\MYAPP\EMAILS directory.
4.187 SELECTINDEX

The SELECTINDEX command constructs a select list from an alternate key index.

Format

```
SELECTINDEX {DICT} file.name index.name [, value] {TO list} {COUNT.SUP}
```

where

- `file.name` is the name of the file.
- `index.name` is the name of an index within the specified file.
- `value` is the value that the indexed field must have to be included in the result list.
- `TO list` specifies the list to be created. If omitted, the default list (list 0) is created.
- `COUNT.SUP` suppresses display of the number of items in the select list.

If the `value` element is omitted, the SELECTINDEX command constructs a select list containing all the indexed values in the specified alternate key index. It is equivalent to use of the QMBasic SELECTINDEX statement with no indexed value.

If the `value` element is included, the SELECTINDEX command constructs a select list containing the ids of all records in which the indexed field identified by `index.name` has the given value. It is equivalent to use of the QMBasic SELECTINDEX statement with an indexed value.

Both `@SELECTED` and `@SYSTEM.RETURN.CODE` will be set to the number of item selected.
4.188 SET

The SET command sets a value into an @-variable.

Format

\[
\text{SET variable value} \\
\text{SET variable EVAL expression}
\]

where

- **variable** is the name of the variable to be set. The leading @ character is optional. The name may be up to 32 characters and is case insensitive.
- **value** is the value to be stored. This may not include the mark characters. Quotes should not be used unless they are part of the value to be stored. Leading and trailing spaces within value are removed; embedded spaces are retained.
- **expression** is an arithmetic expression to be evaluated.

The SET command sets a value into a user defined @-variable. The values of system defined variables \@SYSTEM.RETURN.CODE, \@USER0 to \@USER4 and \@USER.RETURN.CODE can also be set.

In the second form, the expression may include the four arithmetic operators +, -, *, and /. These operators must be surrounded by spaces. Any @-variables in the expression will be expanded.

Examples

\[
\text{SET USER.RETURN.CODE 1}
\]

The above command sets the @USER.RETURN.CODE variable to 1.

```
PA
SET CT 5
LOOP
  DISPLAY <<@CT>>
  SET CT EVAL @CT - 1
  IF @CT = 0 THEN STOP
REPEAT
```

This paragraph executes the loop five times, displaying the decreasing values stored in @CT on each cycle.

```
PA
SET @PRINT ""
IF <<I2,Select branch>> EQ "" THEN STOP
IF <<I3,Print (Y/N)?>> EQ "Y" THEN SET @PRINT LPTR
LIST LOANS WITH BRANCH EQ "<<Select branch>>" 
  BY NAME PAN SCROLL TITLE DATE NAME ID.SUP <<@PRINT>>
```
The above paragraph shows use of a user defined @-variable to control whether the paragraph directs the report to the display or to a printer. The @PRINT variable is set to a null string at the start of the paragraph in case it already exists from previous actions in the same session. Later, if the user wants the report sent to a printer, this variable is set to "LPTR". The value of @PRINT is then included in the query sentence executed by the paragraph.

See also:
LIST.VARS and the QMBasic !ATVAR() and !SETVAR() subroutines.
4.189 SET.DEVICE

The SET.DEVICE command opens a tape or pseudo-tape for processing by QM. The synonym T.ATT may be used.

Format

```
SET.DEVICE device.name [format] {NO.QUERY}
```

where

- `device.name` is the pathname of the device to be opened. This must be enclosed in quotes if it commences with a backslash on Windows.
- `format` identifies the media format.

The SET.DEVICE command assigns a tape device or a file representing a pseudo-tape to the current QM process. The device can then be used by other tape processing commands such as ACCOUNT.SAVE, ACCOUNT.RESTORE, T.DUMP, T.LOAD and the T.xxx tape utility commands.

Note: QM does not support Pick style use of floppy disks as tape devices.

If the `format` option is not given, SET.DEVICE attempts to detect the format of the image being attached by examination of the data. A code signifying the format is then stored internally for future use by the other tape processing utilities.

The NO.QUERY option suppresses the confirmation prompt asking if a previously attached device should be detached.

Note that access to real tape devices (as opposed to pseudo-tapes) may offer limited functionality. In particular, the Pick compatible tape transfer tools require the ability to process file mark blocks, a feature which may not be provided by the underlying operating system device driver. Use of pseudo-tapes is strongly recommended.

To use a floppy disk drive (e.g. a Pick style account save) specify the `device.name` as "A:" on Windows or use the device driver name (probably /dev/fd0) on Linux.

The formats currently supported are:
- AS PICK style ACCOUNT-SAVE image (also the QM ACCOUNT.SAVE format)
- R83
- MV
- FS PICK style FILE-SAVE image.
- ULTFS Ultimate style FILE-SAVE image.
- JBS jBASE ACCOUNT-SAVE image.

The AS type can be either a single ACCOUNT-SAVE or several ACCOUNT-SAVEs on the same tape. These multiple saves are created by successive ACCOUNT-SAVE commands issued without rewinding the tape between saves. If multiple accounts exist, this format is handled like the FS type by the tape utilities and RESTORE.ACCOUNTS, FIND.ACCOUNT and SEL.RESTORE operations may all be used.

The ULTFS format is a FILE-SAVE format in which several accounts are expected.
The JBS format is essentially the same as the individual files that comprise the ULTFS set. These files have a single label followed immediately by the account data and are treated like individual AS types.

See also:
ACCOUNT.RESTORE, ACCOUNT.SAVE, FILE.SAVE, FIND.ACCOUNT, RESTORE.ACCOUNTS, SEL.RESTORE, T.DUMP, T.LOAD, T.xxx
4.190 SET.ENCRYPTION.KEY.NAME

The **SET.ENCRYPTION.KEY.NAME** command updates a data file to reference a new name for an encryption key.

**Format**

```
SET.ENCRYPTION.KEY.NAME filename field, keyname ...
SET.ENCRYPTION.KEY.NAME filename keyname
```

where

- **filename** is the name of the encrypted file to be updated.
- **field** is the name or field number of the field to be amended.
- **keyname** is the name of the encryption key to be used. This is case insensitive.

The **SET.ENCRYPTION.KEY.NAME** command is intended for use where an encrypted file is moved to a system on which the original encryption key name is already in use but with a different key value. It updates the encryption key information stored in the file but does not make any changes to the data records.

The first form of the **SET.ENCRYPTION.KEY.NAME** command updates the encryption key for one or more fields within a file that uses field level encryption.

The second form of the **SET.ENCRYPTION.KEY.NAME** command updates the encryption key for record level encryption.

**Examples**

```
SET.ENCRYPTION.KEY.NAME CUSTOMERS CCARD,CARDNO
```

The above command sets the encryption key for the CCARD field of the CUSTOMERS file to be CARDNO.

```
SET.ENCRYPTION.KEY.NAME CUSTOMERS CKEY
```

The above command sets the record level encryption key for encrypts the CUSTOMERS file to be CKEY.

**See also:**

Data encryption, CHANGE.KEY.PASSWORD, CREATE.FILE, CREATE.KEY, CREATE.SECURE.KEY, DELETE.KEY, DISABLE.KEY, DISABLE.KEY (QMBasic), ENABLE.KEY, ENABLE.KEY (QMBasic), ENCRYPT.FILE, GRANT.KEY, LIST.KEYS, RESET.MASTER.KEY, REVOKE.KEY, UNLOCK.KEY.VAULT
4.191 SET.EXIT.STATUS

The SET.EXIT.STATUS command sets the final exit status returned by QM to the operating system.

Format

    SET.EXIT.STATUS value

where

    value    is the numeric exit status value to be set.

By default, QM returns an exit status of zero to the operating system on termination. The SET.EXIT.STATUS command allows an application to return an alternative exit status value to indicate, for example, success or failure. Note that error conditions detected during startup of a QM session return an exit status of 1.

See also the QMBasic SET.EXIT.STATUS statement.
4.192 SET.FILE

The SET.FILE command adds a Q-pointer to the VOC to reference a remote file.

**Format**

```
SET.FILE {account { filename { pointer } } }
```

where

- **account** is the name of the account holding the file to be referenced. This name must exist in the ACCOUNTS file of the QMSYS account. Alternatively, `account` may be specified as a QMNet style reference `servername:account` to create a Q-pointer that links to a file on a remote system.

- **filename** is the name of the file in the remote account. This must correspond to an F or Q-type VOC entry in that account. Creation of Q-pointer chains where one Q-pointer points to another is not recommended.

- **pointer** is the name to be given to the Q-pointer created in the local account.

The SET.FILE command prompts for information not provided on the command line. The `pointer` defaults to QFILE.

The SET.FILE command creates a Q-pointer in the local account to reference the named file in the remote account. Q-pointers should be used in preference to multiple F-type records pointing to the same file as they simplify maintenance and give a sense of ownership of the file to the account containing the F-type entry.

**Examples**

```
SET.FILE DEV SALES DEV.SALES
```

The above command creates a Q-pointer named DEV.SALES that references the SALES file in the DEV account.

```
SET.FILE STANDBY:LIVE INVOICES SBY.INVOICES
```

The above command creates a Q-pointer named SBY.INVOICES that references the INVOICES file in the LIVE account on the server named STANDBY.
4.193 SET.LANGUAGE

The SET.LANGUAGE command selects the language in which messages should be displayed.

**Format**

```
SET.LANGUAGE {code}
```

where

- `code` is the identifier for the language to be used.

QM includes support for text message output in multiple languages. The standard message library installed with QM contains English messages texts. Additional languages can be loaded using the LOAD.LANGUAGE command.

Message sets for each loaded language are identified by an alphabetic code which should preferably correspond to an international standard country code such as FR for France. Application developers can use any case insensitive alphabetic language code up to three characters for message sets that correspond, for example, to regional variations such as CHF for Swiss French.

The SET.LANGUAGE command selects the message set to be used by the QM process in which it is issued. To apply this globally, the command should be included in the MASTER.LOGIN paragraph. Using the command with no language code selects the standard English messages.

Any operation that attempts to use a message for which there is no corresponding entry in the message library will revert to the standard English version.

**See also:**

- Multi-language applications, LOAD.LANGUAGE
4.194 SET.MODE

The SET.MODE command updates the access permissions on records in a directory file. (Not Windows)

Format

SET.MODE filename { ALL | FROM listno | id... } { NO.QUERY } mode

where

filename is the name of a directory file.

ALL indicates that the access modes of all records in the file should be updated.

FROM listno takes record ids from the specified select list.

id... mode is the desired access permissions as a three digit octal value as used by the operating system chmod command.

The SET.MODE command is broadly equivalent to the Linux/Unix chmod command, setting the access permissions for the requested records in a directory file.

The record ids may be specified as ALL, taken from a select list or supplied on the command line. If none of these methods is used, the command checks whether the default select list is active and, if so, prompts to ask if it should be used. The NO.QUERY option, valid only when defaulting the select list, suppresses the prompt.
4.195 SET.QUEUE

The SET.QUEUE command creates a relationship between a Pick style form queue number and the corresponding SETPTR print unit options.

Format

SET.QUEUE queue {, width, depth, top.margin, bottom.margin, mode {, options }}

SET.QUEUE DISPLAY {LPTR {unit}}

SET.QUEUE DELETING queue

where

queue is the form queue number in the range 0 to 999.
width is the page width in characters, excluding any left margin.
depth is the total page length in lines, including the top and bottom margins. A value of zero implies no pagination of the output data.
top.margin is the number of lines to be left blank at the top of the page.
bottom.margin is the number of lines to be left blank at the bottom of the page.
mode is the print unit mode.
options qualify the output destination. The option syntax is the same as in the SETPTR command. There should be a comma between each option.

If only queue is given, the current settings of the form queue are reported.

The SET.QUEUE command creates or modifies an entry in the $FORMS file to relate a Pick style form queue number to the corresponding SETPTR options. This form queue number can then be used in the SP.ASSIGN command. See SETPTR for details of the command options.

The SET.QUEUE DISPLAY command displays a report of the settings of all defined form queues. The optional LPTR keyword directs this report to a printer.

The SET.QUEUE DELETING command deletes a form queue definition.

The $FORMS file is normally shared between all accounts by use of a VOC entry that references the $FORMS file in the QMSYS account. An account can have its own forms definition file by modifying this VOC entry to point to a file within the account. It is also possible for users to have a personal forms definition file based on their login id by setting the pathname to be of the form

@HOME\FORMS

where the @HOME element will be replaced by the user’s home directory pathname.

SET.QUEUE performs limited validation of the parameters as later use of SP.ASSIGN may change some of them. For example, it is valid for a SET.QUEUE command to have both the AS and AT options (see SETPTR). Whichever is not applicable to the mode settings when the SP.ASSIGN command is used will be discarded.
If then $SETPTR.DEFAULTS record exists (see $SETPTR), the default options are applied at the point when the $P.ASSIGN command is used.

Examples

SET.QUEUE 0,80,66,3,3,1, AT laser
Directs form queue 0 output to a printer named "laser" with a page shape of 80 columns by 66 lines and a 3 line top and bottom margin.

SET.QUEUE 4,80,66,0,0,3, AS SALES_REPORT
Directs form queue 4 output to a record named "SALES_REPORT" in the $HOLD file.

See also:
$SETPTR, $P.ASSIGN
4.196 SET.SERVER, SET.PRIVATE.SERVER

The SET.SERVER command defines a QMNet server available to all users. The SET.PRIVATE.SERVER command defines a QMNet server visible only from the QM session in which the command is executed.

**Format**

```
SET.SERVER name ip.address {;port} username password {NO.QUERY}
SET.SERVER name ip.address {;port} LIKE server.name {NO.QUERY}
SET.PRIVATE.SERVER name ip.address {;port} username password
```

where

- **name** is the name to be used within QM to reference this server (maximum 15 characters). The name does not need to be related to the network name of the server.
- **ip.address** is the IP address or network name of the server.
- **port** is the tcp/ip port number on which connection to this server will be made. It must correspond to the port on which the server listens for incoming QMClient connections. If omitted, the default QMClient port (4243) is used.
- **username** is the user name that will be used to connect to the server. This must be defined at the operating system level on the server but does not need to be defined on the local system. For a domain style login use `user@domain` or the older `domain\user` format.
- **password** is the password for the supplied `username`. It is stored internally in encrypted form for security.
- **server.name** is the name of another server from which the security data is to be copied.

The command will prompt for command line elements that are omitted.

To allow for IPV6 format addresses, the colon separator between the `ip.address` and the `port` may alternatively be entered as a semicolon. This applies either on the command line or in response to a prompt for the address.

QMNet allows an application to access QM data files on other servers as though they were local file, with complete support for concurrency control via file and record locks. The remote server must have remote access enabled by setting the `NETFILES` configuration parameter to 2.

Public servers may be defined by any user with system administrator rights using the `SET.SERVER` command in the QMSYS account. The `NO.QUERY` keyword suppresses the confirmation prompt if the server name is already defined. A public server defined with this command can be accessed by all users. The `ADMIN.SERVER` command can be used to create or modify server definitions to apply restrictions on which users can access the server. Alternatively, the `LIKE` keyword can be used to copy the security settings from another public server definition.

Private servers are defined using the `SET.PRIVATE.SERVER` command. As part of the QM security system (see Application Level Security), the `SET.PRIVATE.SERVER` command is only
available to users for which it is enabled in their user register entry. On a system running with security disabled, any user may define private servers. The default behaviour of QM is to allow a private server definition to have the same name as a public server, effectively redirecting connections to that server. The `SECURITY` configuration parameter can be used to disallow creation of such definitions.

**Example**

```
SET.SERVER ADMIN 193.100.13.18:4000 root
```

This example will create a server known within QM as ADMIN. The server IP address is 193.100.13.18 and connection will use port 4000. The user name (root) has been included in the command but, because the password has been omitted, the command will prompt for this.

**See also:**

OMNet, `ADMIN.SERVER`, `DELETE.SERVER`, `LIST.SERVERS`
4.197 SET.TRIGGER

The SET.TRIGGER command sets, removes or displays the trigger function associated with a file.

Format

\[
\text{SET.TRIGGER \text{file.name} \ function.name \ \{\text{modes}\}} \\
\text{SET.TRIGGER \text{file.name} \ ""} \\
\text{SET.TRIGGER \text{file.name}}
\]

where

- \text{file.name} is the file to be processed.
- \text{function.name} is the name of the catalogued trigger function. Maximum 32 characters.
- \text{modes} is any combination of the following tokens indicating when the trigger will be executed.
  
  - \text{PRE.WRITE} Before a write operation
  - \text{PRE.DELETE} Before a delete operation
  - \text{PRE.CLEAR} Before a clear file operation
  - \text{POST.WRITE} After a write operation
  - \text{POST.DELETE} After a delete operation
  - \text{POST.CLEAR} After a clear file operation
  - \text{READ} After a read operation
  - \text{ALL} Sets all of the modes listed above

If no modes are specified, the default is \text{PRE.WRITE} and \text{PRE.DELETE}.

The first form of the SET.TRIGGER command sets the name of the trigger function to be associated with the named file. Any existing trigger function is replaced by this action.

The second form of the SET.TRIGGER command removes the trigger function for the named file.

The third form of the SET.TRIGGER command displays the name and modes of the trigger function for the named file.

Setting or removing a trigger while the file is open may not take effect until after the next access to the file. For directory files, a change to the trigger will only take effect when the file is next opened.

A trigger function on a hashed file will be called by all updates to the file in the modes for which the trigger is active. A trigger function on a directory file will be called only by updates from within QM and not for use of sequential file operations (WRITESEQ, WRITEBLK, etc), ONSWRITE or OSDELETE. Some implications of this are that query processor CSV or delimited reports directed to a file and QMBasic compiler listing files will not call the trigger function.

See also:

LIST.TRIGGERS
4.198 SET.VFS.SERVER

The SET.VFS.SERVER command defines a Virtual File System (VFS) server.

Format

```
SET.VFS.SERVER name {EXT} handler ip.address{/port} username password
{NO.QUERY}
```

where

- **name** is the name to be used within QM to reference this server (maximum 15 characters). The name does not need to be related to the network name of the server.
- **handler** is the VFS handler name. For an external VFS handler, this must be prefixed with EXT. This name is case sensitive on Linux/Unix systems.
- **ip.address** is the IP address or network name of the server.
- **port** is the tcp/ip port number on which connection to this server will be made, if relevant.
- **username** is the user name that will be used to connect to the server.
- **password** is the password for the supplied username. It is stored internally in encrypted form for security.

The command will prompt for command line elements that are omitted.

Because the **ip.address**, **port** and **username** may not be null strings, a dummy value such as "none" must be entered when defining a VFS handler that does not require this data.

To allow for IPV6 format addresses, the colon separator between the **ip.address** and the **port** may alternatively be entered as a semicolon. This applies either on the command line or in response to a prompt for the address.

VFS servers may be defined by any user with system administrator rights using the SET.VFS.SERVER command in the QMSYS account. The **NO.QUERY** keyword suppresses the confirmation prompt if the server name is already defined. The ADMIN.SERVER command can be used to create or modify server definitions to apply restrictions on which users can access the server.

**Example**

```
SET.VFS.SERVER U2 EXT uv 193.100.13.18 sales
```

This example will create a server known within QM as U2 that uses the external handler named vfs_uv to connect to IP address is 193.100.13.18. Note that the "vfs_" prefix needed on an external VFS handler program is added automatically. If this is included in the handler name supplied to this command, it will be removed. The user name (sales) has been included in the command but, because the password has been omitted, the command will prompt for this.

**See also:**
The Virtual File System, ADMIN.SERVER, DELETE.SERVER, LIST.SERVERS
The **SETPORT** command sets communications parameters of a serial port.

**Format**

```
SETPORT port {BAUD rate} {BITS bits.per.byte} {PARITY parity} {STOP.BITS stop} {BRIEF}
```

where

- `port` is the name of the port to be accessed (e.g. COM1 on Windows or /dev/cua0 on Linux, FreeBSD or AIX).
- `rate` is the baud rate for the port.
- `bits.per.byte` is the number of bits per byte (5 to 8).
- `parity` is the parity mode (**NONE**, **ODD** or **EVEN**).
- `stop` is the number of stop bits (1 or 2).
- **BRIEF** Suppresses the normal confirmation prompt

The port must not be open in the application when this command is executed.

If only `port` is given, the current settings of the port are reported.

On some systems, it may be necessary to change the permissions on the device driver to make it accessible to users.

**Example**

```
SETPORT COM1 BAUD 9600 BITS 7 PARITY ODD STOP.BITS 1
```

**See also:**

- [SETPORT.PRAMS()](#)
4.200 SETPTR

The SETPTR command sets print unit characteristics.

Format

```
SETPTR unit { , width, depth, top.margin, bottom.margin, mode { , options } }
SETPTR DISPLAY { LPTR { printer } }
SETPTR unit, DISPLAY
```

where

- `unit` is the print unit number in the range 0 to 255 or the keyword DEFAULT.
- `width` is the page width in characters, excluding any left margin. This value is used to calculate alignment in query processor reports and headings/footings but does not prevent application programs emitting data that exceeds this width.
- `depth` is the total page length in lines, including the top and bottom margins. This value is used to control insertion of form feeds, headings and footings into the printed data. A value of zero implies no pagination of the output data.
- `top.margin` is the number of lines to be left blank at the top of the page.
- `bottom.margin` is the number of lines to be left blank at the bottom of the page.
- `mode` is the print unit mode:
  - 1 Output is sent to a printer.
  - 3 Output is directed to a hold file.
  - 4 Output is directed to stderr (standard error, console sessions only).
  - 5 Output is directed to the terminal auxiliary printer port.
  - 6 Output is written to a file and also printed.
- `options` qualify the destination as described below. There should be a comma between each option.

If only `unit` is given, the current settings of the print unit are reported.

Use of the DEFAULT keyword in place of a unit number records the default values to be used when a new print unit is accessed without prior use of SETPTR to define its settings. Note that this operation does not affect the default printer, print unit 0, which is configured with standard default settings on entry to QM. These can be changed with a SETPTR command specifying unit 0.

The third form of SETPTR with a unit number and the DISPLAY keyword shows the current settings in a form that can be captured by a program and later used to restore the settings by executing a SETPTR command with the captured value appended.

When using printer modes 3 or 6, a check is made to see if there is a catalogued subroutine named HOLD.FILE.LOGGER and, if there is, this is called when the file is opened and again when it is closed. This subroutine can be used, for example, to build a log of hold file entries or to take some...
action after the file has been closed. The subroutine takes three arguments; the print unit number, a flag indicating if this is an open (1) or a close (0), and the pathname of the file being created.

The options available are:

**AS { FILE name}**
Specifies an alternative hold file name instead of the default $HOLD for modes 3 and 6. Use of this option with other mode values will result in an error. The name must reference a directory file.

**AS { NEXT } { id }**
Specifies the hold file record name in modes 3 and 6. Use of this option with other mode values will result in an error. At least one of the optional components must be present.

id is the name of the record to be created in the $HOLD file. If omitted, a default name of P$unit is used.

The optional NEXT keyword causes QM to attach an underscore and a cyclic sequence number to the end of the name so that successive output is stored separately. Note that this sequence number is shared across all printer output directed to the $HOLD file by all processes, thus two successive jobs from one process may have non-adjacent sequence numbers. The default behaviour is for the sequence number to be four digits, cycling through the range 0001 to 9999. The next available sequence number is stored in field 2 of a record named $NEXT in the dictionary of the $HOLD file. An alternative number of digits in this value may be specified in field 3 of this dictionary record and must be in range 3 to 9. The sequence number can be determined using the `GETPU()` function or the `@SEQNO` variable. If id is a quoted null string, the underscore is omitted.

These two variants of the AS clause may be combined (AS FILE name NEXT id). When used in this way the sequence number management process described above uses the dictionary of the named file. Use of AS NEXT "" will create items where the name is just the sequence number.

**AS PATHNAME path**
Specifies the pathname of the output file to be created (not the directory in which it is to be created) for output in modes 3 and 6. Use of this option with other mode values will result in an error.

**AT printer.name**
Specifies the printer name in modes 1 and 6. This name must be enclosed in quotes if it contains spaces or backslashes. The name is case sensitive except on Windows. Note that printing occurs on the server on which QM is running and the printer name must be known to the server.

Mode 5 provides a way to print on printers connected to the client system. For users connecting to QM from AccuTerm using a terminal type with the -at suffix, the AT option can be used to select a printer name which is known to the client PC.

**BANNER text**
Set the text to appear on a banner page.
BRIEF Suppresses the normal confirmation prompt before setting the printer characteristics. This is typically used in SETPTR commands from paragraphs or QMBasic programs.

COPIES $n$ Specifies the number of copies to be printed. Some printers may not handle this option.

EJECT Appends a form feed to the end of the print data.

ENCODING $mode$ Sets the character encoding to be used for this printer. On ECS mode systems, the default behaviour is to output the low order byte of each character which will lead to incorrect printed output if characters outside the 8 bit set appear in the data. On all systems, ECS or non_ECS, the ENCODING option can be used to set a character encoding such as UTF-8. Encoding can be used with all print modes except mode 4 (print to standard error). Used with mode 6 (file and printer), the same encoding applies to both destinations.

FORM.FEED $mode$ Determines the form feed sequence used by the QMBasic PRINT statement. This may be FF or CRFF. The default is CRFF.

GDI Specifies that the GDI mode API calls are to be used to initiate printing on Windows systems.

INFORM Displays file details when opening a print file in mode 3. Except in phantom or QMClient processes, the session will pause for 1.5 seconds after the message is displayed to allow for situations where the screen would be overwritten.

KEEP.OPEN Keeps the printer open to merge successive printer output. Use the PRINTER CLOSE command (not the QMBasic PRINTER statement) to terminate the print job.

LANDSCAPE When used without the PCL option, this option is passed to the underlying print driver to request landscape format printing where this is supported. On non-Windows platforms, this is equivalent to use of OPTIONS "landscape".

LEFT.MARGIN $n$ Inserts a margin of $n$ spaces to the left of the printed data.

NEWLINE $mode$ Determines the newline sequence used by the QMBasic PRINT statement. This may be CR, LF or CRLF. The default is LF.

NFMT Specifies that no page formatting is to be applied to the output data. The entire output is treated as a single page with no further inserted form feeds or top and bottom margins. Use of this mode is essential when sending binary data such as images to the printer.

NHEAD Suppress banner page (default)

NODEFAULT Omitted options normally take their default values. Use of this keyword leaves the option at its current value.

NOEJECT Suppresses the normal page throw at the end of a print job. This option applies to Windows only.

NOHEAD Suppress banner page (default)

OPTIONS $xxx$ Passes the given option(s) to the underlying operating system print spooler (e.g. OPTIONS "landscape" on non-Windows
systems. Multiple options may be given as a single quoted qualifier to the OPTIONS setting.

**OVERLAY subr** Identifies a catalogued subroutine that will be executed at the start of each page of output. This subroutine takes a single argument which is the print unit number and can be used to send printer control codes for a graphical page overlay, if required. It should not perform any other printer output. This option is ignored for Windows GDI mode printing.

**PCL** Specifies that this printer supports PCL. This option cannot be used with GDI mode Windows printers.

**PORTRAIT** Where supported by the underlying print driver, this keyword specifies that the output is to be printed in portrait format.

**PREFIX path** Sends the contents of the named file to the printer at the start of each job. This can be used to send printer specific commands for features that are not available through other SETPTR options. This option is ignored for Windows GDI mode printing.

**RAW** Specifies that the non-GDI mode API calls are to be used to initiate printing.

**SPOOLER name** Specifies and alternative spooler to be used on non-Windows systems. If not specified, the spooler selected by the SPOOLER configuration parameter is used or, if this is not set, the standard lp spooler is used. The name can include other spooler options if required but must be quoted if it includes spaces or other reserved characters. The actual command executed to print the job will be name with options appropriate to lp added as follows:

- `-n copies` If COPIES is greater than 1.
- `-d prt.name` To set the printer name if AT is used.
- `-t banner` Banner text if BANNER is used.
- `-o "options"` Text from OPTIONS if used.
- `-o "landscape"` If LANDSCAPE is used.

The SPOOLER option can be used to access another standard operating system spooler package or to direct output to a user written shell script or program to perform custom processing. The $$SPOOLERS$$ record in the VOC of the QMSYS account can be used to configure different options from those described above. See Printing for details. As an aid to diagnosing printing problems, the SPOOL.COMMAND option of the OPTION command can be used to display the operating system command used to initiate printing.

**STYLE name** Specifies the name of an X-type VOC record that defines the query processor report style to be used for all reports directed to this print unit unless overridden by the STYLE option in the query command.

**GDI Printers**
The following additional options can be used with GDI mode printers on Windows. If neither of the FONT and FONT.SIZE options is present, the printer's default font is used, otherwise default values are as shown below.

**FONT name**
Sets the font name. Defaults to Courier New if omitted. The font name must be quoted if it contains spaces.

**FONT.SIZE n**
Specifies the font size. If n is omitted, specified as zero or DEFAULT, 10 point size is used.

**PCL Printers**
The following additional options are available. Although the values set will be saved and can be accessed by application software, they only affect printing when used with the PCL option. In many cases, the list of acceptable parameter values can be extended by modifying the SYSCOM $PCLDATA record.

**CPI n**
Specifies the number of characters per inch. The value may be non-integer.

**DUPLEX**
Selects duplex (double sided) printing, binding on the long edge.

**DUPLEX SHORT**
Selects duplex (double sided) printing, binding on the short edge.

**FONT name**
Sets the font name. The font name must be quoted if it contains spaces.

**LANDSCAPE**
Prints the page in landscape format.

**LPI n**
Specifies the number of lines per inch. The value must be 1, 2, 3, 4, 6, 8, 12, 16, 24 or 48.

**PAPER.SIZE xx**
Specifies the paper size. Valid size names are A4, LETTER, LEGAL, LEDGER, A3, MONARCH, COM_10, DL, C5, B5.

**SYMBOL.SET xx**
Specifies the character set. Valid values of xx are ROMAN8 (the default), LATIN1, ASCII, PC8.

**WEIGHT xx**
Specifies the font weight. Valid values of xx are ULTRA-THIN, EXTRA-THIN, THIN, EXTRA-LIGHT, LIGHT, DEMI-LIGHT, SEMI-LIGHT, MEDIUM, SEMI-BOLD, DEMI-BOLD, BOLD, EXTRA-BOLD, BLACK, EXTRA-BLACK, ULTRA-BLACK though specific printers might not support all values.

When setting a print unit to PCL mode, if the above parameters are not specified, printing defaults to 10 cpi, 6 lpi, medium weight using A4 paper and the Roman8 symbol set. These defaults can be modified by adding an X-type record named $PCL to the VOC. This record must have an X as the first character of field 1. Remaining fields may contain any of the CPI, LPI, PAPER.SIZE, SYMBOL.SET and WEIGHT parameters as described above. For example:

```
0001: X
0002: PAPER.SIZE LETTER
0003: LPI 5
```

**Note:** The quality of PCL implementations varies widely and these options may not give the expected results on some printers. In particular, setting some font metrics may cause inconsistent character placement. It is the application developer's responsibility to ensure that the printed results are acceptable.
Windows Printing

On Windows systems, two styles of interface with the underlying Print Manager are supported. The GDI mode uses the Windows Graphical Device Interface API calls. Non-GDI mode uses an alternative set of API calls. For most purposes, the non-GDI mode is likely to be preferable.

The GDI parameter to SETPTR sets GDI mode and the RAW parameter sets non-GDI mode. The default is normally non-GDI but this can be modified by setting the GDI configuration parameter to 1.

Special Print Modes

Use of mode 4 (stderr) allows an application developer to direct output from a QM console session to the standard error file handle. It is the user's responsibility to ensure that this points to an appropriate destination as the default settings may cause the screen display to be overwritten. Data sent to a mode 4 printer is output immediately rather than being buffered by QM. Headings, footings and other pagination related features are ignored for printers in mode 4.

Mode 5 print units direct output to the terminal auxiliary printer port, typically to a printer local to the client. The data is buffered by QM until the print unit is closed and then sent to the terminal prefixed by the string defined in the mc5 terminfo entry for the selected terminal type. The data will be followed by the string defined by the mc4 terminfo entry. The mc5/mc4 pair should be set to the control codes necessary to enable/disable the printer port. For users connecting to QM via AccuTerm with a terminal type that has a -at suffix, the AT option to SETPTR can be used to select a printer name that is known to the client PC.

The SETPTR DISPLAY command displays a report of the settings of all print unit. The optional LPTR keyword directs this report to a printer.

The $SETPTR.DEFAULTS VOC Record

When setting print unit configuration, the SETPTR command checks for an X-type record named $SETPTR.DEFAULTS in the VOC or, if not found, in the QMSYS VOC. If this record exists, fields 2 onwards can be used to set default options that will effectively be inserted after the mode value. Each field corresponds to a different print mode such that field 2 is used for mode 1, field 3 is used for mode 2 (currently not supported on QM), field 4 is used for mode 3 and so on. Thus, a $SETPTR.DEFAULTS record that reads

```
1: X
2: NEWLINE CRLF, INFORM
```

would behave as though these two options were present when setting a print unit into mode 1. Because the default options are inserted between the mode and options elements of the command, they can be overridden by other options specifically set in the command.

Examples

```
SETPTR 0,80,66,3,3,1,AT laser,BRIEF
```

Directs print unit 0 output to a printer named "laser" with a page shape of 80 columns by 66 lines and a 3 line top and bottom margin. The BRIEF option suppresses the normal confirmation prompt.

```
SETPTR 0,80,66,0,0,3,AS SALES_REPORT,BRIEF
```

Directs print unit 0 output to a record named "SALES_REPORT" in the $HOLD file. The BRIEF option suppresses the normal confirmation prompt.
SETPTR 0,,3
Directs print unit 0 output to the default $HOLD file record (P0), leaving all page shape parameters unchanged.

See also:
Printing, PRINTER, SET.QUEUE, SP.ASSIGN
4.201 SETUP.DEMO

The **SETUP.DEMO** command creates a set of demonstration files.

**Format**

```
SETUP.DEMO {UPDATING}
```

The **SETUP.DEMO** command creates two sets of three demonstration files that are used by the examples in the QM Tutorial Guide and other training materials.

**The Library Database**

This represents a simple library application

- **TITLES** Data relating to a book as a title (Title, author, subject, etc)
- **BOOKS** Data relating to a copy of a book (Date out, reader id, etc)
- **READERS** Data relating to users of the library

**The Sales Database**

This represents a simple sales order processing application

- **STOCK** Data relating to stock held by the shop (Part number, product description, price, etc)
- **CUSTOMERS** Data relating to customers of the shop (Name, address, etc)
- **SALES** Data relating to sales (Date, customer, items bought, etc)

If the **UPDATING** option is present, the original demonstration data and dictionary items are reset but any items added by the user are retained. Without this option, the files revert to their initial state.

The command includes safety checks to avoid overwriting files of the same names that do not appear to be part of the demonstration database.

**See also:**

[DELETE.DEMO](#)
4.202 SH

The **SH** command executes a shell (operating system) command.

**Format**

```text
SH command
!command
```

where

```
command
```

is the shell command to be executed.

The **SH** command executes the given operating system command. No checks are performed on the command to be executed so care needs to be taken not to do anything that would interfere with correct operation of QM.

The ! synonym is provided for compatibility with other systems. This form does not need a space between the ! and the command.

Use of **SH** without a command starts an interactive shell. Use the shell **exit** command to return to QM.

**Example**

```
SH DIR
```

This command lists the current directory on a Windows system.

**See also:**

**OS.EXECUTE**
4.203 SHOW.LIST

The SHOW.LIST command displays the content of an active numbered select list, allowing the user to remove unwanted entries.

**Format**

 SHOW.LIST {listno}

where

*listno* is the select list number. This defaults to 0 if omitted.

SHOW.LIST uses a command set based on that of the query processor SHOW command. Using the commands listed below, the user can scroll through the displayed select list items, setting or clearing a marker (displayed as an asterisk next to the item) which indicates whether the item is to be included in the revised select list. All items are initially in the selected state.

- **T** Move to the top of the list (first page).
- **N** Move to the next page. The return key with no command text has the same effect.
- **P** Move to the previous page.
- **Q** Quit from record selection. Any items marked with an asterisk are entered into the revised select list.
- **QC** Quit, clearing all item selection.
- **?** Display help text. Press any key to return to the main screen.
- **S item** Select *item*. The space before the item description is optional. An asterisk will be displayed next to all selected items.
- **C item** Clear *item*, removing the asterisk marker from the screen.

*item* Synonym for S *item*.

The *item* specification may be

- The number shown next to a displayed item.
- A range of numbers in the form *a*-*b* which indicates that the command is to be applied to all items from that tagged with number *a* to that tagged with number *b*. There must be no spaces either side of the hyphen.
- The keyword VISIBLE to apply the command to all items on the current page.
- The keyword ALL to apply the command to all items in the list.

Multiple item specifications may be included in a single command by using either a space or a comma as a separator. For example "1,4,8-11".

The VISIBLE and ALL keywords may be abbreviated by omitting any number of trailing letters (e.g. VIS or V).
4.204 SLEEP

The SLEEP command suspends execution of further commands until a given number of seconds have elapsed or until a specified time of day.

**Format**

```
SLEEP time
```

where

- `time` is either a number of seconds or a time of day in any format accepted by the MT input time conversion.

If `time` is a positive integer value, the process is suspended for that number of seconds.

If `time` is a time of day such as 8:27PM or 22:00:30, the process is suspended until that time. Unlike the QMBasic SLEEP statement, the SLEEP command allows for sleeping past midnight. A SLEEP to 12:00 executed at 13:00 would sleep for 23 hours.

If `time` is omitted, the process is suspended for one second.

The SLEEP command reports an error if the value of `time` is not a positive integer and cannot be converted to a time of day.
4.205 SORT.LIST

The SORT.LIST command sorts a saved select list.

Format

```
SORT.LIST  src.list  {TO tgt.list}  {sort.rule}
```

where

- `src.list` is the name of a previously saved select list in the $SAVEDLISTS file.
- `tgt.list` is the name of the new sorted list to be created in the $SAVEDLISTS file. If omitted, `src.list` is replaced.
- `sort.rule` identifies sorting sequence. If omitted, an ascending left aligned sort is used.

The SORT.LIST command sorts a previously saved select list, either creating a new list or replacing the original.

The `sort.rule` is formed from the following single letter elements:

- **A**  Sort in ascending order (default)
- **D**  Sort in descending order
- **L**  Sort as left aligned values (default)
- **R**  Sort as right aligned values
- **N**  Ignore null elements
- **U**  Return unique items. Multiple occurrences of an item are replaced by just one item.

Invalid or conflicting `sort.rule` elements are ignored.

Example

```
SORT.LIST INVENTORY DL
```

This example sorts the list name INVENTORY into descending left aligned sequence, replacing the original list with the result.

See also: COPY.LIST, DELETE.LIST, EDIT.LIST, FORM.LIST, GET.LIST
4.206 SP.ASSIGN

The SP.ASSIGN command uses a form queue number to set the destination and other options for printer output.

Format

```plaintext
SP.ASSIGN {options} {, setptr.options}
```

where options are chosen from:

- `n` is the number of copies to print.
- `Fqno` specifies the form queue number in the range 0 to 999.
- `H` directs output to the hold file.
- `O` keeps the print unit open until the PRINTER CLOSE command is used.
- `Qqno` Same as `Fqno`.
- `Runit` uses the specified print unit, zero if omitted.
- `S` Suppresses printing.

The SP.ASSIGN command references the $FORMS file (set up using SET.QUEUE) to relate Pick style form queue numbers to the corresponding SETPTR options. The options to SP.ASSIGN can be used to override the settings in the form queue definition.

The `Fqno` or `Qqno` options specify the form queue to be used. This defaults to zero if omitted.

The queue number is used to read the corresponding print unit details from the $FORMS file. The remaining options can be used to override settings in the $FORMS entry.

Use of both the `H` and `S` options selects mode 3 printing, directing output to the hold file. When the `H` option is used without the `S` option, print mode 6 is selected causing the output to be written to a file and also printed.

SP.ASSIGN sets the characteristics of print unit zero, the default printer, unless the `Runit` option is given.

If no options are given, the SP.ASSIGN command shows the form queue number associated with the default print unit and the settings of this printer. This is equivalent to the SP-ASSIGN ? command in Pick style systems.

If the $SETPTR.DEFAULTS record exists (see SETPTR), the default options are applied at the point when the SP.ASSIGN command is used.

The setptr.options clause can be used to specify additional SETPTR options that are to be applied to the selected form queue. A comma is required before this clause and between each SETPTR option.

Note: SP.ASSIGN is provided to ease migration of applications from Pick style environments. It is strongly recommended that new applications should make direct use of SETPTR.
Examples

SP.ASSIGN F3
This command uses the definition of form queue 3 to process output sent to the default printer (print unit 0).

SP.ASSIGN HSF3
This command uses the definition of form queue 3 but forces use of print mode 3 to send the data to a file.

SP.ASSIGN HF3
This is similar to the previous example except that the data is saved to a file and also printed.

SP.ASSIGN 2R4F6
This command uses the definition of form queue 6 to process output sent to print unit 4 but forces the number of copies to be 2.

SP.ASSIGN F3,BANNER "Sales"
This command uses the definition of form queue 3 but also includes the BANNER option to set the banner page text.

See also:
Printing, SETPTR, SET.QUEUE
4.207 SP.OPEN, SP.CLOSE

The SP.OPEN command sets the "keep open" flag on the default printer, merging multiple print requests into a single print job. The SP.CLOSE command resets this option.

Format

SP.OPEN
SP.CLOSE

The SP.OPEN command is equivalent to use of the O option of the SP.ASSIGN command or the KEEP.OPEN option of the SETPTR command.

When this mode is set, output directed to the default printer (print unit 0) from successive commands or programs is merged into a single print job. The job is terminated and queued for printing by use of SP.CLOSE or PRINTER CLOSE.

Example

SP.OPEN
LIST CUSTOMERS,NORTH LPTR
LIST CUSTOMERS,SOUTH LPTR
SP.CLOSE

The above sequence of commands lists records from two elements of the CUSTOMERS multifile as a single print job.

See also:
SETPTR, SP.ASSIGN
4.208 SP.VIEW

The SP.VIEW command views and, optionally, prints records from $HOLD or other files.

Format

```
SP.VIEW {file} {item | sort} { LPTR n }
```

where

- **file** is the file to be processed. If omitted, $HOLD is used.
- **item** is the record id of the record to be processed. If omitted, a pick list of records is displayed.
- **sort** controls the sort order of the displayed item list.
- **LPTR n** specifies the default print unit to be used. If omitted, print unit 0 is used.

If only one name is specified on the command line, it is assumed to be the file name if a corresponding VOC F or Q-type record exists and there is no record of that name in $HOLD.

When the item name is omitted, the default behaviour of SP.VIEW is to show an alphabetically sorted list of available items. The optional sort clause allows this sort order to be modified. It has the same form as a query processor **BY** or **BY.DSND** clause, including possible use of the **NO.CASE** qualifier, for example,

```
BY DTM
```

where DTM is a data item defined in the dictionary of the file being processed. Use of the sort clause will destroy any existing default select list.

The SP.VIEW command allows users to view and print records. If no item is specified, a pick list of records in file is displayed. The cursor keys, page up, page down, home and end keys can be used to move around within this list. Equivalent actions can be performed using the letter keys or the **SED** style control keys.

```
  Cursor up          U    Ctrl-Z  Move up one line
  Cursor down        D    Ctrl-N  Move down one line
  Page up            P    Ctrl-V  Move to previous page
  Page down          N    Esc-V  Move to next page
  Home               T    Esc-<   Move to top of list
  End                B    Esc->   Move to bottom of list
  Return             Q                Quit from program. Requires use of return key to confirm
```

After an item has been selected, either as described above or by specifying the item name on the command line, the text of the given record is displayed. The cursor keys, page up, page down, home and end keys can be used to move around within the data. Equivalent actions can be performed using the letter keys or the **SED** style control keys.
Cursor up U Esc-Z Move to previous page
Cursor down D Ctrl-N Move to next page
Cursor right R Ctrl-F Pan right
Cursor left L Ctrl-B Pan left
Page up P Ctrl-V Move to previous page
Page down, Return N Esc-V Move to next page
Home T Esc-< Move to top of record
End B Esc-> Move to bottom of record
Q QUIT from record. Requires use of return key to confirm
S Spool item
X Delete item. Prompts for confirmation.

The S option prompts for the print unit (defaulting to that in the LPTR option, if used), asks whether line numbering is to be used and then prints the item.
4.209 SPOOL

The SPOOL command sends specified records to the printer.

Format

\[ \text{SPOOL file record(s) \{ LINES m n \} \{ LNUM \} \{ LPTR n \}} \]

where

- **file** is the file holding the record(s) to be printed.
- **record(s)** is a list of records to be printed. If no record ids are specified, SPOOL will use the default select list to identify the records to be printed.
- **LINES m n** specifies that only line \( m \) to \( n \) of the record(s) are to be printed. The value of \( m \) must be greater than 0 and the value of \( n \) must be greater than \( m \).
- **LNUM** specifies that line numbers are to be printed.
- **LPTR n** specifies the print unit to be used. If omitted, print unit 0 is used.

The SPOOL command sends the specified records to print unit 0. The actual destination is determined by use of either PRINTER or SETPTR.

Use of the LNUM keyword prefixes each line by its line number, a colon and a single space. The line number is printed as a minimum of four digits but expands if the record has more than 9999 lines.

Example

\[ \text{SPOOL INVOICES 01249 01250} \]

The above command would print records 01249 and 01250 from the INVOICES file. The SPOOL command does no formatting of the data except to replace field marks by newlines.
4.210 STATUS

The STATUS command displays a list of active phantom processes.

Format

STATUS { ALL }

where

ALL Displays status information for all phantom processes. Without this option, only phantoms started by the current user are displayed.

A list of phantom processes is displayed in the form

User Started Command
2 10:30:28 18 May 1994 BASIC BP ACC

The login time is shown in the local time zone of the user executing the command.

If there are no phantom processes, the STATUS command displays

There are no phantom processes

See also:
CHILD(), LIST.PHANTOMS, PHANTOM, !PHLOG()
4.211 STOP

The STOP command terminates the currently active paragraph.

Format

    STOP \{text\}

The STOP command is intended as a means of exiting from the middle of a paragraph. The active paragraph is discarded, control returning to the paragraph, menu, etc from which the paragraph was started or, if none, to the command prompt.

If \textit{text} is present, the text is displayed on the terminal.

The value of \texttt{@SYSTEM.RETURN.CODE} is not affected by the STOP command.

See also:

ABORT
4.212 SUBSCRIBE

The **SUBSCRIBE** command starts a phantom process that will poll a replication publisher for updates.

**Format**

```
SUBSCRIBE server[:port] username password {options}
```

where

- **server** is the network name or ip address of the publisher.
- **port** is the port number for connection to the publisher. This must correspond to the value of the `REPLPORT` configuration parameter on the publisher and defaults to 4244 if omitted.
- **username** is the user name to be used for the server process established on the publisher.
- **password** is the password associated with **username**. For security, this should be encrypted using the `AUTHKEY` command and specified with an ENCR: prefix.
- **options** are chosen from:
  - **INTERVAL** $n$ specifies the interval in seconds at which the publisher will be polled for updates. This must be in the range 1 to 300 and defaults to 5.
  - **RAW** specifies that data transferred between the publisher and subscriber systems should not be encrypted. There is a marginal performance benefit in using this when the network is considered to be secure.

The **SUBSCRIBE** command starts a phantom process that polls for and applies data updates from the specified publisher. It should normally be executed using the `STARTUP` configuration parameter. The command can only be executed in the QMSYS account and by users with administrator rights.

When subscribing to files that use encryption, the user name of the subscriber process must have full access to the associated encryption keys.

The **REPLSRVR** configuration parameter must be set to match the subscriber server name as used when publishing the files with the **PUBLISH** command.

**See also:**
- Replication, **DISABLE.PUBLISHING**, **ENABLE.PUBLISHING**, **PUBLISH**, **PUBLISH.ACCOUNT**
4.213 T.DUMP

The **T.DUMP** command saves data to a Pick style T-DUMP tape.

**Format**

```
T.DUMP {DICT} filename {id...} {BINARY} {COUNT.SUP} {DET.SUP} {FROM listno}
```

where

- `filename` is the name of the file to be saved.
- `id...` is a list of record ids to be saved.
- `BINARY` suppresses conversion of newlines to field marks in directory files. Use this mode when saving binary data.
- `COUNT.SUP` suppresses display of the count of records saved.
- `DET.SUP` suppresses display of the detailed list of records saved.
- `FROM listno` uses the specified select list to identify the records to be saved.

The **T.LOAD** command processes the named file to produce a Pick style T-DUMP tape.

The tape or pseudo-tape to be created must first be assigned to the process using the `SET.DEVICE` command.

By default, the entire content of the named file is saved. If the default select list is active or the `FROM` option is used to identify an active select list, that list is used to determine the records to be saved. Alternatively, a list of ids may be given on the command line.

**Encryption**

**T.DUMP** operates within the security rules imposed by use of QM's encryption features. Files that use record level encryption cannot be saved if the user performing the save does not have access to the encryption key. The data saved for files that use field level encryption will have all fields for which the user is denied access set to null strings. All saved data is recorded in decrypted form and hence storage of **T.DUMP** media may reduce system security. The media format used by **T.DUMP** is an industry standard that does not provide a way to record details of data encryption. Restoring a save in which encrypted fields have been omitted is unlikely to yield a usable file. Backup of accounts that include encrypted data should be performed using operating system level tools.

**Restrictions**

The media format of **T.DUMP** also imposes a restriction that makes it impossible to save an item that contains a field mark (character 254) immediately followed by a text mark (character 251). In practical terms, this means that it is unlikely to be possible to use **T.DUMP** to save binary data such as compiled programs or dictionary items. For this reason, **T.DUMP** is not considered as suitable for routine system backups.
See also:
ACCOUNT.RESTORE, ACCOUNT.SAVE, FILE.SAVE, FIND.ACCOUNT, QMSAVE, RESTORE.ACCOUNTS, SEL.RESTORE, SET.DEVICE, T.ATT, T.LOAD, T.xxx
4.214 T.LOAD

The T.LOAD command restores a Pick style T-DUMP tape.

Format

```
T.LOAD {DICT} filename {item.list} {BINARY} {COUNT.SUP} {DET.SUP} {OVERWRITING} {NO.QUERY}
```

where

- **filename** is the name of the file to receive the restored data.
- **item.list** is the list of items to be restored. The default select list may be used instead.
- **BINARY** suppresses translation of field marks to newlines in directory files. Use this mode when restoring binary data.
- **COUNT.SUP** suppresses display of the count of records restored.
- **DET.SUP** suppresses display of the detailed restore progress.
- **OVERWRITING** causes existing records to be replaced.
- **NO.QUERY** suppresses the confirmation prompt when restoring using a select list. The NO.SEL.LIST.QUERY mode of the OPTION command can be used to imply this option.

The T.LOAD command processes a Pick style T-DUMP pseudo tape and restores data from it into the named QM file.

The tape or pseudo-tape to be read must first be assigned to the process using the SET.DEVICE command.

See also:
ACCOUNT.RESTORE, ACCOUNT.SAVE, FILE.SAVE, FIND.ACCOUNT, QMSAVE, RESTORE.ACCOUNTS, SEL.RESTORE, SET.DEVICE, T.ATT, T.DUMP, T.xxx
The **T.xxx** utility commands perform various functions to a tape device.

**Format**

- **T.DET**: Detach device
- **T.EOD**: Position to end of data
- **T.FWD**: Skip forwards one file
- **T.RDLBL**: Read tape label
- **T.READ**: Read tape
- **T.REW**: Rewind tape
- **T.STAT**: Report device status
- **T.WEOF**: Write end of file marker

The tape or pseudo-tape to be processed must first be assigned using the **SET.DEVICE** command.

**See also:**
4.216 TANDEM

The TANDEM command allows an administrative user to monitor or control another QM session.

**Format**

```
TANDEM userno
```

where

```
userno  is the QM user number of the session to be monitored.
```

The TANDEM command is primarily intended to allow users with administrator rights to monitor a QM session where the user needs assistance. All terminal output sent to the monitored user’s screen is also sent to the monitoring user’s screen.

Because use of TANDEM potentially weakens system security, it must be enabled using the FEATURE configuration parameter. The VOC entry for TANDEM is initially only present in the QMSYS VOC but it may be copied elsewhere.

Execution of the TANDEM command sends a request to the process to be monitored, asking if it is willing to accept the connection. A prompt box is displayed on the user’s screen, showing the user number and user name of the session attempting to make a connection. The user must respond Y or N within 30 seconds. Either response clears the prompt box from the screen, restoring the overwritten text if the terminal type in use supports screen save/restore. Use of the SILENT.TANDEM mode of the OPTION command suppresses this confirmation prompt.

If the connection is accepted, the monitoring session enters "view mode" where all output sent to the monitored user's screen is duplicated on the monitoring user's screen. For this to work correctly, both sessions must be using the same terminal type and settings.

The monitoring user can switch to "feed mode" by entering Esc-Esc-F. When in this mode, all keyboard input typed in the monitoring session is passed to the monitored session and handled as though that user had typed it. This mode allows the monitoring user, for example, to demonstrate correct use of the application. Function keys that send multi-byte escape sequences should work correctly when in feed mode. If a lone escape is needed, this can be sent with Esc-Esc-E. The link can be returned to view mode by entering Esc-Esc-V.

When in either mode, the tandem connection can be terminated by entering Esc-Esc-X.
4.217 TERM

The TERM command specifies the terminal display dimensions.

Format

TERM
TERM { columns } { , lines } { , prt.cols } { , prt.lines } { term.type }  
TERM COLOUR { bgc } { , fgc }  
TERM DEFAULT  
TERM DISPLAY

where

(columns) is the width of the display device.
(lines) is the depth (number of lines) of the display device.
(prt.cols) is the width (number of columns) of the default printer.
(prt.lines) is the depth (number of lines) of the default printer.
(bgc) is the required background colour
(fgc) is the required foreground colour

The TERM command with no qualifying information reports the display device settings. It shows the page width and depth together with the terminal type (as in the @TERM.TYPE variable).

The four numeric parameters (columns, lines, prt.cols and prt.lines) are all optional. The intervening commas must be included except after the final used parameter. The columns and lines values specify the page shape of the display device. The prt.cols and prt.lines values apply to the default printer. The value of any parameter that is not included in the command remains unchanged. If used for a console session or a QMTerm connection with values of columns and lines less than 256, the command will also set the device window to the given shape.

The columns values must be in the range 20 to 32767. The lines values must be in the range 10 to 32767.

The term.type option selects the terminal device type. Note that there is no comma before the terminal type. The name given must correspond to an entry in the terminfo library.

The COLOUR (or COLOR) option can be used to set the background and foreground colours. The colour names are case insensitive and chosen from BLACK, BLUE, GREEN, CYAN, RED, MAGENTA, BROWN, WHITE, GREY, BRIGHT BLUE, BRIGHT GREEN, BRIGHT CYAN, BRIGHT RED, BRIGHT MAGENTA, YELLOW, BRIGHT WHITE. The American spelling GRAY can be used in place of GREY and the two word names can be written with either a space or a dot between the words.

Note that some terminal emulators only apply the background colour to characters output by the application, not to the entire window. Thus clearing the screen, for example, may lead to unexpected results.
The **TERM DEFAULT** command resets the device to 80 x 24. The printer characteristics are unaffected.

The **TERM DISPLAY** command displays a list of the more important input key codes and output control sequences for the currently selected terminal type in terminfo format.

**Example**

```
TERM 120,32
```

The above command sets the terminal to be 120 columns wide by 32 lines. If the default printer was set to be 132 columns and 66 lines per page, a subsequent **TERM** command with no qualifying information would report

```
Screen width : 120
Screen depth : 32
Printer width: 132
Printer depth: 66
Terminal type: QMTERM
```

```
TERM ,,120,66
```

The above command sets the default printer to be 120 columns wide by 66 lines.

**See also:**

PTERM
4.218 TIME

The TIME command displays the current date and time or translates a time between internal and external format.

Format

TIME \{time\}

TIME INTERNAL

If no time is specified, the date and time are reported in the form

14:30:00  12 May 1993

This format is fixed, regardless of use of the DATE FORMAT command.

If time is specified, the converted form of this time (internal to external or vice versa) is reported.

The TIME INTERNAL form displays the current time in internal form (seconds since midnight).

See DATE for an alternative format date and time report.

@SYSTEM.RETURN.CODE is not affected by this command.
The **UMASK** command sets the default access rights for files created within the QM session.

**Format**

```
UMASK { rights }
```

The **UMASK** command operates in the same way as the umask command of Unix/Linux, specifying the default access rights for files created within this QM session.

The *rights* code consists of three octal digits that specify the access permissions that the file is not to have. The first digit sets the rights for the owner of the file. The second digit sets the rights for users in the group to which the file is assigned except for the owner. The third digit sets the rights for all other users. Each digit is formed by adding the values:

- 4  Do not allow read access
- 2  Do not allow write access
- 1  Do not allow execute access (attach access for directories)

Thus, for example,

```
UMASK 022
```

allows full access to the file's owner but prevents write access by all other users.

If the *rights* are omitted, the **UMASK** command reports the current access rights.
4.220 UNLOCK

The **UNLOCK** command, available only in the QMSYS account, releases task, record or file locks set by any process.

**Format**

\[
\text{UNLOCK} \{ \text{USER} \ user.no \} \ {\text{FILE} \ file.no} \ {\text{ALL}} \ {\text{record.ids}} \ldots \\
\text{UNLOCK} \{ \text{USER} \ user.no \} \ {\text{FILE} \ file.no} \ \text{FILELOCK} \\
\text{UNLOCK} \ \text{TASKLOCK} \ lock.no\ldots
\]

In the first form, **UNLOCK** releases record ids set by user *user.no* on file *file.no*. At least one of these options must be present. The values of *user.no* and *file.no* can be found from the output of the **LIST.READU** command. The **UNLOCK** requires either a list of record ids or the **ALL** keyword to determine which records to unlock.

In the second form, the file lock set by the specified user on the given file is released. Again, at least one of the **USER** and **FILE** options must be specified.

The third form allows task locks owned by other users to be released. Any number of locks may be released in a single command.

The **UNLOCK** command is restricted to users with administrator rights and should be used with great care. Locks are taken by application software to protect critical operations. Releasing a lock can cause data integrity problems.

**Example**

```
LIST.READU
File  Path
  17  D:\SALES\INVENTORY
File  User  Type  Id
  17  4  RU  18464
  17  4  RU  21968
UNLOCK USER 4 18464
```

In this example, **LIST.READU** is used to check which locks are outstanding and the **UNLOCK** command is used to release a specific record lock.
4.221 UNLOCK.KEY.VAULT

The UNLOCK.KEY.VAULT command enables access to the encryption key vault when using the optional vault access security. This command can only be executed by users with administrator rights.

Format

UNLOCK.KEY.VAULT

The command prompts for the master key string.

The UNLOCK.KEY.VAULT command enables access to encrypted files. This optional level of security is selected when the master key is created and requires that the key is entered using this command every time that QM is started (once only, not for each user entering the system). Whilst potentially inconvenient, this mechanism provides security against data access if the entire system is stolen. It is recommended for use with portable systems that hold encrypted data.

The key string entered must be the same as when the key vault was created. The master key cannot be changed unless the key vault is cleared and rebuilt.

See also:
Data encryption, CHANGE.KEYPASSWORD, CREATE.FILE, CREATE.KEY,
CREATE.SECURE.KEY, DELETE.KEY, DISABLE.KEY, DISABLE.KEY(QMBasic),
ENABLE.KEY, ENABLE.KEY(QMBasic), ENCRYPT.FILE, GRANT.KEY, LIST.KEYS,
RESET.MASTER.KEY, REVOKE.KEY, SET.ENCRYPTION.KEY.NAME
4.222 UPDATE.ACCOUNT

The **UPDATE.ACCOUNT** command copies all system VOC entries from NEWVOC, setting the correct locations for system files.

**Format**

```
UPDATE.ACCOUNT
```

The **UPDATE.ACCOUNT** command copies all NEWVOC items to the VOC file of the current account. It is useful after an upgrade or if the VOC has been damaged. Unlike a simple **COPY** from NEWVOC to the VOC, **UPDATE.ACCOUNT** checks for changes that may have a detrimental effect.

After copying items from NEWVOC, the **UPDATE.ACCOUNT** command looks for an optional file named NEWVOC.MODS in the QMSYS account and, if found, copies its content to the VOC file of the account being updated. This allows creation of a site specific set of items that should appear in all VOC files.

Executing this command in the QMSYS account when logged in as a user with administrator rights prompts to ask whether all registered accounts are to be updated. This can be useful after an upgrade on a system with many accounts.

**See also:**

**CREATE.ACCOUNT, DELETE.ACCOUNT**
4.223 UPDATE.LICENCE

The UPDATE.LICENCE command, available only in the QMSYS account, applies new licence details.

Format

UPDATE.LICENCE

The UPDATE.LICENCE command prompts for the licence information supplied with the product or at an upgrade.

Care should be taken to enter the information exactly as it appears on the licence information sheet. When relicensing an existing system, the old licence data is displayed allowing amendment of only the fields that have changed.

Example

LICENCE DETAILS
-----------------------------------------------

Licence number [1961491396] System id WLHX-YRTZ
Max users [50 ]
Expiry date [31 Dec 2000]
Authorisation code [YKSAJ-KKRFW-CNNDK-LRRTW-CJRDT]
Security number [65859]
Site text [Manor Developments Limited]

Use Return, Tab or Cursor keys to move between fields.
Use Ctrl-X to abort licence data entry.

-----------------------------------------------

Enter 10 digit licence number
4.224 UPDATE.RECORD

UPDATE.RECORD simplifies amendment of database files. In batch mode, it allows the same update to be made to multiple records in a file with just one command, possibly changing the content of more than one field. In visual mode, it displays a full screen image of fields from the data record in their external (converted) form, allowing modifications to be entered as the cursor is moved around the displayed data.

Format

```
UPDATE.RECORD {DICT} file {USING {DICT} dict}
    FROM listno
    ALL
    id { id ...}
    INQUIRING {prompt}
```

Batch mode:

```
field.value {CONV "spec"} { field.value ...}
DELETING field
    {COUNT.SUP}
    {VERIFY.SUP}
    {EXCLUSIVE}
    {WAIT} or {NO.WAIT}
    {CREATING}
    {OVERWRITING}
    {REPORTING}
    {LPTR [n]}
    {NO.PAGE}
```

Visual mode:

```
{ID.SUP}
{COL.SUP}
```

Identifying the file to be processed:

```
{DICT} file
```
specifies the file to be updated. The optional DICT keyword indicates that the dictionary portion of the file is to be updated.

```
USING {DICT} dict
```
specifies that dict is to be used as the dictionary for file. The optional DICT keyword uses the dictionary portion of dict.

Selection of the records to be updated:

```
FROM listno
```
specifies that select list listno (0 to 10) is to be used as the list of records to process.

```
ALL
```
specifies that all records in file are to be updated. UPDATE.RECORD will use select list 0 internally to generate the list of records to be processed.

```
id { id ...}
```
is a list of specific record ids to process. Record ids should be enclosed in single or double quotes if they contain spaces or commas or match command keywords.
INQUIRING \{prompt\} causes UPDATE.RECORD to prompt for the id of each record to be processed. The prompt string is optional and must be enclosed in single or double quotes if it contains spaces or commas. If omitted, a default of "Record id" is used. Inquiry mode is the default in visual mode if no record ids are specified and the ALL and FROM keywords are not used.

Only one of these selection styles can be used in a single command. If none of the above record selection criteria is given and the default select list is active, that list is used.

Specifying the updates to be made in batch mode:

- \textit{field} identifies the field to be updated. The value given may be a numeric field position or the name of an A, D or S-type item from the dictionary.
- \textit{value} identifies the new value to be placed in \textit{field}. This may be a number, a quoted string, the name of a field or I-type from the dictionary or the keyword EVAL followed by an I-type expression in either single or double quotes.

- \texttt{CONV "spec"} Allows use of a non-default conversion code.

- \texttt{DELETING field} Deletes the specified \textit{field} from the data record, moving all subsequent fields back by one position.

Options controlling the action of UPDATE.RECORD in batch mode:

- \texttt{COUNT.SUP} suppresses the normal report of the number of records processed.
- \texttt{VERIFY.SUP} suppresses the confirmation prior to an update using a select list or the ALL keyword.
- \texttt{EXCLUSIVE} causes UPDATE.RECORD to lock the entire file during the update to ensure exclusive access. If the file cannot be locked, UPDATE.RECORD will display a message and terminate unless the WAIT keyword is also given, in which case it will wait until it can obtain exclusive access.
- \texttt{WAIT} can be used with the EXCLUSIVE keyword as described above. It can also be used alone to specify that UPDATE.RECORD should wait whenever it encounters a locked record.
- \texttt{NO.WAIT} turns off WAIT. It is not normally required as this is the default. UPDATE.RECORD honours the last WAIT or NO.WAIT on the command line.
- \texttt{CREATING} causes UPDATE.RECORD to create records that are not found in the file.
- \texttt{OVERWRITING} allows overwriting of existing records when changing the record id.
REPORTING produces a detailed report of each update.

LPTR \{n\} directs the commentary from UPDATE.RECORD to the specified print unit. Print unit 0 is used if \(n\) is omitted.

NO.PAGE suppresses pagination of output to the display.

The sequence of the command components must be file specification, record ids, fields and values, options.

Options controlling the action of UPDATE.RECORD in visual mode:

ID.SUP suppresses display of the field names.

COL.SUP suppresses display of the display names (column headings in a query report).
**UPDATE.RECORD batch mode**

**UPDATE.RECORD** runs in batch mode of the command line includes one or more *field/value* specifications.

If the **EXCLUSIVE** keyword is used, **UPDATE.RECORD** will obtain exclusive access to the file by acquiring the file lock. If another user holds a record lock or the file lock when the command is issued, **UPDATE.RECORD** will terminate with an error message unless the **WAIT** keyword has been used, in which case a message is displayed and it repeats the attempt to lock the file at five second intervals.

For each record to be processed, **UPDATE.RECORD** first locks and reads the record. If another user has the record locked and the **WAIT** keyword has been used, **UPDATE.RECORD** will wait for the record to become available. If the **WAIT** keyword was not used, the record id is added to a list of locked records. If either the **INQUIRING** or **REPORTING** keywords was used, a message is output showing the id of the record that could not be locked.

If the specified record does not exist in the file and the **CREATING** keyword has been used, a new, empty record is created to which the amendments are then applied. If the **CREATING** keyword is not present, the record id is added to a list of missing records.

Once the record has been successfully locked, **UPDATE.RECORD** performs each of the specified amendments in the order in which they appear on the command line. Each amendment may draw on the result of preceding amendments. Note in particular, that use of the **DELETING** option will result in new field positions for all later fields. The record id may be changed by specifying the target as field 0 or a dictionary item such as @ID that refers to field 0. Any I-type or EVAL expressions after the change of record id will see the new id in @ID.

The *field* specification may be a numeric field position or the name of a D-type item from the dictionary or the VOC. The *value* specification may be:

A number or a quoted string.

The name of an A, D, I or S-type dictionary or VOC item identifying the field from which data is to be copied. If this is an I-type item or an A/S-type item with a correlative expression, it must have been compiled prior to use by **UPDATE.RECORD**.

The keyword **EVAL** followed by an I-type expression in single or double quotes. This expression will be compiled by **UPDATE.RECORD** and evaluated to determine the new value of the field.

**UPDATE.RECORD** honours dictionary conversion specifications for both the source and target data items. Where the new *value* for a field is derived via a dictionary name, the value is first converted to its external form. Where the field to be updated is identified via a dictionary name, the new value is converted to its internal form prior to storing in the record. If the *value* specification is followed by the **CONV** keyword and a quoted conversion specification, this will overrule any conversion specified in the dictionary. **UPDATE.RECORD** will report a warning message if the conversion generates an error when applied to *value*.

The **REPORTING** keyword causes **UPDATE.RECORD** to produce a detailed report of each amendment made, including the record id together with the old and new values of each field amended. This report can be directed to a specific print unit using the **LPTR** keyword.

In the absence of the **REPORTING** and **INQUIRING** keywords or if the report is directed to a print unit rather than the display, **UPDATE.RECORD** displays a progress report in the form of a series of asterisks.
On completion of the update, **UPDATE.RECORD** may create two entries in the $SAVEDLISTS file. If one or more records could not be updated because they were locked by other users, a select list named &LOCK\_userno& is created containing a list of such records. If one or more records specified for update did not exist in the file and the **CREATING** keyword was not used, a select list named &MISS\_userno& is created containing a list of the missing record ids. **UPDATE.RECORD** deletes any old versions of these lists at the start of the update.

Note that the database locking system is of a voluntary nature. Both the file and record level locking performed by **UPDATE.RECORD** only prevent other users acquiring the locks. A program that does not use locks correctly may still be able to access the file during the update.

On completion of the update, **@SYSTEM.RETURN.CODE** will be set to the number of records updated. If the command terminates due to a command line error, **@SYSTEM.RETURN.CODE** will be negative.

**Examples**

**UPDATE.RECORD** **STOCK** 01-1745 **IN.STOCK**, 0

This update amends field **IN.STOCK** of record 01-1745 in the **STOCK** file to be zero.

**UPDATE.RECORD** **SALES** **ALL** **TOTAL.SOLD**, **EVAL** "TOTAL.SOLD + THIS.MONTH" **THIS.MONTH**, 0 **EXCLUSIVE** **REPORTING** **LPR** 2

This update processes all records from the **SALES** file, adding the current value of the **THIS.MONTH** field into the **TOTAL.SOLD** field, resetting **THIS.MONTH** to zero. The **EXCLUSIVE** keyword ensures that the file is not updated by other users during the amendment. A detailed report of the changes is sent to print unit 2.

**UPDATE.RECORD** **PAYROLL** **INQUIRING** **SALARY**, **EVAL** "SALARY * 1.05"

This update processes entries of the **PAYROLL** file specified in response to a record id prompt, increasing the value of **SALARY** by 5%.

**UPDATE.RECORD** **STOCK** **FROM** 2 5,"12/9/96" **CONV** "D2/"

This update uses select list 2 to identify records of the **STOCK** file in which field 5 is to be set to the internal form of the date 12/9/96.

**The Report**

The **REPORTING** keyword causes **UPDATE.RECORD** to generate a detailed report of its actions. This can be used as a simple audit trail of the changes made to the file and also contains information that enables recovery from an incorrect amendment.

The report shows the start of the amendment for a given record, the changes made to each field and the termination of the update at the point when the record has been written back to the file. Missing records and records which have been created by **UPDATE.RECORD** are highlighted in this report.

A command such as

**UPDATE.RECORD** **STOCK** **FROM** 2 **OLD.VALUE**, **VALUE** **VALUE**, **EVAL** "VALUE*1.1" **CREATING"
might produce a report including the following lines

Start of update of record '01-1268'
  Field OLD.VALUE was '' now '90'
  Field VALUE was '90', now '99'
Completed update of record '01-1268'
Record '01-6723' locked by another user.
Creating record '01-7491'
  Field OLD.VALUE was '75' now '80'
  Field VALUE was '80', now '88'
Completed update of record '01-7491'

This example shows the report entries for update of a record found in the file, a record that was not updated because it was locked by another user and a record that was created by UPDATE.RECORD.

If the CREATING option was not used, record 01-7491 would have produced the following report line

  Record '01-7491' not found.
UPDATE.RECORD visual mode

UPDATE.RECORD runs in visual mode if the command line contains no field/value specifications.

Visual mode presents a full screen display of the external (converted) form of fields from a data record. Changes are made by moving the cursor to the desired position and entering or deleting characters. Modified lines are converted back into their internal form within UPDATE.RECORD when the cursor is moved to a new line or when the data is to be written back to the file.

```
1: NAME            : Site Name    : Acme Software Limited
2: ADDRESS         : Address      : 42 High Street, Anytown
3: POSTCODE        : Postcode     : AN11 1XX
4: DATEPAID        : Date Paid   : 12 Feb 98\-17 Mar 98
5: INVOICES        : Invoices     : 001763\-001966
6: LICENCES        : Licences     : 907881792\-1907881802\-1907881808
7: EXPIRED.LICENCES: Expired      :
8: COUNTRY         : Country      :
9: CLASS           : Class        : 3
10: DEALER.SALES   : Sales        :
11: CALLBACK.DATE  : Callback     : 01 Jul 98
12: CALLBACK.TEXT  : Callback note: Interested in new product range
13: VAT.NO         : VAT no       : 614 1210 25
14: SITE.TEXT      : Site text    : Acme Software
15: CONTACT        : Contact      : Anne McIntosh
16: POSITION       : Position     :
17: TEL.NO         : Tel no       : 01234-56789\-01234-64526
18: FAX.NO         : Fax no       : 01234-21767
19: MOBILE.NO      : Mobile no    :
20: EMAIL          : E-mail       : acme@mailer.com
21: SALES.TOTAL    : Sales        : £12783.33
22: NOTES          : Notes        :
23: *CLIENTS 00106 <6,1,1> D2DMYL[,A3] 9R PAYDATE S
```

By default, the display shows all fields for which a D-type dictionary entry or an A/S=type entry with no correlative exists. A specific subset of fields can be displayed by creating a dictionary phrase named @UPDATE.RECORD which lists the required fields (and possibly keywords).

For each field, the display shows:
- The field number
- The field name (unless suppressed by use of ID.SUP)
- The display name (unless suppressed by COL.SUP)
- The data in its external form

The last two lines of the screen are used as a status area. The upper status line displays the file name and record id. An asterisk is shown at the start of this line if the data has been changed.

The lower status line shows the field, value and subvalue in which the cursor is positioned and the dictionary conversion code, format code, association name and single/multivalue flag for the field. The final field of this status line shows a letter O if UPDATE.RECORD is in overlay mode (see below).

UPDATE.RECORD uses a subset of the default key bindings of the SED full screen editor. These all consist of keystrokes which are
- Control shift + key
- ESCape followed by another key
- Ctrl-X followed by another key
The table below summarises the key bindings. All other keystrokes except for unused control shift codes cause the character to be inserted into the record text at the current cursor position.

<table>
<thead>
<tr>
<th>Key</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ctrl-</td>
<td>Home</td>
</tr>
<tr>
<td>Esc-</td>
<td>Cursor left</td>
</tr>
<tr>
<td>Ctrl-X -</td>
<td>Repeat</td>
</tr>
<tr>
<td>Ctrl-X -</td>
<td>Quit</td>
</tr>
<tr>
<td>A</td>
<td>Delete char</td>
</tr>
<tr>
<td>B</td>
<td>End</td>
</tr>
<tr>
<td>C</td>
<td>Cursor right</td>
</tr>
<tr>
<td>D</td>
<td>Cancel</td>
</tr>
<tr>
<td>E</td>
<td>Backspace</td>
</tr>
<tr>
<td>F</td>
<td>(Return)</td>
</tr>
<tr>
<td>G</td>
<td>Kill line</td>
</tr>
<tr>
<td>H</td>
<td>Refresh</td>
</tr>
<tr>
<td>I</td>
<td>(Return)</td>
</tr>
<tr>
<td>J</td>
<td>Cursor down</td>
</tr>
<tr>
<td>O</td>
<td>Overlay</td>
</tr>
<tr>
<td>P</td>
<td>Overlay</td>
</tr>
<tr>
<td>Q</td>
<td>Cursor up</td>
</tr>
<tr>
<td>R</td>
<td>Quote char</td>
</tr>
<tr>
<td>S</td>
<td>Quote char</td>
</tr>
<tr>
<td>T</td>
<td>Save</td>
</tr>
<tr>
<td>U</td>
<td>Repeat</td>
</tr>
<tr>
<td>V</td>
<td>Page down</td>
</tr>
<tr>
<td>W</td>
<td>Page up</td>
</tr>
<tr>
<td>X</td>
<td>Cut</td>
</tr>
<tr>
<td>Y</td>
<td>Copy</td>
</tr>
<tr>
<td>Z</td>
<td>Command</td>
</tr>
<tr>
<td>.</td>
<td>Mark</td>
</tr>
<tr>
<td>=</td>
<td>Expand char</td>
</tr>
<tr>
<td>&lt;</td>
<td>Top</td>
</tr>
<tr>
<td>&gt;</td>
<td>Bottom</td>
</tr>
<tr>
<td>Bkspc</td>
<td>Backspace</td>
</tr>
<tr>
<td>Del</td>
<td>Delete char</td>
</tr>
<tr>
<td>Return</td>
<td>Cursor down</td>
</tr>
</tbody>
</table>

**UPDATE.RECORD** also recognises the following terminal control keys:
Some functions are available using alternative key sequences. Such alternatives are shown above and in the descriptions that follow.

The **repeat** function (Ctrl-C or Ctrl-U) repeats the previous function.

The **cancel** function (Ctrl-G) can be used to abort partially entered incorrect key sequences and to terminate certain functions as described below.

**Cursor Movement Functions**

**Note:** A confirmation prompt appears if the cursor is moved from a line that contains a data conversion error.

**Home** (Ctrl-A or Home)
Moves the cursor to the start of the current line.

**End** (Ctrl-E or End)
Moves the cursor to the position after the last character in the current line.

**Top** (Esc-<)
Moves to the start of the first displayed field.

**Bottom** (Esc- >)
Moves to the start of the last displayed field.

**Cursor down** (Ctrl-N or Ctrl-P or Cursor down)
Moves the cursor vertically down one line. If this position is beyond the end of the data in the new line, the cursor is displayed immediately to the right of the final character.

**UPDATE.RECORD** remembers the column position from which the cursor was moved so that a further vertical movement will continue to place the cursor at the lesser of its original column position and the end of the current line.

**Cursor up** (Ctrl-P or Ctrl-Z or Cursor up)
Moves the cursor vertically up one line. The same process is used for determining the column position as for the **cursor down** operation described above.

**Cursor right** (Ctrl-F or Cursor right)
Moves the cursor right.

**Cursor left** (Ctrl-B or Cursor left)
Moves the cursor left.

**Page down** (Ctrl-V or Page down)
Moves the cursor down by one screen or to the last line.

**Page up** (Esc-V or Page up)
Moves the cursor up by one screen or to the first line.

**Goto** (Esc-G)
Prompts for a field, value and subvalue position and moves the cursor to that position. The position may be specified as:

- **field**
- **field, value**
- **field, value, subvalue**

Fields may be specified by number or name. Omitted field or value components mean "within the current field/value" unless a higher level component is specified in which case the default is 1. For example:

- **,value**
- **,value,subvalue**
- **,,subvalue**
- **field,,subvalue**

An asterisk can be use to imply "no change". This is useful when processing associated multivalued fields. For example:

- ***,value**
- **field,***
- **field,*,***

If the specified value or subvalue does not exists, **UPDATE.RECORD** will offer to create it.

### Data Insertion

Data is inserted at the current cursor position. If overlay mode is set the new data overwrites any existing data at this position, otherwise it is inserted before the character under the cursor. Overlay mode may be toggled using the **overlay** function (Ctrl-O or Ctrl-X O or Insert).

Any character other than a field mark or item mark may be inserted. The **quote character** function (Ctrl-Q or Esc-Q) allows insertion of non-printing characters. It may be used in four ways:

- Followed by a number of up to three digits, it inserts the character with that decimal ASCII sequence.
- Followed by X and a hexadecimal number, it inserts the character with that ASCII or Unicode sequence. The number of digits is limited to 2 on non-ECS systems, 4 on ECS systems.
- Followed by V, S or T, it inserts a value mark, subvalue mark or text mark respectively.
- Followed by any other character, usually a non-printing character, it will insert that character.

### Copying, Deleting and Restoring Data

**Delete char** (Ctrl-D or Del, Delete)

The character at the current cursor position is deleted.

**Backspace** (Backspace or Ctrl-H)

The **backspace** function removes the character to the left of the cursor.

**Kill line** (Ctrl-K)

The **kill line** function deletes all characters following the cursor.
Copy (Esc-W)
The copy copies part of a field to the clipboard buffer. The required sequence of actions is:
  Position the cursor on the first character to be copied.
  Execute the mark function (Esc-.).
  Position the cursor after the last character to be copied. Where the terminal device allows, the selected characters will be highlighted.
  Press the copy key.

The copy function can be cancelled using the cancel function (Ctrl-G).

Cut (Ctrl-W)
The cut function cuts (deletes) part of a field, placing a copy of the deleted text in the clipboard buffer. The required sequence of actions is:
  Position the cursor on the first character to be cut.
  Execute the mark function (Esc-.).
  Position the cursor after the last character to be cut. Where the terminal device allows, the selected characters will be highlighted.
  Press the cut key.

The cut function can be cancelled using the cancel function (Ctrl-G).

Paste (Ctrl-Y or Esc-Y)
The paste function inserts a copy of the clipboard buffer as set using copy or cut at the current cursor position.

Miscellaneous Functions

Save (Ctrl-X S or Ctrl-X Ctrl-S)
The save function saves the modified data record. UPDATE.RECORD remains in the current record allowing further changes if required.

The save function cannot be executed if the current line contains a data conversion error.

Quit (Ctrl-X C or Ctrl-X Ctrl-C)
The quit function moves to the next record to be processed (if any). A confirmation prompt is displayed if the current record has been modified and not saved.

Expand char (Ctrl-X =)
Certain control characters (e.g. tab, form feed) are represented on the screen by question marks. The expand char function displays the character sequence number for the character at the cursor position on the lower status line.

Command (Esc-X)
The command function allows executing of any valid QM command from within UPDATE.RECORD. In addition, it supports the following built-in commands:

SPOOL
  Used without any following file name etc. , this command spools a copy of the record to the default printer.

QUIT
  Terminates processing of the current record and exits UPDATE.RECORD, abandoning any further records specified for processing.
4.225 VERIFY.CONSTRAINTS

The VERIFY.CONSTRAINTS command is used to check whether data in a file complies with data integrity constraints.

**Format**

```
VERIFY.CONSTRAINTS file.name
VERIFY.CONSTRAINTS file.name,subfile.name
VERIFY.CONSTRAINTS file.name,*
```

where

- `file.name` is the name of the file for which data integrity constraints are to be checked. This must correspond to a VOC F-type record.
- `subfile.name` is the name of the subfile if `file.name` references a multi-file. Use of an asterisk as the subfile name checks the constraints in all subfiles.

The data integrity constraint rules are defined in field 12 of D, A or S-type records in the dictionary. If the rules are changed, the VERIFY.CONSTRAINTS command can be used to check that existing records meet the new rules.

On completion of the command, `@SYSTEM.RETURN.CODE` will be zero if all records comply with the new rules, a positive number of errors detected or a negative error number.

**See also:**
- Data integrity constraints, COMPILE.CONSTRAINTS, VALIDATE()
4.226 WEBSVC

The WEBSVC command listens for incoming simple web services connections.

Format

```
WEBSVC subr {FILE n} {MAX n} {POOL name} {PORT portno}
```

where

- **subr** Specifies the name of the subroutine to be called to process the incoming requests.
- **DEBUGGING** Causes the child process to write the request header and response header to the phantom log file.
- **FILE n** Sets the threshold size in kb above which the request body will be passed to the subroutine via a file instead of in memory. If not set, a file is never used.
- **MAX n** Sets the maximum number of simultaneous child processes. Defaults to 10 if omitted.
- **POOL name** Specifies the connection pool name to be used for the child processes.
- **PORT portno** Is the port number on which to listen for incoming connections. Defaults to 80 if omitted. Linux systems only allow processes running as root to listen on port numbers less than 1024.

The WEBSVC command, typically used in a phantom process, listens for incoming web services connections on the specified port number. When a connection arrives, a phantom process is started to service this request. If the limit on simultaneous connections has been reached, the listener process pauses briefly until some other connection terminates.

The phantom process parses the incoming web request, separating the header from the request body, and calls the catalogued processing subroutine specified in the command line. This subroutine should be declared as

```
SUBROUTINE name(IP.ADDR, ACTION, uri, rqst.header, rqst.body, response.header, response.body, close.connection, info)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP.ADDR</td>
<td>IP Address</td>
</tr>
<tr>
<td>ACTION</td>
<td>Request Type (GET, POST, etc)</td>
</tr>
<tr>
<td>uri</td>
<td>URI</td>
</tr>
<tr>
<td>rqst.header</td>
<td>Request Header as a data collection</td>
</tr>
<tr>
<td>rqst.body</td>
<td>Request Body and associated</td>
</tr>
<tr>
<td>response.header</td>
<td>Response Header and associated</td>
</tr>
<tr>
<td>response.body</td>
<td>Response Body and associated</td>
</tr>
<tr>
<td>close.connection</td>
<td>Close connection after sending response?</td>
</tr>
<tr>
<td>info</td>
<td>Additional information</td>
</tr>
</tbody>
</table>
The final argument, INFO, was introduced at release 3.4-16 as a data collection containing miscellaneous information that the user supplied subroutine may find useful. This argument is optional and may be omitted. If present, the data collection may contain:

- **pool**: Pool name if connection pooling is being used.
- **port**: Server port number.
- **request.body.path**: Temporary file pathname when passing the request body via a file. This collection element is absent for requests that are passed in memory. The user supplied subroutine should delete the file when it is no longer needed.
- **response.body.path**: On return from the user supplied subroutine, if this collection element is present, it contains the pathname of a file that holds the response body.
- **delete.response.file**: On return from the user supplied subroutine, if this collection element is True and the response was returned via a file, the file will be deleted.

As an example of how the incoming request is processed, consider a simple request submitted via a browser:

```
http://mysite.com/a/b?x&y&z
```

- **ACTION**: is the HTTP request type (GET, POST, etc).
- **URI**: is the URI string "/a/b?x&y&z"
- **RQST.HEADER**: is a data collection with an element for each of the name/value pairs in the HTTP request such as:
  - `ACCEPT text/html, application/xhtml+xml, */*`  
  - `ACCEPT-ENCODING gzip, deflate`  
  - `ACCEPT-LANGUAGE en-GB`  
  - `CONNECTION Keep-Alive`  
  - `HOST mysite.com`  
  - `USER-AGENT Mozilla/5.0`

- **RQST.BODY**: would be a null string in this example.

The subroutine processes this data and constructs the response body as a field mark delimited dynamic array. For example,

```
RESPONSE.BODY = '<!DOCTYPE html>'  
RESPONSE.BODY<-1> = '<html>'  
RESPONSE.BODY<-1> = '<p>Hello ' : IP.ADDR : '</p>'  
RESPONSE.BODY<-1> = '</html>'  
RESPONSE.BODY = CHANGE(RESPONSE.BODY, @FM, CRLF)
```

The final line in the above example is needed from QM release 3.4-15 onwards as the data returned via the subroutine's RESPONSE.BODY argument is treated as binary data and not altered in any way before transmission.

The default values for RESPONSE.HEADER and CLOSE.CONNECTION if not updated by the user supplied subroutine are equivalent to:

```
RESPONSE.HEADER = 'HTTP/1.1 200 OK'  
RESPONSE.HEADER<-1> = 'Content-Type: text/html'
```
RESPONSE.HEADER<-1> = 'Server: QM/:change(system(SYS$VERSION), '-', '.')
RESPONSE.HEADER<-1> = 'Date: ' : OCONV(EPOCH(), 'E4WADMYTSZL[A3,"", "A3]')
RESPONSE.HEADER<-1> = 'Connection: close'

CLOSE.CONNECTION = @TRUE

If the POOL command line parameter has been used, the phantom process will go into an idle wait state until either a new connection arrives or the pool timeout is reached. Use of connection pools can improve performance as the process can potentially omit initialisation tasks such as opening files.
4.227 WHO

The **WHO** command displays the current user number and account name.

**Format**

```
WHO {USER}
```

Each directory holding a VOC file is termed an **account**. Multiple accounts are useful where there are several distinct projects. They can also be used to separate development and production versions of an application.

The **WHO** command displays the current user number and account name. If the current account is not the same as the initial account on entry to QM, the initial account name is also displayed.

If the **USER** qualifier is present, the command also shows the user name under which the session is logged in.

**Examples**

```
WHO
  1 AC1

LOGTO AC2
WHO
  1 AC2 from AC1
```

In this example, the user enters QM in account AC1. The **WHO** command shows the user number and this account name. A **LOGTO** command is used to transfer to AC2. The **WHO** command now shows the new account and the original.

```
WHO USER
  1 AC2 from AC1 user SARAH
```

In this example, the command additionally shows that the session is running as user SARAH.
4.228 WHERE

The **WHERE** command displays the pathname of the current account.

**Format**

```
WHERE
```

This command is a simple sentence that displays the value of the **@PATH** system variable.

**Example**

```
WHERE
   C:\QMACC\SALES
```
Part 5

Query Processing
5 Query Processing

QM verbs that select records from files or produce reports are handled by the query processor. All query processor verbs follow a common format.

The query processor verbs are

- **LIST** List records meeting specified criteria
- **LIST.ITEM** List records meeting specified criteria in internal format
- **LIST.LABEL** List records meeting specified criteria in address label format
- **REFORMAT** Builds a new file from data in the source file
- **SCAN** Search for records containing specified text
- **SHOW** Interactive select list generation
- **SORT** List records meeting specified criteria in order of record id
- **SORT.ITEM** List records meeting specified criteria in order of record id in internal format
- **SORT.LABEL** List records meeting specified criteria in address label format, in order of record id
- **SELECT** Create a select list of records meeting specified criteria
- **SREFORMAT** Builds a new file from data in the source file, in order of record id
- **SSELECT** Create a select list of records meeting specified criteria in order of record id
- **SEARCH** Create a select list of records meeting specified criteria which include text matching over the entire record
- **COUNT** Count records meeting specified criteria
- **SUM** Report total of named fields

**The General Form of a Query Processor Verb**

All query processor verbs follow the same general format though not all parts are applicable to all verbs. The components of the command may be in any order except that the file name must immediately follow the verb and the order may be significant in repeated instances of an element.

```
verb {DICT} file.name
{USING {DICT} file.name}
{field.name [field.qualifier] ...}
{selection.clause}
{sort.clause}
{display.clause}
{record.id...}
```
{FROM select.list.no}  
{TO select.list.no}

where

verb  
is the query processor verb name

{DICT} file.name  
identifies the file to be processed. The optional DICT keyword indicates that the dictionary part of the file is to be processed. The DICT.DICT file will be used as the dictionary defining the items in the dictionary being reported.

USING {DICT} file.name  
indicates that a dictionary other than the one normally associated with the file is to be used.

field.name  
is the name of a field (D or I-type) to be displayed. Multiple fields may be specified in a single command. In addition, the special construct F\text{n} or A\text{n} may be used to reference field n and the EVAL keyword may be used to introduce an evaluated expression. Similarly, the ELEMENT keyword may be used to reference a data collection element by element path.

field.qualifier  
provides qualifying information about the immediately preceding field.name such as the format in which it is to be displayed.

selection.clause  
specifies criteria determining which records from the file are to be included.

sort.clause  
specifies the order in which records are to be processed.

display.clause  
controls the manner in which data is displayed or printed.

record.id  
specifies a particular record id is to be processed. Multiple record ids may be specified.

FROM select.list.no  
specifies that a select list is to be used to control which records are processed. If the FROM option is not used and the default select list (list 0) is active, this list will be used automatically. If used in conjunction with one or more record.id items, only records that appear in the select list and are named record.ids will be processed.

TO select.list.no  
for verbs that produce a select list, specifies which list is to be created. If the TO option is not used, the default select list (list 0) is used.

During processing of a query sentence, each word and symbol on the command line is looked up first in the dictionary of the file being processed and then, if not found there, in the VOC file. Quoted items are always treated as literal values.

Phrases defined in the VOC or the dictionary may be included at any point in a query processor command and will be expanded at that position in the command line.

Literal values used in selection or sort clauses do not need to be enclosed in quotes unless they correspond to names defined in either the VOC or the file’s dictionary or if the contain spaces, commas or quotes. Use of quotes is recommended to prevent incorrect interpretation of commands.
The $QUERY.DEFAULTS Record

The default actions of the query processor can be controlled by adding an X-type record to the VOC file or to the dictionary of the file referenced by the query command.

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Field 1</td>
<td>X</td>
</tr>
<tr>
<td>Field 2</td>
<td>Query processor command line elements that will be inserted into a LIST, SORT, LIST.LABEL, or SORT.LABEL command after the file name but before any further command line options.</td>
</tr>
<tr>
<td>Field 3</td>
<td>Query processor command line elements that will be inserted into a SELECT, SSELECT or SEARCH command.</td>
</tr>
<tr>
<td>Field 4</td>
<td>Format code to be applied to the automatically inserted column for the record if there is no @ID definition in the dictionary. If not present, &quot;12L&quot; is used by default. Only simple width and justification format codes are allowed.</td>
</tr>
</tbody>
</table>

In all cases, the options may extend over multiple lines by use of an underscore as the last character of the line to indicate that a continuation line is present. The lines are merged together with the underscore replaced by a single space. The field numbers described above are applied to the record after merging continuation lines.

The query processor checks first for this optional record in the dictionary of the file. If it is not found or there is no dictionary, it then looks in the VOC. It is therefore possible to use a VOC record to set account level defaults which can be overridden by an alternative record in individual dictionaries. A $QUERY.DEFAULTS record in the dictionary with a type code of X but no further content will effectively disable use of the VOC $QUERY.DEFAULTS record whilst not applying any defaults of its own.

Links

Dictionary L-type records can be used to represent a relationship between two files without the need to include a separate I-type TRANS() expression for each field.

In a query command, a link is used by specifying a field name that is constructed from the link name and the name of a field in the linked file, separated by a percent sign (%).

For example, consider a library application where the BOOKS file representing a physical copy of a book uses a composite key constructed from the id of a record in the TITLES file and the copy number, separated by a hyphen. A link record could be placed in the dictionary of the BOOKS file:

```
TITLES 1: L
2: @ID['-', 1, 1]
3: TITLES
```

A query against the BOOKS file may then refer to fields from the TITLES file as, for example, TITLES%AUTHOR. The linked field (AUTHOR in this example) may be an A, C, D, I or S-type item.

To allow use of field names that contain % characters, the query processor only interprets a field name containing a % character as a link if there is no dictionary or VOC item corresponding to the entire name.

Data items can also be referenced in I-type expressions via links. For example,

```
OCONV(TITLES%AUTHOR, "MCT")
```
Report Styles

A style definition sets visual attributes that highlight specific parts of a report such as headings, subtotals and totals. When the report is directed to the screen, the visual attribute settings enable use of colour. For reports sent to a PCL printer, font weight can be set in style definition.

The style definition appears in an X-type record in either the dictionary or the VOC. Details of how the style definition operates can be found with the description of the STYLE query processor keyword.
5.1 The Selection Clause

A selection clause may be provided to specify criteria governing which records are processed by the command. If omitted, all records are processed. Selection clauses can be used with all query processor verbs.

The selection clause is described in detail under the `WITH` and `WHEN` keywords. The records to be processed by a query can also be specified by use of a select list. The `FROM` keyword can be used to specify the list to be used. If this is not present and the default list (list zero) is active, this is used automatically.

The performance of queries against large files can be improved dramatically by use of alternate key indices. These are index files that relate a particular value of a data field or virtual attribute to the ids of records that have that value. Alternate key indices are created in a two step operation using the `CREATE.INDEX` and `BUILD.INDEX` commands. Once an index has been built, it is maintained automatically by QM and is used by the query processor whenever it is advantageous to do so.

Ranges of values can also be satisfied using indices if the upper and lower limits are defined in the first two conditional elements. For example,

```
LIST STOCK WITH QOH > 3 AND QOH < 10 AND SUPPLIER = 27
```

will use an index on the QOH field to access the data whereas

```
LIST STOCK WITH QOH > 3 AND SUPPLIER = 27 AND QOH < 10
```

will not. Use of unnecessary brackets may also defeat the indexing system. For example,

```
LIST STOCK WITH QOH > 3 AND (QOH < 10 AND SUPPLIER = 27)
```

will not use the index as the second conditional element is not a simple item.

Selection clause comparisons are case sensitive by default. Case insensitivity can be applied by including the `NO.CASE` qualifier after the relational operator or by use of the `QUERY.NO.CASE` mode of the `OPTION` command. See Alternate Key Indices for a discussion of why a case sensitive index cannot be used to resolve a case insensitive selection clause element or vice versa.

The `PICK.IMPLIED.EQ` mode of the `OPTION` command can be used to select Pick style behaviour where a field name followed by a literal value enclosed in double quotes has an implied equals operator. Thus

```
LIST CLIENTS WITH CUST.NO "1234" "5678"
```

is equivalent to

```
LIST CLIENTS WITH CUST.NO = "1234" "5678"
```

Without this option, the semantics of the query are such that the two literal values are treated as record ids and the selection element restricts processing to records in which the CUST.NO field is not empty.

See also:
Qualified display clauses
5.2 The Sort Clause

The optional sort clause determines the order in which records are inserted into the select list (SELECT, SSELECT, SEARCH) or reported (LIST, SORT, LIST.ITEM, SORT.ITEM).

Sorting is performed before conversion of data to its external format. Thus sorts of date fields, for example, will correctly sequence dates regardless of their conversion.

The justification mode of the field's format is used to determine whether a left or right aligned sort is performed. For dates, a right aligned sort is required to avoid problems with dates with internal values of differing numbers of digits.

There are two sort clause operators for single valued fields, BY and BY.DSND, which differ only in that BY sorts into ascending order and BY.DSND sorts into descending order. Similarly, there are two exploded sort operators, BY.EXP and BY.EXP.DSND, for use with multivalued fields. These explode the multivalued records to their single valued equivalents, allowing a query to process values in sequence.

Where multiple sort items are specified, the query processor examines them in the order in which they appear in the command. The second and subsequent sort items are only examined where the previous sort was not sufficient to identify the record sequence.

The SORT, SORT.ITEM and SSELECT verbs are equivalent to the LIST, LIST.ITEM and SELECT verbs with a BY @ID clause as the final sort.

The Sort Algorithm

The sorting process compares items to establish their correct position in sorted order.

When using a left aligned sort, items are compared character by character from the left hand end until a difference is encountered. The relative sort position of the two items is then determined by the collating sequence order of the characters that differ.

When using a right aligned sort, the comparison process is considerably more complex. Each item is considered to be formed from a series of elements that may be numeric or non-numeric. The numeric elements are compared as numeric values, including allowing for a leading sign character on the first element only. The non-numeric elements are compared character by character, left to right as for a left aligned sort. Where the first element of one item is numeric and the first element of the other is non-numeric, the numeric one is positioned earlier in the sorted result.

Although right aligned sorts are most commonly used with data that is entirely numeric, the above process ensures that it operates correctly with mixed data types, producing a logical and consistent sorted result. The following example shows the correct sort order for the same data using both left and right aligned sort modes.

Left:  +3, -6, 103, 10A, 1943, 1A1, 1B1, 7CX, A1A, A1C, AA, BD24, BD7, BF20, XX90
Right: -6, 1A1, 1B1, +3, 7CX, 10A, 103, 1943, A1A, A1C, AA, BD7, BD24, BF20, XX90
5.3 The Display Clause

The optional display clause determines which fields (columns) are reported and how they are displayed. This clause is applicable to the LIST and SORT verbs only. If omitted, the query processor uses the default listing phrase to determine what is shown.

There are a wide variety of options in this clause. Some determine the actual layout of the data while others set breakpoints at which totals, averages, etc are to be reported.

Fields appear in the report left to right in the order of the display clause elements. The default view of the record id (@ID) is always shown as the leftmost column unless it is suppressed using the ID.SUP keyword. If there is no @ID dictionary item, the default action is to show this as 12 characters, left justified but this can be amended using the QUERY.DEFAULTS record.

The display clause is constructed from the elements in the table below.

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Data Item</th>
<th>Suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG</td>
<td>D-type item</td>
<td>CONV &quot;code&quot;</td>
</tr>
<tr>
<td>BREAK.ON [&quot;text&quot;]</td>
<td>I-type item</td>
<td>FMT &quot;spec&quot;</td>
</tr>
<tr>
<td>BREAK.SUP [&quot;text&quot;]</td>
<td>A/S-type item</td>
<td>COL.HDG &quot;text&quot;</td>
</tr>
<tr>
<td>CALC</td>
<td>EVAL &quot;expr&quot; [AS xx]</td>
<td>ASSOC &quot;name&quot;</td>
</tr>
<tr>
<td>CUMULATIVE</td>
<td>Fn</td>
<td>ASSOC.WITH field</td>
</tr>
<tr>
<td>ENUM</td>
<td>An</td>
<td>DISPLAY.LIKE field</td>
</tr>
<tr>
<td>MAX</td>
<td></td>
<td>SINGLE.VALUE</td>
</tr>
<tr>
<td>MEDIAN</td>
<td></td>
<td>MULTI.VALUE</td>
</tr>
<tr>
<td>MIN</td>
<td></td>
<td>NO.NULLS</td>
</tr>
<tr>
<td>MODE</td>
<td></td>
<td>RPT</td>
</tr>
<tr>
<td>PCT [n]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>RANGE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Each data item may optionally be prefixed by one of the qualifiers in the first column and followed by any number of compatible options from the third column.

Where no such item is defined in the dictionary or the VOC, the Fn or An data item is recognised by the query processor as a reference to field n, treating the data as single valued with a default format code of "15T". These display characteristics can be modified using other elements from the table above.
Qualified Display Clauses

For improved compatibility with other multivalue databases, QM supports the concept of qualified display clauses. These combine the role of the display clause with simple selection clause elements. Because qualified display clauses lead to a potential ambiguity in the interpretation of a query, this feature must be enabled using the QUALIFIED.DISPLAY mode of the OPTION command.

A qualified display clause element inserts a conditional test after the data item but before any items from the third column of the table above. This conditional test consists of an operator and a field or value against which the test is to be performed. It may not include the AND or OR operator or the use of brackets.

For example, the query

```
LIST STOCK SUPPLIER = 14 DESCRIPTION
```

would list the record id (default), SUPPLIER and DESCRIPTION fields, showing only those record where the SUPPLIER field contains 14. This is equivalent to

```
LIST STOCK SUPPLIER DESCRIPTION WITH SUPPLIER = 14
```

Qualified display clauses can also use Pick style wildcards which are much less powerful than the LIKE operator. For details, see the EQ operator.

The Default Listing Phrase

If a query sentence contains no display clause, the query processor looks in the dictionary for a PH-type (phrase) entry named @. If this is found, it is attached to the end of the query sentence. Typically, this phrase contains a default list of fields to be shown but it may also include other query sentence elements. If there is no @ phrase, only the record id will be shown.

Where a report is directed to a printer by using the LPTR keyword, the query processor's search for a default listing phrase is extended by first looking for a phrase named @LPTR. If this is not found, the query processor uses the @ phrase as for reports directed to the screen. This extra stage allows users to set up a different set of default fields for the printer and the screen, usually because printers tend to be wider than the screen and can therefore fit more data.
5.4 SCAN

The SCAN verb prompts for entry of one or more text strings. Records that meet any other selection criteria given on the command line are tested for the presence of the search strings at any position in the record, displaying the lines that contain the specified strings.

```
SCAN {DICT} file.name
  {USING {DICT} file.name}
  {selection.clause}
  {sort.clause}
  {record.id...}
  {ALL}
  {FOR string1 string2 ...}
  {FROM select.list.no}
  {STRINGS file.name record.id}
```

The SCAN command is equivalent to SEARCH with the DISPLAY option.

Multiple search strings may be specified. The default action is to prompt for entry of search strings, terminating the list by entering a blank response to the prompt. Two alternative methods of specifying the search strings are available. Use of the FOR keyword allows search strings to be specified on the command line. Use of the STRINGS keyword takes the search strings from the specified file and record, each field being taken as a separate search string.

The default action is for the search to scan only the record data. Use of the ALL keyword extends the search to include the record id.

The optional NO.CASE keyword makes the search string test case insensitive.

By default, the SCAN command shows the lines that contain any of the supplied search strings. This can be changed by use of the ALL.MATCH keyword which specifies that the selected records must contain all of the supplied strings.

**Example**

```
SCAN BP FOR "!SORT"
```

The above command would search all QMBasic source programs in the BP file for references to the !SORT() subroutine and display the lines on which they occur.

**See also:**
SEARCH
5.5 **SELECT and SSELECT**

The **SELECT** and **SSELECT** verbs build a select list containing the keys of records meeting specified criteria. **SSELECT** is equivalent to **SELECT** with a final sort by record id.

**SELECT**

```plaintext
{DICT} file.name
{USING {DICT} file.name}
{selection.clause}
{sort.clause}
{record.id...}
{FROM select.list.no}
{SAVING {UNIQUE} {MULTI.VALUE} field.name {NO.NULLS}}
{TO select.list.no}
```

**SSELECT**

```plaintext
{DICT} file.name
{USING {DICT} file.name}
{selection.clause}
{sort.clause}
{record.id...}
{FROM select.list.no}
{SAVING {UNIQUE} {MULTI.VALUE} field.name {NO.NULLS}}
{TO select.list.no}
```

**Example**

```
SELECT VOC WITH F1 LIKE F...
```

This command builds a select list containing the ids of VOC records with field one starting with an upper case F. Such a list corresponds to all files defined by the VOC.

The list of record ids (or other data when using **SAVING**) is stored in select list 0 unless the **TO** clause is used to specify a different list.

**Chained Selects**

For compatibility with other multivalue products, if the CHAINED.SELECT mode of the **OPTION** command is active, a **GET.LIST**, **SELECT**, **SSELECT** or **QSELECT** that returns a list with at least one entry will examine the **DATA** queue. If there is anything in this queue, the first queue item is read and executed as a command. This allows applications to create sequences of commands that will be executed if the **SELECT** is successful. For example, a QMBasic program might contain

```
DATA 'LIST STOCK'
EXECUTE 'SELECT SALES SAVING MULTIVALUED PART'
```

or a paragraph might perform the same action with

```
PA
SELECT SALES SAVING MULTIVALUED PART
DATA LIST STOCK
```

Note that **DATA** statements always follow the command to which they apply in a paragraph.

This mechanism originated in Pick style databases where a Proc might contain

```
PQ
STON
HLIST STOCK
```
STOFF
HSELECT SALES SAVING MULTIVALUED PART
P
or, resequencing the commands,

PQ
HSELECT SALES SAVING MULTIVALUED PART
STON
HLIST STOCK
P

The chaining process can continue over multiple selections, taking one command at a time from the data queue:

PA
SELECT VOUCHERS SAVING SALE
DATA SELECT SALES SAVING UNIQUE MULTIVALUED PART
DATA LIST STOCK
5.6 SEARCH

The SEARCH verb is similar to the SELECT except that it also prompts for entry of one or more text strings. Records that meet any other selection criteria given on the command line are tested for the presence of the search strings at any position in the record.

```
SEARCH {DICT} file.name
   {USING {DICT} file.name}
   {selection.clause}
   {sort.clause}
   {record.id...}
   {ALL}
   {FOR string1 string2 ...}
   {NO.CASE}
   {FROM select.list.no}
   {STRINGS file.name record.id}
   {SAVING [UNIQUE] [MULTI.VALUE] field.name [NO.NULLS]}
   {DISPLAY}
   {TO select.list.no}
```

Multiple search strings may be specified. The default action is to prompt for entry of search strings, terminating the list by entering a blank response to the prompt. Two alternative methods of specifying the search strings are available. Use of the FOR keyword allows search strings to be specified on the command line. Use of the STRINGS keyword takes the search strings from the specified file and record, each field being taken as a separate search string.

The default action is for the search to scan only the record data. Use of the ALL keyword extends the search to include the record id.

The optional NO.CASE keyword makes the search string test case insensitive.

By default, the SEARCH verb builds a list of all records that contain any of the supplied search strings. This can be changed by use of the ALL.MATCH or NO.MATCH keywords. The ALL.MATCH keyword specifies that the selected records must contain all of the supplied strings. The NO.MATCH keyword specifies that the selected records must not contain any of the supplied strings.

Use of the DISPLAY option causes the command to show the line number and content of each line that contains any of the search strings. No select list will be generated unless the TO option is also used. The DISPLAY option cannot be used with NO.MATCH. The SCAN command is equivalent to SEARCH with the DISPLAY option.

All other options of the SELECT verb may be used in SEARCH.

Examples

```
SEARCH BP
String: !SORT
```

The above lines entered at the keyboard would search all QMBasic source programs in the BP file for references to the !SORT() subroutine and build a select list of these records.

```
SEARCH BP DISPLAY FOR "!SORT"
```

The above command would search all QMBasic source programs in the BP file for references to the !SORT() subroutine and display the lines on which they occur.
See also:

ALL.MATCH, NO.CASE, NO.MATCH, SCAN
5.7 LIST and SORT

The LIST and SORT verbs produce reports from QM files. The LIST verb displays records in the order in which they are encountered in the file unless a sort clause is present in the command. The SORT verb is equivalent to LIST with a final sort by record id.

```
LIST {DICT} file.name
{USING {DICT} file.name}
{field.name [field.qualifier] ...}
{selection.clause}
{sort.clause}
{display.clause}
{record.id...}
{FROM select.list.no}
```

```
SORT {DICT} file.name
{USING {DICT} file.name}
{field.name [field.qualifier] ...}
{selection.clause}
{sort.clause}
{display.clause}
{record.id...}
{FROM select.list.no}
```

The record id is always reported as the first item in the output unless the ID.SUP keyword has been used to suppress it. The format of this item is determined by the @ID dictionary record. If this record is not found in the dictionary, a default format is used.

If field names are specified in the command, these fields are displayed in the order specified. If no field names are present, the query processor looks for a phrase in the dictionary defining a default set of fields to be reported. If the LPTR keyword has been included, the query processor first looks for a phrase record named @LPTR. If this cannot be found or the LPTR keyword was not used, it looks for a phrase record named @. The @LPTR and @ phrases can be used to create separate default field name lists for output to the printer and the display respectively. If no @LPTR or @ record is found, only the record id is reported.

The default listing phrase may include field qualifiers, selection, sort and display clause items.

The LIST and SORT verbs normally produce a tabular format report with items listed side by side. If the total width of the items to be reported exceeds the width of the display or printer to which the report is directed, a vertical format report is produced. This can be forced by use of the VERTICALLY keyword.

The PAN keyword allows reports wider than the display to be produced using the left and right cursor keys to pan part or all of the displayed data.

The SCROLL keyword allows scrolling back and forward using the up and down cursor keys.

When LIST or SORT are used to list a dictionary with the no display clause, the action of the default listing phrase (@), includes transformation of A and S-type dictionary items into a form that maps onto the standard dictionary display format used for other types.

Example

```
LIST STOCK QTY REORDER.LEVEL WITH QTY < REORDER.LEVEL
```
This command lists all records from the STOCK file for which the quantity in stock (QTY field) is less than or equal to the reorder level (REORDER.LEVEL field). These two fields are displayed together with the record id.
5.8  LIST.ITEM and SORT.ITEM

The **LIST.ITEM** and **SORT.ITEM** verbs display data from QM files in its internal format. The **LIST.ITEM** verb displays records in the order in which they are encountered in the file unless a sort clause is present in the command. The **SORT.ITEM** verb is equivalent to **LIST.ITEM** with a final sort by record id.

```
LIST.ITEM {DICT} file.name
    {USING {DICT} file.name}
    {selection.clause}
    {sort.clause}
    {record.id...}
    FROM select.list.no
```

```
SORT.ITEM {DICT} file.name
    {USING {DICT} file.name}
    {selection.clause}
    {sort.clause}
    {record.id...}
    FROM select.list.no
```

The output from these verbs displays the record id followed by each field on a separate line. No conversion or formatting is performed on the data. Multivalued data will be displayed with embedded mark characters.

The display normally prefixes each field with its field number. Either the **COL.SUP** or **NUM.SUP** keywords can be used to suppress this.

The **ID.SUP** keyword can be used to suppress display of the record id.

The **SCROLL** keyword allows scrolling back and forward using the up and down cursor keys.

**Example**

```
LIST.ITEM STOCK 16798
```

This command lists the content of record 16798 of the STOCK file.
5.9 LIST.LABEL and SORT.LABEL

The LIST.LABEL and SORT.LABEL verbs are used to print address labels from QM files. The LIST.LABEL verb processes records in the order in which they are encountered in the file unless a sort clause is present in the command. The SORT.LABEL verb is equivalent to LIST.LABEL with a final sort by record id.

**LIST.LABEL**

```
LIST.LABEL [DICT] file.name
 {USING [DICT] file.name}
 {field.name {field.qualifier} ...}
 {selection.clause}
 {sort.clause}
 {display.clause}
 {record.id...}
 FROM select.list.no
```

**SORT.LABEL**

```
SORT.LABEL [DICT] file.name
 {USING [DICT] file.name}
 {field.name {field.qualifier} ...}
 {selection.clause}
 {sort.clause}
 {display.clause}
 {record.id...}
 FROM select.list.no
```

The optional clauses to LIST.LABEL and SORT.LABEL work in exactly the same way as for LIST and SORT except that arithmetic field modifiers (AVERAGE, ENUMERATE, MAX, MIN, PERCENTAGE, TOTAL), breakpoints (BREAK.ON, BREAK.SUP) and the page format keywords (COL.SUP, COL.HDR.SUPP, COL.SPACES, HDR.SUP, DBL.SPC, FOOTING, GRAND.TOTAL, HEADING, PAN, SCROLL, VERTICALLY) are not allowed.

The LIST.LABEL and SORT.LABEL commands produce a vertical style report set out into the positions of labels on the printed page. The label page shape may be defined by a record in the dictionary of the file or in the VOC file, or it may be entered in response to prompts.

If the command includes the LABEL keyword, this may be followed by the name of an X-type label template record stored in the dictionary or in the VOC. This record contains the page shape as a series of lines. Only the leading numeric part of each line is used thus allowing comments to be inserted explaining what each number represents. A typical label definition might read:

```
1: X
2: 2  Count of labels across the page
3: 8  Count of labels per column
4: 42 Characters per line on each label
5: 7  Lines per label
6: 0  Indentation to first column of leftmost label
7: 6  Horizontal space between labels
8: 3  Lines between labels
9: 1  Omit blanks
```

The final field determines whether blank lines within a label should be omitted. It should be set to 1 to omit or 0 to include such lines.

If no LABEL keyword is present in the command line (or any phrase use by the query), the query processor looks for a default label template stored in a record named @LABEL in the dictionary of
the file or in the VOC. This action can be suppressed by use of \texttt{LABEL NO.DEFAULT} in the command.

If no label template has been specified and either there is no \texttt{@LABEL} record or the \texttt{NO.DEFAULT} keyword has been used, the query processor will prompt the user to enter the label shape parameters in the same order as above. The omit blanks option must be entered as Y or N. The items may be entered in response to each prompt in turn or as a comma separated list.

It may be necessary to check that a printer font is chosen where the line spacing fits correctly onto the label page. The \texttt{SETPTR} command top margin may also need to be set to fit the page.

\textbf{Example}

\begin{verbatim}
LIST.LABEL CUSTOMERS NAME ADDRESS ID.SUP LABEL ADDR.LABELS LPTR
\end{verbatim}

This command prints address labels from all records in the CUSTOMERS file. Each label contains data from the NAME and ADDRESS fields. A label template named ADDR.LABELS is used.
5.10 REFORMAT and SREFORMAT

The **REFORMAT** verb constructs a new file from data in the source file. It is of use, for example, when constructing intermediate files in complex reporting processes.

```
REFORMAT {DICT} file.name
    {USING {DICT} file.name}
    {field.name [field.qualifier] ...}
    {selection.clause}
    {sort.clause}
    {display.clause}
    {record.id...}
    {FROM select.list.no}
    {TO new.file.name | TO DATA}
```

**REFORMAT** behaves like **LIST** except that the data identified by the display clause is used to populate a new file instead of being displayed or printed. The first item in the data is used as the record id for the item in the new file. The remaining items form the fields within the record. **REFORMAT** does not automatically prefix the display clause with @ID.

The **SREFORMAT** command is identical to **REFORMAT** except that it sorts by record id after any other sorting has been applied.

The **TO** clause can be used to name the target file on the command line. If this clause is not present, a prompt is displayed for the file name. The file must already exist.

Use of **TO DATA** causes the query processor to create a dynamic array containing the output and placing this in the **@DATA** variable. Each "record" in this array is separated by an item mark character (char 255) and the fields within the item correspond to the display clause elements in the **REFORMAT** command.

Use of **@RECORD** as the final element of the display clause appends the entire data record with no conversion or formatting applied. This is most useful with **TO DATA** to construct views of the data record for further processing.

When used with breakpoints, data from the detail and total lines is directed to the output file but the grand total is omitted.

**Example**

```
REFORMAT CUSTOMERS ZIP.CODE CUST.NO NAME TO CUST.BY.ZIP
```

This command constructs a new file, **CUST.BY.ZIP**, keyed by zip code and containing two data fields, the customer number and name. Note that if two or more customers share the same zip code, the record will be overwritten by the second and subsequent items.
5.11 COUNT

The **COUNT** verb reports the number of records meeting specified criteria.

```
COUNT {DICT} file.name
   {USING {DICT} file.name}
   {selection.clause}
   {record.id...}
   {FROM select.list.no}
```

**Example**

```
COUNT INVOICES WITH NO PAYMENT.DATE
```

This command counts records on the INVOICES file for which the PAYMENT.DATE field is null. This might be a valid means of identifying unpaid invoices.
5.12 SUM

The SUM verb reports the total of the values in named fields.

```
SUM {DICT} file.name
   {USING {DICT} file.name}
   field.name {field.qualifier} ...
   {selection.clause}
   {record.id...}
   {FROM select.list.no}
```

Example

```
SUM INVOICES BALANCE WITH NO PAYMENT.DATE
```

This command the BALANCE field of records on the INVOICES file for which the PAYMENT.DATE field is null.
5.13 **SHOW**

The **SHOW** command provides an interactive means of building select lists.

**Format**

```
SHOW [DICT] file.name
   {USING [DICT] file.name}
   {field.name [field.qualifier] ...}
   {selection.clause}
   {sort.clause}
   {display.clause}
   {record.id...}
   {FROM select.list.no}
   {TO select.list.no}
```

The **SHOW** command supports two special options that limit the number of selected items:

- **MAX** \( n \) specifies the maximum number of items that may be selected for the returned list.
- **MIN** \( n \) specifies the minimum number of items that may be selected for the returned list. Returning no items is always valid regardless of the value of \( n \).

Also, use of the **SORT.SUP** keyword suppresses the default sort by record id when no sort clause is present. This is useful if a sorted select list is being used to control the records to be processed.

**Examples**

```
SHOW BP
```

Displays a list of records in the BP file from which items may be chosen to build a select list.

```
SHOW CLIENTS COMPANY EVAL "BALANCE - CREDIT" ID.SUP WITH BALANCE > CREDIT
```

For each client in the CLIENTS file with an outstanding balance greater than their credit limit, display the company name and the calculated amount by which the client has exceeded their credit limit. Display of the CLIENTS file record id is suppressed. The result of the **SHOW** operation becomes the default select list (list 0).

```
SHOW STOCK QTY REORDER.LEVEL TO 3
```

Displays the id, quantity and reorder level fields of each item in the STOCK file. The result of the **SHOW** operation is saved in select list 3.

```
SHOW CLIENTS COMPANY FMT "30T" LAST.CALL CONV "D2/"
```

Shows a list of CLIENTS file ids, the company name formatted to fit a 30 character wide field and the date on which the client was last called using the D2/ conversion for this date.
Using the SHOW Command

The SHOW command displays a list of records from the file being processed. This display consists of:

- A page heading which may be omitted using the HDR.SUP keyword. A default page heading is used unless specifically set by use of the HEADING keyword. The SHOW command does not support use of embedded control codes in page headings.

- Column headings which may be omitted using the COL.SUP keyword. The heading is taken from the display name field of the dictionary entry for the item in the column. If blank, the field name is used.

- Data from records being processed. The items displayed are the record id (unless the ID.SUP keyword has been used) and other fields named on the command line.

  If the total width of the named fields exceeds the available space, SHOW will drop trailing fields until the data fits the display width or only one (plus the record id, if not suppressed) remains. If the data still does not fit after dropping fields, the remaining fields are displayed in reduced space.

  The SHOW command splits multivalued items onto successive lines and correctly relates values and subvalues in associated fields. A record is never split between two pages, a new page being started if necessary. If a single record requires more lines than will fit on a screen page, it is truncated.

  Each item on the page is numbered for reference in the commands that manipulate the list. The number starts at one for the first item on each page.

- A status line showing the number of selected records.

- An input line on which commands are entered.

- An error line on which error messages and help text appears.

Using the commands listed below, the user can scroll through the displayed records setting or clearing a marker (displayed as an asterisk next to the record number) which indicates whether the record is to be included in the generated select list.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Move to the top of the list (first page).</td>
</tr>
<tr>
<td>B</td>
<td>Move to the bottom of the list (last page).</td>
</tr>
<tr>
<td>N</td>
<td>Move to the next page. The return key with no command text has the same effect.</td>
</tr>
<tr>
<td>P</td>
<td>Move to the previous page.</td>
</tr>
<tr>
<td>Q</td>
<td>Quit from record selection. Any records marked with an asterisk are entered into the new select list.</td>
</tr>
<tr>
<td>QC</td>
<td>Quit, clearing any record selection.</td>
</tr>
<tr>
<td>R</td>
<td>Redisplay the screen. This is useful if a data transmission error causes screen corruption.</td>
</tr>
<tr>
<td>^^</td>
<td>Synonym for R.</td>
</tr>
<tr>
<td>?</td>
<td>Display help text. Press any key to return to the main screen.</td>
</tr>
</tbody>
</table>
**S item** Select *item*. The space before the item description is optional. An asterisk will be displayed next to all selected items.

**C item** Clear *item*, removing the asterisk marker from the screen.

*item* Synonym for **S item**.

The *item* specification may be

The number shown next to a displayed record.

A range of numbers in the form *a*-*b* which indicates that the command is to be applied to all items from that tagged with number *a* to that tagged with number *b*. There must be no spaces either side of the hyphen.

The keyword **VISIBLE** to apply the command to all items on the current page.

The keyword **ALL** to apply the command to all items in the list.

Multiple *item* specifications may be included in a single command by using either a space or a comma as a separator. For example "1,4,8-11".

The **VISIBLE** and **ALL** keywords may be abbreviated by omitting any number of trailing letters (e.g. **VIS** or **V**).
5.14 Query processor keywords

**Field qualifiers**

- `%` Synonym for PERCENTAGE
- `AS` Define synonym for field and qualifiers
- `ASSOC` Include field in association
- `ASSOC.WITH` Associate two fields
- `AVERAGE` Report average of field values
- `AVG` Synonym for AVERAGE
- `BREAK ON` Define field as breakpoint control item.
- `BREAK-ON` Synonym for BREAK.ON
- `BREAK.SUP` Define field as non-displayed breakpoint control item.
- `BREAK-SUP` Synonym for BREAK.SUP
- `CALC` Calculate total of I-type item
- `CALCULATE` Synonym for CALC
- `COL.HDG` Set column heading for displayed item
- `CONV` Specify conversion to be applied to field
- `CUMULATIVE` Report cumulative value of field
- `DISPLAY.LIKE` Display field using attributes of another field
- `DISPLAY.NAME` Synonym for COL.HDG
- `ENUM` Synonym for ENUMERATE
- `ENUMERATE` Count number of values in specified field
- `EVAL` Defines expression to be evaluated
- `FMT` Specify format for display of field
- `MAX` Report maximum value of a field
- `MEDIAN` Report median value
- `MIN` Report minimum value of a field
- `MODE` Report mode value
- `MULTI.VALUE` Treat field as multivalued
- `MULTIVALUED` Synonym for MULTI.VALUE
- `NO NULLS` Suppress null fields in MIN and AVG calculations
- `PCT` Synonym for PERCENTAGE
- `PERCENT` Synonym for PERCENTAGE
- `PERCENTAGE` Report percentages
- `RPT` Repeat single-valued field on each line
- `SINGLE.VALUE` Treat field as single-valued
- `SINGLEVALUED` Synonym for SINGLE.VALUE
- `TOTAL` Report total of field values

**Selection clause options**

- `FIRST` Synonym for SAMPLE
- `FROM` Process only records from given select list
- `REQUIRE SELECT` Query must have an active select list
- `SAMPLE` Report only specified number of records
- `SAMPLED` Report only a sample of records
- `SORTED.FIRST` Synonym for SORTED.SAMPLE
- `SORTED.SAMPLE` Report only a specified number of records after sorting
- `WHEN` Introduces multivalued field selection criteria
- `WITH` Introduces record selection criteria

**SEARCH options**

- `ALL.MATCH` Record must contain all given strings
- `FOR` Specifies search strings on the command line
- `NO.CASE` Case insensitive option for SEARCH
- `NO.MATCH` Record must contain none of the given strings
- `STRINGS` Specifies a file and record containing the search strings
Selection clause operators used in WITH constructs

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#</td>
<td>Not equal. Synonym for NE</td>
</tr>
<tr>
<td>&amp;</td>
<td>Logical AND. Synonym for AND</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than. Synonym for LT</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to. Synonym for LE</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not equal. Synonym for NE</td>
</tr>
<tr>
<td>=&lt;</td>
<td>Less than or equal to. Synonym for LE</td>
</tr>
<tr>
<td>=&gt;</td>
<td>Greater than or equal to. Synonym for GE</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than. Synonym for GT</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Not equal. Synonym for NE</td>
</tr>
<tr>
<td>&lt;=&gt;</td>
<td>Greater than or equal to. Synonym for GE</td>
</tr>
<tr>
<td>~</td>
<td>Soundex matching. Synonym for SAID</td>
</tr>
<tr>
<td>AFTER</td>
<td>Greater than. Synonym for GT</td>
</tr>
<tr>
<td>AND</td>
<td>Logical AND</td>
</tr>
<tr>
<td>BEFORE</td>
<td>Less than. Synonym for LT</td>
</tr>
<tr>
<td>BETWEEN</td>
<td>Closed range test</td>
</tr>
<tr>
<td>EQ</td>
<td>Equals</td>
</tr>
<tr>
<td>EQUAL</td>
<td>Equals. Synonym for EQ</td>
</tr>
<tr>
<td>GE</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>GREATER</td>
<td>Greater than. Synonym for GT</td>
</tr>
<tr>
<td>GT</td>
<td>Greater than</td>
</tr>
<tr>
<td>IN</td>
<td>Item is in named list</td>
</tr>
<tr>
<td>LE</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>LESS</td>
<td>Less than. Synonym for LT</td>
</tr>
<tr>
<td>LIKE</td>
<td>Pattern match</td>
</tr>
<tr>
<td>LT</td>
<td>Less than</td>
</tr>
<tr>
<td>MATCHES</td>
<td>Pattern match. Synonym for LIKE</td>
</tr>
<tr>
<td>MATCHING</td>
<td>Pattern match. Synonym for LIKE</td>
</tr>
<tr>
<td>NE</td>
<td>Not equal</td>
</tr>
<tr>
<td>NO</td>
<td>Test for null field</td>
</tr>
<tr>
<td>NOT</td>
<td>Invert a selection expression. Also a synonym for NE</td>
</tr>
<tr>
<td>NOT IN</td>
<td>Item is not in named list</td>
</tr>
<tr>
<td>OR</td>
<td>Logical OR</td>
</tr>
<tr>
<td>SAID</td>
<td>Soundex matching</td>
</tr>
<tr>
<td>SPOKEN</td>
<td>Soundex matching. Synonym for SAID</td>
</tr>
<tr>
<td>UNLIKE</td>
<td>Inverse pattern match. Opposite of LIKE</td>
</tr>
</tbody>
</table>

Sort clause options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BY</td>
<td>Sort by ascending field value order</td>
</tr>
<tr>
<td>BY.DSND</td>
<td>Sort by descending field value order</td>
</tr>
<tr>
<td>BY-DSND</td>
<td>Synonym for BY.DSND</td>
</tr>
<tr>
<td>BY.EXP</td>
<td>Ascending exploded multivalued sort</td>
</tr>
<tr>
<td>BY-EXP</td>
<td>Synonym for BY.EXP</td>
</tr>
<tr>
<td>BY.EXP.DSND</td>
<td>Descending exploded multivalued sort</td>
</tr>
<tr>
<td>BY-EXP.DSND</td>
<td>Synonym for BY.EXP.DSND</td>
</tr>
</tbody>
</table>

Display options

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BOXED</td>
<td>Generates a boxed report on a PCL printer</td>
</tr>
<tr>
<td>CAPTION</td>
<td>Synonym for GRAND.TOTAL</td>
</tr>
<tr>
<td>COL.HDG.ID</td>
<td>Use field names as default column headings</td>
</tr>
<tr>
<td>COL.HDR.SUPP</td>
<td>Suppress page and column headings</td>
</tr>
<tr>
<td>COL.HDR.SUP</td>
<td>Synonym for COL.HDR.SUP</td>
</tr>
<tr>
<td>COL-HDR.SUPP</td>
<td>Synonym for COL.HDR.SUPP</td>
</tr>
<tr>
<td>COL-HDR.SUP</td>
<td>Synonym for COL.HDR.SUPP</td>
</tr>
<tr>
<td>COL.SPACES</td>
<td>Specifies inter-column spacing</td>
</tr>
<tr>
<td>Option</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>COL.SPCS</td>
<td>Synonym for COL.SPACES</td>
</tr>
<tr>
<td>COL.SUP</td>
<td>Suppress column headings</td>
</tr>
<tr>
<td>COL.SUPP</td>
<td>Synonym for COL.SUP</td>
</tr>
<tr>
<td>COUNTSUP</td>
<td>Suppress record count message at end of report</td>
</tr>
<tr>
<td>CSV</td>
<td>Specifies a comma separated delimited report</td>
</tr>
<tr>
<td>DBL.SPC</td>
<td>Output report with double line spacing</td>
</tr>
<tr>
<td>DBL.SPC</td>
<td>Synonym for DBL.SPC</td>
</tr>
<tr>
<td>DELIMITER</td>
<td>Specifies delimited report</td>
</tr>
<tr>
<td>DET.SUP</td>
<td>Suppress detail lines</td>
</tr>
<tr>
<td>DET.SUPP</td>
<td>Synonym for DET.SUP</td>
</tr>
<tr>
<td>FOOTER</td>
<td>Synonym for FOOTING</td>
</tr>
<tr>
<td>FOOTING</td>
<td>Specify page footing</td>
</tr>
<tr>
<td>FORCE</td>
<td>Force print of headings in empty report</td>
</tr>
<tr>
<td>GRAND.TOTAL</td>
<td>Specify format for totals line</td>
</tr>
<tr>
<td>GRAND.TOTAL</td>
<td>Synonym for GRAND.TOTAL</td>
</tr>
<tr>
<td>HDR.SUP</td>
<td>Suppress default page heading</td>
</tr>
<tr>
<td>HDR.SUPP</td>
<td>Synonym for HDR.SUP</td>
</tr>
<tr>
<td>HEADER</td>
<td>Synonym for HEADING</td>
</tr>
<tr>
<td>HEADING</td>
<td>Specify page heading</td>
</tr>
<tr>
<td>ID.ONLY</td>
<td>Suppress use of @ phrase</td>
</tr>
<tr>
<td>ID.SUP</td>
<td>Suppress default inclusion of @ID in report</td>
</tr>
<tr>
<td>ID-SUPP</td>
<td>Synonym for ID.SUP</td>
</tr>
<tr>
<td>LPTR</td>
<td>Direct report to a printer</td>
</tr>
<tr>
<td>MARGIN</td>
<td>Specify width of left margin</td>
</tr>
<tr>
<td>NEW PAGE</td>
<td>Display or print each record on a separate page</td>
</tr>
<tr>
<td>NO.PAGE</td>
<td>Suppress pause between displayed pages</td>
</tr>
<tr>
<td>NOPAGE</td>
<td>Synonym for NO.PAGE</td>
</tr>
<tr>
<td>NO.SPLIT</td>
<td>Avoid splitting a record across pages if possible</td>
</tr>
<tr>
<td>ONLY</td>
<td>Synonym for ID.ONLY</td>
</tr>
<tr>
<td>OVERLAY</td>
<td>Sets a graphical page overlay</td>
</tr>
<tr>
<td>PAN</td>
<td>Pan columns of a wide report</td>
</tr>
<tr>
<td>REPEATING</td>
<td>Repeats single valued items or the final value of a multivalued item</td>
</tr>
<tr>
<td>SCROLL</td>
<td>Allow scrolling through report pages</td>
</tr>
<tr>
<td>STYLE</td>
<td>Sets a report style</td>
</tr>
<tr>
<td>SUPP</td>
<td>Synonym for HDR.SUP</td>
</tr>
<tr>
<td>VERT</td>
<td>Synonym for VERTICALLY</td>
</tr>
<tr>
<td>VERTICALLY</td>
<td>Report in vertical (one field per line) format</td>
</tr>
<tr>
<td>XML</td>
<td>Report in XML format</td>
</tr>
<tr>
<td>XML</td>
<td>Report in XML format</td>
</tr>
</tbody>
</table>

**SELECT and SSELECT options**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NO.NULLS</td>
<td>Ignore null fields with SAVING option</td>
</tr>
<tr>
<td>SAVING</td>
<td>Save field value in place of record id</td>
</tr>
<tr>
<td>TO</td>
<td>Specifies target select list number</td>
</tr>
<tr>
<td>UNIQUE</td>
<td>Omit duplicates with SAVING</td>
</tr>
</tbody>
</table>

**Miscellaneous**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSENT.IGNORE</td>
<td>Ignore unfound records</td>
</tr>
<tr>
<td>ABSENT.NULL</td>
<td>Treats an absent record as a null item rather than an error</td>
</tr>
<tr>
<td>DEFAULT</td>
<td>Parse the query using default option settings</td>
</tr>
<tr>
<td>LABEL</td>
<td>Specifies the label template for LIST.LABEL and SORT.LABEL</td>
</tr>
<tr>
<td>LOCKING</td>
<td>Takes a file lock on the file being processed, preventing updates</td>
</tr>
<tr>
<td>NO.INDEX</td>
<td>Do not use an alternate key index for record selection</td>
</tr>
<tr>
<td>REPORTING</td>
<td>Display periodic progress information during COUNT or SELECT with selection criteria</td>
</tr>
</tbody>
</table>
**REQUIRE INDEX**  Do not perform the query unless an alternate key index can be used

**TO**  Specifies the output file name for REFORMAT. See also CSV and DELIMITER

**USING**  Use an alternative dictionary
5.15 ABSENT.IGNORE

The **ABSENT.IGNORE** keyword suppresses the message at the end of a query command showing records that have not been found.

**Format**

```
ABSENT.IGNORE
```

The query processor normally builds a list of records that were specified on the command line or via a select list but were not found. This list is displayed on completion of the command. Sometimes it might be valid for the list to contain items that might not be present and should be ignored. In this case the list of unfound records is not required. The **ABSENT.IGNORE** keyword causes the query processor to ignore missing records.

**Example**

```
SELECT VOC
LIST NEWVOC
```

This pair of commands builds a list of all records in the VOC and then uses this list to display a report of the same records from the NEWVOC file. There will be records in the VOC that do not appear NEWVOC and a list of these will be shown on completion of the report.

```
SELECT VOC
LIST NEWVOC ABSENT.IGNORE
```

This pair of commands produces the same report but omits the list of records that were not found in NEWVOC.
5.16 ABSENT.NULL

The **ABSENT.NULL** keyword treats an absent record as a null item rather than an error.

**Format**

```
ABSENT.NULL
```

There are situations where two or more related files share a common (or related) record id but some ids may not appear in all of the files. The **ABSENT.NULL** keyword, normally used with a select list, allows reports to be constructed that draw data from the complete set of files, returning a null record for any item that is not present in the main file processed by the query. A dictionary I-type entry can then be used to retrieve data from the related files.

**Example**

FILE1 contains records 1 and 2. FILE2 contains records 2 and 3.

The dictionary for FILE1 contains an I-type entry, F2REF, that is a simple **TRANS()** using the record id of FILE1 to access the same record in FILE2, returning data from this file.

```
TRANS(FILE2, @ID, FIELD.NAME, 'X')
```

If we have a select list containing 1, 2 and 3,

```
LIST FILE1 F1 F2REF
```

would report records 1 and 2 but give an error for record 3.

Adding the **ABSENT.NULL** option,

```
LIST FILE1 F1 F2REF ABSENT.NULL
```

will process records 1, 2, and 3 from both files, using a null record for the absent records (3 in FILE1 and 1 in FILE2).
5.17 ALL.MATCH

The **ALL.MATCH** keyword used in a **SEARCH** command specifies that the records to be selected must contain all of the given search strings.

**Format**

```
ALL.MATCH
```

Without this keyword, the **SEARCH** command builds a list of records containing any of the supplied search strings. With **ALL.MATCH**, the records must contain all of the supplied strings.

**Example**

```
SEARCH BP ALL.MATCH
String: STOCK.FILE
String: STK.F
String: 
```

This command builds a list of records in the BP file containing both the given strings.

**See also:**

**NO.CASE, NO.MATCH, SEARCH**
5.18 AND

The **AND** selection clause operator links two selection criteria where both must be True for the record to be selected. The synonym **&** can be used.

**Format**

```
WITH condition.1 AND condition.2
```

where

```
condition.1, condition.2
```

are record selection criteria.

The **AND** selection clause operator returns True if both `condition.1` and `condition.2` are True. Alternatively, multiple WITH clauses can be used.

The **AND** and **OR** operators are normally of equal priority and will be evaluated strictly left to right. Brackets may need to be used to enforce evaluation in an different order. Thus a query such as

```
LIST CLIENTS WITH REGION = 1 AND VALUE > 1000 OR REGION = 2 AND VALUE > 500
```

may need brackets to achieve the desired effect

```
LIST CLIENTS WITH (REGION = 1 AND VALUE > 1000) OR (REGION = 2 AND VALUE > 500)
```

Pick style multivalue database products give **AND** priority over **OR** such that the above query would not need the brackets. This behaviour can be enabled in QM by use of the QUERY.PRIORITY.AND mode of the **OPTION** command.

**Example**

```
LIST STOCK WITH QTY GT 100 AND REORDER LT 300
```

This command lists items found on the STOCK file with a QTY field of over 100 and a REORDER field of less than 300.

The same results can be achieved with

```
LIST STOCK WITH QTY GT 100 WITH REORDER LT 300
```
The **AS** keyword is a field qualifier which defines a synonym for a field and any qualifying information.

**Format**

\[ \text{field.name} \{\text{field.qualifier}\} \text{ AS synonym} \]

where

- **field.name** is the name of the field (D or I-type) or an evaluated expression to be given a synonym.
- **field.qualifier** is any qualifying information such as **CONV** or **FMT** keywords.
- **synonym** is the name by which the field and its qualifiers is to be known. This name must not correspond to an existing entry in the file's dictionary or in the VOC.

The **AS** keyword creates a synonym for **field.name** together with any **field.qualifier**. It is normally only used in conjunction with a **field.qualifier** or to name an evaluated expression.

**Example**

```
LIST INVOICES EVAL "SUM(RECEIVED)" AS AMT BY.DSND AMT
```

This command processes records from the INVOICES file and reports the total of the multivalued RECEIVED field. The report is displayed in descending order of total received amount. The **AS** keyword is used to give the synonym AMT to the evaluated sum so that it can be referred to again in the sort clause after the **BY.DSND** keyword without full expansion.

**See also:**

**EVAL**
5.20 ASSOC

The ASSOC keyword is a field qualifier to specify that the field is to be treated as part of a named association.

Format

\[ field.name \ ASSOC "name" \]

where

\[ field.name \]

is the name of the field (D or I-type) or an evaluated expression to be associated.

\[ name \]

is the name of the association in which \( field.name \) is to be included. The name must be quoted to avoid its expansion as a phrase.

The ASSOC keyword causes \( field.name \) to be treated as part of the named association.

Example

\[ \text{LIST ORDERS PART.NO QTY EVAL "SELL * QTY" ASSOC LINE.ITEMS} \]

This command processes records from the ORDERS file and reports the multivalued part numbers, quantities and calculated line total value for each record. The evaluated expression is treated as a member of the association LINE.ITEMS to which the other reported fields already belong.

See also:
ASSOC.WITH
5.21 ASSOC.WITH

The ASSOC.WITH keyword is a field qualifier to specify that the field is to be associated with some other named field.

Format

\[ field.name \text{ ASSOC.WITH } name \]

where

\[ field.name \]

is the name of the field (D or I-type) or an evaluated expression to be associated.

\[ name \]

is the name of the field with which \( field.name \) is to be associated.

The ASSOC.WITH keyword causes \( field.name \) to be treated as associated with field \( name \).

Example

\[ \text{LIST ORDERS PART.NO QTY EVAL } "\text{SELL } \ast \text{ QTY}" \text{ ASSOC.WITH PART.NO} \]

This command processes records from the ORDERS file and reports the multivalued part numbers, quantities and calculated line total value for each record. The evaluated expression is treated as associated with PART.NO.

See also:

ASSOC
5.22 AVERAGE

The AVERAGE field qualifier keyword causes a field to be reported together with its mean average value. The synonym AVG may be used.

Format

\[ \text{AVERAGE} \text{field} \{\text{field.qualifiers}\} \{\text{NO.NULLS}\} \]

where

- \text{field} is the field or evaluated expression to be displayed.
- \text{field.qualifiers} are other field qualifying keywords
- \text{NO.NULLS} causes null values to be ignored.

The AVERAGE field qualifier keyword is placed before the field name to which it applies and causes the query processor to report the value of the field for each record processed and also to report the mean average value at the end of the report. Used with breakpoints, the AVERAGE keyword will also report the average value of the field at each breakpoint.

If the field is defined as multivalued, the AVERAGE keyword operates on each value in turn.

The AVERAGE keyword operates only on numeric data. Non-numeric values are ignored except that the default behaviour is to treat null items as zero. The NO.NULLS keyword can be used to prevent null values being included in the calculation of the average value.

Example

The sentence

\[ \text{LIST INVOICES AVERAGE VALUE CUSTOMER.NAME WITH NO AMT.PAID} \]

would produce a display such as that below in which the average value of the VALUE field is included at the end of the report.

\[
\begin{array}{lll}
\text{LIST INVOICES AVERAGE VALUE CUSTOMER.NAME WITH NO AMT.PAID} \\
\text{Invoice} & \text{...Value} & \text{Customer.}................. \\
74529 & £1712.43 & \text{J McTavish} \\
74273 & £95.23 & \text{County Newspapers} \\
63940 & £141.00 & \text{R Bryant} \\
74993 & £9.29 & \text{Write Right Stationery} \\
\text{========} & \text{=} & \text{£489.49} \\
4 \text{ records listed.} \\
\end{array}
\]

See also:
MEDIAN, MODE, RANGE
5.23 BETWEEN

The BETWEEN selection clause operator compares a field or evaluated expression against two other fields, evaluated expressions or literal values, testing whether the value of the first item lies between the other two values.

**Format**

```
field BETWEEN {NO.CASE} value1 value2
```

where

- **field** is the first field or evaluated expression to be compared.
- **value1** is the low end of the range of values to be selected.
- **value2** is the high end of the range of values to be selected.

The BETWEEN selection clause operator returns True if *field* is greater than or equal to *value1* and less than or equal to *value2*. The optional NO.CASE qualifier causes a case insensitive comparison to be applied.

When applied to multivalued fields, the test is applied to each value in turn. Note that the query

```
LIST ORDERS WITH PART.NO BETWEEN 200 299
```

is not the same as

```
LIST ORDERS WITH PART.NO >= 200 AND PART.NO <= 299
```

if PART.NO is a multivalued field. The first query selects only those records which include part numbers in the range 200 to 299. The second query selects those records which include a part number that is greater than or equal to 200 and another part number that is less than or equal to 299.
5.24 BOXED

The **BOXED** display clause option causes the query processor to generate a boxed report. This option is ignored if report destination is not a printer or file set in PCL mode.

**Format**

```
BOXED
```

The **BOXED** option draws a box around the page border as defined by the page width and depth specified in the **SETPTR** command. The effective width of the page is reduced by two characters to ensure that there is a margin between the text and sides of the box.

If the report includes a page heading or footing, these are separated from the body of the report by a horizontal line.

**Note:** The quality of PCL implementations varies widely and this option may not give the expected results on some printers. It is the application developer’s responsibility to ensure that the printed results are acceptable.
The **BREAK.ON** field qualifier keyword causes the query processor to display the named field, generating a breakpoint whenever the field value changes.

**Format**

```
BREAK.ON { "options" } field
```

where

- **options** controls the appearance of the breakpoint.
- **field** is the field name or evaluated expression to be reported.

The **BREAK.ON** keyword appears before the field name and causes the query processor to generate a breakpoint whenever the field value changes. The field is also printed as part of the report. Queries using breakpoints should also sort on the breakpoint field(s).

The action taken at the breakpoint depends on the optional **options** component and whether any field value accumulations (**AVERAGE**, **CALC**, **MAX**, **MIN**, **PERCENTAGE** or **TOTAL**) are in use.

A breakpoint with no **options** and no field accumulations prints a line with two asterisks in the column for the field causing the breakpoint followed by a blank line. If field accumulations are present, subtotals are also printed for each accumulated column. A line of hyphens may appear above the subtotals depending on the use of the **U** breakpoint control code.

The **options** text will be used in place of the default two asterisks when a breakpoint occurs. This text may also contain control codes enclosed in single quotes. The available control codes are:

- **B{n}** Start a new page, retaining the value of the breakpoint field for inclusion in the page heading/footing by use of the **B** heading text option. The optional single digit qualifier, **n**, allows collection of values from multiple breakpoints for inclusion in a composite heading. If omitted, the value of **n** defaults to zero. Thus use of **B** alone is equivalent to use of **B0**.

- **D** Omit the subtotal line if there is only one line of detail for this breakpoint.

- **L** Emit a blank line in place of the breakpoint. Any text in the **options** string will be ignored.

- **N** Resets page number to one at each breakpoint. This implies the **P** option if not used with **B** or **P**.

- **O** Only show the value of the breakpoint field on the first detail line within the breakpoint.

- **P** Start a new page.

- **U** The default behaviour of QM is to insert a line of hyphens above any subtotals, etc. The **U** breakpoint option suppresses this action. Enabling the **PICK.BREAKPOINT.U** mode of the **OPTION** command inverts this mode such that the line of hyphens appears only if the **U** breakpoint option is present.

- **V** Print the breakpoint field value in place of the default two asterisks. The **V** control code can be embedded in text into which the value will be inserted.
**X** Save previous value of breakpoint field in @BPV.

Combinations of control codes may be used together.

**Pick Syntax**

If the PICK.BREAKPOINT mode of the **OPTION** command is in effect, the **options** element of the **BREAK.ON** appears after the **field** rather than before.

**Examples**

The command

```
LIST SALES BY REGION BREAK.ON REGION SALESMAN TOTAL ORDER.VALUE
```

might produce a display such as that below.

```
LIST SALES BY REGION BREAK.ON REGION SALESMAN TOTAL ORDER.VALUE

SALES..... REGION SALESMAN ORDER VALUE
19887 North Roberts 279.40
19859 North Sharp 384.43
19858 North Sharp 845.50
19845 North Harris 234.53
** ------------
North 1743.86

19866 South Abbott 465.31
19886 South Abbott 397.23
19830 South Smith 324.39
** ------------
South 1186.93

------------
2930.79

7 records listed.
```

For this same data, the command

```
LIST SALES BY REGION BREAK.ON "Total'O'" REGION SALESMAN TOTAL ORDER.VALUE
```

would produce

```
LIST SALES BY REGION BREAK.ON "Total'O'" REGION SALESMAN TOTAL ORDER.VALUE

SALES..... REGION SALESMAN ORDER VALUE
19887 North Roberts 279.40
19859 Sharp 384.43
19858 Sharp 845.50
19845 Harris 234.53
Total ------------
North 1743.86

19866 Abbott 465.31
19886 Abbott 397.23
19830 Smith 324.39
Total ------------
South 1186.93
```
7 records listed.

See also:
BREAK.SUP
5.26 **BREAK.SUP**

The **BREAK.SUP** field qualifier keyword causes the query processor to generate a breakpoint whenever the field value changes. The field is not displayed in the report.

**Format**

```
BREAK.SUP  { "options" } field
```

where

- `options` controls the appearance of the breakpoint.
- `field` is the field name or evaluated expression to be reported.

The **BREAK.SUP** keyword appears before the field name and causes the query processor to generate a breakpoint whenever the field value changes. Queries using breakpoints should also sort on the breakpoint field(s).

The action taken at the breakpoint depends on the optional `options` component and whether any field value accumulations (**AVERAGE**, **CALC**, **MAX**, **MIN**, **PERCENTAGE** or **TOTAL**) are in use.

A breakpoint with no `options` and no field accumulations prints a line with two asterisks in the column for the field causing the breakpoint followed by a blank line. If field accumulations are present, subtotals are also printed for each accumulated column. A line of hyphens may appear above the subtotals depending on the use of the **U** breakpoint control code.

The `options` item is as for **BREAK.ON** though the text will never appear and only some control options are of use with **BREAK.SUP**. The useful control codes are:

- **B{n}** Start a new page, retaining the value of the breakpoint field for inclusion in the page heading/footing by use of the B heading text option. The optional single digit qualifier, `n`, allows collection of values from multiple breakpoints for inclusion in a composite heading. If omitted, the value of `n` defaults to zero. Thus use of **B** alone is equivalent to use of **B0**.
- **D** Omit the subtotal line if there is only one line of detail for this breakpoint.
- **L** Emit a blank line in place of the breakpoint. Any text in the `options` string will be ignored.
- **N** Resets page number to one at each breakpoint. This implies the **P** option if not used with **B** or **P**.
- **P** Start a new page.
- **U** The default behaviour of QM is to insert a line of hyphens above any subtotals, etc. The **U** breakpoint option suppresses this action. Enabling the **PICK.BREAKPOINT.U** mode of the **OPTION** command inverts this mode such that the line of hyphens appears only if the **U** breakpoint option is present.
- **X** Save previous value of breakpoint field in @BPV.

Combinations of control codes may be used together.

**Pick Syntax**
If the PICK.BREAKPOINT mode of the OPTION command is in effect, the options element of the BREAK.SUP appears after the field rather than before.

Examples

The command

```
LIST SALES BY REGION BREAK.SUP REGION SALESMAN TOTAL ORDER.VALUE
```

might produce a display such as that below.

```
LIST SALES BY REGION BREAK.SUP REGION SALESMAN TOTAL ORDER.VALUE Page 1
SALES.... SALESMAN ORDER VALUE
19887 Roberts 279.40
19859 Sharp 384.43
19858 Sharp 845.50
19845 Harris 234.53
-----------
1743.86
19866 Abbott 465.31
19886 Abbott 397.23
19830 Smith 324.39
-----------
1186.93
-----------
2930.79
7 records listed.

For this same data, the command

```
LIST SALES BY REGION BREAK.SUP "'B'" REGION SALESMAN TOTAL ORDER.VALUE HEADING "SALES FOR REGION: 'B'"
```

would produce

```
SALES FOR REGION: North
SALES.... SALESMAN ORDER VALUE
19887 Roberts 279.40
19859 Sharp 384.43
19858 Sharp 845.50
19845 Harris 234.53
-----------
1743.86
<<<page>>
SALES FOR REGION: South
SALES.... SALESMAN ORDER VALUE
19866 Abbott 465.31
19886 Abbott 397.23
19830 Smith 324.39
-----------
1186.93
```
Consider a report such as the following:

<table>
<thead>
<tr>
<th>Order No.</th>
<th>Region</th>
<th>Salesman</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10928</td>
<td>Central</td>
<td>Hughes</td>
<td>777.00</td>
</tr>
<tr>
<td></td>
<td>**</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Central</td>
<td></td>
<td>777.00</td>
</tr>
<tr>
<td>10275</td>
<td>North</td>
<td>Roberts</td>
<td>268.00</td>
</tr>
<tr>
<td>10274</td>
<td>North</td>
<td>Jones</td>
<td>876.43</td>
</tr>
<tr>
<td>10268</td>
<td>North</td>
<td>Roberts</td>
<td>1033.77</td>
</tr>
<tr>
<td></td>
<td>**</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>North</td>
<td></td>
<td>2178.20</td>
</tr>
<tr>
<td>10272</td>
<td>South</td>
<td>Edwards</td>
<td>768.33</td>
</tr>
<tr>
<td>10269</td>
<td>South</td>
<td>Edwards</td>
<td>564.98</td>
</tr>
<tr>
<td>10000</td>
<td>South</td>
<td>Abbott</td>
<td>111.00</td>
</tr>
<tr>
<td>10273</td>
<td>South</td>
<td>Smith</td>
<td>879.34</td>
</tr>
<tr>
<td></td>
<td>**</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>South</td>
<td></td>
<td>2323.65</td>
</tr>
</tbody>
</table>

This report could be improved by omitting the region column and instead showing the region name on the subtotal line for the salesman column:

<table>
<thead>
<tr>
<th>Test....</th>
<th>Salesman</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>10928</td>
<td>Hughes</td>
<td>777.00</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>North</td>
<td>777.00</td>
</tr>
<tr>
<td>10275</td>
<td>Roberts</td>
<td>268.00</td>
</tr>
<tr>
<td>10274</td>
<td>Jones</td>
<td>876.43</td>
</tr>
<tr>
<td>10268</td>
<td>Roberts</td>
<td>1033.77</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>North</td>
<td>2178.20</td>
</tr>
<tr>
<td>10272</td>
<td>Edwards</td>
<td>768.33</td>
</tr>
<tr>
<td>10269</td>
<td>Edwards</td>
<td>564.98</td>
</tr>
<tr>
<td>10000</td>
<td>Abbott</td>
<td>111.00</td>
</tr>
<tr>
<td>10273</td>
<td>Smith</td>
<td>879.34</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>North</td>
<td>2323.65</td>
</tr>
</tbody>
</table>

Total 5278.85
The X breakpoint control option saves the previous value of the breakpoint field, the region name in this example, in the @BPV variable. The SALESMAN.REPAY dictionary item is an I-type expression:

\[
\text{if } @nb = 0 \text{ then salesman else if } @nb = 255 \text{ then 'Total' else } @bpv
\]

The effect of this expression, when used with the **CALC** keyword, is that the detail lines (for which @NB is zero) return the salesman name, the grand total line (@NB = 255) returns "Total" and the other breakpoint lines return the value of @BPV.

See also:

**BREAK.ON**
The **BY** sort clause keyword causes the query processor to sort records prior to display or when building a select list.

**Format**

```
BY {NO.CASE} field
```

where

`field` is the field name or evaluated expression to be used to determine the sort order.

The **BY** keyword causes records to be sorted into ascending order of the specified field. The comparison is performed before conversion of the data to its display format. If the display format is left justified, a left justified sort is performed. Conversely, if the display format is right justified, a right justified sort is performed.

The optional **NO.CASE** keyword causes the sort to be performed in a case insensitive manner. Effectively, all comparisons are done using an uppercase version of the data being compared. Use of the QUERY.SORT.NO.CASE setting of the **OPTION** command implies the NO.CASE qualifier.

If more than one sort clause is present, sort criteria are applied in the order in which they are specified.

The command

```
LIST BOOKS BY @ID
```

is identical to

```
SORT BOOKS
```

**Example**

The command

```
LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAID BY VALUE
```

would produce a display such as that below.

```
LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAID BY VALUE
Invoice ...Value Customer.......................   
74993   £9.29  Write Right Stationery     
74273   £95.23 County Newspapers        
63940   £141.00  R Bryant              
74529   £1712.43  J McTavish          
4 records listed.
```

**See also:**

- **BY.DSND, BY.EXP, BY.EXP.DSND**
The BY.DSND sort clause keyword causes the query processor to sort records prior to display or when building a select list. The synonym BY-DSND may be used.

**Format**

```
BY.DSND [NO.CASE] field
```

where

```
field
```

is the field name or evaluated expression to be used to determine the sort order.

The BY.DSND keyword causes records to be sorted into descending order of the specified field. The comparison is performed before conversion of the data to its display format. If the display format is left justified, a left justified sort is performed. Conversely, if the display format is right justified, a right justified sort is performed.

The optional NO.CASE keyword causes the sort to be performed in a case insensitive manner. Effectively, all comparisons are done using an uppercase version of the data being compared. Use of the QUERY.SORT.NO.CASE setting of the **OPTION** command implies the NO.CASE qualifier.

If more than one sort clause is present, sort criteria are applied in the order in which they are specified.

**Example**

The command

```
LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAID BY.DSND VALUE
```

would produce a display such as that below.

```
LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAID BY.DSND VALUE
Invoice  ...Value  Customer.......................  
74529    £1712.43  J McTavish
63940    £141.00   R Bryant
74273    £95.23    County Newspapers
74993    £9.29     Write Right Stationery
4 records listed.
```

**See also:**

BY, BY.EXP, BY.EXP.DSND
5.29 **BY.EXP**

The **BY.EXP** sort clause keyword applied to a multivalued field causes the query processor to explode the multivalued items to form separate single valued records and to sort these into ascending order prior to display or when building a select list.

**Format**

```
BY.EXP { NO.CASE } field { explosion.limiter }
```

where

*field* is the field name or evaluated expression to be used to determine the sort order.

*explosion.limiter* is a condition similar to that specified in a **WHEN** clause to limit the records and values included in the report. Use of explosion limiters requires that the QUALIFIED.DISPLAY setting of the **OPTION** command is enabled.

The **BY.EXP** keyword causes records to be sorted into ascending order of the values stored in the specified field. The comparison is performed before conversion of the data to its display format. If the display format is left justified, a left justified sort is performed. Conversely, if the display format is right justified, a right justified sort is performed.

The optional **NO.CASE** keyword causes the sort to be performed in a case insensitive manner. Effectively, all comparisons are done using an uppercase version of the data being compared. Use of the QUERY.SORT.NO.CASE setting of the **OPTION** command implies the **NO.CASE** qualifier.

If more than one sort clause is present, sort criteria are applied in the order in which they are specified. If more than one exploded sort clause is present, they must all be on the same association.

**Example**

The command

```
LIST ORDERS PART.NO QTY LINE.TOTAL
```

might produce a display such as that below.

```
ORDER   PART   QTY   PRICE   TOTAL
24842    648     7   10.00    70.00
         216     3   8.00     24.00
24851    107     2   12.50    25.00
24856    319     6   4.50     27.00
         372     1  18.75    18.75
3 records listed.
```

The command

```
LIST ORDERS PART.NO QTY LINE.TOTAL BY.EXP PART.NO
```

applied to the same data would produce the display below.

```
LIST ORDERS PART.NO QTY PRICE LINE.TOTAL
```
<table>
<thead>
<tr>
<th>ORDER</th>
<th>PART</th>
<th>QTY</th>
<th>PRICE</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>24851</td>
<td>107</td>
<td>2</td>
<td>12.50</td>
<td>25.00</td>
</tr>
<tr>
<td>24842</td>
<td>216</td>
<td>3</td>
<td>8.00</td>
<td>24.00</td>
</tr>
<tr>
<td>24856</td>
<td>319</td>
<td>6</td>
<td>4.50</td>
<td>27.00</td>
</tr>
<tr>
<td>24856</td>
<td>372</td>
<td>1</td>
<td>18.75</td>
<td>18.75</td>
</tr>
<tr>
<td>24842</td>
<td>648</td>
<td>7</td>
<td>10.00</td>
<td>70.00</td>
</tr>
</tbody>
</table>

3 records, 5 values listed.

Adding an explosion limiter to this query to show only part numbers greater than 300:

```
LIST ORDERS PART.NO QTY LINE.TOTAL BY.EXP PART.NO > 300
```

would produce the report below.

```
ORDER   PART   QTY   PRICE   TOTAL.
24856   319     6    4.50    27.00
24856   372     1   18.75   18.75
24842   648     7   10.00   70.00
2 records, 3 values listed.
```

See also:

`BY, BY.DSND, BY.EXP.DSND, REPEATING`
5.30 BY.EXP.DSND

The **BY.EXP.DSND** sort clause keyword applied to a multivalued field causes the query processor to explode the multivalued items to form separate single valued records and to sort these into descending order prior to display or when building a select list.

**Format**

```plaintext
BY.EXP.DSND [NO.CASE] field {explosion.limiter}
```

where

- **field** is the field name or evaluated expression to be used to determine the sort order.
- **explosion.limiter** is a condition similar to that specified in a **WHEN** clause to limit the records and values included in the report. Use of explosion limiters requires that the QUALIFIED.DISPLAY setting of the **OPTION** command is enabled.

The **BY.EXP.DSND** keyword causes records to be sorted into descending order of the values stored in the specified field. The comparison is performed before conversion of the data to its display format. If the display format is left justified, a left justified sort is performed. Conversely, if the display format is right justified, a right justified sort is performed.

The optional **NO.CASE** keyword causes the sort to be performed in a case insensitive manner. Effectively, all comparisons are done using an uppercase version of the data being compared. Use of the QUERY.SORT.NO.CASE setting of the **OPTION** command implies the NO.CASE qualifier.

If more than one sort clause is present, sort criteria are applied in the order in which they are specified. If more than one exploded sort clause is present, they must all be on the same association.

**Example**

The command

```plaintext
LIST ORDERS PART.NO QTY LINE.TOTAL
```

might produce a display such as that below.

```plaintext
LIST ORDERS PART.NO QTY PRICE LINE.TOTAL
ORDER PART QTY PRICE TOTAL
24842 648 7 10.00 70.00
216 3 8.00 24.00
24851 107 2 12.50 25.00
24856 319 6 4.50 27.00
372 1 18.75 18.75
3 records listed.
```

The command

```plaintext
LIST ORDERS PART.NO QTY LINE.TOTAL BY.EXP.DSND PART.NO
```

applied to the same data would produce the display below.

```plaintext
LIST ORDERS PART.NO QTY PRICE LINE.TOTAL
```
ORDER  PART  QTY  PRICE  TOTAL.
24842  648    7  10.00   70.00
24856  372    1  18.75   18.75
24856  319    6  4.50     27.00
24842  216    3  8.00    24.00
24851  107    2 12.50    25.00
3 records, 5 values listed.

Adding an explosion limiter to this query to show only part numbers greater than 300:

LIST ORDERS PART.NO QTY LINE.TOTAL BY.EXP.DSND PART.NO > 300

would produce the report below.

LIST ORDERS PART.NO QTY PRICE LINE.TOTAL
ORDER  PART  QTY  PRICE  TOTAL.
24842  648    7  10.00   70.00
24856  372    1  18.75   18.75
24856  319    6  4.50     27.00
2 records, 3 values listed.

See also:  
BY, BY.DSND, BY.EXP
5.31 CALC

The CALC keyword prefixes an I-type field name or an evaluated expression and causes the calculation to be performed on the total lines, possibly using accumulated values from the detail lines.

Format

CALC field

where

field

is the I-type field or expression for which the calculation is to be performed.

The CALC keyword is normally used in conjunction with the I-type TOTAL() function. During detail lines, the TOTAL() function accumulates values which are then used on the subtotal and grand total lines to calculate the value in the column to which the CALC keyword applies.

The CALC keyword can be used for any other expression to be evaluated at breakpoints and at the end of the report. See the BREAK.SUP keyword for an example of using this with the X breakpoint control option.

Example

We have a file which includes a calculated PROFIT item defined as

\[ 100 \times (\text{SELL} - \text{COST}) / \text{COST} \]

The command

LIST PARTS AVG COST AVG SELL AVG PROFIT

This might produce a report such as that below

<table>
<thead>
<tr>
<th>Part</th>
<th>Cost</th>
<th>Sell</th>
<th>Profit%</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>10.00</td>
<td>13.00</td>
<td>30.00</td>
</tr>
<tr>
<td>102</td>
<td>15.00</td>
<td>18.00</td>
<td>20.00</td>
</tr>
<tr>
<td>103</td>
<td>14.00</td>
<td>17.00</td>
<td>21.43</td>
</tr>
</tbody>
</table>

The average profit figure (23.81) is the average of the figures in the column above it. Perhaps what we really want to show is the percentage profit selling for 16.00 something that cost us 13.00 (the average cost and selling prices). To do this, the PROFIT expression is changed to

\[ 100 \times (\text{TOTAL(SELL)} - \text{TOTAL(COST)}) / \text{TOTAL(COST)} \]

or, more simply,

\[ 100 \times \text{TOTAL(SELL} - \text{COST}) / \text{TOTAL(COST)} \]

The command

LIST PARTS AVG COST AVG SELL CALC PROFIT

now produces
<table>
<thead>
<tr>
<th>Part</th>
<th>Cost</th>
<th>Sell</th>
<th>Profit%</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>10.00</td>
<td>13.00</td>
<td>30.00</td>
</tr>
<tr>
<td>102</td>
<td>15.00</td>
<td>18.00</td>
<td>20.00</td>
</tr>
<tr>
<td>103</td>
<td>14.00</td>
<td>17.00</td>
<td>21.43</td>
</tr>
</tbody>
</table>

3 records listed.
5.32 COL.HDG

The COL.HDG keyword defines an alternative column heading for reported data. The synonym DISPLAY.NAME may be used.

Format

\[ field \text{ COL.HDG } text \]

where

- \( field \) is the field or expression to which the new column heading is to be applied.
- \( text \) is the new column heading. This must be enclosed in single or double quotes.

The default column heading for reported data is the display name from the dictionary entry or, for evaluated expressions, the expression. The COL.HDG field qualifier can be used to set an alternative column heading.

The \( text \) may include the control tokens that control how the column heading is displayed. The codes are enclosed in single quotes which implies that a column heading specification that uses the codes must itself be enclosed in double quotes.

- 'L' appearing within \( text \) breaks the heading onto a new line at that point. This is equivalent to use of a value mark in a dictionary heading definition.
- 'R' at the start of the heading \( text \) right aligns the heading.
- 'X' at the start of the heading \( text \) suppresses the dot fillers normally inserted into unused columns of the heading.

If both R and X are to be used, they should be enclosed in a single set of quotes.

Examples

```
LIST INVOICES AMT.DUE COL.HDG "Outstanding" SITE.NAME WITH NO AMT.RECEIVED
```

This command reports records from the INVOICES file where no payment has been recorded. The AMT.DUE field has the column heading set to "Outstanding".

```
LIST SALES CUST.NO COL.HDG "Client'L'Number" VALUE COL.HDG "'RX'Value"
```

This command reports records from the SALES file. The heading for the CUST.NO field occupies two lines. The column heading for VALUE is right justified with the normal dot filler suppressed.
5.33 COL.HDG.ID

The **COL.HDG.ID** keyword causes the query processor to use the display clause field names as the default column headings.

**Format**

**COL.HDG.ID**

The query processor normally uses the display name entry from the dictionary as the column heading in a report. Use of the **COL.HDG.ID** keyword causes use of the actual field name as the column heading for all fields unless overridden by use of **COL.HDG**.

**Example**

The command

```
LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAID
```

would normally produce a display such as

```
LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAID
Invoice  Value  Customer.......................  
74529    £1712.43    J McTavish
74273    £95.23       County Newspapers
63940    £141.00       R Bryant
74993    £9.29       Write Right Stationery
4 records listed.
```

Adding the **COL.HDG.ID** option, the command

```
LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAID COL.HDG.ID
```

would produce a display such as

```
LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAID
INVOICE  VALUE  CUSTOMER.NAME...............  
74529    £1712.43    J McTavish
74273    £95.23       County Newspapers
63940    £141.00       R Bryant
74993    £9.29       Write Right Stationery
4 records listed.
```
The **COL.HDR.SPACE** keyword inserts a blank line under the column headings.

**Format**

**COL.HDR.SPACE**

The **COL.HDR.SPACE** keyword causes the query processor to emit a blank line after the column headings in a tabular format report. Used with **COL.SUP**, the blank line replaces the column headings.

**Example**

The command

```
LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAID
```

would normally produce a display such as

```
Invoice ...Value Customer...................
74529   £1712.43  J McTavish
74273   £95.23   County Newspapers
63940   £141.00  R Bryant
74993   £9.29   Write Right Stationery
4 records listed.
```

Adding the **COL.HDR.SPACE** option, the command

```
LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAID
COL.HDR.SPACE
```

would produce a display such as

```
LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAID
COL.HDR.SPACE
Invoice ...Value Customer...................
74529   £1712.43  J McTavish
74273   £95.23   County Newspapers
63940   £141.00  R Bryant
74993   £9.29   Write Right Stationery
4 records listed.
```
5.35 COL.HDR.SUPP

The COL.HDR.SUPP display clause keyword suppresses page and column headings. The synonyms COL-HDR-SUPP, COL.HDR.SUP and COL-HDR-SUP can be used.

Format

COL.HDR.SUPP

The COL.HDR.SUPP keyword suppresses both the page heading (normally the command that invoked the query processor) and the column headings derived from the dictionary display names or COL.HDG keywords.

Example

The command

LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAID

would normally produce a display such as

LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAID
Invoice ...Value Customer....................... 74529 £1712.43 J McTavish 74273 £95.23 County Newspapers 63940 £141.00 R Bryant 74993 £9.29 Write Right Stationery 4 records listed.

Adding the COL.HDR.SUPP option, the command

LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAID COL.HDR.SUPP

would produce a display such as

74529 £1712.43 J McTavish 74273 £95.23 County Newspapers 63940 £141.00 R Bryant 74993 £9.29 Write Right Stationery 4 records listed.
5.36 COL.SPACES

The COL.SPACES keyword determines the number of spaces inserted between columns of a tabular report. The synonym COL.SPCS may be used.

Format

COL.SPACES \( n \)

where

\( n \) is the number of spaces to be used. If omitted, this defaults to one.

Tabular reports are automatically adjusted to fit the available page space. The query processor will try to put three spaces between each column. If this will not fit the available width, the column spacing is reduced to fit the output device or, if this cannot be done, the query processor switches to vertical mode.

The COL.SPACES keyword allows a user specified column spacing to be used.

Example

The command

LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAID

would produce a display such as that below.

<table>
<thead>
<tr>
<th>Invoice</th>
<th>Value</th>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>74529</td>
<td>£1712.43</td>
<td>J McTavish</td>
</tr>
<tr>
<td>74273</td>
<td>£95.23</td>
<td>County Newspapers</td>
</tr>
<tr>
<td>63940</td>
<td>£141.00</td>
<td>R Bryant</td>
</tr>
<tr>
<td>74993</td>
<td>£9.29</td>
<td>Write Right Stationery</td>
</tr>
</tbody>
</table>

4 records listed.

Use of the COL.SPACES keyword could modify this to become

LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAID COL.SPACES 8

<table>
<thead>
<tr>
<th>Invoice</th>
<th>Value</th>
<th>Customer</th>
</tr>
</thead>
<tbody>
<tr>
<td>74529</td>
<td>£1712.43</td>
<td>J McTavish</td>
</tr>
<tr>
<td>74273</td>
<td>£95.23</td>
<td>County Newspapers</td>
</tr>
<tr>
<td>63940</td>
<td>£141.00</td>
<td>R Bryant</td>
</tr>
<tr>
<td>74993</td>
<td>£9.29</td>
<td>Write Right Stationery</td>
</tr>
</tbody>
</table>

4 records listed.
5.37 CONV

The CONV keyword defines an alternative conversion for reported data.

Format

\[ \text{field CONV conv.spec} \]

where

- \text{field} is the field or expression to which the new conversion is to be applied.
- \text{conv.spec} is the new conversion specification. This must be enclosed in single or double quotes.

The default conversion for reported data is taken from the dictionary entry for field or, for evaluated expressions, the first field referenced in the expression. The CONV field qualifier can be used to set an alternative conversion specification.

Example

\begin{verbatim}
LIST ORDERS ORDER.DATE CONV "DDMY"
\end{verbatim}

This command reports records from the ORDERS file where using a non-default conversion specification for the ORDER.DATE field.
5.38 **COUNT.SUP**

The **COUNT.SUP** display option keyword suppresses display of the number of records listed or selected at the end of the command.

**Format**

```
COUNT.SUP
```

Query processor commands normally displays the number of records listed or selected at the end of the command. The **COUNT.SUP** keyword can be used to suppress this action.

**Example**

Compare the two commands and their displayed results shown below.

```
SELECT STOCK WITH QTY < REORDER.LEVEL
78 records selected.

SELECT STOCK WITH QTY < REORDER.LEVEL COUNT.SUP
```

The first command shows the normal display of the count of records selected. The second command includes the **COUNT.SUP** keyword to suppress this display.
5.39 **COL.SUP**

The **COL.SUP** display clause keyword suppresses column headings. The synonym **COL-SUPP** can be used.

**Format**

```
COL.SUP
```

The **COL.SUP** keyword suppresses the column headings derived from the dictionary display names or **COL.HDG** keywords. Used with **LIST.ITEM** or **SORT.ITEM**, this keyword suppress display of field numbers.

**Example**

The command

```
LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAID
```

would normally produce a display such as

```
LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAID
Invoice ....Value Customer.......................  
74529    £1712.43  J McTavish            
74273     £95.23  County Newspapers     
63940     £141.00  R Bryant             
74993      £9.29  Write Right Stationery  
4 records listed.
```

Adding the **COL.SUP** option, the command

```
LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAID COL.SUP
```

would produce a display such as

```
LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAID COL.SUP
74529    £1712.43  J McTavish            
74273     £95.23  County Newspapers     
63940     £141.00  R Bryant             
74993      £9.29  Write Right Stationery  
4 records listed.
```
5.40 CSV

The **CSV** display option keyword specifies that the report should be produced in CSV format.

**Format**

```
CSV {"mode"} {"delimiter"} {TO pathname | AS file id {NO.QUERY | APPENDING}}
{ENCODING name}
```

where

- **mode** is a numeric value specifying the format rules to be applied.
- **delimiter** is an alternative delimiting character. This may not be a double quote.
- **pathname** is the pathname of the file to receive the output.
- **file id** is a filename and record id pair identifying the destination for the output.
- **name** is the name of an encoding to be applied.

The **CSV** keyword produces a report in CSV (comma separated variable) format as used by many software products. In this format, each item in the report is separated by a comma instead of the usual tabular style of report. QM extends this format by allowing use of an alternative **delimiter** character.

The **mode** option specifies the format rules to be applied. A **mode** value of 1 (the default if no **mode** is given) produces output that conforms to the CSV format specification (RFC 4180). This requires that items containing double quotes or the delimiter character are enclosed in double quotes with embedded double quotes replaced by two adjacent double quotes.

A **mode** value of 2 encloses all non-null values in double quotes except for numeric items that do not contain a comma. Embedded double quotes are replaced by two adjacent double quotes.

A **mode** value of 3 encloses all values in double quotes. Embedded double quotes are replaced by two adjacent double quotes.

The **delimiter** may be set to a tab character by use of the special syntax "<TAB>". Other non-printing characters can be specified by use of the ^nnn notation where nnn is the three digit character number from the ASCII character set.

Multivalued items will be split over multiple lines of CSV output. If an item is defined as single valued in the dictionary (or by use of the **SINGLE.VALUE** keyword) but the corresponding data contains value marks, the marks are treated as normal data characters and the entire multivalued item, including the embedded value marks, will be output as a single element of the CSV output.

The **TO** option directs output to an operating system file by pathname. The **AS** option directs output to a named record in a QM directory file. If the destination item already exists, the user will be prompted to confirm whether it should be overwritten. The **NO.QUERY** option suppresses this prompt, overwriting the existing file. The **APPENDING** option causes the output to be appended to the output file if it already exists. The **NO.QUERY** and **APPENDING** options may not be used together. The optional **ENCODING** qualifier can be used to set or override the character encoding for the destination file.

Use of **TO** or **AS** implies use of **HDR.SUP** as the output is not paginated.
In normal usage, the page heading and record counts would probably need to be suppressed using the **HDR.SUP** and **COUNT.SUP** keywords. The **COL.SUP** keyword can be used to suppress column headings.

**Examples**

The command

```
LIST CUSTOMERS NAME TEL HDR.SUP COL.SUP COUNT.SUP CSV
```

would produce a display such as that below.

```
17463, Arkright Tool Hire, 01726-48745
56221, "Smith, Price and Samuel", 01876-28414
```

Note how the customer name in the second line has been quoted because it contains a comma.

```
LIST STOCK PROD.NO QOH DESCR CSV TO C:\STOCK.CSV
```

This command constructs comma separated format report of the STOCK file into the C:\STOCK.CSV file.

```
LIST STOCK PROD.NO QOH DESCR CSV AS $HOLD STK.RPT
```

This command constructs comma separated format report of the STOCK file in a record named SRK.RPT in the $HOLD file.

A command such as

```
LIST SALES ITEM.NO CSV
```

where ITEM.NO is multivalued might produce a report containing

```
26832, 176
,218
,398
```

Modifying the command to be

```
LIST SALES ITEM.NO SINGLE.VALUE CSV
```

would change the output to be

```
16832, 176
218
398
```

See also: **DELMIT**
5.41 **CUMULATIVE**

The **CUMULATIVE** field qualifier keyword displays the cumulative value of a field.

**Format**

```
CUMULATIVE field {field.qualifiers}
```

where

- `field` is the field or evaluated expression to be displayed.
- `field.qualifiers` are other field qualifying keywords

The **CUMULATIVE** field qualifier keyword is placed before the field name to which it applies and causes the query processor to report the cumulative value of the field. A total line is also produced. Used with breakpoints, the **CUMULATIVE** keyword will also report the total value of the field at each breakpoint.

The **CUMULATIVE** keyword operates only on numeric data. Non-numeric values are ignored.

**Example**

The command

```
LIST INVOICES TOTAL VALUE CUMULATIVE VALUE CUSTOMER.NAME WITH
NO AMT.PAID
```

would produce a display such as that below in which the total value of the VALUE field is included at the end of the report.

```
LIST INVOICES TOTAL VALUE CUMULATIVE VALUE CUSTOMER.NAME WITH
NO AMT.PAID
Invoice   ...Value   ...Value  Customer......................
74529     £1712.43  £1712.43  J McTavish
74273     £95.23    £1807.66  County Newspapers
63940     £141.00   £1948.66  R Bryant
74993     £9.29     £1957.95  Write Right Stationery
           =======   =======
           £1957.95  £1957.95
4 records listed.
```
5.42 DBL.SPC

The DBL.SPC display option keyword causes records in a tabular report to be double spaced. The synonym DBL-SPC may be used.

Format

DBL.SPC

The DBL.SPC keyword inserts a blank line between each record displayed in a tabular format report. It has no effect in a vertical format report.

Example

The command

LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAIRED

would produce a display such as that below.

LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAIRED
Invoice ...Value Customer....................
74529   £1712.43  J McTavish
74273   £95.23  County Newspapers
63940   £141.00  R Bryant
74993   £9.29  Write Right Stationery
4 records listed.

Using the DBL.SPC keyword modifies this report to become

LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAIRED DBL.SPC
Invoice ...Value Customer....................
74529   £1712.43  J McTavish
74273   £95.23  County Newspapers
63940   £141.00  R Bryant
74993   £9.29  Write Right Stationery
4 records listed.
5.43 DEFAULT

The DEFAULT keyword causes the query processor to parse the command using the default QM option settings.

**Format**

```
DEFAULT
```

The DEFAULT keyword allows a query to be written in a form that will be parsed and executed in the same way regardless of any option settings (see the OPTION command). Although it can be included in user written query commands, it is primarily intended for inclusion in queries that form standard parts of QM and might behave incorrectly if options have been set.

The keyword takes effect from the point where it appears in the query sentence. Any option sensitive components appearing in the query prior to this keyword will be parsed in accordance with the current option settings. Note in particular that because query elements appearing in the optional SQUERY.DEFaulTs record are effectively inserted into the query sentence immediately after the filename but before any other options, the DEFAULT keyword is permitted to be placed before the filename.

See also:

OPTION
5.44 DELIMITER

The DELIMITER display option keyword specifies the separating character(s) to be used in a delimited report.

Format

```
DELIMITER "string" {TO pathname | AS file id [ENCODING name] [NO.QUERY | APPENDING]}
```

where

- **string** is the character sequence to be placed between report "columns".
- **pathname** is the pathname of the file to receive the output.
- **file id** is a filename and record id pair identifying the destination for the output.

A delimited report displays its output as a series of items separated by the given **string** instead of the usual tabular style of report. The DELIMITER keyword causes the query processor to produce this style of report and specifies the separator to be used. The output from a delimited report can, for example, be structured with comma separators, sent to a file using the **LPTR** keyword and then read into applications such as Microsoft Excel.

The **string** may contain tab characters by use of the special syntax "<TAB>". Other non-printing characters can be included by use of the ^nnn notation where nnn is the three digit character number from the ASCII character set.

Multivalued items will be split over multiple lines of delimited output. If an item is defined as single valued in the dictionary (or by use of the SINGLE.VALUE keyword) but the corresponding data contains value marks, the marks are treated as normal data characters and the entire multivalued item, including the embedded value marks, will be output as a single element of the delimited output.

The TO option directs output to an operating system file by pathname. The AS option directs output to a named record in a QM directory file. If the destination item already exists, the user will be prompted to confirm whether it should be overwritten. The NO.QUERY option suppresses this prompt, overwriting the existing file. The APPENDING option causes the output to be appended to the output file if it already exists. The NO.QUERY and APPENDING options may not be used together. The optional **ENCODING** qualifier can be used to set or override the character encoding for the destination file.

Use of TO or AS implies use of **HDR.SUP** as the output is not paginated.

In normal usage, the page heading and record counts would probably need to be suppressed using the **HDR.SUP** and **COUNT.SUP** keywords. The **COL.SUP** keyword can be used to suppress column headings.

Examples

The command

```
LIST INVOICES VALUE CUSTOMER.NAME DELIMITER ",," HDR.SUP
COL.SUP COUNT.SUP
```

would produce a display such as that below.
The command

    LIST INVOICES VALUE CUSTOMER.NAME DELIMITER "<tab>" HDR.SUP COUNT.SUP

would produce a display such as that below where the spacing is performed by tab characters.

```
74529      £1712.43  J McTavish
74273      £95.23    County Newspapers
63940      £141.00   R Bryant
74993      £9.29     Write Right Stationery
```

A command such as

    LIST SALES ITEM.NO DELIMITER "",""

where ITEM.NO is multivalued might produce a report containing

```
26832,176
,218
,398
```

Modifying the command to be

    LIST SALES ITEM.NO SINGLE..VALUE DELIMITER

would change the output to be

```
16832,176vM218vM398
```

See also: CSV
5.45 DET.SUP

The DET.SUP keyword (synonym DET-SUPP) suppresses reporting of detail lines, leaving only page and column headers, totals, footers and the final record count.

Format

DET.SUP

The DET.SUP keyword removes all detail lines from a report. It allows easy reporting of totals without the data that contributed to the totals.

When a query uses both DET.SUP and breakpoints, any column for which no breakpoint is defined and no arithmetic field qualifier is in use will show the value of that field for the last record processed within that breakpoint group. This is probably only meaningful when the value of the field is the same for all records in the breakpoint group.

When used with report styles (see the STYLE option), subtotal lines will be reported using the detail line style.

Example

The command

```
LIST SALES BY REGION BREAK.ON REGION SALESMAN TOTAL ORDER.VALUE
```

would produce a display such as that below.

```
LIST SALES BY REGION BREAK.ON REGION SALESMAN TOTAL ORDER.VALUE
Page 1
SALES..... REGION SALESMAN ORDER VALUE
19887 North Roberts 279.40
19859 North Sharp 384.43
19858 North Sharp 845.50
19845 North Harris 234.53
** 9859 North Sharp
** 9858 North Sharp
** 9845 North Harris
** 9887 North Roberts
1743.86

19886 South Abbott 465.31
19886 South Abbott 397.23
19830 South Smith 324.39
** 9886 South Abbott
** 9886 South Abbott
1186.93

7 records listed.
```

Adding the DET.SUP keyword and omitting the salesman changes this report to be as below.

```
LIST SALES BY REGION BREAK.ON REGION TOTAL ORDER.VALUE DET.SU
Page 1
```
<table>
<thead>
<tr>
<th>REGION</th>
<th>ORDER VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>North</td>
<td>1743.86</td>
</tr>
<tr>
<td>South</td>
<td>1186.93</td>
</tr>
</tbody>
</table>

=========
2930.79

7 records listed.
The **DISPLAY.LIKE** keyword is a field qualifier which causes the field to be displayed using the attributes of another field defined in the dictionary.

**Format**

\[
\text{field.name DISPLAY.LIKE other.field}
\]

where

- **field.name** is the field or an evaluated expression to be displayed.
- **other.field** is the field whose attributes are to be used when **field.name** is displayed.

The **DISPLAY.LIKE** keyword causes the attributes of **other.field** to be used when displaying **field.name**. The attributes are the display name, conversion, format, single / multiple value flag and association. The **COL.HDG**, **CONV**, **FMT**, **SINGLE.VALUE**, **MULTI.VALUE**, **ASSOC** and **ASSOC.WITH** field qualifiers can be used to further modify the attributes.

**Example**

\[
\text{LIST INVOICES EVAL } \text{"ISSUE.DATE + 90" DISPLAY.LIKE DUE.DATE COL.HDG "Reminder date"}
\]

This command processes records from the INVOICES file and reports the record id (probably the invoice number) and the date 90 days after that stored in the ISSUE.DATE field using the attributes of the DUE.DATE field except for the column heading which is specifically set.
5.47  **ELEMENT**

The **ELEMENT** keyword prefixes a data collection element path when processing a collection file.

**Format**

```
ELEMENT path
```

where

```
path
```

is the data collection element path. This may need to be enclosed in single or double quotes to avoid syntactic ambiguity.

The **ELEMENT** keyword allows a data collection element to be referenced by element path on the command line in much the same way as a field can be referenced using *An* or *Fn* instead of by name. It may be used in the selection, sort or display clauses.

The *path* may include an asterisk in a position that corresponds to an array in the data collection. This will result in a multivalued result where each value corresponds to an array element. For example, a simple sales order record might have an element named "detail" which is an array where each element is a further collection with elements "prodno" and "qty". Use of

```
ELEMENT "detail/*/prodno"
```

would reference a multivalued list of product numbers in the order.

Taking this example further, it may be necessary to store the serial numbers of each item sold as an array named "serial" that is stored as a third element in the detail collection. Use of

```
ELEMENT "detail/*/serial/*"
```

would reference data in which there is a value for each product and a subvalue for each serial number.

The default display characteristics used for an **ELEMENT** item in the absence of a field qualifiers treat it as 10 characters left justified, single valued and use the element path as the column heading.

**See also:**

[AS, Data collections](#)
5.48 ENUMERATE

The **ENUMERATE** field qualifier keyword causes a field to be reported together with a count of values. The synonym **ENUM** may be used.

**Format**

```
ENUM field {field.qualifiers} {NO.NULLS}
```

where

- **field** is the field or evaluated expression to be displayed.
- **field.qualifiers** are other field qualifying keywords
- **NO.NULLS** causes null values to be ignored.

The **ENUMERATE** field qualifier keyword is placed before the field name to which it applies and causes the query processor to report the value of the field for each record processed and also to report the number of values at the end of the report. Used with breakpoints, the **ENUMERATE** keyword will also report the number of values at each breakpoint.

If the field is defined as multivalued, the **ENUMERATE** keyword counts each value.

The **NO.NULLS** keyword can be used to ignore null values.

**Example**

The command

```
LIST INVOICES ENUMERATE VALUE CUSTOMER.NAME WITH NO AMT.PAID
```

would produce a display such as that below in which the number of items in the VALUE field is shown at the end of the report.

```
LIST INVOICES ENUMERATE VALUE CUSTOMER.NAME WITH NO AMT.PAID
Invoice ....Value Customer....................
  74529 £1712.43 J McTavish
  74273 £95.23 County Newspapers
  63940 £141.00 R Bryant
  74993 £9.29 Write Right Stationery
        ========
        4
4 records listed.
```
5.49 EQ

The EQ selection clause operator compares a field or evaluated expression against another field, evaluated expression or literal value and tests for the first item being equal to the second. The synonyms EQUAL and = can be used.

**Format**

```
field EQ [NO.CASE] value
```

where

- `field` is the first field or evaluated expression to be compared.
- `value` is the second field, evaluated expression or literal value to be compared. The optional NO.CASE qualifier causes a case insensitive comparison to be applied.

The EQ selection clause operator returns True if `field` is equal to `value`.

The default behaviour of the query processor is that if `field` is defined as left justified in the dictionary, an exact match string comparison is used. For right aligned fields (typically numeric values) a numeric comparison will be used if the field content and `value` are both integer values. The QUERY.STRING.COMP or QUERY.STR.COMP.ALL modes of the OPTION command can be used to apply string character comparisons to all data.

**Example**

```
LIST STOCK WITH QTY EQ 100
```

This command lists items found on the STOCK file with a QTY field of 100. If QTY is defined as left justified, the content of this field must be the three digits 100 for the record to be included in the report. If QTY is defined as right justified, records for which QTY holds any numeric data equal to 100 will be included (e.g. 0100).

**Pick Style Wildcards**

If the PICK.WILDCARD option is enabled (see the OPTION command) and the `value` item is a literal value, the interpretation is extended to include use of Pick style wildcard characters:

- A `[]` character at the start of the `value` replaces any number of leading characters. It is equivalent to the … action of the LIKE operator.
- A `[]` character at the end of the `value` replaces any number of trailing characters. It is equivalent to the … action of the LIKE operator.
- A `^` character within the `value` replaces a single character. It is equivalent to the 1X action of the LIKE operator.
The **EVAL** (or **EVALUATE**) keyword prefixes an expression to be evaluated and used as though it were an I-type defined in the dictionary.

**Format**

```
EVAL expr
```

where

- **expr** is the expression. This must be enclosed in single or double quotes.

The **EVAL** keyword prefixes an expression which is handled as a temporary I-type. It may be used as a field for display or in selection or sort clauses.

The **expr** string conforms to the same rules as I-type expressions. It is compiled by the query processor for use within the command being executed but is not saved in the dictionary. Where the value of **expr** is a constant, the expression is evaluated only once and then treated as a literal value.

The default display name used for an **EVAL** expression in the absence of a **COL.HDG** qualifier is the expression itself.

The default display format and conversion are taken from the first field referenced by the expression. If no fields are referenced, the format defaults to 10L with no conversion. Alternative format or conversion may be specified with the **FMT** or **CONV** field qualifiers.

For increased data privacy, the **SECURITY** configuration parameter can be used to set a mode whereby use of **EVAL** requires the user to have write access to the file's dictionary. Note that this is the dictionary associated with the file being reported regardless of possible use of the **USING** clause to select an alternative dictionary.

**Example**

```
LIST INVOICES EVAL "SUM(RECEIVED)"
```

This command processes records from the INVOICES file and reports the record id (probably the invoice number) and the total of the RECEIVED field which is multivalued to allow for more than one payment against an invoice.

**See also:**

**AS**
5.51 FMT

The FMT keyword defines an alternative format for reported data.

Format

\[ field \text{ FMT } fmt.spec \]

where

- \textit{field} is the field or expression to which the new format is to be applied.
- \textit{fmt.spec} is the new format specification. This must be enclosed in single or double quotes.

The default format for reported data is taken from the dictionary entry for \textit{field} or, for evaluated expressions, the first field referenced in the expression. The FMT field qualifier can be used to set an alternative format specification.

Example

```
LIST ORDERS SITE.NAME FMT "32L"
```

This command reports records from the ORDERS file where using a non-default format specification for the SITE.NAME field.
5.52 FOOTING

The FOOTING keyword defines a page footing for the report. The synonym FOOTER may be used.

**Format**

```
FOOTING "text"
```

where

- `text` is the footing text to appear on each page.

A page footing defined with the FOOTING keyword will appear at the bottom of each page of the report. The footing text may contain control codes enclosed in single quotes to insert variable data or to alter the appearance of the footing. These control codes are:

- **B{n}** Insert data from the corresponding B control code in a BREAK.ON or BREAK.SUP option string. The optional single digit qualifier, n, defaults to zero if omitted.
- **C** Centres the current line of the footing text.
- **D** Inserts the date. The default format is dd mmm yyyy (e.g. 24 Aug 2005) but can be changed using the DATE.FORMAT command.
- **F{n}** Inserts the file name in a field of n spaces. If n is omitted, a variable width is used.
- **G** Inserts a gap. Spaces are inserted in place of the G control code to expand the text to the width of the output device. If more than one G control code appears in a single line, spaces are distributed as evenly as possible.

When a footing line uses both G and C, the footing is considered as a number of elements separated by the G control options. The element that contains the C option will be centered. The items either side of the centered element are processed separately when calculating the number of spaces to be substituted for each G option.

- **Hn** Sets horizontal position (column) numbered from one. Use of H with C or with a preceding G token may have undesired results.
- **I{n}** Inserts the record id in a field of n spaces. If n is omitted, a variable width is used.
- **L** Inserts a new line at this point in the text.
- **N** Suppresses pagination of the output to the display.
- **O** Reverses the elements separated by G tokens in the current line on even numbered pages. This is of use when printing double sided reports.
- **P{n}** Insert page number. The page number is right justified in n spaces, widening the field if necessary. If omitted, n defaults to four.
- **R{n}** Same as I{n}.
- **S{n}** Insert page number. The page number is left justified in n spaces, widening the field if necessary. If omitted, n defaults to one.
- **T** Inserts the time and date in the form hh:mm:ss  dd mmm yyyy. The format of the date component can be changed using the DATE.FORMAT command.
A single quote may be inserted in the footing by use of two adjacent single quotes in the text.

If more than one FOOTING definition is given in a single query, the first one is used. This allows a query sentence to include a footing that will override an alternative footing in the default listing phrase.

Example

The command

```
LIST SALES FOOTER "C'SALES REPORT'LDG'Confidential" HDR.SUP
```
might produce a report such as that below.

```
SALES..... REGION   SALESMAN  ORDER  VALUE
19845   North  Harris   234.53
19858   North  Sharp    845.50
19859   North  Sharp    384.43
19887   North  Roberts  279.40
19830   South  Smith    324.39
19886   South  Abbott   397.23
19866   South  Abbott   465.31

7 records listed.
```

```
SALES REPORT
26 Jun 2000              Confidential
```

See also:

**HEADING**
5.53 FOR

The **FOR** keyword used in a **SEARCH** command specifies that the search strings will be specified on the command line.

**Format**

```
FOR string1 string2 ...
```

Without this keyword, the **SEARCH** command prompts for the strings for which it is to search. The **FOR** keyword allows these strings to be specified on the command line.

The **FOR** keyword must be followed by at least one search string. Although not recommended, other query processor keywords may appear between the search strings but all literal values appearing later in the command that would otherwise be treated as record ids will be used as search strings. Where a search string includes a space, a quote or other characters that may cause ambiguity, the string should be enclosed in quotes.

**Example**

```
SEARCH BP FOR PARSER.H NO.CASE
```

This command builds a list of records in the BP file containing the string "PARSER.H" regardless of casing.

**See also:**

**NO.CASE, ALL.MATCH, NO.MATCH, SEARCH, STRINGS**
5.54  FORCE

The **FORCE** keyword forces output of page headings in an empty report.

**Format**

```
FORCE
```

A query report that finds no records to output normally shows only the zero record count. The **FORCE** option causes the page and column headings to be displayed in this situation.
5.55 FROM

The **FROM** keyword specifies the select list to be used as a source of record ids for processing by the query.

**Format**

```
FROM list.no | NONE
```

where

```
list.no
```

is the select list number (0 to 10) to be used.

If the **FROM** keyword is not present, the query processor will automatically use the default list, list 0, if it is active. Otherwise, all records in the file are processed.

Use of **FROM NONE** instructs the query processor to ignore the default select list.

**Example**

```
LIST STOCK FROM 4
```

This command lists items from the STOCK file using select list 4 as the source of record ids to be processed.
5.56 GE

The GE selection clause operator compares a field or evaluated expression against another field, evaluated expression or literal value and tests for the first item being greater than or equal to the second. The synonyms >= and => can be used.

Format

\[
\text{field GE} \{\text{NO.CASE}\} \text{ value}
\]

where

- \text{field} is the first field or evaluated expression to be compared.
- \text{value} is the second field, evaluated expression or literal value to be compared. The optional \text{NO.CASE} qualifier causes a case insensitive comparison to be applied.

The GE selection clause operator returns True if field is greater than or equal to value. For right aligned fields (typically numeric values) a numeric comparison will be used if the field content and value are both integer values. The QUERY.STR.COMP.ALL modes of the \text{OPTION} command can be used to apply string character comparisons to all data.

Example

\[
\text{LIST STOCK WITH QTY GE 100}
\]

This command lists items found on the STOCK file with a QTY field of 100 or over.
5.57 GRAND.TOTAL

The GRAND.TOTAL keyword (synonyms GRAND-TOTAL and CAPTION) specifies text to appear at the left edge of the grand total line when any field accumulations (AVERAGE, ENUMERATE, MAX, MIN, PERCENTAGE or TOTAL) are used.

Format

GRAND.TOTAL "text"

where

text is the text to appear on the grand total line.

The GRAND.TOTAL text may include control options enclosed in single quotes. These are:

L Suppress the grand total line completely (see also NO.GRAND.TOTAL).
P Print the grand total line on a new page.
U Toggle inclusion of the underline above the grand total values.

The default form of the grand total is to show a line of equals signs (=) above the total values. The grand total text appears on the left of this line.

The PICK.GRAND.TOTAL keyword of the OPTION command provides closer compatibility with Pick style systems where there is no underline and the text is displayed on the line that holds the total values.

The U control option inverts inclusion of the underline. Without PICK.GRAND.TOTAL, this mode omits the underline. With PICK.GRAND.TOTAL, the underline is shown.

Example

The command

LIST SALES TOTAL ORDER.VALUE GRAND.TOTAL "TOTAL"

might produce the report below.

LIST SALES TOTAL ORDER.VALUE GRAND.TOTAL "TOTAL"  
Page 1
SALES..... ORDER VALUE
19845  234.53
19858  845.50
19859  384.43
19887  279.40
19830  324.39
19886  397.23
19866  465.31
TOTAL ===========
          2930.79

7 records listed.
See also:

NO.GRAND.TOTAL
5.58 GT

The GT selection clause operator compares a field or evaluated expression against another field, evaluated expression or literal value and tests for the first item being greater than the second. The synonyms AFTER, GREATER and > can be used.

Format

\[ \text{field GT \{NO.CASE\} value} \]

where

- \( \text{field} \) is the first field or evaluated expression to be compared.
- \( \text{value} \) is the second field, evaluated expression or literal value to be compared. The optional \text{NO.CASE} qualifier causes a case insensitive comparison to be applied.

The GT selection clause operator returns True if \( \text{field} \) is greater than \( \text{value} \). For right aligned fields (typically numeric values) a numeric comparison will be used if the field content and \( \text{value} \) are both integer values. The QUERY.STR.COMP.ALL modes of the \text{OPTION} command can be used to apply string character comparisons to all data.

Example

\[ \text{LIST STOCK WITH QTY GT 100} \]

This command lists items found on the STOCK file with a QTY field of over 100.
5.59 HDR.SUP

The **HDR.SUP** display clause keyword suppresses the default page heading. The synonyms **HDR-SUPP** and **SUPP** can be used.

**Format**

```
HDR.SUP
```

The **HDR.SUP** keyword suppresses the default page heading (the command that invoked the query processor). Any heading specified by use of the **HEADING** keyword will still be output.

**Example**

The command

```
LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAID
```

would normally produce a display such as

```
LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAID
Invoice ...Value Customer.........................
  74529   £1712.43  J McTavish
  74273   £95.23  County Newspapers
  63940   £141.00  R Bryant
  74993    £9.29  Write Right Stationery
4 records listed.
```

Adding the **HDR.SUP** option, the command

```
LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAID HDR.SUP
```

would produce a display such as

```
Invoice ...Value Customer.........................
  74529   £1712.43  J McTavish
  74273   £95.23  County Newspapers
  63940   £141.00  R Bryant
  74993    £9.29  Write Right Stationery
4 records listed.
```
5.60 HEADING

The HEADING keyword defines a page heading for the report. The synonym HEADER may be used.

Format

HEADING "text"

where

`text` is the heading text to appear on each page.

A page heading defined with the HEADING keyword will appear at the top of each page of the report in place of the default heading. Except when used with the SHOW verb, the heading text may contain control codes enclosed in single quotes to insert variable data or to alter the appearance of the heading. These control codes are:

- **B{n}** Insert data from the corresponding B control code in a BREAK.ON or BREAK.SUP option string. The optional single digit qualifier, `n`, defaults to zero if omitted.
- **C** Centres the current line of the heading text.
- **D** Inserts the date. The default format is dd mmm yyyy (e.g. 24 Aug 2005) but can be changed using the DATE.FORMAT command.
- **F{n}** Inserts the file name in a field of `n` spaces. If `n` is omitted, a variable width is used.
- **G** Inserts a gap. Spaces are inserted in place of the G control code to expand the text to the width of the output device. If more than one G control code appears in a single line, spaces are distributed as evenly as possible.
  
  When a heading line uses both G and C, the heading is considered as a number of elements separated by the G control options. The element that contains the C option will be centered. The items either side of the centered element are processed separately when calculating the number of spaces to be substituted for each G option.
- **H{n}** Sets horizontal position (column) numbered from one. Use of H with C or with a preceding G token may have undesired results.
- **I{n}** Inserts the record id in a field of `n` spaces. If `n` is omitted, a variable width is used.
- **L** Inserts a new line at this point in the text.
- **N** Suppresses pagination of the output to the display.
- **O** Reverses the elements separated by G tokens in the current line on even numbered pages. This is of use when printing double sided reports.
- **P{n}** Insert page number. The page number is right justified in `n` spaces, widening the field if necessary. If omitted, `n` defaults to four.
- **R{n}** Same as I{n}.
- **S{n}** Insert page number. The page number is left justified in `n` spaces, widening the field if necessary. If omitted, `n` defaults to one.
- **T** Inserts the time and date in the form hh:mm:ss dd mmm yyyy. The format of the date component can be changed using the DATE.FORMAT command.
A single quote may be inserted in the heading by use of two adjacent single quotes in the text.

If more than one **HEADING** definition is given in a single query, the first one is used. This allows a query sentence to include a heading that will override an alternative heading in the default listing phrase.

**Example**

The command

```
LIST SALES HEADING "'C'SALES REPORT'LDG'Confidential" HDR.SUP
```

might produce a report such as that below.

<table>
<thead>
<tr>
<th>SALES REPORT</th>
<th>Confidential</th>
</tr>
</thead>
<tbody>
<tr>
<td>26 Jun 2000</td>
<td></td>
</tr>
<tr>
<td>SALES......</td>
<td>REGION</td>
</tr>
<tr>
<td>19845</td>
<td>North</td>
</tr>
<tr>
<td>19858</td>
<td>North</td>
</tr>
<tr>
<td>19859</td>
<td>North</td>
</tr>
<tr>
<td>19887</td>
<td>North</td>
</tr>
<tr>
<td>19830</td>
<td>South</td>
</tr>
<tr>
<td>19886</td>
<td>South</td>
</tr>
<tr>
<td>19866</td>
<td>South</td>
</tr>
</tbody>
</table>

7 records listed.

**See also:**

[FOOTING](#)
5.61 ID.ONLY

The ID.ONLY keyword causes the query processor to ignore the default listing phrase and show only record ids. The synonym ONLY may be used.

Format

ID.ONLY

Used with no field names on the command line, the ID.ONLY keyword causes the query processor to ignore the default listing phrase (@ or @LPTR) and to show the record its only. If field names are given on the command line this keyword has no effect.

For compatibility with other multivalue products, the ID.ONLY keyword may appear before the filename in a query processor sentence. For example:

LIST ID.ONLY ORDERS
5.62 ID.SUP

The ID.SUP display option keyword causes the record id to be omitted from the report. The synonym ID-SUPP may be used.

Format

ID.SUP

The record id is normally included automatically as the first item in a LIST or SORT report. The ID.SUP keyword can be used to suppress this item either where it is not required or where it is named explicitly in the command so that, for example, non-standard display attributes can be used.

Used with LIST.ITEM or SORT.ITEM, the ID.SUP keyword suppresses display of the record id above the data.

Example

The command

    LIST STOCK DESCRIPTION QTY WITH QTY < REORDER.LEVEL

might produce a report such as

    LIST STOCK DESCRIPTION QTY WITH QTY < REORDER.LEVEL
    Item code  Description...............  Stock
    A17439     A4 four hole binder, black  12
    A50993     Adhesive tape, 25mm x 30m  3
    D94266     Fibre pens, 10 pack, blue  6
    3 records listed.

including the ID.SUP keyword would change this to become

    LIST STOCK DESCRIPTION QTY WITH QTY < REORDER.LEVEL ID.SUP
    Description...............  Stock
    A4 four hole binder, black  12
    Adhesive tape, 25mm x 30m  3
    Fibre pens, 10 pack, blue  6
    3 records listed.
5.63  **IN**

The **IN** selection clause operator tests whether the content of a field is in a named select list.

**Format**

\[ \text{field IN [NO.CASE] listname} \]
\[ \text{field IN [NO.CASE] "filename id"} \]

where

- **field** is the field or evaluated expression to be compared.
- **listname** is the name of an item previously saved to the $SAVEDLISTS file that holds a field mark delimited list of acceptable values for field. The optional **NO.CASE** qualifier causes a case insensitive comparison to be applied. The list name should be enclosed in quotes if it may clash with a dictionary or VOC item name.
- **filename** is the name of a file containing a record that holds a field mark delimited list of acceptable values for field. The optional **NO.CASE** qualifier causes a case insensitive comparison to be applied. The quotes around the **filename** and **id** are mandatory.
- **id** is the record id of the record in **filename** to be processed.

The **IN** selection clause operator causes the report output to contain only records where the content of field is in the list identified by listname or filename and id. When applied field is multivalued, at least one value must meet the condition. The **EVERY** qualifier can be used to test that all values are in the list.

**Example**

```
SELECT COUNTRIES SAVING SHORTCODE SAVE.LIST CTRY
```

This sequence of commands processes the COUNTRIES file to build a select list of country codes that it then saves as CTRY. We could then go on to do

```
LIST CLIENTS WITH COUNTRY IN CTRY
```

to list only CLIENTS file records that had a valid country code.
5.64 LABEL

The LABEL keyword specifies the label template record name for LIST.LABEL and SORT.LABEL.

Format

    LABEL  template.name

    LABEL  NO.DEFAULT

If this keyword is not present, the query processor looks for a record named @LABEL in the dictionary or, if not found, in the VOC. This record should contain a label page layout definition as described with the LIST.LABEL command.

The first format of the LABEL keyword specifies an alternative name of a label template record stored in the dictionary of the file being processed or in the VOC.

The second format specifies that the default @LABEL record is not to be used. The query processor will then prompt for the label page shape parameters. Prompts also appear if the LABEL keyword is not present and no @LABEL record is found.
5.65 LE

The LE selection clause operator compares a field or evaluated expression against another field, evaluated expression or literal value and tests for the first item being less than or equal to the second. The synonyms <= and =< can be used.

Format

\[ \text{field} \ LE \ \{\text{NO.CASE}\} \ value \]

where

- \textit{field} is the first field or evaluated expression to be compared.
- \textit{value} is the second field, evaluated expression or literal value to be compared. The optional NO.CASE qualifier causes a case insensitive comparison to be applied.

The LE selection clause operator returns True if \textit{field} is less than or equal to \textit{value}. For right aligned fields (typically numeric values) a numeric comparison will be used if the field content and \textit{value} are both integer values. The QUERY.STR.COMP.ALL modes of the \textbf{OPTION} command can be used to apply string character comparisons to all data.

Example

\[ \text{LIST STOCK WITH QTY LE 100} \]

This command lists items found on the STOCK file with a QTY field of 100 or less.
5.66 LIKE

The **LIKE** selection clause operator compares a field or evaluated expression against another field, evaluated expression or literal value and tests for the first item matching the pattern template given by the second. The synonyms **LIKE**, **MATCHES** and **MATCHING** can be used.

**Format**

```plaintext
field LIKE {NO.CASE} template
```

where

- **field** is the first field or evaluated expression to be compared.
- **template** is the field, evaluated expression or literal value representing the pattern against which **field** is to be compared. The optional **NO.CASE** qualifier causes a case insensitive comparison to be applied.

The **LIKE** selection clause operator returns True if **field** matches **template**.

The **LIKE** operator treats characters that do not correspond to any valid component of a pattern as literal values which must be matched exactly. Thus it is possible to find all the **QMBasic** include records (which have a suffix of .H) in the BP file by a command of the form

```plaintext
SELECT BP WITH @ID LIKE ....H
```

The initial three dots are a valid template component. The remaining two characters are not and are hence treated as literals. It would be better to enter this as

```plaintext
SELECT BP WITH @ID LIKE "...'.H'"
```

to avoid confusion. In some cases quotes must be used to handle literal values which are also valid components of a pattern template.

**Example**

```plaintext
LIST STOCK WITH PRODUCT.CODE LIKE A...
```

This command lists items found on the STOCK file with a PRODUCT.CODE starting with A.

**See also:**

*Pattern Matching*
5.67 LOCKING

The LOCKING keyword locks the file during a report, preventing updates.

Format

    LOCKING

The LOCKING keyword causes the query processor to take a file lock on the file named in the query sentence. This prevents any other process from modifying the file during the report, ensuring that the report reflects a snapshot view of the data at the point when the query began.

Files accessed using the TRANS(0) function in I-type dictionary records or using the T conversion code will not be locked and hence may be modified during the report.

It is the user's responsibility to ensure that use of the lock in lengthy reports does not cause operational problems. In particular, be aware that a user who leaves a query on a "Press return to continue" prompt may severely affect the ability of other users to access the file.
5.68 LPTR

The **LPTR** keyword directs the output of the query to a printer.

**Format**

```
LPTR { unit }
```

where

```
unit
```

is the print unit number. If omitted, print unit 0 is used.

The **LPTR** keyword directs the query processor output to the specified print unit. It also changes the way in which the default listing phrase operates. With this option, if no report fields are specified on the command line, the query processor first looks for a phrase named @LPTR and then, if this does not exist, it reverts to the @ phrase. This allows different default report formats for the printer and the display.

A query using the **LPTR** option normally closes the printer at the end of the report. When using the default print unit (printer 0), if the QUERY.MERGE.PRINT mode of the **OPTION** command is active and the printer was already active (**PRINTER ON**) before the query command started, the printer is left active at the end of the report. This allows an application program to embed a query report into other data printed by the program.
5.69 LT

The LT selection clause operator compares a field or evaluated expression against another field, evaluated expression or literal value and tests for the first item being less than the second. The synonyms BEFORE, LESS and < can be used.

Format

\[
\text{field \ LT \{NO.CASE\} value}
\]

where

\[
\begin{align*}
\text{field} & \quad \text{is the first field or evaluated expression to be compared.} \\
\text{value} & \quad \text{is the second field, evaluated expression or literal value to be compared. The optional NO.CASE qualifier causes a case insensitive comparison to be applied.}
\end{align*}
\]

The LT selection clause operator returns True if field is less than value. For right aligned fields (typically numeric values) a numeric comparison will be used if the field content and value are both integer values. The QUERY.STR.COMP.ALL modes of the OPTION command can be used to apply string character comparisons to all data.

Example

\[
\text{LIST STOCK WITH QTY LT 100}
\]

This command lists items found on the STOCK file with a QTY field of less than 100.
5.70 MARGIN

The MARGIN keyword specifies the width of a blank left margin to appear in the report output.

Format

\[ \text{MARGIN } width \]

where

\[ width \]

specifies the margin width in characters.

The MARGIN keyword allows a blank left margin to be produced at the edge of the report. It is intended for use when the printed report is to be bound and thus requires clear space at the left edge.
5.71 MAX

The MAX field qualifier keyword causes a field to be reported together with its maximum value.

Format

MAX field \{field.qualifiers\}

where

field is the field or evaluated expression to be displayed.

field.qualifiers are other field qualifying keywords

The MAX field qualifier keyword is placed before the field name to which it applies and causes the query processor to report the value of the field for each record processed and also to report the maximum value at the end of the report. Used with breakpoints, the MAX keyword will also report the maximum value of the field at each breakpoint.

If the field is defined as multivalued, the MAX keyword operates on each value in turn.

The MAX keyword operates on all types of data. Where the field holds non-numeric data, a string comparison is performed.

Used with the SHOW verb, the MAX keyword specifies the maximum number of records allowed in the resultant select list.

Example

The command

LIST INVOICES MAX VALUE CUSTOMER.NAME WITH NO AMT.PAID

would produce a display such as that below in which the maximum value of the VALUE field is repeated at the end of the report.

```
LIST INVOICES MAX VALUE CUSTOMER.NAME WITH NO AMT.PAID
Invoice ...Value Customer.........................
  74529  £1712.43  J McTavish
  74273  £95.23  County Newspapers
  63940  £141.00  R Bryant
  74993  £9.29  Write Right Stationery
        ========         £1712.43
4 records listed.
```
5.72 MEDIAN

The MEDIAN field qualifier keyword causes a field to be reported together with its median average value.

Format

MEDIAN field {field.qualifiers} {NO.NULLS}

where

field is the field or evaluated expression to be displayed.

field.qualifiers are other field qualifying keywords

NO.NULLS causes null values to be ignored.

The MEDIAN field qualifier keyword is placed before the field name to which it applies and causes the query processor to report the value of the field for each record processed and also to report the median value (the midpoint value) at the end of the report. Used with breakpoints, the MEDIAN keyword will also report the median value of the field at each breakpoint.

If the field is defined as multivalued, the MEDIAN keyword operates on each value in turn.

The MEDIAN keyword operates only on numeric data. Non-numeric values are ignored except that the default behaviour is to treat null items as zero. The NO.NULLS keyword can be used to prevent null values being included in the calculation of the median value.

The median is defined as the middle value in a sorted list of all values, including repeated values, in the data being averaged. Where there are an even number of values, the median value is the average of the two values in the centre of the list.

Example

The sentence

LIST INVOICES MEDIAN VALUE CUSTOMER.NAME WITH NO AMT.PAID

would produce a display such as that below in which the median value of the VALUE field is included at the end of the report.

| Invoice | ...Value | Customer | |
|---------|----------|----------|
| 74529   | £1712.43 | J McTavish |
| 74273   | £95.23   | County Newspapers |
| 28137   | £26.20   | J Higgins |
| 63940   | £141.00  | R Bryant |
| 74993   | £9.29    | Write Right Stationery |
|         |          | ========== |
|         |          | £95.23 |
| 5 records listed. | | |
AVERAGE, MODE, RANGE
The **MIN** field qualifier keyword causes a field to be reported together with its minimum value.

**Format**

```
MIN field [field.qualifiers] {NO.NULLS}
```

where

- `field` is the field or evaluated expression to be displayed.
- `field.qualifiers` are other field qualifying keywords
- `NO.NULLS` causes null values to be ignored.

The **MIN** field qualifier keyword is placed before the field name to which it applies and causes the query processor to report the value of the field for each record processed and also to report the minimum value at the end of the report. Used with breakpoints, the **MIN** keyword will also report the minimum value of the field at each breakpoint.

If the field is defined as multivalued, the **MIN** keyword operates on each value in turn.

The **MIN** keyword operates on all types of data. Where the field holds non-numeric data, a string comparison is performed.

The **NO.NULLS** keyword can be used to prevent null values being included in the test for the minimum value.

Used with the **SHOW** verb, the **MIN** keyword specifies the minimum number of records allowed in the resultant select list.

**Example**

The command

```
LIST INVOICES MIN VALUE CUSTOMER.NAME WITH NO AMT.PAID
```

would produce a display such as that below in which the minimum value of the VALUE field is repeated at the end of the report.

```
LIST INVOICES MIN VALUE CUSTOMER.NAME WITH NO AMT.PAID
  Invoice   ...Value  Customer............................
    74529     £1712.43  J McTavish
    74273     £95.23   County Newspapers
    63940     £141.00  R Bryant
    74993     £9.29    Write Right Stationery
          ======

£9.29
4 records listed.
```
5.74 MODE

The **MODE** field qualifier keyword causes a field to be reported together with its mode value.

**Format**

```
MODE field {field.qualifiers} {NO.NULLS}
```

where

- `field` is the field or evaluated expression to be displayed.
- `field.qualifiers` are other field qualifying keywords
- `NO.NULLS` causes null values to be ignored.

The **MODE** field qualifier keyword is placed before the field name to which it applies and causes the query processor to report the value of the field for each record processed and also to report the mode value (the most frequently occurring value) at the end of the report. Used with breakpoints, the **MODE** keyword will also report the mode value of the field at each breakpoint.

Where more than more than one value shares the most frequent occurrence, it is not defined which one will be shown.

If the field is defined as multivalued, the **MODE** keyword operates on each value in turn.

The **NO.NULLS** keyword can be used to prevent null values being included in the calculation of the mode value.

**Example**

The sentence

```
LIST SALES MODE ITEM DESCR
```

would produce a display such as that below in which the most frequently sold item code is shown at the bottom of the ITEM column.

<table>
<thead>
<tr>
<th>Sales</th>
<th>Item</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>12140</td>
<td>003</td>
<td>Pen, blue</td>
</tr>
<tr>
<td></td>
<td>122</td>
<td>Whiteboard, medium</td>
</tr>
<tr>
<td></td>
<td>234</td>
<td>Paper, 100g A4 5 x 500 sheets</td>
</tr>
<tr>
<td></td>
<td>121</td>
<td>Whiteboard, small</td>
</tr>
<tr>
<td>12347</td>
<td>004</td>
<td>Pen, green</td>
</tr>
<tr>
<td></td>
<td>131</td>
<td>Flipchart paper</td>
</tr>
<tr>
<td></td>
<td>232</td>
<td>Paper, 80g A4 5 x</td>
</tr>
<tr>
<td>...etc...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12012</td>
<td>031</td>
<td>Ruler, 6 inch</td>
</tr>
<tr>
<td></td>
<td>003</td>
<td>Pen, blue</td>
</tr>
<tr>
<td></td>
<td>===</td>
<td></td>
</tr>
<tr>
<td></td>
<td>003</td>
<td></td>
</tr>
</tbody>
</table>
See also:

AVERAGE, MEDIAN, RANGE
5.75 **MULTI.VALUE**

The **MULTI.VALUE** keyword is a field qualifier that forces the field to be processed as a multivalued item. The synonym **MULTIVALUED** can be used.

**Format**

```
field MULTI.VALUE
```

where

```
field
```

is the field or expression which is to be treated as multivalued.

The query processor verbs normally use the dictionary to determine whether a field should be treated as single or multivalued. The **MULTI.VALUE** field qualifier forces the field to be processed as a multivalued item regardless of the dictionary definition. It is normally only used with an **EVAL** expression.

**See also:**

**SINGLE.VALUE**
5.76 NE

The NE selection clause operator compares a field or evaluated expression against another field, evaluated expression or literal value and tests for the first item being not equal to the second. The synonyms NOT, #, <> and >< can be used.

Format

\[ field \ NE \ [NO.CASE] \ value \]

where

- \( field \) is the first field or evaluated expression to be compared.
- \( value \) is the second field, evaluated expression or literal value to be compared. The optional NO.CASE qualifier causes a case insensitive comparison to be applied.

The NE selection clause operator returns True if \( field \) is not equal to \( value \).

The default behaviour of the query processor is that if \( field \) is defined as left justified in the dictionary, an exact match string comparison is used. For right aligned fields (typically numeric values) a numeric comparison will be used if the field content and \( value \) are both integer values. The QUERY.STRING.COMP or QUERY.STR.COMP.ALL modes of the OPTION command can be used to apply string character comparisons to all data.

Example

```plaintext
LIST STOCK WITH QTY NE 0
```

This command lists items found on the STOCK file with a non-zero QTY. If QTY is defined as left justified, the content of this field must be the single digit 0 for the record to be included in the report. If QTY is defined as right justified, records for which QTY holds any numeric data equal to zero will be included (e.g. 00).

Pick Style Wildcards

If the PICK.WILDCARD option is enabled (see the OPTION command) and the \( value \) item is a literal value, the interpretation is extended to include use of Pick style wildcard characters:

- A [ character at the start of the \( value \) replaces any number of leading characters. It is equivalent to the … action of the UNLIKE operator.
- A ] character at the end of the \( value \) replaces any number of trailing characters. It is equivalent to the … action of the UNLIKE operator.
- A ^ character within the \( value \) replaces a single character. It is equivalent to the 1X action of the UNLIKE operator.
5.77 NEW.PAGE

The NEW.PAGE display option keyword causes each record in a report to start on a new page.

Format

    NEW.PAGE
5.78 NO

The NO selection clause operator tests whether a field or evaluated expression is null.

Format

\textbf{WITH NO} \textit{field}

where

\textit{field} is the field or evaluated expression to be tested.

The NO selection clause operator returns True if \textit{field} is null.

Example

\textbf{LIST STOCK WITH NO SUPPLIER\_CODE}

This command lists items found on the STOCK file with a null SUPPLIER\_CODE.
5.79 NOT

The NOT selection clause operator inverts the result of the immediately following condition.

**Format**

```
WITH NOT condition
```

where

- `condition` is a selection expression.

The NOT operator can be used to invert the result of a selection clause expression. For most expressions, it is easier and more readable to use the inverse operator but there are occasions when the NOT operator is the only way to achieve the desired effect.

```
LIST STAFF WITH NOT SURNAME = 'Smith'
```

is an alternative to

```
LIST STAFF WITH SURNAME # 'Smith'
```

If we wanted to use Soundex matching to include alternative spellings such as "Smyth", there is no inverse of the SAID operator but we could do

```
LIST STAFF WITH NOT SURNAME SAID 'Smith'
```

There is an important difference in QM’s implementation of the NOT operator and that of other multivalue products. A query such as

```
LIST ORDERS WITH STATUS = "P" "D"
```

uses the implied OR to test STATUS against a list of values. This is equivalent to

```
LIST ORDERS WITH STATUS = "P" OR STATUS = "D"
```

Inserting a NOT operator to invert the test

```
LIST ORDERS WITH NOT STATUS = "P" "D"
```

associates the NOT only with the first value and is equivalent to

```
LIST ORDERS WITH NOT STATUS = "P" OR STATUS = "D"
```

Other multivalue products treat the list of status values as a single operation and apply the NOT to the result. In QM, it is necessary to write this as

```
LIST ORDERS WITH NOT (STATUS = "P" "D")
```

The NOT keyword can also be used as a synonym for NE

```
LIST ORDERS WITH STATUS NOT "P"
```
5.80 NO.CASE

The NO.CASE keyword used in the SEARCH verb specifies that the string comparison is to be performed in a case insensitive manner.

Format

NO.CASE

The SEARCH verb is normally case sensitive in its comparison of the given search strings. The NO.CASE option performs a case insensitive comparison.

The NO.CASE keyword can also be used as a qualifier with relational operators in the selection clause. In this context, the QUERY.NO.CASE mode of the OPTION command can be used to imply case insensitivity.

See also:
ALL.MATCH, NO.MATCH, SEARCH
5.81 NO.GRAND.TOTAL

The `NO.GRAND.TOTAL` phrase suppresses the grand total in a query report.

Format

```
NO.GRAND.TOTAL
```

The `NO.GRAND.TOTAL` phrase suppresses the grand total line. It is only of use when the query includes breakpoints that produce subtotals.

Example

```
LIST ORDERS BY CUST.NO BREAK.ON CUST.NO TOTAL ORDER.VALUE
NO.GRAND.TOTAL
```

This is equivalent to

```
LIST ORDERS BY CUST.NO BREAK.ON CUST.NO TOTAL ORDER.VALUE
GRAND.TOTAL "'L'"
```

See also:

`GRAND.TOTAL`
5.82 **NO.INDEX**

The **NO.INDEX** keyword causes the query processor to ignore any index that would otherwise be used to handle the selection clause of the query.

**Format**

```
NO.INDEX
```

Use of the **NO.INDEX** keyword may result in faster query processing where the selection clause includes a large proportion of the records in the file.
5.83 NO.MATCH

The NO.MATCH keyword used in a SEARCH command specifies that the records to be selected must contain none of the given search strings.

Format

    NO.MATCH

Without this keyword, the SEARCH command builds a list of records containing any of the supplied search strings. With NO.MATCH, the records must contain none of the supplied strings.

Example

    SEARCH BP NONE.MATCH
    String: STOCK.FILE
    String: STK.F
    String:

This command builds a list of records in the BP file containing neither of the given strings.

See also:

ALL.MATCH, NO_CASE, SEARCH
5.84 NO.NULLS

The NO.NULLS keyword suppresses null items.

Format

NO.NULLS

The NO.NULLS keyword has three uses:

1. With the query processor AVERAGE, ENUMERATE, MEDIAN, MIN and MODE keywords, this keyword ignores null items.

2. With the query processor SAVING keyword, this keyword omits null field values from the generated select list.

3. With the CREATE_INDEX command, it omits records with null field values from the index.
5.85 NO.PAGE

The NO.PAGE display option keyword suppresses the normal page end prompt. The synonym NOPAGE may be used.

Format

    NO.PAGE

The LIST and SORT commands normally pause at the end of each page when output is directed to the display. The NO.PAGE keyword suppresses this page end prompt.
5.86 NO.SPLIT

The NO.SPLIT keyword causes the query processor to avoid splitting records across pages where possible.

Format

    NO.SPLIT

The NO.SPLIT causes the query processor to start a new page when there is insufficient space on the current page for the record about to be reported.
5.87 NOT.IN

The NOT.IN selection clause operator tests whether the content of a field is not in a named select list.

Format

field NOT.IN {NO.CASE} listname
field NOT.IN {NO.CASE} "filename id"

where

field is the field or evaluated expression to be compared.

listname is the name of an item previously saved to the $SAVEDLISTS file that holds a field mark delimited list of unacceptable values for field. The optional NO.CASE qualifier causes a case insensitive comparison to be applied. The list name should be enclosed in quotes if it may clash with a dictionary or VOC item name.

filename is the name of a file containing a record that holds a field mark delimited list of unacceptable values for field. The optional NO.CASE qualifier causes a case insensitive comparison to be applied. The quotes around the filename and id are mandatory.

id is the record id of the record in filename to be processed.

The NOT.IN selection clause operator causes the report output to contain only records where the content of field is not in the list identified by listname. When applied field is multivalued, at least one value must meet the condition. The EVERY qualifier can be used to test that all values are not in the list.

Example

SELECT COUNTRIES SAVING SHORTCODE
SAVE.LIST CTRY

This sequence of commands processes the COUNTRIES file to build a select list of country codes that it then saves as CTRY. We could then go on to do

LIST CLIENTS WITH COUNTRY NOT.IN CTRY

to list only CLIENTS file records that had an invalid country code.
5.88 **OR**

The **OR** selection clause operator links two selection criteria where either may be True for the record to be selected.

**Format**

```
WITH condition.1 OR condition.2
```

where

```
condition.1, condition.2
```

are record selection criteria.

The **OR** selection clause operator returns True if either or both of `condition.1` and `condition.2` are True.

The **AND** and **OR** operators are normally of equal priority and will be evaluated strictly left to right. Brackets may need to be used to enforce evaluation in a different order. Thus a query such as

```
LIST CLIENTS WITH REGION = 1 AND VALUE > 1000 OR REGION = 2 AND VALUE > 500
```

may need brackets to achieve the desired effect

```
LIST CLIENTS WITH (REGION = 1 AND VALUE > 1000) OR (REGION = 2 AND VALUE > 500)
```

Pick style multivalue database products give **AND** priority over **OR** such that the above query would not need the brackets. This behaviour can be enabled in QM by use of the `QUERY.PRIORITY.AND` mode of the **OPTION** command.

**Example**

```
LIST STOCK WITH QTY GT 100 OR REORDER LT 300
```

This command lists items found on the STOCK file with a QTY field of over 100 or a REORDER field of less than 300.
5.89 OVERLAY

The OVERLAY option specifies a catalogued subroutine to emit a graphical page overlay.

Format

    OVERLAY subr.name

where

    subr.name    is the name of a catalogued subroutine.

The OVERLAY option allows a report to include a graphical overlay to draw a form on each page of a report directed to a printer or a file. This is equivalent to use of the OVERLAY option of the SETPTR command except that it applies only to the one report.

The subr.name qualifier is the name of a catalogued subroutine that will emit the page overlay. This subroutine takes a single argument, the print unit number, and should not perform any other action.

Example

    LIST ORDERS OVERLAY ORD.OV LPTR

This command lists the ORDERS file, overlaying each page with a graphical image generated by the ORD.OV subroutine.
5.90 PAGESSEQ

The PAGESSEQ option identifies a record that controls page numbering.

Format

PAGESSEQ filename id

where

filename is the name of the file containing the control record.

id is the record id of the control record.

The PAGESSEQ option provides a way in which successive uses of the same report can produce sequentially numbered pages allowing, for example, separate monthly business reports to be assembled into a single item. The option is ignored for reports directed to the screen.

The PAGESSEQ option specifies the name of a file and a record within that file. This record should contain a single field which holds the page number to be applied to the first page of the report. The query processor will retain a lock on this record for the duration of the query and will update it on completion to contain a value one greater than the number of the final page in the report.

If the control record does not exist, a default page number of 1 is used for the first page and the record will be created when the query terminates.

Example

The command

LIST MONTHLY.INVOICES PAGESSEQ SEQFILE INV LPTR

would produce a report of the MONTHLY.INVOICES file, using the value in SEQFILE INV as the start page number and updating this value on completion of the report.
The **PAN** keyword, used in reports directed to the display, permits the total width of the report to exceed that of the display by allowing the user to pan columns.

**Format**

```
PAN
```

The **PAN** keyword causes the query processor to buffer the report page and allows the user to pan columns using the left and right cursor key. The panning operation never displays only part of a column.

The **PAN** keyword operates differently depending on where it is placed in the query sentence. If it appears before or after all the displayed fields, the entire display is panned. If it appears between displayed fields, only those fields following the keyword are panned; the remaining fields being locked in position.

The L and R keys or Ctrl-B and Ctrl-F can be used in place of the cursor keys.
5.92 PERCENTAGE

The PERCENTAGE field qualifier keyword causes a field to be reported as a percentage of the total of the value of the field in all selected records. Short forms PERCENT, PCT or %.

Format

PERCENTAGE {dp} field {field.qualifiers}

where

dp is the number of decimal places to be displayed. This defaults to zero if omitted.

field is the field or evaluated expression to be displayed.

field.qualifiers are other field qualifying keywords

The PERCENTAGE field qualifier keyword is placed before the field name to which it applies and causes the query processor to report the value of the field for each record processed as a percentage of the total value of the field in all selected records. The total percentage (always 100 unless the data has changed during execution of the query) is shown at the end of the report. Used with breakpoints, the PERCENTAGE keyword will also report the percentage value at each breakpoint.

If the field is defined as multivalued, the PERCENTAGE keyword operates on each value in turn.

The PERCENTAGE keyword operates only on numeric data. Non-numeric values are treated as zero.

Example

The command

```
LIST INVOICES TOTAL VALUE PCT VALUE CUSTOMER.NAME WITH NO AMT.PAID
```

would produce a display such as that below.

```
LIST INVOICES TOTAL VALUE PCT VALUE CUSTOMER.NAME WITH NO AMT.PAID
Invoice  ...Value  Value  Customer.........................
74529     £1712.43  87  J McTavish
74273     £95.23    5  County Newspapers
63940     £141.00   7  R Bryant
74993     £9.29     1  Write Right Stationery
          ========== =====
£1957.95   100
```

4 records listed.
5.93 RANGE

The RANGE field qualifier keyword causes a field to be reported together with its range value.

Format

```
RANGE field {field.qualifiers} [NO.NULLS]
```

where

- `field` is the field or evaluated expression to be displayed.
- `field.qualifiers` are other field qualifying keywords
- `NO.NULLS` causes null values to be ignored.

The RANGE field qualifier keyword is placed before the field name to which it applies and causes the query processor to report the value of the field for each record processed and also to report the range value (the difference between the minimum and maximum value) at the end of the report. Used with breakpoints, the RANGE keyword will also report the range value of the field at each breakpoint.

Where more than more than one value shares the most frequent occurrence, it is not defined which one will be shown.

If the field is defined as multivalued, the RANGE keyword operates on each value in turn.

The RANGE keyword operates only on numeric data. Non-numeric values are ignored except that the default behaviour is to treat null items as zero. The NO.NULLS keyword can be used to prevent null values being included in the calculation of the range value.

Example

The sentence

```
LIST INVOICES RANGE VALUE CUSTOMER.NAME WITH NO AMT.PAID
```

would produce a display such as that below in which the range value of the VALUE field is included at the end of the report.

```
LIST INVOICES RANGE VALUE CUSTOMER.NAME WITH NO AMT.PAID
Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customer.......................           Invoice  ...Value  Customers...
5.94 REQUIRE.INDEX

The **REQUIRE.INDEX** keyword causes the query processor to terminate the query unless it can make use of an alternate key index.

**Format**

```
REQUIRE.INDEX
```

Use of the **REQUIRE.INDEX** keyword can help determine whether a query has been phrased in a manner that can make use of an alternate key index. If it cannot, a diagnostic message explaining the reason is displayed.
5.95 REQUIRE.SELECT

The **REQUIRE.SELECT** keyword indicates that the query should only proceed if there is an active select list.

**Format**

```
REQUIRE.SELECT
```

The **REQUIRE.SELECT** keyword is useful in automated queries from paragraphs, etc. where a preceding **SELECT** might have found no items and hence not left an active list. The following query would therefore process all records instead of none.

If no select list is active when a query using this keyword is initiated, an error message is displayed.

**Example**

```
SELECT ORDERS WITH VALUE > 1000 SAVING UNIQUE CUST.NO
0 records selected to list 0
LIST CUSTOMERS REQUIRE.SELECT
Select list required - Processing terminated
```

The above sequence shows how the **REQUIRE.SELECT** keyword causes the query processor to terminate the **LIST** operation when no records were found matching the selection criteria. Without this keyword, the **LIST** would have reported all the customers.
5.96 REPEATING

The **REPEATING** keyword causes the query processor to repeat single valued data against further values in other fields.

**Format**

```
REPEATING
```

In a report that includes multivalued fields, the value of any single valued items normally only appears once. The **REPEATING** keyword duplicates the single valued items against each multivalued element of other fields. It is related to the **RPT** field qualifier that can be applied to specific single-valued items included in the report.

The decision as to whether an item is single or multivalued is based on the S/M flag in the D/I-type dictionary definition or use of the equivalent field qualifiers, not by whether the data includes value marks. This is to ensure that a multivalued field with only a single entry does not get repeated.

Pick style A/S-type dictionary items are always treated as multivalued.

**Example**

A file containing a multivalued list of order numbers corresponding to each customer might produce a report that includes the following section:

```
LIST CUST.SALES ORDER.NO HDR.SUP
Customer   Order No
1447       10045
1587       10051
             10059
```

Using the **REPEATING** keyword changes this to:

```
LIST CUST.SALES ORDER.NO HDR.SUP REPEATING
Customer   Order No
1447       10045
1587       10051
1587       10059
```

See also: **BY.EXP, RPT**
5.97 REPORTING

The **REPORTING** keyword causes the query processor to display progress information when doing **COUNT** or **SELECT** with selection criteria.

**Format**

```
REPORTING {interval}
```

where

```
interval
```

is the number of records selected between each update of the display. This defaults to 200 if omitted.

This keyword causes the query processor to periodically display a line of the form

```
12600/28658 (17%)
```

where the first number is the number of records selected, the second number is the number of records examined, and the third number is the approximate percentage of the data records processed.

The **REPORTING** keyword is ignored in query processor operations to which it does not apply.
The RPT keyword is a field qualifier that repeats a single-valued item on each line.

**Format**

```
field RPT
```

where

- `field` is the field or expression which is to be repeated.

The RPT field qualifier applied to a single-valued item causes the item value to be repeated on each line when the report includes multi-valued items in other columns. It is related to the REPEATING keyword which is equivalent to using RPT on every single-valued item included in the report.

Use of RPT with a data item that is defined in the dictionary as being multivalued either explicitly (C/D/E/I-type) or implicitly (A/S-type) forces the query processor to treat the data as single valued.

**Example**

A file containing a multivalued list of order numbers corresponding to each customer might produce a report that includes the following section:

```
LIST CUST.SALES DATE ORDER.NO HDR.SUP
Customer   Date       Order No
1447       12 Jan 19  10045
1587       14 Jan 19  10051
               10059
```

Using the RPT field qualifier on the DATE field changes this to:

```
LIST CUST.SALES DATE RPT ORDER.NO HDR.SUP
Customer   Date       Order No
1447       12 Jan 19  10045
1587       14 Jan 19  10051
               15 Jan 19  10059
```

See also: **REPEATING**
The **SAID** selection clause operator compares a field or evaluated expression against another field, evaluated expression or literal value and tests for the first item having the Soundex phonetic code given by the second. The synonyms **SPOKEN** and ~ can be used.

**Format**

\[
\text{field \ SAID \ value}
\]

where \( \text{field} \) is the first field or evaluated expression to be compared. \( \text{value} \) is the second field, evaluated expression or literal value to be compared.

The **SAID** selection clause operator returns True if the Soundex phonetic code for \( \text{field} \) is \( \text{value} \).

The Soundex phonetic code for a word is made up from the first letter of the word in upper case followed by three digits which are found by examination of further characters of the word according to the following table.

<table>
<thead>
<tr>
<th>Digit</th>
<th>Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A E H I O U W Y</td>
</tr>
<tr>
<td>1</td>
<td>B F P V</td>
</tr>
<tr>
<td>2</td>
<td>C G J K Q S X Z</td>
</tr>
<tr>
<td>3</td>
<td>D T</td>
</tr>
<tr>
<td>4</td>
<td>L</td>
</tr>
<tr>
<td>5</td>
<td>M N</td>
</tr>
<tr>
<td>6</td>
<td>R</td>
</tr>
</tbody>
</table>

Other letters such as accented vowels are ignored. Consecutive letters that result in the same value result in only a single character. If the result is less than four characters long, zeros are added to fill the remaining positions. Thus the word SOUNDEX encodes to S532.

**Example**

```
LIST STAFF WITH SURNAME SAID 'REED'
```

This command names in the STAFF file that sound like Reed (Read, Reid, etc).
5.100 SAMPLE

The **SAMPLE** selection clause keyword causes only a limited number of records to be selected or displayed. The synonym **FIRST** can be used in place of **SAMPLE**.

**Format**

```
SAMPLE {n}
```

where

```
n
```

is the number of records to be processed. If $n$ is omitted, this defaults to 10.

The **SAMPLE** keyword causes selection or listing of records to terminate after the given number of records have been found.

Sampling occurs after selection criteria but before sorting. Thus it cannot be used to show, for example, only the first three records in sorted order of the whole file. See the **SORTED.SAMPLE** keyword for a way to do this.

**Example**

```
LIST STOCK WITH QTY > 100 SAMPLE 5
```

This command lists the first 5 items found on the STOCK file that have a QTY field of over 100.

**See also:**

**SORTED.SAMPLE**
5.101 SAMPLED

The **SAMPLED** selection clause keyword causes only a proportion of records to be selected or displayed.

**Format**

```
SAMPLED {n}
```

where

\[ n \]

is the sample interval. If \( n \) is omitted, this defaults to 10.

The **SAMPLED** keyword causes only every \( n \)'th record meeting the selection criteria (if any) to be selected or listed.

**Example**

```
LIST STOCK WITH QTY > 100 SAMPLED 5
```

This command lists every fifth item found on the STOCK file with a QTY field of over 100.
5.102 SAVING { UNIQUE}

The SAVING clause can be used in a SELECT or SSELECT command to save the content of a field in place of the record id.

Format

SAVING {UNIQUE} {MULTI.VALUE} field.name {NO.NULLS}

where

UNIQUE specifies that duplicate values are not to be repeated in the saved list.
MULTI.VALUE specifies that values and subvalues in the field are to be saved as separate list entries. The alternative spelling, MULTIVALUED, may be used.
field.name is the field or evaluated expression to be saved.
NO.NULLS causes null values to be omitted from the saved list.

The SAVING clause changes the action of SELECT or SSELECT to save the content of a field (D or I-type) or evaluated expression into the target select list in place of the record id. It is normally used to save fields which are ids of records in some other file.

Use of the UNIQUE keyword suppresses multiple inclusion of the same field value in the list.

For compatibility with other products, the SAVING clause normally saves only the first value in a multivalued field. Use of the MULTI.VALUE keyword causes each value or subvalue to be inserted in the list as a separate entry.

Also for compatibility with Pick style systems, a command such as

SELECT ORDERS PART.NO

where PART.NO is a field name is equivalent to

SELECT ORDERS SAVING MULTI.VALUE PART.NO

Example

SELECT INVOICES SAVING UNIQUE SITE.REF

This command creates a save list of all the site references appearing in the invoices file. The UNIQUE keyword ensures that site references only appear once regardless of the number of invoices that refer to them.
5.103 SCROLL

The SCROLL keyword used in a report directed to the display enables scrolling back through report pages.

Format

```
SCROLL \{ pages \}
```

where

```
pages
```

The `pages` qualifier is retained for backward compatibility but the value of `pages` is not significant except that a value of zero will disable scrolling, perhaps enabled by use of the SCROLL keyword in the $QUERY.DEFAULTS record.

Use of the SCROLL keyword allows paging back through a displayed report. The following options are available at the end of page prompt:

- **A** Abort. The query is terminated, returning to the command prompt. The ON.ABORT paragraph will be executed, if it exists.
- **Q** Quit. The query is terminated, returning to the menu or paragraph from which the query was initiated or to the command prompt.
- **N** Next. The next page of the report is shown. The cursor down key or ctrl-N can also be used.
- **P** Previous. The previous page of the report is shown. The cursor up key, ctrl-P or ctrl-Z can also be used.
- **n** Page number. The specified page number is shown. Note that when used with the N breakpoint option which resets page numbers, the numbering used by scroll is the physical page number within the report which may be different from the page number printed in the heading or footing.
- **C** Continue. Used when the display is showing a saved page, this continues with the first unseen page.
- **S** Suppress pagination. The query continues with no further screen pagination.
5.104 SINGLE.VALUE

The SINGLE.VALUE keyword is a field qualifier that forces the field to be processed as a single-valued item.

Format

\[ field \ \text{SINGLE.VALUE} \]

where

\[ field \] is the field or expression which is to be treated as single-valued.

The query processor verbs normally use the dictionary to determine whether a field should be treated as single or multivalued. The SINGLE.VALUE field qualifier forces the field to be processed as a single-valued item regardless of the dictionary definition. It is normally only used with an EVAL expression.

See also:
MULTI.VALUE
The `SORT.SUP` keyword suppresses the default sort by record id in the `SHOW` command.

**Format**

```
SORT.SUP
```

The default action of the `SHOW` command if no sort clause is present is to show records sorted by record id. The `SORT.SUP` keyword suppresses this sort. It is useful where a sorted select list is being used to determine the records to be processed.

The keyword is ignored if the query includes a sort clause.

**Example**

```
SELECT SALES BY CUSTOMER
SHOW SALES SORT.SUP
```

This command displays records in customer order.
The **SORTED.SAMPLE** selection clause keyword causes only a limited number of records to be selected or displayed. The synonym **SORTED.FIRST** can be used in place of **SORTED.SAMPLE**.

### Format

```
SORTED.SAMPLE {n}
```

where

- \( n \) is the number of records to be processed. If \( n \) is omitted, this defaults to 10.

The **SORTED.SAMPLE** keyword causes selection or listing of records to terminate after the given number of records have been found.

The record count restriction is applied after sorting. Thus it can be used to show, for example, only the first three records in sorted order of the whole file.

The keyword is ignored if the query does not include a sort clause.

### Example

```
LIST STOCK BY PRICE SORTED.SAMPLE 5
```

This command lists the first 5 lowest priced items found on the STOCK file.

### See also:

[SAMPLE](#)
5.107 STRINGS

The STRINGS keyword used in a SEARCH command specifies the file and record from which the search strings are to be taken.

Format

    STRINGS file.name record.id

Without this keyword, the SEARCH command prompts for the strings for which it is to search. The STRINGS keyword allows these strings to be taken from a file.

The file should be formatted such that each field represents a separate search string. Take care to avoid blank lines as these will be taken as being search strings.

Example

    SEARCH BP STRINGS SRCH.DATA PROGS

This command builds a list of records in the BP file containing any of the strings found in the PROGS record of the SRCH.DATA file.

See also:
NO.CASE, ALL.MATCH, FOR, NO.MATCH, SEARCH
5.108 STYLE

The STYLE keyword selects the report style to be used, overriding any other style selection methods.

Format

STYLE name

where

name is the name of a VOC style record. Use of STYLE NONE will disable other style selection mechanisms.

Each line of a report falls into one of the following classifications: Heading, Column heading, Detail, Subtotal, Total, Footing, Other.

Report styles allow users to attribute each of these classifications a colour for a displayed report or a font weight for a report directed to a PCL printer. An additional style, Exit, is used to determine how the screen is left on exit from the query processor. If this is absent, the query processor sends the terminfo sgr0 code to turn off all display attributes.

Report styles are defined using an X-type record in either the dictionary or the VOC where fields 2 onwards consist of a line classification, foreground colour, background colour and font weight in the form:

   Heading=Bright blue,Black,Bold

Only the first character of the line classification name is used. Thus the above line could be written as

   H=Bright blue,Black,Bold

The colour names are taken from the following list:
   Black, Blue, Green, Cyan, Red, Magenta, Brown, White, Grey, Bright Blue, Bright Green, Bright Cyan, Bright Red, Bright Magenta, Yellow, Bright White

Any non-alphabetic characters are ignored. Thus Bright Green can also be written as, for example, Bright-Green, Bright-Green or BrightGreen. Numeric colour values of 0 to 15 can be used where these correspond to the order of the colour names above.

Note that the colour palette used by AccuTerm may need to be amended from its default settings to improve the rendering of the non-bright colours.

Font weights are taken from the list defined in SYSCOM $PCLDATA which defaults to:

Any non-alphabetic characters are ignored in the same way as for colour names. Numeric font weight values in the range -7 to +7 can be used where these correspond to the order of the font weight names above.

All components of a style definition are case insensitive.
Any classification not defined in the style record, or any omitted component within a classification, takes on the values of the Other classification which itself defaults to White foreground, Black background, Medium font weight if not defined.

When the DET.SUP keyword is used, subtotals are reported using the detail line style.

Style selection uses the following sequence:
1. Use of the STYLE option on the command line to reference a VOC record
2. The optional $STYLE dictionary record
3. A style set for the print unit using the STYLE option of SETPTR
4. A default style set using REPORT.STYLE

Example

X
H=Bright Blue,,Bold
S=Blue
T=Bright Red,,Bold

In this example, headings are displayed with bright blue foreground colour and the default background colour. When sent to a PCL printer, the headings will be in bold face. Subtotals are displayed in blue, using the default background colour and printer font weight. Totals are displayed with bright red foreground colour and the default background colour, using bold face when printed.

See also:
REPORT.STYLE
5.109 TO (Selection verbs)

The TO keyword used in a SELECT, SSELECT or SEARCH command specifies the select list to be created.

**Format**

```
TO list.no
```

where

```
list.no
```

is the select list number (0 to 10) to be created.

If the TO keyword is not present, the default list, list 0, will be created.

**Example**

```
SELECT STOCK WITH COST > 100 TO 4
```

This command builds a list of records in the STOCK file with the COST field greater than 100, placing the resultant record ids in select list 4.
5.110 TO (REFORMAT)

The TO keyword used in a REFORMAT command specifies the name of the output file.

Format

TO new.file.name
TO DATA

where

new.file.name is the name of an existing file into which the REFORMAT output is to be written.

If the TO keyword is not present, the REFORMAT command prompts for the file name.

Use of TO DATA causes the query processor to create a dynamic array containing the output. This can be accessed via the @DATA variable. Each "record" in this array is separated by an item mark character (char 255) and the fields within the item correspond to the display clause elements in the REFORMAT command.

Examples

REFORMAT CUSTOMERS ZIP.CODE CUST.NO.NAME TO CUST.BY.ZIP

This command constructs a new file, CUST.BY.ZIP, keyed by zip code and containing two data fields, the customer number and name. Note that if two or more customers share the same zip code, the record will be overwritten by the second and subsequent items.

EXECUTE "REFORMAT CUSTOMERS ZIP.CODE CUST.NO.NAME TO DATA"
LOOP
   S = REMOVEF(@DATA, @IM)
UNTIL STATUS()
   ...processing...
REPEAT

This is the same command executed from within a QMBasic program, directing the output to the @DATA variable. The loop after the EXECUTE then extracts each item mark delimited entry and processes it.
5.111 TOTAL

The TOTAL field qualifier keyword causes a field to be reported together with its total value.

Format

\[ \text{TOTAL field \{field.qualifiers\}} \]

where

- \text{field}\quad \text{is the field or evaluated expression to be displayed.}
- \text{field.qualifiers}\quad \text{are other field qualifying keywords}

The TOTAL field qualifier keyword is placed before the field name to which it applies and causes the query processor to report the value of the field for each record processed and also to report the total value at the end of the report. Used with breakpoints, the TOTAL keyword will also report the total value of the field at each breakpoint.

If the field is defined as multivalued, the TOTAL keyword operates on each value in turn.

The TOTAL keyword operates only on numeric data. Non-numeric values are ignored. If the values are a mixture of numerics and non-numerics, the total of the numerics will be reported. If all the values are non-numeric, zero will be reported.

When used with breakpoints, the NO.GRAND.TOTAL keyword can be used to suppress the grand total line, leaving only the subtotals for each breakpoint.

Example

The command

\[ \text{LIST INVOICES TOTAL VALUE CUSTOMER.NAME WITH NO AMT.PAID} \]

would produce a display such as that below in which the total value of the VALUE field is included at the end of the report.

\begin{verbatim}
LIST INVOICES TOTAL VALUE CUSTOMER.NAME WITH NO AMT.PAID
Invoice     ...Value Customer.......................  
74529     £1712.43  J McTavish            
74273     £95.23 County Newspapers     
63940     £141.00 R Bryant                
74993     £9.29 Write Right Stationery  
          =======
          £1957.95
4 records listed.
\end{verbatim}
5.112 UNLIKE

The UNLIKE selection clause operator compares a field or evaluated expression against another field, evaluated expression or literal value and tests for the first item not matching the pattern template given by the second. The synonym NOT.MATCHING can be used.

Format

\[
\text{field UNLIKE } \{\text{NO.CASE}\} \text{ template}
\]

where

- \textit{field} is the first field or evaluated expression to be compared.
- \textit{template} is the field, evaluated expression or literal value representing the pattern against which \textit{field} is to be compared. The optional NO.CASE qualifier causes a case insensitive comparison to be applied.

Example

LIST STOCK WITH PRODUCT.CODE UNLIKE A...

This command lists items found on the STOCK file with a PRODUCT.CODE not starting with A.

See also:
Pattern Matching
5.113 USING

The USING clause allows a query to be processed using the dictionary of another file.

Format

USING \{ DICT \} file.name

where

DICT specifies that the dictionary of the named file is to be used.

file.name is the file to be used as the dictionary for the query.

The USING keyword allows a query to be processed with an alternative dictionary. It is of particular use where files share a dictionary.

The query processor uses the dictionary of the file being reported until the USING clause is encountered. All subsequent command line items are parsed using the specified dictionary. It is therefore usual to place the USING clause immediately after the query file name.

Example

LIST ARCHIVED.CUSTOMERS USING DICT CUSTOMERS

This command lists an archive file of customer data using the dictionary of the main CUSTOMERS file.
5.114 VERTICALLY

The VERTICALLY display option keyword causes a vertical format report to be produced. The synonym VERT may be used.

**Format**

```
VERTICALLY {FMT {"name.format"} {, "data.format"}}
```

where

- `name.format` is an alternative format to be applied to the field heading text.
- `data.format` is a default format to be applied to all data items except those that are defined in the dictionary as being right justified. If `data.format` is specified but `name.format` is omitted. the comma is still required.

The LIST and SORT commands normally produce a tabular report unless the total width of the data to be reported exceeds that of the display or printer to which it is directed. The VERTICALLY keyword forces a vertical format report regardless of display width. A blank line is produced between each record in the report.

In the absence of any of the overrides discussed here, the default action of the query processor is to show the field heading text (display name) as a left justified 12 character wide column, effectively using a format code of "L#12". The optional FMT qualifier to the VERTICAL keyword can be used to change this. Note that the default code uses a mask rather than "12.L" which would insert text marks if the name was longer than 12 characters.

These defaults can be overridden by including a record named $VLIST in the dictionary of the file being processed. This must have a type code of X in field 1. If field 2 is not blank, the content is used as the default name format. If field 3 is not blank, the content is used as the default data format. This will impact all vertical mode reports from the data file(s) that use this dictionary. Placing the $VLIST record in the VOC will apply the revised format codes to queries against all files except those for which there is a $VLIST record in the dictionary.

For either use of the $VLIST record or the FMT qualifier, specifying the `name.format` as the word FIT (case insensitive) will apply a name format that exactly fits the longest field display name in the report.

The FMT qualifier also allows a default format to be applied to all data items included in report with the exception of those that are defined in the dictionary as being right justified. Thus, for example, a field defined in the dictionary as having a format of "20T" may be good for tabular reports but introduce unwanted line breaks in a vertical report. Use of the `data.format` element of the FMT qualifier to the VERTICALLY keyword allows a more appropriate field width to be used. Any display clause element that itself includes a FMT qualifier will override the `data.format` default for that one field.

**Examples**

The command

```
LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAID
```

would produce a display such as that below.
LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAID
Invoice ....Value Customer...........................
74529      £1712.43  J McTavish
74273      £95.23   County Newspapers
63940      £141.00  R Bryant
3 records listed.

Including the VERTICALLY keyword would modify the display format to become

LIST INVOICES VERTICALLY VALUE CUSTOMER.NAME WITH NO AMT.PAID
Invoice... : 74529
Value..... : £1712.43
Customer.. : J McTavish

Invoice... : 74273
Value..... : £95.23
Customer.. : County Newspapers

Invoice... : 63940
Value..... : £141.00
Customer.. : R Bryant

3 records listed.

Adding a format code to the VERTICALLY keyword reduces the name width as below.

LIST INVOICES VERTICALLY FMT "8.L" VALUE CUSTOMER.NAME WITH NO AMT.PAID
Invoice. : 74529
Value... : £1712.43
Customer : J McTavish

Invoice. : 74273
Value... : £95.23
Customer : County Newspapers

Invoice. : 63940
Value... : £141.00
Customer : R Bryant

3 records listed.

In this second example, the same effect could be achieved with VERTICALLY FMT "FIT".
5.115 WHEN

The **WHEN** keyword introduces a selection clause for a multivalued field.

**Format**

```
WHEN condition
```

where

- `condition` is a boolean expression.
- `field1 operator field2` to compare two fields.
- `field1 operator value` to compare a field with a literal value.

**operator** is any of the query processor operators:

- **EQ** = EQUAL
- **NE** # NOT <> >
- **LT** < LESS BEFORE
- **LE** <= =<
- **GT** > GREATER AFTER
- **GE** >= =>
- **LIKE** MATCHES MATCHING
- **UNLIKE** NOT.MATCHING
- **SAID** SPOKEN ~
- **NO**
- **BETWEEN**

A **selection clause** specifies criteria governing which records are processed by the command. If omitted, all records are processed. The **WHEN** clause performs selection on exploded values from within the named multivalued field, showing only the selected value of the named field and all associated fields.

Field comparisons are performed using the internal format of `field1`, converting the `field2` or `value` item to this format if required. Thus a field holding an internal date, for example, may be compared with the more natural external form of the date. For example,

```
LIST INVOICES WHEN ISSUE.DATE > "12 OCT 96"
```

will list all invoice records with an issue date after 12 October 1996.

Prior to release 3.2-5 the **WHEN** clause filtering was not applied to **SELECT** commands. This incorrect behaviour can be continued in later releases by use of the NO.SEL.WHEN.FILTER setting of the **OPTION** command if needed.

**Example**

...
The command

    LIST SALES WITH PART = 100

to find only orders containing part 100 might produce a report such as

    LIST SALES WITH PART = 100
    Order    Part    Qty
    10001    100       4
    107       3
    219       3
    10021    100       3
    206       3
    219       7
    10014    105       3
    100       1
    210       7

    3 records listed.

For the same data, use of the \texttt{WHEN} keyword to select only part 100 with a query such as

    LIST SALES WHEN PART = 100

would produce the report below.

    LIST SALES WHEN PART = 100
    Order    Part    Qty
    10001    100       4
    10021    100       3
    10014    100       1

    3 records listed.
5.116 WITH

The **WITH** keyword introduces a selection clause.

**Format**

```
WITH {EVERY} condition {rel.op {EVERY} condition...}
```

where

- `condition` is `field1 operator field2` to compare two fields
- or `field1 operator value` to compare a field with a literal value

- `rel.op` is **AND** or **OR**

- `operator` is any of the query processor operators:
  - `EQ` = **EQUAL**
  - `NE` ≠ **NOT** <> **<>** >>>
  - `LT` < **LESS** BEFORE
  - `LE` ≤ **=** <= **<=**
  - `GT` > **GREATER** AFTER
  - `GE` ≥ **=>** =>
  - `LIKE` MATCHES **MATCHING**
  - `UNLIKE` NOT.MATCHING
  - `SAID` SPOKEN ~
  - `NO`
  - `BETWEEN`

A **selection clause** specifies criteria governing which records are processed by the command. If omitted, all records are processed.

The relational operators may be followed by the keyword **NO.CASE** to apply a case insensitive comparison. This also occurs if the QUERY.NO.CASE mode of the **OPTION** command is in effect.

The **EVERY** keyword indicates that every value or subvalue of `field1` must match `field2` in the manner defined by the **operator**. For example, the command

```
LIST EXAM.RESULTS STUDENTS SUBJECTS WITH EVERY GRADE = "A"
```

might be used to report a list of students achieving grade A in every examination. The SUBJECTS and GRADE fields in this example are a pair of associated multivalued fields recording examination subjects and grades.

**Testing for Empty Fields**
Use of the **WITH** clause with a field name but no operator or second field name tests whether the field is not empty. This allows queries such as

```
LIST SALES WITH PAYMENT
```
or

```
LIST SALES WITH NO PAYMENT
```
The second of these examples can also be written as

```
LIST SALES WITHOUT PAYMENT
```

Note that the implementation of this construct is inconsistent across different multivalue products if the field is multivalued. In QM, the second of the examples above shows only records in which all values in the PAYMENT field are empty.

**Complex Conditions**

The **AND** and **OR** operators may be used to build complex conditions. For example,

```
LIST STOCK WITH QTY < REORDER AND SUPPLIER = 26
```
selects only those records where the content of the QTY field is less than the content of the REORDER field and the SUPPLIER field contains the value 26.

The **AND** and **OR** operators are of equal priority and, if both appear in a single **WITH** clause, are evaluated left to right. Brackets may be used to modify the evaluation sequence. For example,

```
LIST STOCK WITH QTY < REORDER AND (SUPPLIER = 26 OR WAREHOUSE = 14)
```

**Multiple WITH Clauses**

A query may contain more than one **WITH** clause. There is normally an implied **AND** relationship between these clauses. Thus the command

```
LIST STOCK WITH QTY < REORDER WITH SUPPLIER = 26
```
is identical in effect to

```
LIST STOCK WITH QTY < REORDER AND SUPPLIER = 26
```

The **WITH.IMPLIES.OR** mode of the **OPTION** command changes the effect of multiple **WITH** clauses to have an implied **OR** between the clauses so that the above query with two **WITH** clauses is equivalent to.

```
LIST STOCK WITH QTY < REORDER OR SUPPLIER = 26
```

**Fields with Conversion Codes**

Field comparisons are performed using the internal format of *field1*, converting the *field2* or *value* item to its internal form if a conversion code is present. Thus a field holding an internal date, for example, may be compared with the more natural external form of the date. For example,

```
LIST INVOICES WITH ISSUE.DATE > "12 OCT 96"
```
will list all invoice records with an issue date after 12 October 1996.

This leads to a potential problem if the conversion code is not "reversible", where data converted from internal form to external form cannot be converted back again. As an example, a date conversion that returns just the month in which the date lies is not reversible. An internal date of 15171 (14 July 2009) would convert to "July" but this cannot be converted back to the original date.
The date conversions are defined such that a date with no year is assumed to be in the current year and a date with no day number is assumed to refer to the first day of the month. Thus a query of the form

```
LIST SALES WITH MONTH = "July"
```

is actually asking for orders placed on the first of July in the current year.

**Short forms**

The query processor offers a variety of short forms for selection clause elements.

**Implicit field names**

If two or more tests are to be performed against the same field, the field name only needs to appear in the first test. A relational operator without a preceding field name uses the same field as in the previous operator or the record id if this is the first relational operator.

In each of the following examples, the second query is an abbreviated form of the first

```
LIST VOC WITH TYPE = F OR TYPE = Q
LIST VOC WITH TYPE = F OR = Q

LIST ORDERS WITH DATE AFTER '31 DEC 99' AND DATE BEFORE '1 JAN 01'
LIST ORDERS WITH DATE AFTER '31 DEC 99' AND BEFORE '1 JAN 01'

LIST SALES WITH @ID > 10000
LIST SALES > 10000
```

**Implicit OR relation**

A relational operator followed by a series of values tests each of the values against the given field in an implicit OR relationship.

```
LIST ORDERS WITH REGION = 'SOUTH' OR REGION = 'NORTH'
LIST ORDERS WITH REGION = 'SOUTH' 'NORTH'
```
5.117 WITHOUT

The WITHOUT phrase is a synonym for use of WITH NO.

Format

WITHOUT condition \{rel.op condition\...\}

Example

LIST ORDERS WITHOUT PAYMENT.DATE

This is equivalent to

LIST ORDERS WITH NO PAYMENT.DATE
The **XML** keyword causes the query processor to produce output in XML format.

**Format**

```
XML {ELEMENTS} {WITH.DTD | DTD.ONLY} {WITH.SCHEMA | SCHEMA.ONLY}
```

where

- **ELEMENTS** produces a report in which each field or value is output as a separate XML element.
- **WITH.DTD** includes the Document Type Definition in the output.
- **DTD.ONLY** produces a report containing only the Document Type Definition.
- **WITH.SCHEMA** includes the schema in the output.
- **SCHEMA.ONLY** produces a report containing only the schema.

An XML mode report command may not include breakpoints or arithmetic field qualifiers. Use of the PAN, DET.SUP, HEADING or FOOTING options will be ignored.

The tag name applied to top level (record) items is the file name. For associated multivalue items, the association name is used as the tag. For non-associated multivalued items, the tag is the name of the item with a `_MV` suffix.

**Example**

The following example is based on QM's demonstration sales database as created using the **SETUP.DEMO** command but without creating any of the additional items developed from material in the Teach Yourself OpenQM training package.

```
LIST SALES XML
<?xml version="1.0" encoding="UTF-8"?>
<ROOT>
<SALES SALE = "12140" DATE = "14 Oct 07" CUST = "1056">
  <LINE ITEM = "003" QTY = "1" PRICE = "19.85"/>
  <LINE ITEM = "055" QTY = "1" PRICE = "2.87"/>
  <LINE ITEM = "021" QTY = "6" PRICE = "0.25"/>
</SALES>
<SALES SALE = "12347" DATE = "07 Feb 08" CUST = "1087">
  <LINE ITEM = "004" QTY = "1" PRICE = "1.70"/>
  <LINE ITEM = "131" QTY = "1" PRICE = "6.98"/>
  <LINE ITEM = "232" QTY = "1" PRICE = "19.85"/>
  <LINE ITEM = "055" QTY = "1" PRICE = "2.87"/>
  <LINE ITEM = "021" QTY = "6" PRICE = "0.25"/>
</SALES>
</ROOT>
etc
```

With the **ELEMENTS** keyword, the first record from this example becomes
<?xml version="1.0" encoding="UTF-8"?>
<ROOT>
  <SALES>
    <SALE>12140</SALE>
    <DATE>14 Oct 07</DATE>
    <CUST>1056</CUST>
    <LINE>
      <ITEM>003</ITEM>
      <QTY>1</QTY>
      <PRICE>1.70</PRICE>
    </LINE>
    <LINE>
      <ITEM>122</ITEM>
      <QTY>3</QTY>
      <PRICE>44.50</PRICE>
    </LINE>
    <LINE>
      <ITEM>234</ITEM>
      <QTY>2</QTY>
      <PRICE>26.00</PRICE>
    </LINE>
    <LINE>
      <ITEM>121</ITEM>
      <QTY>6</QTY>
      <PRICE>38.00</PRICE>
    </LINE>
  </SALES>
</ROOT>
Part 6

QMBasic
There are times when the powerful facilities available using the standard commands of QM are not sufficient to meet application demands. For these occasions, the QMBasic programming language provides a very easy to use means of developing components of the application. User written programs can be mixed with standard commands to give maximum capabilities with minimum development costs.

QMBasic is not difficult to learn. As the name implies, it has its origin in the Basic language found on many personal computers, however, the powerful string handling and screen formatting functions make development extremely fast. QMBasic has very high compatibility with the equivalent languages found in other similar data management products but also has some major extensions such as object oriented programming.
6.1 QMBasic overview

QM applications are written using QMBasic. Unlike many other programming languages, the individual source modules are not linked together to form a single executable program but remain separate items that are loaded into memory dynamically when they are first needed. This approach generally results in lower memory usage and easier maintenance.

The program modules are stored as simple text records, normally in directory files though use of hashed files is allowed. Each field of the record represents a line of the program (a transformation that corresponds exactly to how directory file records are stored by the underlying operating system). Although you may place your program modules in any file you wish or scatter them over several files, by convention programmers often use a file named BP (Basic Programs). The BASIC and RUN commands will look here for programs by default if no file name is given in the commands.

QMBasic modules are of four types:

Programs
A program is a simple module that can be run directly from the command line. It can also be called from other programs using CALL in the same way as a subroutine that has no arguments. A program optionally starts with a PROGRAM statement though this is implied if none of the statements used to start the module types below are present.

Subroutines
A subroutine is a module that is called from another QMBasic element using CALL. Subroutines usually take arguments, variables that are passed in or out of the subroutine to transfer data between modules. A subroutine module starts with a SUBROUTINE statement.

Functions
A function is very similar to a subroutine but returns a value to the program that executed it. A function module starts with a FUNCTION statement.

Class modules
A class module contains the property and method routines that are used for object oriented programming. A class module starts with a CLASS statement.

Throughout all documentation, the word program is used to refer to all of the above module types unless the context explicitly states otherwise.

Before a program can be executed, the source form written by the developer must be compiled (translated into corresponding executable program modules) using the BASIC command. The executable items are written to records of the same name as the source in a separate directory file that has the same name as the source file but with a .OUT suffix added. For example, the compiled version of a program stored as MYPROG in BP will be in MYPROG in the BP.OUT file. Programs may be executed directly from the .OUT file or may be moved into the system catalogue using the CATALOGUE command. Subroutines, functions and class modules must be catalogued before use.

Often, it is useful to place QMBasic source code elements that are used in more than one program in a separate record which is read during compilation as though it was part of the main program. In particular, common data structures or names representing keys to subroutines may be handled in this way to ensure that all components of the application have a common view of the information instead of needing to make changes in many places. The SYSCOM file is an example of this technique with records containing keys and other values that you may need in many programs. The QMBasic $INCLUDE directive described later in this section is used to direct the compiler to include text from another record. Include records may be stored in any file and are not separately compiled as the text is imported into other programs. It is recommended that a suffix of .H is used on include record names as the compiler will automatically skip these when using a select list. This suffix has its origins...
in the C programming language where it is used to denote a "header file" that serves the same purpose as QMBasic include records.

A QMBasic program has a very simple to understand format. The program is made up of a series of **statements**. Each statement normally corresponds to a single line of source program text though it is possible to place multiple statements on a single line by separating them with semicolons. Some statements have a syntax which allows them to span multiple lines without special action. Any statement that includes a comma in its syntax may start a new line immediately after the comma. Other statements may be split over multiple lines by ending each line except the last with a tilde (\~) character. Note that continuation lines are handled before any other analysis of the line and therefore a comment that ends with a tilde will be treated as continuing on the next line.

Lines commencing with an asterisk or an exclamation mark are treated as comments and ignored by the compiler. Comments can be included on the same line as a source program statement by using a semicolon to start a new statement followed by an asterisk or an exclamation mark. Blank lines and leading spaces are ignored by the compiler.

```
* A comment on a line of its own
A = 44 ;* This is a trailing comment
B = "abc" ; C = LEN(B) ;* Two statements on a single line
CALL MYSUBR(TITLE, DATA, ;* A subroutine call
            ITEM.COUNT) ;* on a separate line
```

The compiler is not case sensitive in language keywords. By default, variable names are also case insensitive but this can be altered using the `$MODE` directive or the `$BASIC.OPTIONS` record. All program fragments in this documentation are shown in uppercase partly so that they stand out and partly because historically most multivalue software has been written in this way. Use of lowercase can make programs easier to read. Other conventions such as writing equated token names in uppercase can be used if desired.

A program usually commences with a **PROGRAM**, **SUBROUTINE**, **FUNCTION** or **CLASS** statement. This serves to identify the type of QMBasic item and to assign a name to it. If none of these statements is present it is assumed to be a program.

The formats of these statements are

```
PROGRAM name
SUBROUTINE name(arg1,arg2,...)
FUNCTION name(arg1,arg2,...)
CLASS name
```

where a subroutine may take up to 255 arguments, a function 254.

A program ends with an **END** statement. Only blank lines and comments may follow this final **END**. For compatibility with other multivalue database products there is a compiler directive to make this final **END** optional ($MODE OPTIONAL.FINAL.END).

On an ECS mode system, string constants may contain ECS characters. All other language elements are restricted to the 8-bit character set.
QMBasic - Variable names and values

Variable names must commence with a letter and may contain letters, digits, periods (full stops), percentage signs and dollar signs. Names may also contain underscore characters but not as the last character of the name. Users are discouraged from defining names containing dollar signs for their own purposes as these are reserved to identify system functions and constants. Except as indicated elsewhere, there is no restriction on the length of a name though very long names may appear truncated in debugging information. Variable names are case insensitive such that CLIENT and Client are interchangeable as references to the same variable. Use of the $MODE CASE.SENSITIVE changes this such that CLIENT and Client become different variables.

Except when using the $MODE EXPLICIT compiler directive, variables do not need to be declared. The compiler adds variable names to its internal symbol table when they are first encountered. Use of explicit declaration allows misspelled variable names to be trapped by the compiler but can be inconvenient. When this mode is enabled, variables must be declared using the DIMENSION statement.

Although QMBasic imposes few restrictions on the choice of names, it is advisable to avoid using names which correspond to QMBasic statements, functions and keywords. The only reserved names which may not be usable in some contexts are

```
AND   GOTO   OR
BEFORE GT    REPEAT
BY     IN     RETURNING
CAPTURING LE    SETTING
CAT    LOCKED STEP
DO     LT     THEN
ELSE   MAT    TO
EQ     MATCH  TRAPPING
FROM   MATCHES UNTIL
GE     NE     WHILE
GO     NEXT
GOSUB  ON
```

QMBasic variables are type variant, that is, that they may hold, for example, an integer value at one point in time and a character string later on. The actual form in which the data is held is determined by how it was assigned. If a variable is set to contain a string of digits and is subsequently used in an arithmetic calculation, the value is converted internally to a numeric form without affecting the variable itself. If this arithmetic calculation is performed many times in a loop, it may be worth forcing a type conversion to prevent repeated temporary conversions. For this reason, multivalue Basic programs sometimes contain apparently redundant looking statements of the form

```
A = A + 0 ;* Convert to numeric form
or
S = S : "" ;* Convert to string form
```

QMBasic provides a better way to do this by use of the INT(), NUMERIC(), STR() and BOOL() functions.
Numeric values are held as integers wherever possible, conversion to floating point format occurring when the result of an arithmetic operation is non-integer or when the value is too large to be stored as an integer.

A variable holding a string of no characters is referred to as a **null string** and is treated as a special case in many operations. Users familiar with SQL type environments should take care to distinguish the multivalue database meaning of the word null from its SQL meaning.

A string variable may hold any number of characters. The actual total limit for all strings in a program is imposed by the disk space available for paging and typically far exceeds application needs. Although QMBasic avoids copying strings unnecessarily whenever it can, operations involving very large strings are likely to have a detrimental effect on performance.

A Boolean variable holds a True or False value. In general, these are interchangeable with numeric values 1 and 0 respectively. Support for the internal Boolean data type was added in release 3.2-2. When Boolean values are stored in dynamic arrays, the numeric representation is used.

A variable may hold many other types of information. For example, a **file variable** holds a reference to an open file and is used in all statements that refer to that file. A **subroutine variable** contains a fast reference to a catalogued subroutine that has been loaded into memory. Users cannot directly create subroutine variables; they are the result of transforming a string variable holding the subroutine name when it is first called. Until otherwise determined, variables are initially **unassigned**. Reference to an unassigned variable (where no value has yet been stored) will cause a run time error as an aid to program debugging. This can be changed to become a warning message by use of the **UNASS.WARNING** mode of the **OPTION** command.

**Constants**

Constant values may be numbers, strings or Boolean values.

**Numeric constants** are written as a sequence of digits, optionally preceded by a sign or containing a decimal point. If a sign is used, there must be no space between it and the first digit. Regardless of the setting of the national language support decimal separator character (see **SETNLS**), the decimal separator in a QMBasic numeric constant is always the period.

If the NUMERIC.EXPONENT mode of the **OPTION** command is enabled, conversion of a character string to numeric form will accept exponent style values such as 123.45E-1.

QMBasic also allows hexadecimal numbers in equated tokens and most expressions. These are written with a prefix of 0x as used in the C programming language (e.g. 0x23 is decimal 35).

**String constants** are sequences of characters enclosed by delimiters. Valid delimiter characters are the single quote ('), the double quote (") and the backslash (\). The delimiter at the start and end of the string value must be the same but there is no difference in the internal treatment of the delimiters. On ECS mode systems, string constants may contain characters from the extended character set.

The compiler imposes no limit on the length of a string literal value though it may not extend from one line to the next. To improve program readability, very long strings can be constructed by concatenating component substrings.

The mark characters are available as @FM, @VM, @SM, @TM and @IM. These are described in a later section.

The Boolean constants are written as @TRUE or @FALSE. Prior to release 3.3-0, these evaluated to their numeric equivalents, 1 and 0 respectively. Release 3.3-0 onwards evaluate these as the
internal Boolean data type, however, the NUMERIC.BOOLEAN option of the $MODE compiler directive can be used to revert to the old behaviour. This should only be used if programs must be able to run on releases prior to 3.2-2 that did not support the Boolean data type.
QMBasic provides support for both scalar and matrix variables. A **scalar variable** is a simple value referenced by its name alone. It may contain data of any type.

A **matrix variable** is a one or two dimensional array of values. Matrices must be declared by use of the **DIMENSION** (more usually **DIM**) statement. Because memory for matrices is allocated dynamically, the **DIM** statement must be executed at program run time before the variable is used in any other way. Matrix variables are also known as **dimensioned arrays**.

A one dimensional matrix of ten elements is defined by a statement of the form

```
DIM A(10)
```

For a two dimensional matrix with 5 rows of 8 columns this becomes

```
DIM B(5,8)
```

A single dimensional matrix is effectively a two dimensional matrix with one column. Thus references of the forms A(B) and A(B,1) are totally interchangeable.

By default, all matrices have an additional element, the zero element, which is used by some QMBasic statements. This is referred to as A(0) or B(0,0). The **$MODE** compiler directive can be used to create Pick style matrices which do not have a zero element. Note that, in a two dimensional matrix, this is a single element, not a complete row 0 and column 0.

The elements of a matrix may be of differing types (numbers, strings, file variables, etc).

A variable holding a string value may be considered as a **dynamic array**, the mark characters being used to divide it into fields, values and subvalues. Such a string may correspond to a record in a data file or may be totally internal to the program. Special operations are provided in QMBasic to manipulate dynamic arrays. These include sorted and unsequenced searching, insertion, deletion, replacement and extraction as well as some extremely powerful operations to build or decompose dynamic arrays.

A dynamic array in which each field, value or subvalue contains a numeric value is known as a **numeric array**. Many of the arithmetic operations operate on numeric arrays by processing corresponding elements in turn. For example, a statement

```
A = B + C
```

adds B and C together, storing the result in A. Where B and C are simple numeric values or strings that can be converted to numbers, this operation behaves as in most other computer languages. If B and C are dynamic arrays the operation handles each corresponding pair of values in turn.

```
B = "1" : @FM : "2" : @VM : "3" : @FM : "4"
C = "5" : @FM : "6" : @VM : "7" : @FM : "8"
A = B + C
```

The result of this operation would be to set A to 681012. The effect of operations on numeric arrays where the placement of fields, values and subvalues do not match exactly is determined by the use of the **REUSE()** function.

**See also:**
- Matrix File I/O
Data Collections

A data collection is an arbitrarily multi-dimensional set of name / value pairs. Whereas the elements of a dynamic array are referenced by their numeric position, elements of a collection are referenced by case sensitive name. The value associated with each named element may be of any QM data type, including a further collection, thus allowing nesting to a depth limited only by available memory.

A collection can be created in three ways. The `COLLECTION()` function can be used to create an empty collection

```qmb
CLIENT.DATA = COLLECTION()
```
or to make a copy of an existing collection

```qmb
CLIENT.DATA = COLLECTION(OLD.CLIENT.DATA)
```

Note that a statement such as

```qmb
CLIENT.DATA = OLD.CLIENT.DATA
```
does not copy the collection. Instead, it creates a new reference to the same collection. This is similar to copying file variables or those that reference object oriented programming objects.

Alternatively, a character string containing a JSON (JavaScript Object Notation) representation of the data can be parsed into a collection using the `JPARSE()` function

```qmb
CLIENT.DATA = JPARSE(JSON.STRING)
```

The elements of a collection are referenced using a syntax that is very similar to use of dynamic arrays but uses curly brackets instead of angle brackets and uses names instead of numeric values. Note that in all examples of syntax relating to data collections, the curly brackets are part of the syntax and not indicative of optional elements as elsewhere in the documentation.

```qmb
CLIENT.NAME = CLIENT.DATA{'NAME'}
```

In exactly the same way as with dynamic arrays, the element reference can be a literal value (as above), a variable or an expression that derives the name.

```qmb
ITEM = 'NAME'
CLIENT.NAME = CLIENT.DATA{ITEM}
```

Where an element of a collection is itself a collection, the syntax becomes

```qmb
LOCATION = CLIENT.DATA{'ADDRESS', 'TOWN'}
```
or

```qmb
LOCATION = CLIENT.DATA{'ADDRESS'}{'TOWN'}
```
The second of these two syntaxes is valid because `CLIENT.DATA['ADDRESS']` is itself a collection and hence can have a further collection element reference applied to it.

A third method of referencing nested collections provides a way to access data for which the dimensionality may not have been known when the program was compiled. In this syntax, the name (the **element path**) is formed from multiple parts separated by forward slash characters.

```qmb
LOCATION = CLIENT.DATA{'ADDRESS/TOWN'}
```

Although this example shows this as a literal value for clarity, this syntax would more commonly be used with an indirect reference via a name variable or expression. The three syntaxes can be mixed in any combination.
Referencing a collection element that does not exist returns a null string in the same way as referencing a non-existent dynamic array element, however, the STATUS() value will be set to ER$NOT.FOUND for a failed collection reference.

A collection may be an element of a dimensioned array

\[
\text{CLIENT.NAME} = \text{CLIENT.DATA(CLI.NO)}\{\text{"NAME"}\}
\]

**Enumerating Collection Elements**

The names present in a collection may be enumerated using the ENUMERATE() function.

\[
\text{NAMES} = \text{ENUMERATE(CLIENT.DATA)}
\]

The names are returned as a field mark delimited dynamic array, sorted into ascending order. For a nested collection, only the level directly referenced by the argument to the function is enumerated.

The presence of a single element can be tested using the ELEMENT.EXISTS() function.

**Scanning the content of a data collection**

As an alternative to use of ENUMERATE() followed by a loop that references each element in turn, the SCAN statement provides a simpler and slightly faster way to walk through the elements of a collection.

**Adding or Modifying Collection Elements**

An element may be added or updated in a collection using the collection reference on the left of an assignment operator

\[
\text{CLIENT.DATA\{\text{"PHONE"}\}} = \text{PHONE.NO}
\]

or using the INS statement

\[
\text{INS PHONE.NO AS CLIENT.DATA\{\text{"PHONE"}\}}
\]

If the element name references intermediate levels of a nested data collection, any absent elements are automatically inserted.

Elements may be copied from one collection to another (or a different name in the same collection) using a statement such as

\[
\text{CLIENT.DATA\{\text{"PHONE"}\}} = \text{OTHER.CLIENT.DATA\{\text{"PHONE"}\}}
\]

**Deleting Collection Elements**

An element may be deleted from a collection using the DEL statement

\[
\text{DEL CLIENT.DATA\{\text{"PHONE"}\}}
\]

If the element name references intermediate levels of a nested data collection, all lower level items are also deleted.

**Arrays in Collections**

A data collection may contain single dimensional arrays which are automatically sized to fit their content. An empty array is created using the MATO function
CLIENT.DATA('CONTACTS') = MAT()

Alternatively, a standard QMBasic array variable can be copied into a collection

CLIENT.DATA('CONTACTS') = MAT(CONTACT.NAMES)

In the second syntax, if the array is two dimensional, it will be restructured to become a single dimensional array by copying items row by row to the new array.

The MAT() function has an optional second argument that sets the maximum number of elements to copy. If omitted or zero, all elements are copied.

Once the array has been created, data values are updated by using the element number as the name of the item.

CLIENT.DATA('CONTACTS', N) = CONTACT.NAME

Use of a negative value for the array index will insert a new item on the end of the array in much the same way as use of negative values in dynamic array assignment. There may not be unused elements in an array within a collection. Thus, if the CONTACTS array holds five items, a statement such as

CLIENT.DATA('CONTACTS', 8) = NEW.NAME

is invalid as elements 6 and 7 would be undefined, however,

CLIENT.DATA('CONTACTS', 6) = NEW.NAME

is valid as the newly created element immediately follows the last existing element.

An element can be deleted from an array using the DEL statement

DEL CLIENT.DATA('CONTACTS', N)

or

DEL CLIENT.DATA('CONTACTS'){N}

Deleting an element renumbers all elements that follow the deleted item, again just like deleting a dynamic array element.

An element can be inserted into an array using the INS statement

INS NAME AS CLIENT.DATA('CONTACTS', N)

or

INS NAME AS CLIENT.DATA('CONTACTS'){N}

Inserting an element renumbers all elements that follow the inserted item. In the same way as when adding an element to the end of an array, undefined elements are not permitted. The index value identifying the insertion position must be no greater than one more than the current number of elements. A negative value appends an item to the end of the array.

An empty array may be inserted as an element of a collection using the INS statement

INS MAT() AS CLIENT.DATA('CONTACTS')

or an existing QM array can be copied into a collection using

INS MAT(NAMES) AS CLIENT.DATA('CONTACTS')

Conversely, copying an array from a collection into a standard dimensioned array uses a modified form of the MAT statement

MAT NAMES = MAT CLIENT.DATA('CONTACTS')

The INMAT() function can be used to find the number of elements in an array within a collection.
A collection reference may contain up to a maximum of two uses of an asterisk to indicate that all elements of an array are to be returned as a dynamic array.

\[
\text{PRODUCTS} = \text{ORDER}{'\text{DETAIL}/*/\text{PRODNO}'})
\]

\[
\text{SERIAL} = \text{ORDER}{'\text{DETAIL}/*/\text{SERIAL.NO}/*'})
\]

\[
\text{CONTACT.NAMES} = \text{CLIENT.DAT}A{'\text{CONTACT}/*'})
\]

In this syntax, the data level within the collection corresponding to the position of the asterisk must be an array and the entire reference must lead to an item that can be represented as a string. Where only one asterisk is present, the dynamic array contains a value for each array element. Where two asterisks are present, the dynamic array has a value for each element of the array referenced by the first asterisk and a subvalue for each element of the array referenced by the second asterisk.

### Collections and JSON

JSON is a way to represent arbitrarily multi-dimensional data as a character string that can be stored in a text file or transmitted over a network. It is primarily intended as a communications format for web based applications.

The `JPARSE()` function will parse a JSON string into a data collection. The `JBUILD()` function performs the opposite transformation, building a JSON string from a collection. It is important to understand the issues relating to data types that can occur when using these functions.

JSON supports numeric, string, Boolean and null data types as elements of objects and arrays.

Numeric data may be represented in several formats in JSON but `JPARSE()` will yield either an integer or floating point value that has no reference to the original data format. Using `JBUILD()` to reconstruct the JSON string may result in a different but valid representation of the same value.

Boolean values (True and False) will be correctly handled by QM through use of its internal Boolean data type. Parsing and rebuilding a JSON string will maintain the difference between True/False and their alternative numeric representation as 1/0.

JSON also supports the SQL concept of a null value (not the same as a null string). Support for this in QM is limited to setting and testing null values and will correctly maintain the null value through use of `JPARSE()` and `JBUILD()`. Other operations on the null value will usually result in a run time error.

The elements of a data collection may include data of a type that is not supported by JSON (e.g. a file variable). Although it is unlikely that these data types would be present in collections that will be converted to JSON format, the `JBUILD()` function will fail, returning a null string and a `STATUS()` value of ER$BUILD.ERROR.

### Collection Files

A collection that has been encoded to a JSON string using `JBUILD()` can be stored as a record in any QM file type. If the strings within the collection do not contain field, value or subvalue marks, the JSON string can also be stored as a field, value or subvalue within a data record.

QM also supports data collection files which are a variant of a hashed file that stores data as a collection without the need to convert it to a JSON string. A file of this type is created by including the COLLECTION option in the `CREATE.FILE` command. Because the data in a collection file is
not a dynamic array (although the values of any strings within the collection could be), dictionary items that reference fields by number other than the record id are invalid. Instead, dictionaries may contain E-type (element) records which have the same form as a D-type record except that field 2 holds the element path instead of a field number. The element path may include up to a maximum of two uses of an asterisk to return a multivalued list of array elements as described above.

Dictionary I-type items can reference E-type data definitions and can also use the \{name\} constructs defined above. The third argument to the TRANS() function (the item to be returned) may be an E-type item defined in the dictionary of the file being accessed if it is a collection.

A collection file may have indices built on data defined by an E-type dictionary record in exactly the same way as those based on D-type items in other files. It is also possible to define a trigger function in a collection file in which case the record data passed in to the trigger function will be a collection instead of a string. Only record level encryption is supported for collection files.

The query processor can access collection files by using E-type dictionary items instead of D-type items. The ELEMENT "name" construct can be used to reference a collection element on the command line.

Collection files are fully supported by QMNet, data replication, transactions and record level encryption.

Linked Collections

A data collection stored in a collection file may contain links to other collections in the same file or a different file. The links are stored as a string variable that contains a reference to the linked item as `filename:id`

The filename (but not the colon) can be omitted if the link is to a record in a default file (perhaps the same file) but this reduced syntax has implications for the application as described below. There is nothing about this string item that defines it as a link except for how it is used by the application.

Where a data collection containing a link has been read into a QMBasic variable, the linked item can be joined onto its parent by use of the EXPAND() function

```
OK = EXPAND(VAR('link'))
```

where link is the element path of the string that defines the link.

If the link string uses the reduced syntax that has no filename, an extended form of the EXPAND() function must be used

```
OK = EXPAND(VAR('link'), FILEVAR)
```

where FILEVAR is a file variable that references the file containing the linked item. This optional function argument is ignored when using the full syntax of the link string.

The EXPAND() function returns True if successful.

The Collection Editor, CED

The collection editor allows the content of a data collection to be viewed or edited. It may be used as a QM command to edit a record in a collection file or as a QMBasic subroutine named !CED() to edit a collection passed in as an argument. See CED for details of this editor.
Orphaned Self-Referential Collections

Because an element of a data collection can be any QM data type, including a collection, it is possible to create a self-referential data collection. In its simplest form, this could be achieved with statements of the form:

\[
A = \text{COLLECTION}()
\]
\[
A\{"CONTENT"\} = A
\]

Although this is unlikely to have any practical use, it can be done by accident. If the program went on to overwrite variable A

\[
A = 99
\]

the only program visible link to the collection has been lost and the entire collection becomes orphaned. This can never be repaired as there is no longer any way to access the collection.

See also:
CED, 'CED(), COLLECTION(), data collection files, DEL, E-type dictionary records, ENUMERATE(), EXPAND(), INS, JBUILD(), JPARSE(), MAT(), SCAN
**QMBasic - Common blocks**

Variables are normally available to all statements within a single QMBasic program or subroutine. Although the language provides an internal subroutine call through the **GOSUB** statement, this does not automatically bring in the concept of the internal subroutine having its own variables or any other aspect of variable scope found in many other languages.

QMBasic extends the language definition by adding the concept of variables that are private to an internal subroutine. This is achieved by use of the **LOCAL** statement and the associated **PRIVATE** variable declaration statement. Variables declared in this way are private to the one internal subroutine and cannot be accessed by other parts of the program. Furthermore, they are stacked if the subroutine calls itself, either directly or indirectly via another intermediate subroutine. For more information, see the description of the **LOCAL** statement.

QMBasic provides **common blocks** for data which is to be shared between two or more programs in the same QM process. These are declared by a statement of the form

```
COMMON /name/ var1, var2, var3,...
```

where **name** is the name by which the common block is to be known. A common block may contain any number of variables and is created when it is first referenced. It remains in existence until the user leaves QM. Once a common block is created, subsequent programs using the same common block name within the same process access the same data. The number of variables in the common block may not be increased by later definition but programs can define fewer variables than in the actual common block. Normally, the structure of a common block is best defined in an include file so that the same definition is used by all parts of the application.

Where programs use separate **COMMON** statements to reference the same block, note that the variables are defined by their position in the list, not the names used. Thus it would be valid (but not a good idea) for one program to have

```
COMMON /MYCOMMON/ A, B, C
```

and another program

```
COMMON /MYCOMMON/ D, E, F
```

where the data stored in B by the first program would be visible to the second program as E.

The name of a common block must conform to the same rules as a variable name. There is also an **unnamed common** (sometimes known as blank or unlabelled common) which is defined by a **COMMON** statement without a name:

```
COMMON A, B, C
```

This operates in exactly the same way except that each command processor level has its own unnamed common. Thus, an **EXECUTE** statement used to run one program from within another would result in a new unnamed common block being created for the executed program, the original being restored on return. The unnamed common is discarded on return to the command processor. Thus, running two unrelated programs that use unnamed common one after the other at the same command processor level will not carry the common block from the first program into the second program.

The variables in a common block are initialised to integer zero when the block is created. It is thus possible to include QMBasic code to perform further initialisation just once by statements of the form

```
COMMON /MYCOMMON/ INITIALISED,                  VAR1,                  VAR2,                  VAR3,...                  IF NOT(INITIALISED) THEN
```
do initialisation tasks
INITIALISED = @TRUE
END

Note how the names of the variables within the common block may extend from one line to the next. The compiler will continue the common block definition over multiple lines wherever the line ends with a comma.

The same common block could be defined as

    COMMON/MYCOMMON/ VAR1
    COMMON/MYCOMMON/ VAR2
    COMMON/MYCOMMON/ VAR3
    ...etc...

The compiler assumes that definitions of variables with the same common block name are a continuation of the previous definitions.

Common blocks may also contain matrices. These are defined by including the row and column bounds in the COMMON statement, for example

    COMMON /MYCOMMON/ MAT1(5,3)

Except when using Pick style matrices, the size of a matrix in common may be changed by a later DIM statement. The size given in the COMMON declaration is the initial size of the matrix.

The matrix in the example above may be made smaller by a statement such as

    DIM MAT1(2,1)

or made larger by a statement such

    DIM MAT1(99,9)

It will then keep the changed row and column bounds, within the current program and in any other program that shares the common block, regardless of the row and column bounds specified in the COMMON statement. This allows matrices to be freely shared between programs, with their dimensions being changed as needed.

Pick style matrices are of fixed size and cannot be re-dimensioned dynamically.
QMBasic - Labels

Any statement of a QMBasic program may be labelled. A label may take one of three formats; a name of the same format as a variable name followed by a colon, a sequence of digits and periods followed by a colon, or a sequence of digits and periods with no trailing colon.

The following are all valid label names.

    REDISPLAY:
    100
    12.9.6:

The label must appear as the first item on the source line. Labels and variables may have the same name though this may lead to some confusion when maintaining a program.

Statements that reference the label (e.g. GOSUB) may optionally include the colon after the label name. This is not recommended as it can make using an editor to search for a label in a program more difficult as the search will also find references to the label.

Numeric labels are provided for compatibility with other products. Use of numeric labels is discouraged as the "names" do not impart any information about the role of the label. For example, a statement such as

    GOSUB 9600

gives the reader no clue about the action performed by the subroutine at label 9600 whereas

    GOSUB GET.CUSTOMER.ID

suggests what the subroutine does.
QMBasic - Expressions and operators

A QMBasic expression consists of one or more data items (constants or variables) linked by operators.

- constant: Use constant value (string or numeric)
- var: Use value of named variable
- var[s,n]: Extract n character substring starting at character s of var
- var[n]: Extract last n characters of string var
- var[d,s,n]: Extract n consecutive sections of string var delimited by character d, starting at section s.
- var[s,*]: Extract from character n onwards of string var
- var[d,s, *]: Extract all sections of string var delimited by character d, starting at section s.
- var<f>: Extract field f of dynamic array variable
- var<f>,v>: Extract field f, value v of dynamic array variable
- var<f>,v,s>: Extract field f, value v, subvalue s of dynamic array variable
- var{path}: Extract the data collection element with the given element path
- func(args): Use the returned value from a function which may take arguments

In all cases above, var may be a matrix reference, for example

var(r,c)[s,n]

where r and c are expressions which evaluate to the desired matrix row and column index values.

There is also a special conditional item of the form

**IF conditional.expr THEN expr.1 ELSE expr.2**

where conditional.expr is evaluated to determine whether the overall value is that of expr.1 or expr.2.

The Boolean (True/False) values used by QMBasic are stored internally as a Boolean data type, however, use of a Boolean value in an operation that requires a numeric value converts True to one and False to zero.

The Boolean constant values are available as @TRUE and @FALSE for use in programs and dictionary I-type expressions.

The substring extraction operation x[s,n] extracts n characters starting at character s of the string x. Character positions are numbered from one. Thus

A = "abcdefgijkl"
Z = A[5,3]

sets Z to the string "efg".

If the bounds of the substring extend beyond the end of the string from which it is to be extracted, the result is truncated. Trailing spaces are not added to make up the shortfall. A start position of less than one is treated as one.
Specifying the number of characters, \( n \), as an asterisk extracts from character \( s \) onwards, regardless of the length of the string.

\[
A = "abcdefgijkl"
Z = A[5,*]
\]

sets \( Z \) to the string "efghijkl".

The trailing substring extraction operation \( x[n] \) extracts the last \( n \) characters of the string \( x \). Thus

\[
A = "abcdefgijkl"
Z = A[3]
\]

sets \( Z \) to the string "jkl".

If the length of the substring to be extracted is greater than the length of the source string, the entire source string is returned.

The third form of substring extraction, known as group extraction, is of the form \( \text{var}[d,s,n] \). It treats \( \text{var} \) as being made up of a series of sections delimited by character \( d \) and returns \( n \) consecutive sections, starting at section \( s \). See the \( \text{FIELD()} \) function for an alternative syntax.

The field extraction operator \( x<f,v,s> \) extracts field \( f \), value \( v \), subvalue \( s \) from the source string \( x \). If \( s \) is omitted or zero, field \( f \), value \( v \) is extracted. If \( v \) is omitted or zero, field \( f \) is extracted. Thus

\[
x<2> \quad \text{extracts field 2}
x<2,7> \quad \text{extracts field 2, value 7}
x<2,7,3> \quad \text{extracts field 2, value 7, subvalue 3}
\]

The operators of QMBasic are set out in the table below. The numbers in the right hand column are the operator precedence, the lower valued operators taking precedence in execution. Operations of equal precedence are processed left to right. Round brackets may be used to alter the order of execution or to improve readability of complex expressions.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Unary plus</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>Unary minus</td>
<td>0</td>
</tr>
<tr>
<td>~</td>
<td>Unary bitwise not</td>
<td>0</td>
</tr>
<tr>
<td>&lt; &gt;</td>
<td>Dynamic array extraction</td>
<td>1</td>
</tr>
<tr>
<td>[ ]</td>
<td>Substring extraction</td>
<td>1</td>
</tr>
<tr>
<td>{ }</td>
<td>Collection element reference</td>
<td>1</td>
</tr>
<tr>
<td>** or ^</td>
<td>Exponentiation (raising to power)</td>
<td>2</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>3</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>3</td>
</tr>
<tr>
<td>//</td>
<td>Integer division</td>
<td>3</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
<td>4</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>4</td>
</tr>
<tr>
<td>:</td>
<td>Implicit format (See ( \text{FMT()} ) function)</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>Concatenation</td>
<td>6</td>
</tr>
</tbody>
</table>
< Less than 7
> Greater than 7
= Equal to 7
== Case sensitive string equality 7
~= Case insensitive string equality 7
# Not equal to 7
<= Less than or equal to 7
>= Greater than or equal to 7
MATCHES Pattern match (see below) 7
AND Logical AND 8
OR Logical OR 8
.& Bitwise logical AND 8
.! Bitwise logical OR 8

The following alternative logical and relational operator formats may be used

< LT
> GT
= EQ
# NE <> ><
<= LE =<
>= GE =>
MATCHES MATCH
AND &
OR !

The relational operators are defined such that, if the two items to be compared can both be treated as numbers, a simple numeric comparison is performed. If one or both items cannot be treated as numbers, they are compared as left aligned character strings. The COMPARE() function can be used to force a string comparison that returns a value indicating the collating sequence relationship of the two items being compared.

If a program is compiled using the $MODE NOCASE.STRINGS directive, string comparisons are case insensitive. The == operator can be used to force a case sensitive comparison. Conversely, in program compiled without this directive, the ~= operator can be used to force a case insensitive comparison.

Note: The language syntax includes an ambiguity with the use of the < and > characters as both relational operators and in dynamic array references. For example, the statement

A = B<C> + 0

could be extracting field C from dynamic array B and adding zero to it (to force it to be stored as a numeric value) or it could be testing whether B is less than C and the result of this comparison is greater than zero. In cases such as this, the compiler looks at the overall structure of the statement and takes the most appropriate view. Use of brackets when mixing relational operators with field references will always avoid possible misinterpretation.
The MATCHES operator matches a string against a pattern consisting of one or more concatenated items from the following list.

- ... Zero or more characters of any type
- 0X Zero or more characters of any type
- nX Exactly n characters of any type
- n-mX Between n and m characters of any type
- 0A Zero or more alphabetic characters
- nA Exactly n alphabetic characters
- n-mA Between n and m alphabetic characters
- 0N Zero or more numeric characters
- nN Exactly n numeric characters
- n-mN Between n and m numeric characters
- "string" A literal string which must match exactly. Either single or double quotation marks may be used. Backslashes may not be used as string quotes in this context.

The values n and m are integers with any number of digits. m must be greater than or equal to n.

The 0A, nA, 0N, nN and "string" patterns may be preceded by a tilde (~) to invert the match condition. For example, ~4N matches four non-numeric characters such as ABCD (not a string which is not four numeric characters such as 12C4).

A null string matches patterns ..., 0A, 0X, 0N, their inverses (~0A, etc) and "."

The 0X and n-mX patterns match against as few characters as necessary before control passes to the next pattern. For example, the string ABC123DEF matched against the pattern 0X2N0X matches the pattern components as ABC, 12 and 3DEF.

The 0N, n-mN, 0A, and n-mA patterns match against as many characters as possible. For example, the string ABC123DEF matched against the pattern 0X2-3N0X matches the pattern components as ABC, 123 and DEF.

The pattern string may contain alternative templates separated by value marks. The MATCHES operator tries each template in turn until one is a successful match against the string. If a match is found, the INMAT() function can be used to retrieve the value position within the pattern that matched.

As an example of the MATCHES operator, the statement

```qmb
IF S MATCHES "1NON'-'1NON" THEN PRINT "OK"
```

would print OK when S contains one or more digits followed by a hyphen and one or more digits. Note the use of 1NON to ensure that at least one digit is present.

The content of two dimensioned matrices may be compared using

```qmb
MAT matrix1 = MAT matrix2
```

as a Boolean element in, for example, an IF statement. The only valid relational operators in this usage are equals and not equals.

**Partial Expression Evaluation**

The default behaviour of QM, in common with many other multivalue database products, is that a logical expression is always completely evaluated before taking any action conditioned by it. For example,
IF IDX MATCHES '1-2N' AND S<IDX> # 0 THEN ...

will test the value of S<IDX> regardless of whether the value in IDX is numeric or not. This would lead to a run time error if IDX is not numeric.

Use of the PARTIAL.EXPRESSIONS setting of the $MODE compiler directive causes logical expression evaluation to terminate as soon as the result can be determined. For a non-numeric value in IDX, the above example would skip the test of S<IDX>.

The difference between these two methods of evaluation is also important if the expression includes a reference to a function that has a side effect. For example,

IF X = 1 AND REMOVE(STR, CODE)

In the default mode of operation, the REMOVE() function is executed regardless of the content of X. With the PARTIAL.EXPRESSIONS mode, the REMOVE() function is only executed if X is 1.

See also:
Data collections, Pattern Matching
**QMBasic - Assignment statements**

Variables may be assigned values by statements of the following forms

- `var op expr` Assign `expr` to `var`
- `var[s,n] = expr` Assign `expr` to substring of `var`
- `var[n] = expr` Assign `expr` to trailing substring of `var`
- `var[d,i,n] = expr` Assign `expr` to delimited substring of `var`
- `var<f> = expr` Assign `expr` to field `f` of `var`
- `var<f,v> = expr` Assign `expr` to field `f`, value `v` of `var`
- `var<f,v,s> = expr` Assign `expr` to field `f`, value `v`, subvalue `s` of `var`
- `var{path} = expr` Assign `expr` to the data collection element with the given element `path`

In all cases, `var` may be a dimensioned matrix element.

The `var op expr` format allows the following operators.

- `=` Simple assignment
- `+=` Add `expr` to original value
- `-=` Subtract `expr` from original value
- `*=` Multiply original value by `expr`
- `/=` Divide original value by `expr`
- `:=` Concatenate `expr` as string to original value
- `&=` Bitwise AND with original value
- `!=` Bitwise OR with original value

Additionally, many other statements set values into variables.

**Substring Assignment**

The substring assignment operator is

`var[s,n] = expr`

In the default compiler modes, substring assignment overlays an existing `n` character long portion of a string, starting at character `s`. If the substring bounds extend beyond the end of the actual value stored in the string, the excess data is lost. If the value of `expr` is longer than the substring to be set, the trailing characters are lost. If the value of `expr` is shorter than the substring to be set, the remainder is filled with spaces.

Two alternative implementations are provided for compatibility with various Pick style multivalue environments. Use of the $MODE$ compiler directive extends the definition above such that the original string is extended if the region to be overwritten extends beyond the end of the current string value.

Use of the $MODE$ compiler directive changes the behaviour of substring assignment considerably. If the value of `s` is negative, the new contents of `var`
are formed by copying the value of \( expr \), adding \( s \) spaces, skipping \( n \) characters of the original value of \( var \) and copying the remainder. If the value of \( s \) is greater than or equal to zero, the new value of \( var \) is formed by copying \( s-1 \) characters of the original value of \( var \), adding spaces if necessary, skipping \( n \) characters, inserting the value of \( expr \) and then copying any remaining characters from the original \( var \).

**Trailing Substring Assignment**

The trailing substring assignment operator is

\[
var[n] = expr
\]

This overwrites the final \( n \) characters of \( var \) with the value of \( expr \). If \( n \) is less than 1, it is treated as 1. If \( n \) exceeds the length of \( var \) it is treated as being the length of \( var \).

If the value of \( expr \) is longer than the substring to be set, the trailing characters are lost. If the value of \( expr \) is shorter than the substring to be set, the remainder is filled with spaces.

The two $MODE compiler directive settings described above do not affect trailing substring assignment.

**Delimited Substring Assignment**

Delimited substring assignment replaces or inserts a portion of a string which is divided into substrings by use of a delimiter character. This character does not have to be one of the mark characters. The first character of string \( d \) is taken as the delimiter character. Starting at substring \( i \), \( n \) substrings are replaced by the value of the assignment expression. For full details of delimited substring assignment, see the description of the FIELDSTORE statement.

**Substring Assignment with Dynamic Arrays**

All variants of substring assignment also allow the operation to be applied to an element of a dynamic array. For example,

\[
\begin{align*}
\text{VAR<FNO>[5]} & \mathbf{= S} \\
\text{VAR<FNO>[1,10]} & \mathbf{= S} \\
\text{VAR<FNO>['*', 5, 1]} & \mathbf{= S}
\end{align*}
\]

**Field Assignment**

Field (or value, or subvalue) assignment replaces an existing field (or value, or subvalue) with the result of the expression. If the specified field, value or subvalue does not already exist within the string, mark characters are added as necessary.

\[
Z<2,3> = expr
\]

When adding a new field at the end of a string, the syntax

\[
Z<-1> = expr
\]

can be used where the field position is specified as any negative value, by convention -1. The QMBasic language will add a new field to receive the result. Similarly, the operations

\[
\begin{align*}
Z<5,-1> & \mathbf{= expr} \\
Z<5,3,-1> & \mathbf{= expr}
\end{align*}
\]
would add a new value or subvalue to the end of existing data.

The way in which the append operation is performed depends on the setting of the COMPATIBLE.APPEND option of the $MODE compiler directive.

By default, QM prefixes the appended data with a field, value or subvalue mark unless the string, field or value in which the item is being appended is completely null. Thus, if $S = "ABC\text{FM}DEF\text{VMFM}XYZ\text{FM}"

$S<-1> = "ghi"$ sets $S$ to "ABC\text{FM}DEF\text{VMFM}XYZ\text{FM}GHI"

$S<1,-1> = "ghi"$ sets $S$ to "ABC\text{VMGHI}FM\text{DEFVMFM}XYZ\text{FM}"

$S<2,-1> = "ghi"$ sets $S$ to "ABC\text{FM}DEF\text{VMGHI}FM\text{XYZFM}"

Setting the COMPATIBLE.APPEND mode modifies the behaviour such that a mark character is not inserted if the final element of the portion of the dynamic array to which data is being appended is null. This is how other multivalue database products work and results in

$S<-1> = "ghi"$ sets $S$ to "ABC\text{FM}DEF\text{VMFM}XYZ\text{FMGHI}"

$S<1,-1> = "ghi"$ sets $S$ to "ABC\text{VMGHI}FM\text{DEFVMFM}XYZ\text{FM}"

$S<2,-1> = "ghi"$ sets $S$ to "ABC\text{FM}DEF\text{VMGHI}FM\text{XYZFM}"

This same rule applies to the INS statement and the INSERT() and REPLACE() functions. Dictionary I-type expressions that use INSERT() or REPLACE() always adopt the default QM behaviour.

Note that the $S<-1>$ syntax should not be used when working with an association because it can lose the relation between the members of the association. Suppose we are working with an inventory in which we are associating a part number, description, and a comment. Every item has a part number and description but the comment is not always present. If we write

$\text{PART.NO}<-1> = \text{part number}$

$\text{DESCRIPTION}<-1> = \text{description}$

$\text{COMMENT}<-1> = \text{text}$

the COMMENT field will not be updated as expected if the text item is a null string. Next time these items are updated, the associations may get out of step.

To avoid this problem, use the field assignment or the REPLACE() function in code such as this:

$\text{PART.NO}<-1> = \text{part number}$

$\text{DESCRIPTION}<-1> = \text{description}$

$\text{COMMENT}<-1> = \text{text}$

where N is the dynamic array field position to be updated. This will ensure that any null values are inserted correctly, so that the association between the values will also be correctly maintained.

See also:

Data collections
QMBasic - Type conversion

QMBasic variables are of variant type, the stored type being determined by the context in which the value was set and conversion being carried out on a temporary basis wherever necessary to perform processing. For example, the following program fragment

\[
A = 962 \\
S = A[2,1]
\]

would result in A containing the integer value 962 and S containing the digit 6 as a string. The type conversion from integer to string is implicit in the use of the substring extraction on the second line.

Where a variable is accessed a very large number of times, there may be performance benefits to be obtained from ensuring that it is stored in an appropriate type thus minimising implicit temporary conversions. The QMBasic language does not have any specific type conversion functions as the automatic type variant nature of the language is adequate for most purposes. Where a variable is to be forced to be a number, this can be achieved by adding zero

\[
A = A + 0
\]

or, more typically, combined with some other operation such as

\[
\text{NUM.INVOICES} = \text{CLI.REC<C.INV.CT>} + 0
\]

Similarly, data can be forced to string format by appending a null string

\[
A = A : ""
\]

This is a much less common operation in real programs.

QMBasic provides a better way to do type conversion by use of the \texttt{INT()}, \texttt{NUMERIC()}, \texttt{STR()} and \texttt{BOOL()} functions.

Developers should avoid over-use of explicit type conversions. The implicit conversion of type variant variables is one of the strong points of the QMBasic language, removing the need for nearly all of the explicit conversions needed in other programming languages.

When the national language support decimal separator character is set to something other than the period, both the selected character and the period are recognised as being a valid decimal separator when converting a numeric value in a string to its internal numeric form.

The Boolean values True and False can be used interchangeably with numeric values 1 and 0. Any operation that is defined to return a Boolean value in numeric form will return 1 or 0. Any operation that uses a Boolean value will treat 0 and a null string as False, everything else as True.
Multivalue Functions

The QMBasic language has many functions that provide multivalued equivalents of their more commonly used single valued counterparts. In each case, these work element by element through the dynamic arrays passed into the functions, performing the operation on each element in turn to produce an equivalent dynamic array of results.

Note that the implementation of the multivalue operations differs across multivalue products with regard to whether the text mark is treated as a data character or as a delimiter. QM treats the text mark as a delimiter character in these functions, processing each text mark delimited item separately.

For example, if we have two dynamic arrays

| A contains   | ABC<sub>FM</sub>DEF<sub>FM</sub>GHI |
| B contains   | 123<sub>FM</sub>456<sub>FM</sub>789 |

We can concatenate these two dynamic arrays in two ways:

- C = A : B sets C to ABC<sub>FM</sub>DEF<sub>FM</sub>GHI123<sub>FM</sub>456<sub>FM</sub>789
- C = CATS(A, B) sets C to ABC123<sub>FM</sub>DEF456<sub>FM</sub>GHI789

The main multivalued string functions are

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CATS()</td>
<td>Concatenate elements of a dynamic array</td>
</tr>
<tr>
<td>COUNTS()</td>
<td>Multivalued variant of COUNT()</td>
</tr>
<tr>
<td>FIELDS()</td>
<td>Multivalued variant of FIELD()</td>
</tr>
<tr>
<td>FMTS()</td>
<td>Format elements of a dynamic array</td>
</tr>
<tr>
<td>ICONVS()</td>
<td>Perform input conversion on a dynamic array</td>
</tr>
<tr>
<td>INDEXS()</td>
<td>Multivalued equivalent of INDEX()</td>
</tr>
<tr>
<td>MATCHES()</td>
<td>Multivalued application of pattern matching (MATCHES operator)</td>
</tr>
<tr>
<td>NUMS()</td>
<td>Multivalued variant of NUM()</td>
</tr>
<tr>
<td>OCONVS()</td>
<td>Perform output conversion on a dynamic array</td>
</tr>
<tr>
<td>SPACES()</td>
<td>Multivalued variant of SPACE()</td>
</tr>
<tr>
<td>STRS()</td>
<td>Multivalued variant of STR()</td>
</tr>
<tr>
<td>SUBSTRINGS()</td>
<td>Multivalued substring extraction</td>
</tr>
<tr>
<td>TRIMBS()</td>
<td>Multivalued variant of TRIMB()</td>
</tr>
<tr>
<td>TRIMFS()</td>
<td>Multivalued variant of TRIMF()</td>
</tr>
<tr>
<td>TRIMS()</td>
<td>Multivalued variant of TRIM()</td>
</tr>
</tbody>
</table>

For compatibility with other multivalue products, QM supports four multivalued arithmetic functions that are equivalent to the implied value by value execution of the arithmetic operators. For example,

\[ X = ADDS(A, B) \]
is equivalent to
\[ X = A + B \]

**ADDS()**
Add corresponding elements of two dynamic arrays

**DIVS()**
Divide corresponding elements of two dynamic arrays

**MULS()**
Multiply corresponding elements of two dynamic arrays

**SUBS()**
Subtract corresponding elements of two dynamic arrays

There are also a number of multivalued logical functions. These provide equivalents to the relational operators and other functions that return Boolean values.

For example, the **GTS(arr1, arr2)** function takes two dynamic arrays and returns a new dynamic array of True / False values indicating whether the corresponding elements of *arr1* are greater than those of *arr2*.

Thus, if A contains 11\text{FM}0\text{VM}17\text{VM}PQR\text{FM}2 and B contains 12\text{FM}0\text{VM}14\text{VM}ACB\text{FM}2

\[ C = \text{GTS}(A, B) \]

Returns C as 0\text{FM}0\text{VM}1\text{VM}1\text{FM}0

The multivalued logical functions are

**ANDS()**
Multivalued logical AND

**EQS()**
Multivalued equality test

**GES()**
Multivalued greater than or equal to test

**GTS()**
Multivalued greater than test

**LES()**
Multivalued less than test

**LTS()**
Multivalued less than or equal to test

**NES()**
Multivalued inequality test

**NOTS()**
Multivalued logical NOT

**ORS()**
Multivalued logical OR

The **IFS()** function returns a dynamic array constructed from elements chosen from two other dynamic arrays depending on the content of a third dynamic array.

**IFS(control.array, true.array, false.array)**

where

- **control.array** is a dynamic array of True / False values.
- **true.array** holds values to be returned where the corresponding element of **control.array** is True.
false.array holds values to be returned where the corresponding element of control.array is False.

The IFS() function examines successive elements of control.array and constructs a result array where elements are selected from the corresponding elements of either true.array or false.array depending on the control.array value.

Example

A contains 1VM0VM0VM1VM1VM0
B contains 6VM2VM3VM4VM9VM6VM3
C contains 2VM8VM5VM0VM3VM1VM3

D = IFS(A, B, C)

D now contains 6VM8VM5VM4VM9VM6VM3

Used in combination, and often with REUSE(), these functions can give very elegant solutions to apparently complex problems. For example, to determine how many elements of a dynamic array contain a value greater than four:

N = SUM(GTS(X, REUSE(4)))
File Processing

QMBasic programs usually need to access database files. This section discusses the various techniques available. Further information can be found by following the links to detailed sections.

The QM file system supports two distinct types of file:

- **Hashed files** use a mathematical approach to locate data such that, when correctly configured, it should be possible to read any record with just one disk access regardless of the number of records in the file. For more information on the creation and configuration of these files see the section on **dynamic files**.

- **Directory files** do not offer the high performance of hashed files but allow access to their data from outside of the QM environment. For this reason, they are typically used for data interchange between applications. They are also ideal for storing extremely large records. For more information on the creation of these files, see the section on **directory files**.

Directory files also allow data to be processed in a line by line manner or as a simple byte stream. There are also special program operations to simplify reading and writing comma separated data as used, for example, by some spreadsheet packages. For more information on this style of access, see **sequential file i/o**.

Opening Files

Before a file can be processed, it must be opened. This is normally done using the OPEN statement, identifying the file by referencing the name of the corresponding F-type VOC record. By having this level of indirection, the physical location of the file can be changed without affecting the application; all that is necessary is to edit the VOC record to reference the new file location. There are three special file name syntaxes available to reference files in other accounts without needing a Q-type VOC record:

- Implicit Q-pointer: account:file
- Implicit QMNet pointer: server:account:file
- Pathname: PATH:pathname

Because these syntaxes potentially weaken the security provided by the VOC indirection, their availability is determined by a system configuration parameter, **FILERULE**.

It is also possible for an application to open files directly by pathname using the OPENPATH statement. This should only be used where the normal VOC indirection is not appropriate.

A typical application may open many files simultaneously and it is therefore necessary to have a way to determine which file is being referenced by subsequent data transfer operations. This link is provided by the OPEN and OPENPATH statements setting up a file variable which is then used in other operations on the same file. The file remains open so long as the file variable remains in place. Overwriting the file variable will implicitly close the file that it referenced. Exit from the program will discard local variables and hence close the file.

Most applications adopt a convention for the names of variables. Many examples in this documentation use a convention where the file name is contracted to three or four characters and a suffix of .F is added to form the file variable name. Thus a file named ORDERS might be referenced via a file variable named ORD.F. It is common to continue the convention into other variables so that, for example, ORDERS file record ids might be stored in ORD.ID and records could be read from the file into the ORD.REC variable. These are examples only. There is no restriction on naming imposed by QM itself.
A file variable may be copied, just like any other variable. In this case, the file remains open until the last file variable referencing it is discarded or overwritten.

There are two factors that limit the number of files that can be open at one time. Firstly, QM has an internal file table that contains a reference to every distinct file open on the system by all QM processes. The size of this table is set by the NUMFILES configuration parameter. If several users all open the same file, that only requires one entry in the table. If a program attempts to open a file when the table is full, the operation will fail, taking the ELSE clause to allow the program to report an error. The LIST.FILES command can be used to monitor how close a system is to reaching this configuration limit.

The second limit is imposed by the operating system. On some systems this may be configurable, on others it is fixed. QM tries to hide this limit by implementing a mechanism whereby, if the limit is reached, the file that has not been accessed for longest is closed internally to make room for the new file. Subsequent access to the file that has been closed will automatically reopen it, probably closing something else to make space. Although this mechanism is totally automatic and gives the developer the illusion that there is no limit, the impact on performance can be quite serious. It is strongly recommended that any configurable limit imposed by the operating system is set to an appropriate value.

Opening a file is a complex process. Although QM maintains a file cache to improve the situation, developers should avoid continually re-opening the same file. One useful way to achieve this is to place the file variable in a common block so that it is not discarded when the program or subroutine exits. By using this technique it is possible for a program to open all of its main data files as it starts up and to keep them open for the entire life of the application. Keeping large numbers of files open will require careful configuration of NUMFILES and possibly the corresponding operating system parameters.

Reading, Writing and Deleting Data

Programs read data using the QMBasic READ statement. With a hashed file, the internal processing of this statement applies the hashing process to read just the group that would contain the requested record and then locates the record within that group. If it is not found, it is not in the file and there is no need to look elsewhere. This process ensures that hashed files give best performance. For a directory file, QM uses operating system functions to locate and read the requested item. This will not give the performance of hashed files as it requires a scan of the directory to locate the item.

The READ statement returns a variable that contains a dynamic array representing the data of the requested record. The program can then use the various dynamic array operations such as field extraction to access the data in the record.

If the record is to be updated by the application, it is essential to ensure that other processes cannot update the record at the same time. This protection is provided by QM's locking mechanisms and corresponding QMBasic statements, most importantly READU. A program should never write or delete a record unless it owns a lock to protect it. There is a configuration parameter, MUSTLOCK, that allows administrators to enforce strong locking rules. Unfortunately, this cannot be made the default behaviour as there is much software which does not use locking because the developer knew that there could never be an interaction with other processes.

A data record is written to the file using the QMBasic WRITE statement. If the record already exists in the file, the new version replaces the previous one. If the record does not already exist, the write operation adds it to the file. The record lock is automatically released when the write completes though there is an alternative statement, WRITEU, that retains the lock.

A data record is deleted from the file using the QMBasic DELETE statement. The record lock is automatically released after the record has been deleted.
The QMBasic statements named above work with dynamic arrays. There is an alternative style of file I/O that uses dimensioned matrices. For details, see Matrix File I/O.

Select Lists and Alternate Key Indices

The READ statement requires that the program knows the id of the record it needs to read. To process a file sequentially or to process only records that meet a specific condition, programs use a select list. This list may be generated by executing a query processor SELECT operation or by use of the QMBasic SELECT statement. Whichever method is used, the program then reads items from the list using the READNEXT statement, typically in a loop that then uses READ or one of its locking variants to process each record from the list.

Building a select list requires the system to traverse the entire file, examining every record. For situations where only a small proportion of records are to be selected, an alternate key index can give substantial performance improvements. Effectively, this is a set of pre-built select lists based on the content of a specific field or the result of evaluating an I-type expression. The index is automatically updated whenever a change is made to the file. For well chosen indexed fields, the cost of this additional update on write is usually significantly outweighed by the performance improvement of being able to go directly to the desired set of records.

The ON ERROR Clause

Most of the QMBasic file handling statements have an optional ON ERROR clause. This is rarely needed by applications but allows a program to trap an error that would otherwise cause QM to abort the program. If an ON ERROR clause is present, the program can take its own recovery action or display alternative diagnostic messages. Developers should avoid using the ON ERROR clause simply to condition an ABORT statement as this will usually give less diagnostic information than would have appeared if no ON ERROR clause had been present.

Examples

```
OPEN 'CLIENTS' TO CLI.F ELSE STOP 'Cannot open CLIENTS'
LOOP   DISPLAY 'Enter client number: ' :   INPUT CLI.IDUNTIL CLI.ID = ''   READ CLI.REC FROM CLI.F, CLI.ID THEN      DISPLAY CLI.REC<2>   END ELSE      DISPLAY 'Client not found'   ENDREPEAT
```

This short program opens the CLIENTS file and then enters a loop in which it prompts for a client number, reads the client record and displays the content of field 2. The loop continues until the user enters a blank client number.

This example shows why direct use of field numbers in programs is a bad idea. Anyone reading this program has no idea what information about the client is being displayed. A better approach is to use the EQUATE statement to define names for each field (typically in an include record). The data display statement might then become

```
DISPLAY CLI.REC<CLI.NAME>
```
which suggests to anyone reading the program that it is displaying the client's name. The GENERATE command can be used to construct an include record of field names from the file dictionary rather than having to maintain two separate descriptions of the data.

```
SELECT @VOC TO 1
LOOP
  READNEXT ID FROM @VOC ELSE EXIT
  READ VOC.REC FROM @VOC, ID THEN
    IF VOC.REC[1,1] = 'F' THEN
      OPEN ID TO TEST.F ELSE
        DISPLAY 'File ' : ID : ' cannot be opened'
      END
    END
  END
END REPEAT
```

This program uses the system defined file variable @VOC to reference the VOC instead of opening it explicitly. The SELECT statement builds a list of all records in the file into select list 1 which is then processed in the loop. For each item in the list, the record is read from the VOC and, if it is an F-type record, the program attempts to open the file. If it cannot be opened, an error message is displayed.

Note the use of EXIT to exit from the loop when the list is exhausted. Some multivalue environments do not support this statement and require developers to devise alternative exit schemes that are generally not as efficient.

Note also that the file opened to variable TEST.F is not explicitly closed. Each OPEN will implicitly close the previous file as the file variable is overwritten. The final file opened will remain open until the program terminates.

This program has needed to include a test to process only F-type VOC records. Alternatively, the program could use the query processor to build a list of F-type records and then process all records in this list:

```
EXECUTE "SELECT VOC WITH TYPE = 'F' TO 1"
LOOP
  READNEXT ID FROM 1 ELSE EXIT
  READ VOC.REC FROM @VOC, ID THEN
    OPEN ID TO TEST.F ELSE
      DISPLAY 'File ' : ID : ' cannot be opened'
    END
  END
END
```

Although this approach may look simpler and does not require the unwanted records to be read, it is actually less efficient than the first method as the query processor will need to read every record and the loop then re-reads the records of interest. In the previous example, use of the QMBasic SELECT actually only sets a pointer to the start of the file and the subsequent READNEXT reads each group when it needs to process the first record from the group, effectively reading the file only once. This exposes an interesting problem that is highlighted in the next example.

```
OPEN 'CLIENTS' TO CLI.F ELSE STOP 'Cannot open CLIENTS'
SELECT CLI.F TO 1
LOOP
  READNEXT CLI.ID FROM 1 ELSE EXIT
```

4.0-0
READU CLI.REC FROM CLI.F, CLI.ID THEN
  RECORDLOCKU CLI.F, '0':CLI.ID
  WRITE CLI.REC TO CLI.F, '0':CLI.ID
  DELETE CLI.F, CLI.ID
END
REPEAT

The above program might be used to convert a CLIENTS file to add a zero on the front of each record id, perhaps to allow more clients on a system where the application requires fixed length ids. Because it is the READNEXT that actually traverses the file rather than the SELECT statement, new records written to higher numbered groups would be seen by a later READNEXT and get processed for a second time. For example, if the record for client number 1234 was in group 6 and the new version of this record with id 01234 hashed to group 10, it would appear in the list constructed when processing reaches group 10 and the record would be renamed once more to become 001234.

Although programs that might suffer from this problem are rare, we need to force completion of the record selection before entering the loop. One way to do this would be to use the query processor SELECT instead of the QMBasic equivalent:

  EXECUTE 'SELECT CLIENTS TO 1'

Note the use of RECORDLOCKU to set an update lock on the record to be added to the file. Although this is probably strictly unnecessary in this example because the new record will not already exist, it does ensure compliance with the locking rules.
Matrix File I/O

QMBasic has two styles of file i/o that may be freely mixed within an application. Using READ and WRITE to transfer data using dynamic arrays is simpler and usually faster for programs that do little processing of the data or work with records that have few fields. For programs that perform a significant amount of processing of the data in records that have many fields, it may be worth the cost of breaking the fields into separate elements of a dimensioned matrix using MATREAD and MATWRITE.

These statement have the same locking variants as their dynamic array counterparts. They also share an almost identical syntax where the prefix MAT is used to select the matrix version of the operation and the variable representing the database record must be a dimensioned array. For example, the dynamic array read:

```
READ var FROM filevar, id
```

becomes

```
MATREAD array FROM filevar, id
```

The MATREAD statement places each field of the record into a separate element of the array, keeping values and subvalues together as these are instances of the same data item.

For example, if a record has three fields, the second of which is multivalued:

```
A FM B1 VM B2 FM C
```

using MATREAD to read this into a three element (plus the zero element) matrix would result in:

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>B1 VM B2</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td></td>
</tr>
</tbody>
</table>
```

The MATWRITE operation joins together each element of the matrix, inserting field marks between them and writes this to the file.

If the matrix has more elements than there are fields in the record, the excess elements are set to null strings:

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>B1 VM B2</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

The INMAT() function can be used after a MATREAD to determine how many elements of the matrix were used. The MATWRITE operation ignores all trailing empty fields so that above situation would not write two empty fields at the end of the record.
If the matrix has fewer elements than there are fields in the record, the zero element is used to store the excess data. Consider the record with five fields and a matrix with three elements:

<table>
<thead>
<tr>
<th>0</th>
<th>D_{FM}E</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>B_{1vm}B2</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
</tr>
</tbody>
</table>

The \texttt{MATWRITE} operation adds the contents of the zero element to the record formed from the remaining elements of the matrix, reconstructing the correctly formed data. The zero element thus acts as an "overflow bucket" allowing programs that did not expect to find the excess data to function correctly.

Pick style matrices do not have a zero element. In this case, the excess data is stored in the final element of the matrix:

<table>
<thead>
<tr>
<th>1</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>B_{1vm}B2</td>
</tr>
<tr>
<td>3</td>
<td>C_{FM}D_{FM}E</td>
</tr>
</tbody>
</table>

This is likely to cause the program to malfunction if it updates element 3 where it expected only to find the third field of the database record. To avoid this, Pick style programmers usually ensure that the matrix has at least one more element than they expect it to need, effectively moving the "overflow bucket" to the end of the matrix.
Sequential File I/O

Directory files are so called because they are represented by an operating system directory. The records in these files are represented by operating system files in the directory. These files do not give the high performance of hashed files but they allow access to the data from outside of QM. They are therefore particularly useful for data interchange.

Records in directory files are sometimes very large and may consist of a number of lines of textual information with a fixed layout. In such cases, it may be useful to process the data line by line. QM provides statements to perform sequential reading or writing of text data. These can only be used with directory files.

An item is opened for sequential processing using the OPENSEQ statement. This has two forms, one that opens a record in a directory file by name:

```
OPENSEQ file, id TO filevar
```

the other opens a file by pathname:

```
OPENSEQ pathname TO filevar
```

In both forms, the statement takes the optional ON ERROR, LOCKED, THEN and ELSE clauses. At least one of the THEN and ELSE clauses must be present. Because the OPENSEQ operation is effectively opening a record, it applies a lock to this record to prevent other users overwriting it.

The OPENSEQ statement will take the ELSE clause for three reasons:
- The file does not exist.
- The file exists but is not a directory file.
- The file exists as a directory file but the record does not exist.

The last of these three situations would be an error in a program that is intending to read the item but is usually not an error in a program that will write to the item. The STATUS() function can be used to determine which of the above three conditions exist as discussed in the detailed OPENSEQ statement description.

OPENSEQ also has options to open the item in read-only mode, append to an existing item, or overwrite an existing item.

The QMBasic statements that can be used to access the sequential item are:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>READSEQ</td>
<td>Read text line by line</td>
</tr>
<tr>
<td>READBLK</td>
<td>Read a given number of bytes</td>
</tr>
<tr>
<td>WRITESEQ</td>
<td>Write text line by line</td>
</tr>
<tr>
<td>WRITESEQF</td>
<td>Write text line by line, flushing to disk before continuing</td>
</tr>
<tr>
<td>WRITEBLK</td>
<td>Write a given number of bytes</td>
</tr>
<tr>
<td>READCSV</td>
<td>Read comma separated variable (CSV) format data</td>
</tr>
<tr>
<td>WRITECSV</td>
<td>Write comma separated variable format data</td>
</tr>
<tr>
<td>SEEK</td>
<td>Position within the sequential item</td>
</tr>
<tr>
<td>NOBUF</td>
<td>Suppress buffering</td>
</tr>
<tr>
<td>WEOFSEQ</td>
<td>Write end of file (truncate the item)</td>
</tr>
<tr>
<td>CLOSESEQ</td>
<td>Close the sequential item, flushing buffers and releasing the lock.</td>
</tr>
</tbody>
</table>

Examples
This short program reads lines from a text file, C:\PRICES. Each line within this file has a stock part number in the first five characters and a new price in external format in the next eight characters. For each line, the program reads the corresponding STOCK file record and updates field STK.PRICE to contain the internal form of the price value. The token STK.PRICE would typically be defined in an include record.

This program is a variation on the first example where the import data contains comma separated items as might have been written by a spreadsheet tool such as Excel. The READCSV statement reads the first two comma separated items in each line of text into STK.ID and PRICE. Any additional values on the line are discarded.

This program is a variation on the first example where the import data contains comma separated items as might have been written by a spreadsheet tool such as Excel. The READCSV statement reads the first two comma separated items in each line of text into STK.ID and PRICE. Any additional values on the line are discarded.

This program is a variation on the first example where the import data contains comma separated items as might have been written by a spreadsheet tool such as Excel. The READCSV statement reads the first two comma separated items in each line of text into STK.ID and PRICE. Any additional values on the line are discarded.

This program is a variation on the first example where the import data contains comma separated items as might have been written by a spreadsheet tool such as Excel. The READCSV statement reads the first two comma separated items in each line of text into STK.ID and PRICE. Any additional values on the line are discarded.
This program creates a text item in C:\EXPORT.CSV where each line contains the stock part number, the price and the quantity on hand as a comma separated list suitable for import into spreadsheets such as Excel.
Default File Variable

For compatibility with other environments, QMBasic supports the concept of the **default file variable** but, because use of this feature can result in mis-typed programs compiling without errors but not functioning correctly, it must be enabled by use of the STDFIL or STDFIL.SHARED option of the `$MODE` compiler directive or the equivalent setting in the `$BASIC_OPTIONS` record.

The default file variable allows a program to open a file without referencing a specific variable name.

When using the STDFIL.SHARED option of `$MODE`, the default file variable is an implied reference to the @STDFIL variable. All programs and subroutines executed at the same command processor level share a single default file variable. It is therefore important that an application uses this feature with care. @STDFIL is maintained separately for each command processor level. Thus a program that uses the default file variable can use the QMBasic EXECUTE statement to run a separate program that uses the default file variable to reference a different file. On return from the EXECUTE, @STDFIL references the original file.

When using the STDFIL option of `$MODE` instead of STDFIL.SHARED, the default file variable is maintained separately for each program or subroutine. The @STDFIL variable is local to the program in which it is referenced.

A single application can validly include a mix of programs that use both modes but use of both STDFIL and STDFIL.SHARED together in a single program is equivalent to use of STDFIL.

When this feature is enabled, the syntax of many of the QMBasic statements that access files is modified to allow the file variable to be omitted. The revised syntax for use with the default file variable is as shown below but not referenced elsewhere in this manual.

```
CLEARFILE {ON ERROR statement(s)}
CLOSE {ON ERROR statement(s)}
DELETE record.id {ON ERROR statement(s)}
DELETEU record.id {ON ERROR statement(s)}
FILELOCK {ON ERROR statement(s)} {LOCKED statement(s)}
FILEUNLOCK {ON ERROR statement(s)}
OPEN {dict.expr, } filename {READONLY} {ON ERROR statement(s)} {THEN statement(s)} {ELSE statement(s)}
OPENPATH pathname {READONLY} {ON ERROR statement(s)} {THEN statement(s)} {ELSE statement(s)}
MATREAD mat FROM record.id {ON ERROR statement(s)} {THEN statement(s)} {ELSE statement(s)}
MATREADL mat FROM record.id {ON ERROR statement(s)} {LOCKED statement(s)} {THEN statement(s)} {ELSE statement(s)}
MATREADU mat FROM record.id {ON ERROR statement(s)} {LOCKED statement(s)} {THEN statement(s)} {ELSE statement(s)}
MATWRITE mat TO record.id {ON ERROR statement(s)}
MATWRITEU mat TO record.id {ON ERROR statement(s)}
READ rec FROM record.id {ON ERROR statement(s)} {THEN statement(s)} {ELSE statement(s)}
```
READL rec FROM record.id {ON ERROR statement(s)} {LOCKED statement(s)} {THEN statement(s)} {ELSE statement(s)}

READU rec FROM record.id {ON ERROR statement(s)} {LOCKED statement(s)} {THEN statement(s)} {ELSE statement(s)}

RELEASE , record.id {ON ERROR statement(s)} [Note comma before record.id]

SELECT {TO list} {ON ERROR statement(s)}

SELECTN {TO list} {ON ERROR statement(s)}

SELECTV {TO list} {ON ERROR statement(s)}

WRITE rec TO record.id {ON ERROR statement(s)}

WRITEU rec TO record.id {ON ERROR statement(s)}

QM does not support use of the default file variable with READV, WRITEV and the locking related variants of these statements.
Object Oriented Programming

QMBasic includes support for object orientated programming. Users familiar with other object oriented languages will find that QM offers many of the same concepts but, because they are integrated into an existing programming environment, there may be some significant differences in usage.

What is an Object?

An object is a combination of data and program operations that can be applied to it. An object is defined by a class module, a QMBasic program that is introduced by the \texttt{CLASS} statement and contains the definitions of persistent data items and public subroutine and functions. An object is a run time instance of the class, \texttt{instantiated} by use of the \texttt{OBJECT()} function

\begin{verbatim}
OBJ = OBJECT("MYCLASS")
\end{verbatim}

where "MYCLASS" is the catalogue name of the class module. The \texttt{OBJ} variable becomes a reference to an \texttt{instance} of the class.

A second use of the \texttt{OBJECT()} function with the same catalogue name will create a second instance of the object. On the other hand, copying the object variable creates a second reference to the same instance. A clone of an existing object can be created by use of the \texttt{OBJECT()} function in which the argument is a reference to the object that is to be cloned.

Class Module Variables

There are five styles of data storage available in a class module.

\textbf{Public variables} are declared using the \texttt{PUBLIC} statement, for example,

\begin{verbatim}
PUBLIC A, B(5)
\end{verbatim}

at the start of the class module before any executable program statements. Each instance of a class has its own separate public variables but the values persist between entries to that object instance. Public variables are initially unassigned and may be accessed from within the class module or, subject to rules set out below, from outside.

\textbf{Private variables} are declared using the \texttt{PRIVATE} statement, for example,

\begin{verbatim}
PRIVATE C, D, E
\end{verbatim}

at the start of the class module before any executable program statements. Each instance of a class has its own separate private variables but the values persist between entries to the object instance. Private variables are initially unassigned and may be accessed only from within the class module.

\textbf{Shared variables} are declared using the \texttt{SHARED} statement, for example,

\begin{verbatim}
SHARED PUBLIC F, G(10), H
\end{verbatim}

at the start of the class module before any executable program statements. Each instance of the same class shares the same variables. Values stored in shared variables persist until the last reference to the class is discarded. Shared variables are initially unassigned and may be defined as either private or public.

\textbf{Common variables} are declared using the \texttt{COMMON} statement, for example,

\begin{verbatim}
COMMON I, J(8)
\end{verbatim}

or

\begin{verbatim}
COMMON /NAME/ K, L
\end{verbatim}
Local variables are either not declared at all or are declared using a DIMENSION (DIM) statement. As in other program types, they exist only while the object instance in which they appear is active and are discarded on leaving the program. Local variables are accessible only from within the object instance and are initially unassigned.

Private and public data items are frequently used to store what other object oriented programming environments would term property values. These data items are initially unassigned unless an initialisation value is present, for example

```qmbasic
PRIVATE WIDTH = 80
```

which will set WIDTH to 80 prior to execution of the CREATE.OBJECT subroutine described below.

Public Functions and Subroutines

Another important difference between class modules and other program types is that a class module usually has multiple entry points, each corresponding to a public function or public subroutine. Indeed, simply calling the class module by its catalogue name will generate a run time error.

Just as with conventional QMBasic functions and subroutines, a public function must return a value to its caller whereas a public subroutine does not (though it can do so by updating its arguments).

A public function is defined by a group of statements such as

```qmbasic
PUBLIC FUNCTION XX(A,B,C)   ...
   ...processing...
   RETURN Z
END
```

where XX is the function name, A, B and C are the arguments (optional), and Z is the value to be returned to the caller.

A public subroutine is defined by a group of statements such as

```qmbasic
PUBLIC SUBROUTINE XX(A,B,C)   ...
   ...processing...
   RETURN
END
```

where XX is the subroutine name and A, B and C are the arguments (optional). A whole matrix can be referenced as an argument by following it with the dimension values. For example,

```qmbasic
PUBLIC SUBROUTINE CALC(CLIENT, CLI.REC(1), TOTVAL)
```

In this example, the dimension value has been shown as 1 to emphasise that the actual value is irrelevant. The compiler uses this purely to determine that CLI.REC is a single dimensional matrix, possibly representing a database record read using MATREAD. The alternative syntax used with SUBROUTINE statements by prefixing the matrix name with MAT and using a DIMENSION statement to set dimensionality is not available for public subroutines and functions.

The number of arguments in a public function or subroutine is normally limited to 32 but this can be increased using the MAX.ARGS option of the CLASS statement.
Both styles of public routine allow use of the VAR.ARGS qualifier after the argument list to indicate that it is of variable length. Argument variables for which the caller has provided no value will be unassigned. The ARG.COUNT() function can be used to find the actual number of arguments passed. A special syntax of three periods (...) used as the final argument specifies that unnamed scalar arguments are to be added up to the limit on the number of arguments. These can be accessed using the ARG() function and the SET.ARG statement. See the PUBLIC statement for more details of this feature.

It is valid for a class module to contain combinations of a PUBLIC variable, PUBLIC SUBROUTINE and PUBLIC FUNCTION with the same name. If there is a public subroutine of the same name as a public variable, the subroutine will be executed when a program using the object attempts to set the value of the public item. If there is a public function of the same name as a public variable, the function will be executed when a program using the object attempts to retrieve the value of the public item. If both are present, the public property variable will never be directly visible to programs using the object.

Sometimes an application developer may wish a public variable to be visible to users of the class for reading but not for update. Although this could be achieved by use of a dummy PUBLIC SUBROUTINE that ignores updates or reports an error, public variables may be defined as read-only by including the READONLY keyword after the variable declaration:

PUBLIC A READONLY
or
PUBLIC B(5) READONLY

Referencing an Object

References to an object require two components, the object variable and the name of a property or method within that object. The syntax for such a reference is

OBJ->PROPERTY

or, if arguments are required,

OBJ->PROPERTY(ARG1, ARG2, ...)

Any argument may reference a whole matrix by prefixing the matrix name with the keyword MAT, for example

OBJ->CALC(CLIENT, MAT CLI.REC, TOTVAL)

The names of public variables, subroutines and functions in object references are case insensitive, even if the class module is compiled with the CASE.SENSITIVE setting of the $MODE compiler directive.

When used in a QMBasic expression, for example,

ITEMS += OBJ->LISTCOUNT

the object reference returns the value of the named item, in this case LISTCOUNT. This may be a public variable or the value of a public function. If the same name is defined as both, the public function is executed.

When used on the left of an assignment, for example,

OBJ->WIDTH = 70

the object reference sets the value of the named item, in this case WIDTH. This may be a public variable or the value of a public subroutine that takes the value to be assigned as an argument. If the same name is defined as both, the public subroutine is executed.
This dual role of public variables and functions or subroutines makes it very easy to write a class module in which, for example, a property value may be retrieved without execution of any program statements inside the object but setting the value executes a subroutine to validate the new value.

**Using Dimensions and Arguments**

Public variables may be dimensioned arrays. Subscripts for index values are handled in the usual way:

```
OBJ->MODE(3) = 7
```

where MODE has been defined as a single dimensional array. If MODE has an associated public subroutine, the indices are passed via the arguments and the new value as the final argument. Thus, if MODE was defined as

```
PUBLIC SUBROUTINE MODE(A,B)
```

the above statement would pass in A as 3 and B as 7.

**Execution of Object Methods**

Other object-oriented languages usually provide *methods*, subroutines that can be executed from calling programs to do some task. QMBasic class modules do this by using public subroutines. The calling program uses a statement of the form:

```
OBJ->RESET
```

where RESET is the name of the public subroutine representing the method. Again, arguments are allowed:

```
OBJ->RESET(5)
```

This leads to an apparent syntactic ambiguity between assigning values to public properties and execution of methods. Actually, there is no ambiguity but the following two statements are semantically identical:

```
OBJ->X(2,3)  
OBJ->X(2) = 3
```

**Expressions as Property Names**

All of the above examples have used literal (constant) property names. QMBasic allows expressions as property names in all contexts using a syntax

```
OBJ->(expr)
```

where *expr* is an expression that evaluates to the property name.

**Object References in Subroutine Calls**

Any reference to an object element in a subroutine call, for example

```
CALL SUBNAME(OBJ->VAR)
```

is considered to be read access and is passed by value. If the subroutine updates the argument, this will not update the object property value.

**The ME Token**
Sometimes an object needs to reference itself. The reserved data name ME can be used for this purpose:

```
ME->RESET
```

Note that setting a private or public variable inside the object to the ME reference, for example

```
PUBLIC MYSELF
MYSELF = ME
```

would create a situation where the object can never be discarded as there is always a reference to it.

**The CREATE.OBJECT Subroutine**

When an object is instantiated using the `OBJECT()` function, part of this process checks whether there is a public subroutine named CREATE.OBJECT and, if so, executes it. This can be used, for example, to preset default values in public and private variables. Up to 32 arguments may be passed into this subroutine by extending the `OBJECT()` call to include these after the catalogue name of the class module.

**The DESTROY.OBJECT Subroutine**

An object remains in existence until the last object variable referencing it is discarded or overwritten. At this point, the system checks for a public subroutine named DESTROY.OBJECT and, if it exists, it is executed. This subroutine is guaranteed to be executed, even if the object variable is discarded as part of a program failure that causes an abort. The only situation where an object can cease to exist without this subroutine running to completion is if the DESTROY.OBJECT subroutine itself aborts.

**The MAIN Subroutine**

If a class module includes a public subroutine named MAIN, an object instantiated from that class can be executed as a main program from the command processor using `RUN` or a catalogue reference in exactly the same way as a PROGRAM module but cannot be executed using `CALL`. The CREATE.OBJECT subroutine, if present, will be executed followed by the MAIN subroutine.

**The UNDEFINED Name Handler**

The optional UNDEFINED public subroutine and/or public function can be used to trap references to the object that use property names that are not defined. This handler is executed if a program using the object references a name that is not defined as a public item. The first argument will be the undefined name. Any arguments supplied by the calling program will follow this. The `ARG.COUNT()` and `ARG()` functions can be used to help extract this data in a meaningful way.

If there is no UNDEFINED subroutine/function, object references with undefined names cause a run time error.

**Inheritance**

Sometimes it is useful for one class module to incorporate the properties and methods of another. This is termed *inheritance*.

Use of the `INHERITS` clause of the `CLASS` statement effectively inserts declaration of a private variable of the same name as the inherited class (removing any global catalogue prefix character) and adds
name = OBJECT(inherited.class)
INHERIT name

to the CREATE.OBJECT subroutine.

Alternatively, inheritance can be performed during execution of the object by direct use of the
INHERIT statement.

The name search process that occurs when an object is referenced scans the name table of the
original object reference first. If the name is not found, it then goes on to scan the name tables of
each inherited object in the order in which they were inherited. Where an inherited object has itself
inherited further objects, the lower levels of inheritance are treated as part of the object into which
they were inherited. If the name is not found, the same search process is used to look for the
undefined name handler.

An inherited object can subsequently be disinherited using DISINHERIT.

Debugging an Object Instance

The names of arguments to a public subroutine are not known to the debugger. Instead, a special
positional reference of the form "*Arg1" is needed when displaying argument variables. The *Arg
prefix is case insensitive and the argument number may include leading zeroes. For example, display
of the second argument could use a debugger command such as
/*arg2

All of the qualifiers to variable references such as array indices and dynamic array element positions
may be used.

Syntax Summary

CLASS name {INHERITS class1, class2...}
  PUBLIC A {READONLY}, B(3), C
  PRIVATE P, Q, R
  SHARED X, Y, Z

  PUBLIC SUBROUTINE SUB1(ARG1, ARG2) {VAR.ARGS}
    ...processing...
  END

  PUBLIC FUNCTION FUNC1(ARG1, ARG2) {VAR.ARGS}
    ...processing...
    RETURN RESULT
  END

  ...Other QMBasic subroutines...
END

There is a sample QMBasic class module named INDEX.CLS in the BP file of the QMSYS
account, catalogued as !INDEX.CLS. This class allows an application to scan an index one record id
at a time instead of one indexed value at a time. Full details of its use and internal operation can be
found in the source code.

See also:
CLASS, DISINHERIT, INHERIT, OBJECT(), PRIVATE, PUBLIC.
Exception Handling

An exception is a named event, typically an error, that can be caught and handled by an application. Exceptions are thrown either by use of the THROW statement in an application program or automatically by QM as an alternative to the abort events normally raised by data type errors and similar failures.

The actual name of an exception has no significance to QM but should ideally relate to the situation that the exception handles. Exception names are case insensitive and may be up to 63 characters. Application developers should avoid creating exception names that begin with "SYS." and names containing a $ symbol as these are reserved as described below.

QMBasic allows creation of exception handlers in a manner broadly similar to that of some other programming languages by use of a TRY/CATCH construct:

```
TRY {statement}
    {statements}
CATCH exception {, exception...} {, exception...} {, exception...}
    {statements}
CATCH exception {, exception...} {, exception...} {, exception...}
END
```

where

- `statement(s)` are any executable QMBasic statements.
- `exception` is an exception name. The name may optionally be enclosed in quotes. An exception name may be followed by the keyword DUMPING to cause a process dump file to be created prior to unwinding the stack back to the exception handler.

A single TRY construct may have multiple CATCH clauses, each handling one or more exceptions.

The TRY/CATCH construct executes the statement(s) in the TRY clause, monitoring for named exceptions being thrown. The TRY clause may include GOSUB, CALL, or EXECUTE to execute lower level code outside of this clause, however, unless an exception is raised, it is essential that these operations return to complete the TRY clause. Use of GOTO to jump into or out of the TRY clause may have undesirable results.

On throwing an exception, the application exits from all lower level programs, subroutines, etc and continues execution in the CATCH clause that handles this exception. Where an object oriented programming object is discarded by an exception, the DESTROY.OBJECT subroutine will be executed in the normal way.

Transactions started by actions within the TRY clause and still active at the point when the exception is thrown will be rolled back.

A single CATCH clause may catch multiple exception names. On arrival in the CATCH clause, the @EXCEPTION variable will contain the name of the exception that has been caught and the @EXCEPTION.DATA variable will contain any data associated with this exception. The @EXCEPTION.ORIGIN variable is set to a dynamic array in which field 1 holds the program name from which the exception was thrown and field 2 holds the line number in that program. If the program has no cross-reference tables, the line number will be -1.
The `CATCH` clause has two optional qualifiers that can appear after an exception name. The DUMPING qualifier causes a process dump file to be created, showing the full state of the call stack at the point when the exception was thrown. The SAVING.STACK qualifier sets @EXCEPTION_STACK to contain the call stack in the same form as produced by the `SYSTEM(1002)` function. See `TRY/CATCH` for an example of a simple code fragment to report this data in a useful form.

Example

```qmbasic
LOOP   READNEXT ACC.ID ELSE EXIT
TRY
   CALL PROCESS.ACCOUNT
CATCH ACCOUNT.INVALID
      DISPLAY 'Account ' : ACC.ID : ' is not valid'
END
REPEAT
```

The above rather simple example shows a program fragment that processes successive items from a select list. If the `PROCESS.ACCOUNT` subroutine, or any lower level action called from it, throws an ACCOUNT.INVALID exception, the program will continue execution at the DISPLAY statement.

Exception Groups

An exception name may be formed from multiple components separated by a period where each component forms a level of grouping. In the above example, the ACCOUNT.INVALID exception can be considered as part of an exception group named ACCOUNT. The `CATCH` clause in the example will be entered on throwing the specific ACCOUNT.INVALID exception. Alternatively, the `CATCH` clause could have been written as

```
CATCH ACCOUNT
```

and this would be executed for any exception in the ACCOUNT group, including the ACCOUNT.INVALID exception. Use of exception groups can significantly simplify programs by using a single `CATCH` clause to trap a whole exception group, the actual exception name being available in the `@EXCEPTION` variable. There is no restriction on the number of levels in an exception group aside from the entire exception name being limited to 63 characters. The example above would also catch further exception levels such as ACCOUNT.INVALID.CLOSED to show the reason for the account to be invalid.

Scope of Exception Handlers

The structure of an application may validly nest `TRY/CATCH` constructs to any depth. On throwing an exception, the system works back through the established exception handlers to find the first that catches the relevant exception. This process will not continue past a program that uses the TRAPPING ABORTS option of the `EXECUTE` statement as this option requests that any errors in lower level programs should cause return from the `EXECUTE`. An application can test whether there is a handler for a specific named exception by use of the `CAUGHT()` function.

The `SYS$ANY` Exception

A `CATCH` clause that references the SYSSANY exception name will catch any exception and can be used as a general error handler. When used, this exception name must be the last one referenced
in the CATCH clauses associated with the TRY. The exception name returned via @EXCEPTION will be that of the actual exception thrown, not SYS$ANY.

This exception can be used in much the same way as the TRAPPING ABORTS option of the EXECUTE statement to act as a firewall beyond which no exceptions will be thrown.

The SYS$UNHANDLED Exception

A CATCH clause that references the SYS$UNHANDLED exception name will be executed if there is no specific handler for the thrown exception or SYS$ANY. The check for this handler occurs after checking the entire exception handler stack for the named exception or SYS$ANY, therefore SYS$ANY effectively takes priority over SYS$UNHANDLED regardless of their relative positions in the exception handler stack.

System Generated Exceptions

Many of the situations that would otherwise cause an abort event to be generated such as a data type error can be trapped by exceptions from the SYS exception group.

If there is no handler for the exception, including the SYS$ANY and SYS$UNHANDLED handlers described above, the application will abort in the usual way. Note that the abort error message is stored in both @ABORT.MESSAGE and @EXCEPTION.DATA but is not displayed if it is caught by an exception handler.

The full list of exception names in the SYS group is shown hierarchically below. In all cases, the name is formed by adding the desired hierarchical elements to a "SYS." prefix. For example, a deadlock situation throws an exception named "SYS.LOCKS.DEADLOCK".

<table>
<thead>
<tr>
<th>Exception Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABORT</td>
<td>User initiated abort</td>
</tr>
<tr>
<td>EXTCALL</td>
<td>External Call Interface</td>
</tr>
<tr>
<td>ABORT</td>
<td>Abort requested by external call handler</td>
</tr>
<tr>
<td>INVALID_ARGCT</td>
<td>Incorrect argument count</td>
</tr>
<tr>
<td>IO_ERROR</td>
<td>Data transfer error</td>
</tr>
<tr>
<td>SETUP_ERROR</td>
<td>Child process creation failed</td>
</tr>
<tr>
<td>SIZE_ERROR</td>
<td>Invalid data size</td>
</tr>
<tr>
<td>FILESYS</td>
<td>File system</td>
</tr>
<tr>
<td>CREATION_ERROR</td>
<td>Error creating file</td>
</tr>
<tr>
<td>INVALID_FILE_TYPE</td>
<td>Invalid file type for this operation</td>
</tr>
<tr>
<td>INVALID_PARTNO</td>
<td>Invalid distributed file part number</td>
</tr>
<tr>
<td>INVALID_PATHNAME</td>
<td>Pathname is invalid</td>
</tr>
<tr>
<td>INVALID_RECORD_ID</td>
<td>Record id is invalid</td>
</tr>
<tr>
<td>IO_ERROR</td>
<td>Data transfer error</td>
</tr>
<tr>
<td>LOCK_REQUIRED</td>
<td>Record is not locked</td>
</tr>
<tr>
<td>NOT_DISTRFL</td>
<td>Distributed file required</td>
</tr>
<tr>
<td>OPEN_ERROR</td>
<td>Error opening file</td>
</tr>
<tr>
<td>PERMISSIONS</td>
<td>Permissions error</td>
</tr>
<tr>
<td>Error Type</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------------</td>
<td>------------------------------------------------------------------</td>
</tr>
<tr>
<td>ERROR</td>
<td>Permissions do not allow this operation</td>
</tr>
<tr>
<td>READ ONLY</td>
<td>File is open in read-only mode</td>
</tr>
<tr>
<td>TRIGGER</td>
<td>Trigger function errors</td>
</tr>
<tr>
<td>ERROR</td>
<td>Error code returned by trigger function</td>
</tr>
<tr>
<td>INVALID_ARGCT</td>
<td>Incorrect argument count in trigger function</td>
</tr>
<tr>
<td>KERNEL</td>
<td>Kernel errors</td>
</tr>
<tr>
<td>NO_MEMORY</td>
<td>Insufficient memory available</td>
</tr>
<tr>
<td>LOCKS</td>
<td>Locking errors</td>
</tr>
<tr>
<td>DEADLOCK</td>
<td>Deadlock detected</td>
</tr>
<tr>
<td>MAXRLOCK</td>
<td>Maximum number of record locks in process reached</td>
</tr>
<tr>
<td>TASK_LOCK</td>
<td>Task lock errors</td>
</tr>
<tr>
<td>INVALID</td>
<td>Invalid task lock number</td>
</tr>
<tr>
<td>PRINTER</td>
<td>Printer and terminal errors</td>
</tr>
<tr>
<td>IO_ERROR</td>
<td>Data transfer error</td>
</tr>
<tr>
<td>INVALID_KEY</td>
<td>Invalid key value</td>
</tr>
<tr>
<td>INVALID_OPERATION</td>
<td>Operation not permitted</td>
</tr>
<tr>
<td>INVALID_REGION</td>
<td>Invalid screen region</td>
</tr>
<tr>
<td>INVALID_UNIT</td>
<td>Invalid print unit</td>
</tr>
<tr>
<td>PROGRAM</td>
<td>QMBasic errors</td>
</tr>
<tr>
<td>DATATYPE</td>
<td>Data type errors</td>
</tr>
<tr>
<td>ECS_CHAR</td>
<td>Data outside 8-bit character set not valid</td>
</tr>
<tr>
<td>INVALID</td>
<td>Data type is invalid for this usage</td>
</tr>
<tr>
<td>INVALID_SOCKET</td>
<td>Invalid socket type for this usage</td>
</tr>
<tr>
<td>NOT_COLLECTION</td>
<td>Data collection variable required</td>
</tr>
<tr>
<td>NOT_FILE</td>
<td>File variable required</td>
</tr>
<tr>
<td>NOT_IMAGE</td>
<td>Screen image variable required</td>
</tr>
<tr>
<td>NOT_MATRIX</td>
<td>Dimensioned matrix required</td>
</tr>
<tr>
<td>NOT_NUMERIC</td>
<td>Non-numeric where numeric required (Not thrown if NON.NUMERIC.WARNING option is enabled)</td>
</tr>
<tr>
<td>NOT_OBJECT</td>
<td>Instantiated object required</td>
</tr>
<tr>
<td>NOT_SEQ_FILE</td>
<td>Sequential file variable required</td>
</tr>
<tr>
<td>NOT_SOCKET</td>
<td>Socket variable required</td>
</tr>
<tr>
<td>NOT_SORT</td>
<td>Sort session variable required</td>
</tr>
<tr>
<td>NULL_STRING</td>
<td>Null string is invalid for this usage</td>
</tr>
<tr>
<td>UNASSIGNED</td>
<td>Variable is unassigned</td>
</tr>
<tr>
<td>DIV_ZERO</td>
<td>Divide by zero (Not thrown if DIV.ZERO.WARNING option is enabled)</td>
</tr>
<tr>
<td>INVALID_ARGCT</td>
<td>Incorrect argument count</td>
</tr>
<tr>
<td>INVALID_CALL_NAME</td>
<td>Invalid subroutine/function/class name</td>
</tr>
</tbody>
</table>
INVALID_DATA_SIZE: Invalid data size for this operation
INVALID_IN_TXN: Operation is not valid in a transaction
INVALID_ITYPE: I-type object code is invalid
INVALID_KEY: Invalid key value in function call
INVALID_MATRIX_INDEX: Matrix index is out of range
INVALID_MATRIX_SIZE: Invalid matrix dimensions
INVALID_NAME: Invalid data collection element name
INVALID_OBJECT_REFERENCE: Invalid object method/property reference
IS_CLASS: Program is a class module
LIMIT: System limit reached
CALL_DEPTH: Subroutine call depth limit
NUM_FILES: System file count limit
GOSUB_DEPTH: Internal subroutine depth limit
MAX_BREAKPOINTS: Debugger breakpoint limit
LOAD_ERROR: Error loading program
READ_ERROR: Error reading object code
VERSION: Unsupported object code version
NO_PDEBUG: Phantom debugger not active
NOPROPERTY_VALUE: Object did not return property value
NOT_FOUND: Program not found
NOT_OBJECT: Program is not an instantiated object
NOT_TXN: Operation is only valid in a transaction
RANGE: Value is out of range for this operation
RESIZE_ERROR: Error resizing common or matrix
SELECT: Select list processing errors
INVALID_LIST_NO: Invalid select list number
SHELL: Shell command execution errors
EXECUTION_ERROR: Command execution error
SORT: Sort subsystem errors
INVALID_DATA_LENGTH: Sort data exceeds maximum allowed length
INVALID_KEY_COUNT: Invalid number of sort keys
INVALID_KEY_LENGTH: Sort key exceeds maximum allowed length
INVALID_OPERATION: Sort operation invalid at this time
IO_ERROR: Data transfer error

See also: CAUGHT(), THROW, TRY/CATCH
Using Socket Connections

QMBasic includes a set of functions that allow network connections to be established with other systems or other software on the same system. Where supported by the underlying operating system, both IPV4 and IPV6 protocol connections are available though IPV6 support must be enabled using the IPV6 configuration parameter.

What is a Socket?

A socket is one end of a bidirectional link between two software components across a network or within the same system. A socket is established using a network address (typically written as four dot separated values such as 192.168.13.11 for IPV4 addresses or up to eight colon separated hexadecimal values for IPV6 addresses) and a port number. These can be considered as being much like a telephone number and extension. The network address identifies the device on the network to which a connection is to be established and the port number identifies a service within that device.

Just as we can look up a telephone number in a directory, so the internet has domain name servers that fulfill the same role, allowing users to reference a network destination by name instead of its number. For example, www.openqm.com translates to 81.31.112.103.

There is a central registry of standard port numbers (e.g. 23 for a telnet connection or 80 for a web server) but these are not rigidly enforced. By default, QM uses ports 4242 and 4243 for terminal and QMClient connections respectively. On Unix and Linux systems, only processes running as root can listen on port numbers less than 1024.

QM also supports Unix domain sockets on Unix and Linux systems. These use a file system pathname for the connection in place of a network address.

A socket may be established in two modes; stream and datagram. A stream connection represents a channel over which a succession of messages may be passed in each direction until connection is broken by one participant. A datagram connection consists of a single message and response pair after which the connection is broken.

There are also two protocols used to manage the internal structure of a message; TCP (transmission control protocol) and UDP (user datagram protocol). These are usually paired up with the corresponding socket modes such that TCP is used over stream connections and UDP over datagram connections but the underlying network system allows variations.

Socket Programming for Stream Connections

Creating a stream connection is very easy. Although one end of this connection is likely to be something other than a QMBasic program, the example below is based on two QMBasic programs communicating across a network.

The server process (the one that is waiting for an incoming connection) starts listening by using a sequence of the form:

```
SRVR.SKT = CREATE.SERVER.SOCKET('', 4000, SKT$STREAM + SKT$TCP)
IF STATUS() THEN STOP 'Cannot initialise server socket'
SKT = ACCEPT.SOCKET.CONNECTION(SRVR.SKT, 0)
IF STATUS() THEN STOP 'Error accepting connection'
```

The `CREATE.SERVER.SOCKE(T)` call listens for incoming TCP stream connections on port 4000. The first argument in this function has been specified as a null string to listen on all local
addresses. A specific address can be specified if required. SRVR.SKT becomes a socket variable that is effectively monitoring for incoming connections. This is then used in a call to ACCEPT.SOCKET.CONNECTION() which will wait for a connection to arrive. A timeout period can be specified. On return from this function, if no error has occurred, SKT will be the socket variable for the specific connection. The program could resume listening for further incoming connections using ACCEPT.SOCKET.CONNECTION().

Once a connection has been established, data can be read or written using READ.SOCKET() and WRITE.SOCKET().

\[
\text{DATA} = \text{READ.SOCKET(SKT, 100, SKT$BLOCKING, 0)} \\
N = \text{WRITE.SOCKET(SKT, DATA, 0, 0)} 
\]

This example simply echoes the data back to the other end of the connection. The SKT$BLOCKING flag specifies that READ.SOCKET() should wait for incoming data rather than returning immediately if there is no data waiting. Note that this waits for any data, not the full 100 bytes specified as the limit. It may be necessary to perform several reads to assemble the data sent from the other end of the connection.

Finally, the connection can be terminated using CLOSE.SOCKET(), remembering that the server socket also needs to be closed when no new connections are to be handled. Like all QBASIC variables, if a program terminates and a socket variable is discarded, the socket will be closed automatically.

\[
\text{CLOSE.SOCKET SKT} \\
\text{CLOSE.SOCKET SRVR.SKT} 
\]

The client program that wants to connect to this server would be of the form

\[
\text{SKT} = \text{OPEN.SOCKET("172.16.13.14", 4000, SKT$BLOCKING)} \\
\text{IF STATUS() THEN STOP 'Cannot open socket'}} \\
\text{N = WRITE.SOCKET(SKT, DATA, 0, 0)} \\
\text{REPLY = READ.SOCKET(SKT, 100, SKT$BLOCKING, 0)} \\
\text{CLOSE.SOCKET SKT} 
\]

Datagram Connections

A datagram connection is typically used to send a single message to a server which returns a single response and then closes the connection. The domain name servers mentioned above use this form of data exchange when looking up a name.

The server program becomes simply:

\[
\text{SKT} = \text{CREATE.SERVER.SOCKET("", 4000, SKT$DGRM + SKT$UDP)} \\
\text{IF STATUS() THEN STOP 'Cannot initialise socket'}} \\
\text{DATA = READ.SOCKET(SKT, 100, SKT$BLOCKING, 0)} \\
\text{N = WRITE.SOCKET(SKT, DATA, 0, 0)} \\
\text{CLOSE.SOCKET SKT} 
\]

The client program becomes:

\[
\text{SKT} = \text{OPEN.SOCKET("172.16.13.14", 4000, SKT$DGRM + SKT$UDP + SKT$BLOCKING)} \\
\text{IF STATUS() THEN STOP 'Cannot open socket'}} \\
\text{N = WRITE.SOCKET(SKT, DATA, 0, 0)} \\
\text{REPLY = READ.SOCKET(SKT, 100, SKT$BLOCKING, 0)} \\
\text{CLOSE.SOCKET SKT} 
\]
Socket Tracing

Socket tracing enables an application to create a diagnostic report of all data transmitted in either direction via a socket. Tracing is enabled by use of the SKT$INFO.TRACE mode of the SET.SOCKET.MODE() function, passing in the pathname of the log file to be created. Any existing data in this log file will be overwritten. Tracing remains active until the socket is closed or a further call to SET.SOCKET.MODE() sets the log file pathname to a null string.

The log consists of a simple text file in with entries tagged as SEND or RECV to indicate the direction of data transfer. Each entry shows the data in both hexadecimal and character form.

Working with Multiple Simultaneous Socket Connections

If an application needs to work with multiple simultaneous socket connections, handling input on each socket as it occurs, the SELECT.SOCKET() function provides a way to wait for an event on any of a collection of sockets.

Socket Inheritance

As an alternative to working with multiple simultaneous socket connections, it is sometimes useful for a network listener process to use ACCEPT.SOCKET.CONNECTION() to wait for and accept an incoming connection and then to start a separate phantom process to handle that connection. This can be achieved using socket inheritance.

After accepting the incoming connection, the server should set the socket into inheritable mode using the SKT$INHERITABLE mode of the SET.SOCKET.MODE() function. Then, when the process starts a phantom, the inheritable socket is passed to the new process via the @SOCKET variable, closing it in the server process. Only a single socket can be put into inheritable mode at one time. Attempting to set inheritability on a second socket will cancel the inheritability of the previous socket. Also, note that server mode sockets opened with CREATE.SERVER.SOCKET() cannot be made inheritable.

A phantom process that inherits a socket from its parent process consumes a licence as it is considered to be an interactive process.

See also:
ACCEPT.SOCKET.CONNECTION, CLOSE.SOCKET, CREATE.SERVER.SOCKET(), OPEN.SOCKET(), READ.SOCKET(), SELECT.SOCKET(), SERVER.ADDR(), SET.SOCKET.MODE(), SOCKET.INFO(), WRITE.SOCKET()
External Functions

Sometimes applications need to be able to call external functions such as those in standard operating system libraries. QM provides an External Call Interface (ECI) by which this can be done from within the security of the managed application environment of QMBasic.

Function Prototypes

Before an external function can be used by an application, it must be defined using the QMBasic \texttt{DEFFUN} statement with the \texttt{EXTERNAL} keyword

\begin{verbatim}
DEFFUN name(arg1, arg2, ...) EXTERNAL \{CALLING "libname"\}
\end{verbatim}

where

- \texttt{name} is the name of the function.
- \texttt{arg1 etc} are placeholders for the function arguments.
- \texttt{libname} is a quoted string defining the name of the executable program providing these library functions. If omitted, it defaults to qmextcall. The .exe suffix on Windows systems is added automatically. The name is case sensitive on Linux systems.

To preserve the managed application environment provided by QM, external function calls are handled by a child process that is created when the first external function is called from a QM session. Use of a child process ensures that errors in this program cannot corrupt memory within the parent QM process. This isolation is essential as corruptions to QM's shared memory area could cause problems in other processes that would be hard to diagnose and might damage files.

Although the overheads of using a separate process are low, the external function call mechanism is intended for execution of user written code that performs a substantial task or is not used intensively by the application. Developers who have a need for intensive use of short library functions should contact QM support as it may be possible to integrate these into the standard product.

To use the external call interface, the application developer must download and modify the skeleton external call server program that is shown below. This sample program shows how to execute the C library stat() function, returning the size of a named file. This operation is available directly from QMBasic and the program is not intended as a realistic use of the external call interface but serves to show how to implement external function calls.

Note that the function names passed by the parent QM process are always in uppercase unless the \texttt{CASE.SENSITIVE} setting of the $MODE$ compiler directive is enabled. A real external call server program may need to support multiple functions and would do so using conditional coding similar to that in the skeleton program.

Up to 31 arguments can be passed into an external function. To optimise the interface between the child and parent processes, an argument placeholder name can be prefixed with \texttt{IN:} to indicate that the argument is passed into the function but no updated value is to be returned, or \texttt{OUT:} to indicate that the argument is used only to return data and the current content of the argument variable should not be passed to the child process. For example,

\begin{verbatim}
DEFFUN MYFUN(IN:A, IN:B, OUT:C) EXTERNAL
\end{verbatim}

Any arguments for which neither qualifier is present are bidirectional though automatic optimisation is applied so as not to pass back arguments that have not changed.

The server process is started when the parent process first calls an external function and will normally stay running until the user exits from QM or uses \texttt{LOGTO} to move to a different account.
The program can maintain internal state data, file variables, etc that persist from one call to the next. The ECI child process can be terminated by use of the \texttt{CLOSE.ECI} command.

Arguments passed into the function are numbered from one. The user written code that will replace the test code in the skeleton program can access the arguments passed from the QMBasic function call by use of \texttt{GetInteger()}, \texttt{GetFloat()} or \texttt{GetString()}, each of which takes the argument number to be retrieved as its only parameter. The number of arguments passed from the QMBasic program can be determined using the \texttt{GetNumArgs()} function and the data type for an argument can be retrieved using the \texttt{GetArgType()} function which returns a single character, I for integer, F for floating point and S for string. Data type conversion will be performed where the actual argument type does not match the form in which is requested, for example, using \texttt{GetInteger()} to retrieve and argument that was passed into the QMBasic function call as a string. Attempting to access an argument position that was not passed in the function call or an argument that was unassigned will return either numeric zero or a null string.

The \texttt{GetString()} function returns a pointer to a null terminated 8 bit character string form of the argument. This string should not be overwritten. Although the string is null terminated, the interface between the parent QM process and the external call server process permits char(0) within strings. In this case, use of the C `strlen()` function will not return the correct string length. The \texttt{StringLength()} function can be used to determine the length of the actual argument string.

Using \texttt{GetString()} with an ECS mode version of QM will return only the low order 8 bits of each character and hence is likely to lead to incorrect results if the QMBasic data passed as the argument contains characters outside the 8-bit set. The \texttt{GetStringW()} function returns data as a null terminated wchar_t data type string and can be used regardless of whether the QM system is in ECS mode. Note that Windows systems define wchar_t as 16 bits whereas the gcc compiler used on most Linux systems defines wchar_t as 32 bits.

Arguments may be updated for return values using the \texttt{ReturnInteger()}, \texttt{ReturnFloat()} or \texttt{ReturnString()} functions. Each of these takes the argument number and new value as its parameters. The \texttt{ReturnString()} function has a third argument through which the caller should pass the string length. Passing a negative value for the string length indicates that the function should treat the data as a null terminated string and determine its length internally.

The \texttt{ReturnStringW()} function returns a wchar_t form string for which the length should be specified in characters, not bytes. Use of \texttt{ReturnStringW()} on a non-ECS mode system to return data that includes characters outside the 8-bit set will return only the low order 8 bits of each character.

The return functions allow reference to argument number zero. This corresponds to the value that will be returned as the result of the QMBasic function call that triggered the child process action.

The new value set by any of the return functions will be ignored for arguments declared as input only in the QMBasic function definition.

The \texttt{SetOSError()} function can be used to set a value for \texttt{OS.ERROR()} in the QMBasic program on return from the external function. \texttt{SetOSError()} takes a single int argument. If this function is not used, the value of \texttt{OS.ERROR()} is unchanged.

The \texttt{CompleteCall()} function must be called as the final stage in processing a request and takes a single argument which is the value to be returned by the QMBasic \texttt{STATUS()} function after the external function call has completed. This allows errors to be reported to the calling process. It is recommended that error code values should correspond to the standard QM error numbers as defined in the SYSCOM ERR.H include record. Passing a negative status value will cause the QMBasic program to abort, reporting the absolute status value in the error message. Any return argument values will not be updated in this situation.
Compiling the Program

Windows users should define the WINDOWS token either from the compiler or by including a line

```c
#define WINDOWS
```

before the `#include` directives.

The appropriate version of the qmextcall library from the bin subdirectory of the QMSYS account must be linked with this program. The versions supplied are:

<table>
<thead>
<tr>
<th>QM Version</th>
<th>Name</th>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows 32 bit</td>
<td>qmextcallms.lib</td>
<td>Microsoft C</td>
</tr>
<tr>
<td>Windows 64 bit</td>
<td>qmextcall64.lib</td>
<td>Microsoft C</td>
</tr>
<tr>
<td>Linux / Mac / Unix</td>
<td>libqmextcall.a</td>
<td>Static library</td>
</tr>
<tr>
<td></td>
<td>qmextcall.so</td>
<td>Shared object library (Linux)</td>
</tr>
<tr>
<td></td>
<td>qmextcall.dylib</td>
<td>Dynamic library (Mac)</td>
</tr>
</tbody>
</table>

The default name for the compiled program is qmextcall (qmextcall.exe on Windows). This program must be placed in the bin subdirectory of the QMSYS account. Alternative names can be used with the CALLING clause to the `DEFFUN` statement.

Skeleton External Call Server

```c
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <sys/stat.h>

#ifdef WINDOWS
    #include <windows.h>
#else
    #include <wchar.h>
#endif

#include <qmextcall.h>

int main(int argc, char * argv[]){
    char function_name[MAX_FUNCTION_NAME_LEN+1];
    struct stat statbuf;
    int err;

    /* Process command line and initialise */
    Initialise(argc, argv);

    /* Process requests */
    while(GetCallRequest(function_name))
    {
        err = 0;  /* Default returned status value */
    }
}
```
//----- Insert code to handle your external functions here

if (!strcmp(function_name, "STAT"))
{
    if (!stat(GetString(1), &statbuf))
    {
        ReturnInteger(0, statbuf.st_size);
    }
    else
    {
        ReturnInteger(0, -1);
        err = errno;
    }
}

//----- End of code to handle your external functions

else /* Function name not recognised */
{
    err = -ER_FUNCNAME;
}

CallCompleted(err); /* Send updated arguments and result */

return 0;
}
6.2 QMBasic - Compiler Directives

Compiler directives control the way in which the compiler processes the QMBasic source programs. They do not result directly in executable statements.

Directive names may be written with a # character in place of the $ character.

The available directives are

- $*
- $CATALOG
- $CATALOGUE
- $COPYRIGHT
- $DEBUG
- $DEFINE
- $EJECT
- $ERROR
- $EXECUTE
- $IFDEF and $IFNDEF (also $ELSE and $ENDIF)
- $INCLUDE
- $INSERT
- $LIST
- $MODE
- $NO.CATALOGUE
- $NO.XREF
- $NOCASE.STRINGS
- $PAGE
- $QMCALL
- $STOP
- $UNDEFINE
- $WARNING
$^*$ compiler directive

The $^*$ directive inserts user supplied text into the object code.

Format

$^* \textit{text}$

where

\textit{text} is the text to be inserted.

The $^*$ directive allows a developer to embed text into the compiled object code. This text can be printed using the \texttt{FIND.PROGRAM} command.

Multiple $^*$ directives may appear in a program, each adding a line to the stored text.

Example

$^* \text{Version 3.7}$

See also:
\texttt{$SCOPYRIGHT$, FIND.PROGRAM}
$CATALOGUE compiler directive

The $CATALOGUE directive (or the American spelling $CATALOG) causes automatic cataloguing of a program after successful compilation.

Format

$CATALOGUE name {GLOBAL | PRIVATE | LOCAL} {NO.QUERY}

where

name is the name to be used when the program is added to the catalogue.

The $CATALOGUE directive causes the compiler to add the program to the system catalogue with the given call name if the compilation is successful. If name is omitted, the source record name is used. When using this default name, an error will be reported if the name is not the same as the name specified in the PROGRAM, SUBROUTINE, FUNCTION or CLASS statement.

If the name does not follow the normal QMBasic name construction rules (e.g. a Pick user exit such as 50BB) it should be enclosed in quotes. The rules for catalogue name format are described with the CATALOGUE command.

The GLOBAL, PRIVATE and LOCAL qualifiers set the cataloguing mode. If none of these is present, the system's default mode is used.

The NO.QUERY keyword suppresses the prompt that normally appears if the program is already catalogued in a different mode.

Automatic cataloguing can also be performed using the CATALOGUE entry in the $BASIC.OPTIONS record as described under the BASIC command. Use of the $CATALOGUE compiler directive will override any alternative settings specified in the $BASIC.OPTIONS record.

Examples

$CATALOGUE
This directive causes the compiler to copy the program to the private catalogue using its default name.

$CATALOGUE PRINT.INVOICE
This directive causes the compiler to copy the program to the private catalogue as PRINT.INVOICE.

$CATALOGUE PRINT.INVOICE LOCAL
This directive causes the compiler to copy the program to the local catalogue as PRINT.INVOICE.

$CATALOGUE "50BB" GLOBAL
This directive causes the compiler to copy the program to the global catalogue as 50BB. Note the need for quotes as the name does not comply with QMBasic name construction rules which require the first character to be a letter.
See also:
$NO.CATALOGUE, $BASIC.OPTIONS
$COPYRIGHT compiler directive

The $COPYRIGHT directive inserts legal copyright data into the object code.

Format

```
$COPYRIGHT "text"
```

where

`text` is the text of the copyright message. This must be enclosed in quotes.

The $COPYRIGHT directive allows a developer to embed a legal copyright notice in the compiled object code. This notice can be printed using the FIND.PROGRAM command.

If more than one $COPYRIGHT directive appears in a program, the last one determines the text to be inserted.

$COPYRIGHT and $* may appear in the same program. In this case, the copyright effectively forms the last line of the comment data.

The copyright text can also be supplied via the $BASIC.OPTIONS record.

Example

```
$COPYRIGHT "Copyright Acme Software, 2011"
```

See also:

$*, FIND.PROGRAM
$DEBUG compiler directive

The $DEBUG directive compiles the program in debug mode.

Format

$DEBUG

The $DEBUG directive is an alternative to use of the DEBUGGING keyword to the BASIC command. The $DEBUG directive must appear before any executable statements in the program.

During development, it is often useful to compile the entire application in debug mode. There is a small performance overhead running debug mode programs so it is recommended that production versions of the application should be compiled without this mode.

See also:
QMBasic debugger
$DEFINE compiler directive

The $DEFINE directive is used to associate a value with a symbolic name at compile time. The standard include records in the SYSCOM file contain many examples of $DEFINE.

Format

$DEFINE name value

where name is the symbol to be used in the program and value is a constant.

Although it is valid for name to correspond to a built-in function, statement or reserved word, use in this way is discouraged as the function, statement or reserved word becomes inaccessible. As an aid to detecting this situation, particularly when migrating an existing application to QM, use of the STRICT.EQUATE setting of the $MODE compiler directive will generate a compiler warning for name clashes.

The token QM is automatically defined and may be used with $IFDEF to determine whether a program is being compiled on the QM database. The token QM.WINDOWS, QM.LINUX, QM.FREEBSD, QM.MAC, QM.AIX, QM.SOLARIS, QM.RPI or QM.PDA is defined corresponding to the underlying operating system. Because these are only relevant at compile time, take care when using them in programs that may be moved between platforms. The SYSTEM(1010) function can be used to determine platform type at run time.

See $IFDEF for an example of use of define tokens to control conditional compilation, allowing a single source stream to be used in different multivalue environments.

See also:
EQUATE
$ERROR compiler directive

The $ERROR directive generates a compilation error.

Format

$ERROR message

where

message is the error message to be displayed by the compiler.

The $ERROR directive causes the QMBasic compiler to generate a compilation error. It is most likely to be used inside a $IFDEF conditional compilation structure to recognise that the program is in some way invalid.

Example

$IFDEF PLATFORM
  $ERROR Platform not defined
$ENDIF

The above example generates a compilation error if the PLATFORM token has not been defined.

See also:
$DEFINE, $IFDEF, $STOP, $WARNING
$EXECUTE compiler directive

The $EXECUTE directive executes a command during program compilation.

Format

$EXECUTE "command"

where command is the command to be executed. This must be enclosed in quotes.

This directive can be useful, for example, when programmatically generated include records are used.
$IFDEF and $IFNDEF compiler directives

The $IFDEF directive provides conditional compilation.

Format

```
$IFDEF name  
...statements...
$ELSE       
...statements...
$ENDIF
```

The $ELSE and associated statements are optional. Statements conditioned by the $IFDEF directive are ignored if `name` has not been defined by a previous $DEFINE directive. Statements under the $ELSE directive are ignored if `name` has been defined by a previous $DEFINE directive.

The $IFNDEF directive provides the inverse action of $IFDEF, compiling the following statements if `name` has not been defined.

Multiple token names may specified in a single directive with an intervening AND or OR operator (or their symbolic equivalents, & and !). The expression is evaluated left to right and brackets are not permitted.

$IFDEF and $IFNDEF statements may be nested to any depth though nesting more than two deep can make program maintenance very difficult.

Portable Software

One use of conditional compilation is to allow a single source stream to be compiled in different multivalue environments rather than having to maintain separate source streams. When compiling on QM, the name QM is always defined thus a construct such as

```
$IFDEF QM
  SETLEFT 'DATE' FROM INV.F
  LOOP
    SELECTRIGHT 'DATE' FROM INV.F SETTING DT
    UNTIL STATUS()
    READLIST IDS THEN GOSUB PROCESS
  REPEAT
$ENDIF
```

might be used scan an alternate key index on QM. The same program might contain an equivalent for UniVerse

```
$IFDEF UV
  BSCAN DT, IDS FROM INV.F USING 'DATE' RESET THEN
  LOOP
    GOSUB PROCESS
    BSCAN DT, IDS FROM INV.F USING 'DATE' ELSE EXIT
  REPEAT
  END
$ENDIF
```

though UniVerse does not define the UV token automatically. This would need to be done in the source code, probably as part of a platform specific include record.
Some environments such as D3 do not support conditional compilation. There is a skeleton pre-processor program in the BP file of the QMSYS account that can be compiled on D3, etc to resolve conditional compilation constructs.

See also:
$DEFINE, $ERROR, $STOP, $WARNING
$INCLUDE compiler directive

The $INCLUDE directive is used to direct the compiler to include text from another record. Include records may be in either directory or hashed files.

Format

$INCLUDE { filename } record.id

For compatibility with some other systems, $INSERT may be used as a synonym for $INCLUDE.

Examples

$INCLUDE SYSCOM ERR.H
$INCLUDE MYKEYS.H

The first example includes record ERR.H from the SYSCOM file. The second example, which has no file name specified, includes MYKEYS.H from the same file as the source program.

Use of a .H suffix is recommended on include records as these will be skipped automatically by the compiler when using a select list.

The file name in the first $INCLUDE in this example is not strictly necessary as the default behaviour of the compiler is always to look in SYSCOM if no file name is given and the include record is not found in the same file as the source program. The sequence of locations searched for the include record when no filename is specified can be modified by use of the SEARCH directive in the $BASIC.OPTIONS record.

Because directory files on non-Windows platforms have case sensitive record names, the compiler will look for the record name as entered and then, if it has not been found, in uppercase. Thus programs may be written in either upper or lowercase if the records are always stored with uppercase name. The standard include records in the SYSCOM file follow this rule.

Include files may be nested (one included from within another) though this can lead to difficulties when maintaining complex programs and is discouraged.

For compatibility with other multivalue databases, the $INCLUDE directive can also be written without the $ prefix. Used in this way, it must be the only statement on the source line and the filename and record.id must not correspond to tokens that have been defined using $DEFINE or EQUATE.
$LIST compiler directive

The $LIST directive can be used to start, suspend and resume generation of a listing of the program and any associated error messages.

Format

$LIST [ ON]
$LIST OFF

The listing is directed to a record of the same name as the program source but with a suffix of .LIS. Any existing listing record is deleted by the compiler at the start of compilation regardless of whether a new listing is to be produced.

A $LIST ON directive in the main program starts generation of a listing record from that point onwards. The compiler LISTING option is equivalent to a $LIST ON at the start of the program.

A $LIST OFF directive stops generation of the listing record. If this is on an include record, listing will resume on return to the source or include record from which it was entered.

A $LIST ON directive in an include record only resumes generation of the listing record if listing was active when processing of the include record began.
$MODE compiler directive

The $MODE directive enables language options for improved compatibility with other multivalue databases.

**Format**

```
$MODE option [, option...]
```

where

*option* is the feature to be turned on. Multiple comma separated option names may be given.

The available options are:

- **DEFAULT** Turn off all options.
- **AUTOLOCK** Automatically locks a record if not already locked on WRITE or DELETE. This is provided for compatibility with some other environments and is not recommended.
- **CASE.SENSITIVE** Enables case sensitivity for names of labels, variables and user defined functions.
- **CHANGE.NO.OVERLAP** Do not allow overlapping substrings in CHANGE(), ERPLACE() and SWAP().
- **COMPATIBLE.APPEND** Modifies the behaviour of the append modes of the $<f,v,sv> assignment operator, the INS statement and the INSERT() and REPLACE() functions to match that of other multivalue products.
- **COMPOSITE.READNEXT** Modifies how a READNEXT statement handles exploded select lists.
- **CONDITIONAL.STATEMENTS** Allow most statements that have a THEN/ELSE clause to be used as conditional elements in WHILE or UNTIL.
- **COUNT.OVERLAP** Allow overlapping substrings in COUNT() and COUNTS().
- **DEBUG.EQUATES** Causes the definition of equate tokens referenced in a program to be saved in the object code when compiling in debug mode such that they can be used in the debugger.
- **DEFAULT.UNASS.ARGS** Enables substitution of a default value for unassigned arguments in a SUBROUTINE or FUNCTION statement.
- **EXCEPTION.SAVE.STACK** Causes the CATCH clause of a TRY/CATCH construct to behave as though the SAVING.STACK option is present.
- **EXECUTE.CLEARLIST** Clears the default select list prior to execution of the command in an EXECUTE statement.
EXPLICIT
All variables, including scalars, must be explicitly declared in a DIMENSION statement.

FOR.STORE.BEFORE.TEST
Stores the new value of the control variable in a FOR/NEXT construct before testing the end condition.

HEADING.NO.EJECT
Makes the NO.EJECT mode of the QMBasic HEADING statement the default, suppressing the automatic page throw on setting a new heading.

IMPLIED.STOP
The QMBasic compiler normally inserts an implied RETURN at the end of a program, subroutine or function. Setting this option inserts an implied STOP for compatibility with other multivalue products. Class modules always insert a RETURN regardless of the setting of this option.

INDEX.OVERLAP
Allow overlapping substrings in INDEX() and INDEXS().

NOCASE.STRINGS
Compiles the program using case insensitive string operations. This directive must appear before any executable statements in the program source and applies to the entire program module. Selecting case insensitive string mode affects the relational operators (=, #, <, >, <=, >=), their multivalue function equivalents (EQS(), NES(), LTS(), GES(), LES(), GES()), the CHANGE(), CONVERT(), COUNT(), DCOUNT(), FIELD(), LAST(), INDEX() and SWAP() functions, the CONVERT, FIND, FINDSTR, GROUPSTORE and LOCATE statements and pattern matching.

NO.ECHO.DATA
Suppresses echo of data for INPUT if it is taken from the DATA queue. See also the NO.ECHO.DATA mode of the OPTION command.

NON.DURABLE.TXN
Make transaction commit non-durable such that nested transactions are inherited by the parent on commit.

NUMERIC.BOOLEAN
QM support for an internal Boolean data type was added at release 3.2-2 and the @FALSE and @TRUE tokens were modified to use this data type at release 3.3-0. Programs that use @FALSE or @TRUE compiled using release 3.3-0 or later will fail with a run-time error if executed on a release prior to 3.2-2. Use of the NUMERIC.BOOLEAN mode causes @FALSE and @TRUE to evaluate to 0 and 1 respectively for backward compatibility but may cause problems in programs that use data collections on later releases.
OPEN.FAIL.ZERO.FVAR If enabled, sets the file variable in an OPEN, OPENPATH or OPENSEQ to zero instead of unassigned if the open fails.

OPTIONAL.FINAL.END Suppresses the warning message if there is no END statement at the end of the program.

OPTIONAL.THEN.ELSE Makes the THEN/ELSE clauses of file read statements optional. The statements affected are MATREAD, MATREADL, MATREADU, READ, READBLK, READL, READU, READY, READVL and READVU.

PARTIAL.EXPRESSIONS Causes evaluation of a logical expression to terminate as soon as the result can be determined. See Partial Expression Evaluation for more details.

PICK.CONVERT Pick style argument sequence for CONVERT().

PICK.ENTER Pick style processing of ENTER.

PICK.ERRMSG Pick style syntax for STOP and ABORT.

PICK.JUMP.RANGE Causes the ON GOSUB and ON GOTO statements to continue at the next statement if the index value is out of range.

PICK.MATRIX Create Pick style matrices. See the COMMON and DIMENSION statements for a discussion of the implications of this mode.

PICK.READ Causes READ, READL, READU, READY, READYU and READVL statements to take on the Pick style behaviour in which the target variable is left unchanged if the record is not found.

PICK.READNEXT Causes the READNEXT statement to leave the record id variable unchanged when taking the ELSE clause.

PICK.SELECT Changes the behaviour of SELECTV (and SELECT when $MODE SELECTV is in effect) such that, if the default select list is active, that list is transferred into the target variable instead of building a new list.

PICK.SUBSTR Causes substring assignment operations to take on the Pick style behaviour in which the variable is extended by adding trailing spaces if the region to be overwritten is beyond the end of the current string value.

PICK.SUBSTR.ASSIGN Causes substring assignment operations to take on the full Pick style behaviour as described under Assignment Statements. Setting this mode overrides the PICK.SUBSTR mode.

PRCLOSE.DEFAULT.0 PRINTER CLOSE defaults to printer 0 rather than closing all printers.
SELECTV Changes the action of SELECT to be as for SELECTV.
SSELECTV Changes the action of SSELECT to be as for SSELECTV.
STDFIL Enables use of the default file variable.
STDFIL.SHARED Enables use of the shared default file variable.
STRICT.EQUATE Applies strict rules to EQUATE token names and SDEFINE tokens, reporting a warning if the token name clashes with a reserved name, statement name or intrinsic function name.
STRING.LOCATE Numeric data in right aligned LOCATE is not treated as a special case.
SYMTAB0 Causes the compiler to emit the symbol table in the format used prior to release 4.0-0.
TIME.MS Causes the TIME() function and SYSTEM(12) to return the local time with a fractional (mS) part. This mode applies to the whole program regardless of where it appears.
TRAP.UNUSED Displays a warning about variables that are assigned a value but never used.
TRAP.UNUSED.MAIN Like TRAP.UNUSED but warns only if the variable is assigned a value in the main body of the program rather than in an include record.
UNASSIGNED.COMMON Variables in common blocks are created unassigned instead of being initialised to zero.
UV.LOCATE UniVerse Ideal / Reality flavour style LOCATE.
WRITE.DELETE.THEN.ELSE WRITE, MATWRITE and DELETE statements have an optional THEN/ELSE clause.

Prefixing a mode name (other than DEFAULT) with a minus sign turns off the named option.

The $MODE directive allows use of shortcut names to enable multiple modes in a single step. Other modes that were already set when these are used will not be cleared. The shortcuts that give closest compatibility when migrating an application to QM are described under "Migrating to QM". The shortcut names below are retained from QM version 3.4-17 and earlier.

D3 Sets CASE.SENSITIVE, CHANGE.NO.OVERLAP, COMPATIBLE.APPEND, COUNT.OVERLAP, FOR.STORE.BEFORE.TEST, HEADING.NO.EJECT, IMPLIED.STOP, INDEX.OVERLAP, OPTIONAL.FINAL.END, PICK.ENTER, PICK.ERRMSG, PICK.JUMP.RANGE, PICK.MATRIX, PICK.READ, PICK.SELECT, PICK.SUBSTR, PICK.SUBSTR.ASSIGN, SELECTV, SSELECTV, STDFIL, UNASSIGNED.COMMON, UV.LOCATE

MVBASE Sets CASE.SENSITIVE, CHANGE.NO.OVERLAP, COMPATIBLE.APPEND, COUNT.OVERLAP, FOR.STORE.BEFORE.TEST, HEADING.NO.EJECT, IMPLIED.STOP, INDEX.OVERLAP, OPTIONAL.FINAL.END, PICK.ENTER,
The modes set by these shortcuts may change between releases.

Default modes can be set by creating a record named $BASIC.OPTIONS in the source file or in the VOC. Details of how to do this can be found with the description of the `BASIC` command.

Some of these modes can also be used in dictionary I-type expressions.

**Examples**

```qmbasic
$MODE TRAP.UNUSED
FUNCTION INV.CLI(INV.REC)
  CLEINT.NO = INV.REC<4>
  IF CLIENT.NO = '' THEN CLIENT.NO = INV.REC<18>
  RETURN CLIENT.NO
END

The above program contains a typographical error in the spelling of the first use of CLIENT.NO. Because it has been compiled with the TRAP.UNUSED option, the compiler will display a warning message that the CLEINT.NO variable is assigned but never used.

PROGRAM INVOICE
FOR I = 1 TO 3
  NEXT I
  DISPLAY "I on exit = " : I
END

The above program displays the loop exit value as being 3 whereas the program below uses the FOR.STORE.BEFORE.TEST mode and displays the loop exit value as being 4.

$MODE FOR.STORE.BEFORE.TEST
PROGRAM INVOICE
FOR I = 1 TO 3
  NEXT I
  DISPLAY "I on exit = " : I
END
```
$NO.CATALOGUE compiler directive

The $NO.CATALOGUE directive (or the American spelling $NO.CATALOG) overrides any use of the CATALOGUE option in the $BASIC.OPTIONS record.

Format

$NO.CATALOGUE

The $NO.CATALOGUE directive specifies that a QMBasic program is not to be catalogued where the $BASIC.OPTIONS record includes a CATALOGUE entry that would otherwise cause the module to be catalogued automatically.

See also:

$CATALOGUE, $BASIC.OPTIONS
$NO.XREF

The $NO.XREF directive specifies that a QMBasic program is to be compiled without the cross-reference tables.

Format

$NO.XREF

The $NO.XREF directive is equivalent to use of the NO.XREF option to the BASIC command.

The cross-reference tables are used by QM to identify source code line numbers and the names of variables when reporting errors. Compiling a program without the cross-reference tables means that error messages are not able to show line numbers or the name of the variable involved.

The QMBasic debugger also uses the cross-reference tables. The $NO.XREF option is ignored if a program is compiled in debug mode.

See also:
BASIC, $BASIC.OPTIONS
$NOCASE.STRINGS compiler directive

The $NOCASE.STRINGS directive compiles the program using case insensitive string operations.

Format

$NOCASE.STRINGS {OFF}

This directive must appear before any executable statements in the program source and applies to the entire program module.

The $NOCASE.STRINGS directive is deprecated and replaced by use of $MODE NOCASE.STRINGS.

Selecting case insensitive string mode affects the relational operators (=, #, <, >, <=, >=), their multivalue function equivalents (EQS(), NES(), LTS(), GES(), LES(), GES()), the CHANGE(), CONVERT(), COUNT(), DCOUNT(), FIELD(), LAST(), INDEX() and SWAP() functions, the CONVERT, FIND, FINDSTR, GROUPSTORE and LOCATE statements and pattern matching.

The OFF qualifier turns off case insensitive string comparisons. It is intended for use where the $BASIC.OPTIONS record has been used to set case insensitivity but should be overridden for specific programs.
$PAGE compiler directive

The $PAGE directive inserts a page break in the compiler listing record. The synonym $EJECT may be used.

Format

$PAGE

The $PAGE directive can be used to improve readability of the record generated by use of the LISTING option to the BASIC command or the $LIST compiler directive.
$QMCALL compiler directive

The $QMCALL compiler directive controls access to a catalogued subroutine or class module via QMClient.

Format

$QMCALL

QMClient includes the ability to restrict which catalogued subroutines and class modules may be called directly by the client interface.

For processes running with the QMCLIENT configuration parameter set to 2, this directive makes the subroutine or class module in which it appears available for use with the QMClient API QMCall or QMCreateObject() functions. This subroutine or class may then call other subroutines or classes that may or may not include this compiler directive.

By using the QMCLIENT configuration parameter and this directive, it is possible to restrict the actions of a QMClient session thus allowing tighter control of QMClient security.
$STOP compiler directive

The $STOP directive terminates compilation.

Format

\[
\text{$STOP \{message\}}
\]

where

\[
message
\]

is an optional message to be displayed by the compiler.

The $STOP directive causes the QMBasic compiler to terminate compilation of the current program. It is most likely to be used inside a $IFDEF conditional compilation structure to recognise that the program is inappropriate to the system being compiled.

Example

\[
\text{$IFDEF QM}
\text{$STOP Program not applicable to QM}
\text{$ENDIF}
\]

The above example appears in the program in the BP file of the QMSYS account. This program is issued in source form for use in environments other than QM.

See also:
$DEFINE, $ERROR, $IFDEF, $WARNING
$UNDEFINE

The $UNDEFINE directive is used to cancel a compile time symbolic name previously established with $DEFINE or EQUATE.

Format

    $UNDEFINE name

where name is the symbol to be undefined.

See also:

$DEFINE, EQUATE
$WARNING compiler directive

The $WARNING directive generates a compilation warning.

Format

$WARNING message

where

message is the warning message to be displayed by the compiler.

The $WARNING directive causes the QMBasic compiler to generate a compilation warning message. It is most likely to be used inside a $IFDEF conditional compilation structure to recognise that the program is in some way invalid.

Example

$IFDEF PLATFORM
  $WARNING Platform not defined
ENDIF

The above example generates a compilation warning if the PLATFORM token has not been defined.

See also: $DEFINE, $ERROR, $IFDEF, $STOP
6.3 QMBasic Limits

Number of local variables or elements in a matrix
Dependent on available virtual memory.

Maximum file size
Dynamic files may have up to 2147483647 groups (16384Gb with 8kb group size) unless a
lower limit is imposed by the underlying operating system.

Maximum string size
There is effectively no limit imposed by QMBasic. The actual limit will be determined by the
total dynamic memory size which must be mapped to the swap file.

Number of open files
The underlying operating system may impose a limit on the number of files which may be open
simultaneously. QMBasic provides an automated file sharing scheme whereby files may be
closed at the operating system level if the limit is reached. QMBasic will save details of the file
which has been closed and will reopen it automatically when required.

This scheme provides access to a virtually unlimited number of files but can have severe
performance effects when many files are used in frequent rotation.

Maximum precision
14 decimal places

Maximum variable name or statement label length
No limit

Maximum characters in a string constant
No limit.

Maximum arguments to a subroutine
255

Maximum labels in an ON GOSUB or ON GOTO
65535

Maximum characters in a catalogue call name
128 (32 for trigger functions) or the value of the MAXIDLEN configuration parameter,
whichever is the lower.

Maximum characters in a record id
Default 63 but can be increased up to 255 using the MAXIDLEN configuration parameter.
6.4 QMBasic Statements and Functions by Type

Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
</tr>
<tr>
<td>~</td>
<td>Negation</td>
</tr>
<tr>
<td>**</td>
<td>Exponentiation</td>
</tr>
<tr>
<td>^</td>
<td>Exponentiation</td>
</tr>
<tr>
<td>//</td>
<td>Integer Division</td>
</tr>
<tr>
<td>=</td>
<td>Equality</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>#</td>
<td>Not equal</td>
</tr>
<tr>
<td>==</td>
<td>Equal to</td>
</tr>
<tr>
<td>:</td>
<td>Division</td>
</tr>
<tr>
<td>[ ]</td>
<td>Array access</td>
</tr>
<tr>
<td>&lt; &gt;</td>
<td>Matches</td>
</tr>
<tr>
<td>MATCHES</td>
<td>Matches</td>
</tr>
<tr>
<td>LIKE</td>
<td>Matches</td>
</tr>
</tbody>
</table>

For all of the above, see Expressions and Operators.

Program Structure, Declaration and Assignment

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS</td>
<td>Declare a class module</td>
</tr>
<tr>
<td>CLEAR</td>
<td>Set all local variables to zero</td>
</tr>
<tr>
<td>CLEARCOMMON</td>
<td>Set all unnamed common variables to zero</td>
</tr>
<tr>
<td>COMMON</td>
<td>Define a common block</td>
</tr>
<tr>
<td>DEFFUN</td>
<td>Define a function</td>
</tr>
<tr>
<td>DIM</td>
<td>Synonym for DIMENSION</td>
</tr>
<tr>
<td>DIMENSION</td>
<td>Set matrix dimensions</td>
</tr>
<tr>
<td>DISINHERIT</td>
<td>Disinherit an object</td>
</tr>
<tr>
<td>END</td>
<td>Terminate program or statement group</td>
</tr>
<tr>
<td>EQUATE</td>
<td>Define a symbolic name for a constant or matrix element</td>
</tr>
<tr>
<td>FUNCTION</td>
<td>Declare function name and arguments</td>
</tr>
<tr>
<td>GET</td>
<td>Synonym for PUBLIC FUNCTION</td>
</tr>
<tr>
<td>INHERIT</td>
<td>Inherit an object</td>
</tr>
<tr>
<td>MAT</td>
<td>Matrix initialisation or copy</td>
</tr>
<tr>
<td>PRIVATE</td>
<td>Declare private variables in a local subroutine or a class module</td>
</tr>
<tr>
<td>PROGRAM</td>
<td>Declare program name</td>
</tr>
<tr>
<td>PUBLIC</td>
<td>Declare public properties in a class module</td>
</tr>
<tr>
<td>PUT</td>
<td>Synonym for PUBLIC SUBROUTINE</td>
</tr>
<tr>
<td>REDIM</td>
<td>Re-dimension a matrix passed as an argument</td>
</tr>
<tr>
<td>REDIMENSION</td>
<td>Re-dimension a matrix passed as an argument</td>
</tr>
<tr>
<td>SHARED</td>
<td>Define shared variables in a class module</td>
</tr>
<tr>
<td>SUBROUTINE</td>
<td>Declare subroutine name and arguments</td>
</tr>
<tr>
<td>VOID</td>
<td>Discard the result of evaluating an expression</td>
</tr>
</tbody>
</table>

Program Control

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABORT</td>
<td>Abort to command prompt</td>
</tr>
<tr>
<td>ABORTE</td>
<td>Abort to command prompt with Pick style message handling</td>
</tr>
<tr>
<td>ABORTM</td>
<td>Abort to command prompt with Information style message handling</td>
</tr>
<tr>
<td>CALL</td>
<td>Call an external subroutine</td>
</tr>
<tr>
<td>CASE</td>
<td>Perform statements according to multiple conditions</td>
</tr>
<tr>
<td>CAUGHT()</td>
<td>Test whether a handler exists for a named exception</td>
</tr>
<tr>
<td>CHAIN</td>
<td>Terminate program and execute a command</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>Continue next iteration of a loop</td>
</tr>
<tr>
<td>ENTER</td>
<td>Synonym for CALL</td>
</tr>
<tr>
<td>ENTER.PACKAGE()</td>
<td>Enter a licensed package</td>
</tr>
<tr>
<td>EXECUTE</td>
<td>Execute a command</td>
</tr>
<tr>
<td>EXIT</td>
<td>Leave a loop</td>
</tr>
<tr>
<td>EXIT.PACKAGE()</td>
<td>Exit from a licensed package</td>
</tr>
<tr>
<td>FOR / NEXT</td>
<td>Iterative loop construct</td>
</tr>
<tr>
<td>GET(ARG.)</td>
<td>Retrieve command line arguments</td>
</tr>
</tbody>
</table>
GO / GOTO Jump to a label
GOSUB Enter an internal subroutine
IF / THEN / ELSE Perform conditional statements
LOCAL Declares an internal subroutine or function that has private local variables
LOOP / REPEAT Define a loop to be repeated
NAP Suspend program for a short period
ON GOSUB Jump to one of a list of labels selected by value
ON GOTO Enter one of a list of internal subroutines selected by value
OS.EXECUTE Execute an operating system command
PAUSE Pause execution until awoken by another process
PERFORM Synonym for EXECUTE
PHANTOM Start a phantom process
POOL.IDLE Move a connection pooling phantom process into the idle state
REMOVE.BREAK.HANDLER Deactivate a break handler subroutine
RETURN Return from CALL or GOSUB
RETURN FROM PROGRAM Return from CALL
RETURN TO Return from program or subroutine to a specific label
RQM Synonym for SLEEP
SERVER.WINDOW() Open an asynchronous server application window
SET.BREAK.HANDLER Establish a break handler subroutine
SIGNAL() Raise a cross-process application signal
SLEEP Suspend program to / for given time
STOP Terminate program
STOPE Terminate program with Pick style message handling
STOPM Terminate program with Information style message handling
SUBR() Call a subroutine as a function
THROW Throw an exception
TRY/CATCH Execute statements with an exception handler
UNTIL Leave loop if condition is met
WAKE Restart execution of a process on a PAUSE
WHILE Leave loop unless condition is met

Mathematical and Logical Functions
ABS() Absolute value
ABSS() Multivalued absolute value
ACOS() Arc-cosine
ANDS() Multivalued logical AND
ASIN() Arc-sine
ATAN() Arc-tangent
BITAND() Bitwise logical AND operation
BITNOT() Bitwise logical NOT operation
BITOR() Bitwise logical OR operation
BITRESET() Turn off specified bit
BITSET() Turn on specified bit
BITTEST() Test specified bit
BITXOR() Bitwise logical exclusive OR operation
CEIL() Smallest integer not less than argument value
COS() Cosine
DIV() Divide
EOS() Multivalued equality test
EXP() Exponential
FIX() Convert a floating point value to a string with a specified precision
FLOOR()  
Largest integer not greater than argument value
GES()  
Multivalued greater than or equal to test
GTS()  
Multivalued greater than test
IDIV()  
Integer division
IFS()  
Multivalued conditional expression
INT()  
Truncate value to integer
LES()  
Multivalued less than test
LN()  
Natural log
LTS()  
Multivalued less than or equal to test
MAX()  
Returns the greater of two values
MAXIMUM()  
Find the greatest value in a dynamic array
MIN()  
Returns the lesser of two values
MINIMUM()  
Find the lowest value in a dynamic array
MOD()  
Modulus value from division
MODS()  
Multivalued modulus value from division
NEG()  
Arithmetic inverse
NEGS()  
Multivalued arithmetic inverse
NES()  
Multivalued inequality test
NOT()  
Logical NOT
NOTS()  
Multivalued logical NOT
ORS()  
Multivalued logical OR
PWR()  
Raise value to power
RDIV()  
Rounded integer division
RANDOMIZE  
Set random number seed value
REM()  
Remainder value from division
REUSE()  
Reuse element of numeric arrays in mathematical functions
REUSING()  
Test whether argument variable has REUSE() applied
RND()  
Generate random number
RNDUP()  
Round a number towards zero in a specified increment
SHIFT()  
Perform bit shift
SIN()  
Sine
SQRT()  
Square root
SUM()  
Sum lowest level elements of a numeric array
SUMMATION()  
Sum all elements of a numeric array
TAN()  
Tangent

String Handling
ALPHA()  
Test if string holds only alphabetic characters
ASCII()  
Convert an EBCDIC string to ASCII
CASING  
Enable/disable case sensitivity in string comparisons
CATS()  
Concatenate elements of a dynamic array
CHANGE()  
Replace substring in a string
CHAR()  
Get ASCII character for a given collating sequence value
CHARS()  
Evaluates the CHAR() function for each element of a dynamic array
COL1()  
Start of substring position from FIELD()
COL2()  
End of substring position from FIELD()
COMPARE()  
Compare strings
COMPARES()  
Multivalued variant of COMPARE()
CONVERT  
Substitute characters with replacements
CONVERT()  
Substitute characters with replacements
COUNT()  
Count occurrences of substring in string
COUNTS()  
Multivalued variant of COUNT()
CROP()  
Remove redundant mark characters
CSVDQ()  
De-quote a CSV string
CSV.MODE
Set CSV conversion mode
DCOUNT()
Count delimited substrings in string
DECRYPT()
Decrypt text
DEL
Delete a field, value or subvalue
DELETE()
Delete a field, value or subvalue
DIGEST()
Form a message digest value for a supplied string or file
DISPLAY.WIDTH()
Return the number of character positions required to display a string
DISPLAY.WIDTHS()
Evaluates the DISPLAY.WIDTH() function for each element of a dynamic array
DOWNCASE()
Convert string to lowercase
DPARSE
Split elements of a delimited string
DPARSE.CSV
Split elements of a CSV format delimited string
QUOTE()
Synonym for QUOTE()
QUOTES()
Synonym for QUOTES()
EBCDIC()
Convert an ASCII string to EBCDIC
ECHAR()
Returns the ECS character with a given codepoint value
ECHARS()
Evaluates the ECHAR() function for each element of a dynamic array
ELEMENT.EXISTS()
Test for presence of a data collection element
ENCRYPT()
Encrypt data
ENUMERATE()
Build a dynamic array of element names in a data collection
EREPLACE()
Replace substring in a string
EXTRACT()
Extract a field, value or subvalue
FIELD()
Extract delimited fields
FIELDS()
Multivalued variant of FIELD()
FIELDSTORE()
Replace or insert delimited fields
FIND
Find a string in a dynamic array element
FINDSTR
Find a substring in a dynamic array element
FMT()
Format a string
FMTDW()
Format a string based on display width
FMTDWS()
Format a dynamic array based on display width
FMTS()
Format a dynamic array
FOLD()
Break a string into sections, splitting at spaces where possible
FOLDDW()
Break a string into sections based on display width, splitting at spaces where possible
FOLDDWS()
Multivalued variant of FOLDDW()
FOLDS()
Multivalued variant of FOLD()
GETREM()
Get remove pointer position
GROUPSTORE
Insert, delete or replace items in a delimited string
ICONV()
Perform input conversion
ICONVS()
Perform input conversion on a dynamic array
INDEX()
Locate occurrence of substring within a string
INDEXS()
Multivalued equivalent of INDEX()
INS
Insert a field, value or subvalue
INSERT()
Insert a field, value or subvalue
IS.ALNUM()
Test whether a character is alphanumeric
IS.ALPHA()
Test whether a character is a letter
IS.DIGIT()
Test whether a character is a digit
IS.ECS()
Test whether a character string contains ECS characters
IS.GRAPH()
Test whether a character is a graphic
IS.MARK()
Test whether a character is a mark
IS.SPACE()
Test whether a character is a space
IS.USER.CHAR()
Test whether a character has the user definable attribute set
IS.WIDESTR() Determine whether a string contains any wide characters
JBUILD() Build a JSON string from a data collection
JPARSE() Parse a JSON string into a data collection
LAST() Extract the final substring from a delimited string
LASTS() Multivalued equivalent of LAST()
LEFT() Return leading substring
LEN() Return length of a string
LENS() Multivalued equivalent of LEN()
LISTINDEX() Return position of an item in a delimited list
LOCATE LOCATE() Locate string in dynamic array
LOCATE() Locate string in dynamic array
LOWER() Convert delimiters to lower level
MATBUILD Build a dynamic array from matrix elements
MATCHES() Matches each element of a dynamic array against a pattern template
MATCHFIELD() Return portion of string matching pattern
MATCHFIELDS() Return portion of string matching pattern for each element of a dynamic array
MATPARSE Break a dynamic array into matrix elements
MATSTR() Convert a dimensioned matrix to a dynamic array
MD5() Convert a string to its 32 digit message digest value.
NS() Return a dynamic array of subvalue positions for supplied data
NUM() Test if string holds a numeric value
NUMS() Multivalued variant of NUM()
NV() Return a dynamic array of value positions for supplied data
OCONV() Perform output conversion
OCONVS() Perform output conversion on a dynamic array
PARSE() Parse a string against a pattern template
QUOTE() Enclose a string in double quotes
QUOTES() Enclose each element of a dynamic array in double quotes
RAISE() Convert delimiters to higher level
RECEX() Test a string against a regular expression
REMOVE() Extract an item from a dynamic array
REMOVE() Extract an item from a dynamic array
REMOVEF() Extract data from a delimited character string
REPLACE() Replace a field, value or subvalue
REVREMOVE() Extract an item from a dynamic array in reverse order
RIGHT() Extract trailing substring
RMVD() Extract an item from a dynamic array
SADD() Perform integer addition of two numeric strings of any length
SCMP() Compare two integer numeric strings of any length
SDIV() Perform integer division of two numeric strings of any length
SEQ() Get collating sequence value for a given ASCII character
SEQS() Evaluates the SEQ() function for each element of a dynamic array
SET.ECS.MAP() Select an ECS character map
SETRE() Set remove pointer position
SMUL() Perform integer multiplication of two numeric strings of any length
SORT_COMPARE() Compare items according to sort rules
SOUNDEX() Form a Soundex code value for a string
SOUNDEXS() Multivalued variant of SOUNDEX()
SPACE() Create a string of spaces
SPACES() Multivalued variant of SPACE()
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPLICE()</td>
<td>Concatenates elements of two dynamic arrays, inserting a string between the items.</td>
</tr>
<tr>
<td>SQUOTE()</td>
<td>Enclose a string in single quotes</td>
</tr>
<tr>
<td>SQUOTES()</td>
<td>Enclose each element of a dynamic array in single quotes</td>
</tr>
<tr>
<td>SSUB()</td>
<td>Perform integer subtraction of two numeric strings of any length</td>
</tr>
<tr>
<td>STR()</td>
<td>Create a string from a repeated substring</td>
</tr>
<tr>
<td>STRS()</td>
<td>Multivalued variant of STR()</td>
</tr>
<tr>
<td>SUBSTITUTE()</td>
<td>Multivalued substring replacement</td>
</tr>
<tr>
<td>SUBSTRDW()</td>
<td>Extract a substring based on its display width</td>
</tr>
<tr>
<td>SUBSTRDWS()</td>
<td>Multivalued equivalent of SUBSTRDW()</td>
</tr>
<tr>
<td>SUBSTRINGS()</td>
<td>Multivalued substring extraction</td>
</tr>
<tr>
<td>SWAP()</td>
<td>Synonym for CHANGE()</td>
</tr>
<tr>
<td>SWAPCASE()</td>
<td>Invert case of alphabetic characters in a string</td>
</tr>
<tr>
<td>SWAPMARKS()</td>
<td>Exchange mark characters with displaced Unicode accented characters</td>
</tr>
<tr>
<td>TRANSLITERATE()</td>
<td>Replace characters by 8 bit transliteration substitutes</td>
</tr>
<tr>
<td>TRIM()</td>
<td>Trim characters from string</td>
</tr>
<tr>
<td>TRIMB()</td>
<td>Trim spaces from back of string</td>
</tr>
<tr>
<td>TRIMBS()</td>
<td>Multivalued variant of TRIMB()</td>
</tr>
<tr>
<td>TRIMF()</td>
<td>Trim spaces from front of string</td>
</tr>
<tr>
<td>TRIMFS()</td>
<td>Multivalued variant of TRIMF()</td>
</tr>
<tr>
<td>TRIMM()</td>
<td>Trim whitespace characters from string</td>
</tr>
<tr>
<td>TRIMMS()</td>
<td>Multivalued variant of TRIMM()</td>
</tr>
<tr>
<td>TRIMW()</td>
<td>Trim whitespace characters from string</td>
</tr>
<tr>
<td>TRIMWS()</td>
<td>Multivalued variant of TRIMW()</td>
</tr>
<tr>
<td>UPCODE()</td>
<td>Convert string to uppercase</td>
</tr>
<tr>
<td>VSLICE()</td>
<td>Extract a value or subvalue slice from a dynamic array</td>
</tr>
</tbody>
</table>

**Date and Time**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE()</td>
<td>Return the current date as a day number</td>
</tr>
<tr>
<td>EPOCH()</td>
<td>Return time and date as an epoch value</td>
</tr>
<tr>
<td>MVDATE()</td>
<td>Extract the multivalue style date from an epoch value</td>
</tr>
<tr>
<td>MVDATE.TIME()</td>
<td>Extract the multivalue style date and time from an epoch value</td>
</tr>
<tr>
<td>MVEPOCH()</td>
<td>Convert a multivalue style date and time to an epoch value</td>
</tr>
<tr>
<td>MVTIME()</td>
<td>Extract the multivalue style time from an epoch value</td>
</tr>
<tr>
<td>SET.TIMEZONE</td>
<td>Set time zone for use by the epoch conversion code</td>
</tr>
<tr>
<td>TIME()</td>
<td>Return the current time</td>
</tr>
<tr>
<td>TIMEDATE()</td>
<td>Return the date and time as a string</td>
</tr>
</tbody>
</table>

**File Handling**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACCOUNT_PATH()</td>
<td>Return pathname for a given account</td>
</tr>
<tr>
<td>BEGIN TRANSACTION</td>
<td>Start a new transaction</td>
</tr>
<tr>
<td>CLEARFILE</td>
<td>Clear a file, deleting all records and releasing disk space</td>
</tr>
<tr>
<td>CLOSE</td>
<td>Close a file</td>
</tr>
<tr>
<td>CLOSESEQ</td>
<td>Close a record opened for sequential access</td>
</tr>
<tr>
<td>COMMIT</td>
<td>Commit transaction updates</td>
</tr>
<tr>
<td>CREATE</td>
<td>Create an empty sequential file record</td>
</tr>
<tr>
<td>CREATE.FILE</td>
<td>Create a file</td>
</tr>
<tr>
<td>DELETE</td>
<td>Delete record from a file</td>
</tr>
<tr>
<td>DELETESEQ</td>
<td>Delete an operating system file</td>
</tr>
<tr>
<td>DELETEU</td>
<td>Delete record from a file preserving locks</td>
</tr>
<tr>
<td>DFPART()</td>
<td>Access a part file within a distributed file</td>
</tr>
<tr>
<td>DIR()</td>
<td>Return the contents of a directory</td>
</tr>
<tr>
<td>DISABLE.KEY</td>
<td>Disable access to a password protected encryption key</td>
</tr>
</tbody>
</table>
Enable access to a password protected encryption key
Terminate a transaction
Perform control action on an open file
Open a file and access data by field name
Create a file event monitoring variable
Return information about an open file
Lock a file
Unlock a file
Flush sequential file data to disk
Flush dynamic file cache
Returns information about file and record locks
Get serial port parameters
Control field mark translation in directory files
Read a record, parsing into a matrix
Read a CSV format text item into a matrix
Read a record setting a read lock, parsing into a matrix
Read a record setting an update lock, parsing into a matrix
Write a record from matrix elements
Write CSV form data from matrix elements to a sequential file
Write a record from matrix elements, retaining any lock
Turn off buffering for a record opened using OPENSEQ
Open a file
Open a file by pathname
Open a record for sequential access
Create and open a temporary file
Delete a file by pathname
Perform action on operating system file
Read a file by pathname
Rename an operating system file or directory
Write a file by pathname
Fetch data from a file using an "outer join"
Read a record from a file
Read bytes from a sequential file
Read a CSV format text item
Read a record from a file, setting a read lock
Read from a sequential file
Read a record from a file, setting an update lock
Read a field from a record in a file
Read a field from a record in a file, setting a read lock
Read a field from a record in a file, setting an update lock
Test if record is locked
Set a read lock on a record
Set an update lock on a record
Release record or file locks
Fetch data from a file
Position a sequential file
Set serial port parameters
Returns a dynamic array of information about an open file
Sets a timeout for READBLK and READSEQ
Fetch data from a file
Abort a transaction
Commit a transaction
Start a new transaction
Resolve a filename to its corresponding pathname
Wait for a file monitoring event to occur
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WEOFSEQ</td>
<td>Write end of file position to sequential file</td>
</tr>
<tr>
<td>WRITE</td>
<td>Write a record to a file</td>
</tr>
<tr>
<td>WRITEBLK</td>
<td>Write bytes to a sequential file</td>
</tr>
<tr>
<td>WRITECSV</td>
<td>Write CSV format data to a sequential file</td>
</tr>
<tr>
<td>WRITESEQ</td>
<td>Write to sequential file</td>
</tr>
<tr>
<td>WRITESEQF</td>
<td>Write to sequential file, flushing to disk</td>
</tr>
<tr>
<td>WRITEU</td>
<td>Write a record to a file, retaining any lock</td>
</tr>
<tr>
<td>WRITEV</td>
<td>Write a field to a record in a file</td>
</tr>
<tr>
<td>WRITEVU</td>
<td>Write a field to a record in a file, retaining any lock</td>
</tr>
<tr>
<td>XLATE()</td>
<td>Synonym for TRANS()</td>
</tr>
</tbody>
</table>

**Printer, Keyboard and Display Handling**

### Terminal Cursor Movement and Control

- @ (x,y): Terminal cursor movement and control
- BREAK: Enable or disable break key handling
- BREAK COUNT(): Return the value of the break inhibit counter
- CLEARDATA: Clear DATA queue
- CLEARINPUT: Clear keyboard type-ahead
- CRT: Synonym for DISPLAY
- DATA: Save text in DATA queue
- DISPLAY: Output to the display
- ECHO: Enable or disable input echo
- ERRMSG: Display a Pick style message from the ERRMSG file
- FOOTING: Set footing text
- FORMCSV(): Transforms a string to a CSV standard compliant item
- GETPU(): Get a characteristic of a print unit
- HEADING: Set heading text
- HUSH: Suppress or enable display output
- IN: Read a single byte from the terminal with an optional timeout
- INPUT: Input a string from the keyboard or data queue
- INPUT @: Input a string from the keyboard or data queue
- INPUTCLEAR: Synonym for CLEARINPUT
- INPUTCSV: Input CSV format data
- INPUTERR: Synonym for PRINTERR
- INPUTFIELD: Input a string with function key handling
- INPUTFIELDV: Input a string with function key handling
- INPUTIF: Input a string from the keyboard type-ahead or data queue
- INPUTIF @: Input a string from the keyboard type-ahead or data queue
- INPUTNULL: Set a special character to return a null string in INPUT @ and INPUTFIELD.
- INPUTTRAP: Defines characters with special actions in INPUT @ and INPUTFIELD
- KEYCODE(): Input a single keystroke from the keyboard with terminfo translation
- KEYCODEV(): Input a single keystroke from the keyboard with terminfo translation to a code point value
- KEYEDIT: Define editing keys for INPUT @
- KEYEXIT: Define exit keys for INPUT @
- KEYINR(): Input a single keystroke from the keyboard in raw mode (no internal processing)
- KEYIN(): Input a single keystroke from the keyboard
- KEYINC(): Input a single keystroke from the keyboard with case inversion
- KEYINCV(): Input a single keystroke from the keyboard with case inversion, returning a code point value
KEYINV()  Input a single keystroke from the keyboard, returning a code point value
KEYREADY() Test for keyboard input
KEYTRAP Define trap keys for INPUT @
MOUSE Enable or disable mouse input
PAGE Start a new page
PRINT Output to a logical print unit
PRINTER CLOSE Close a print unit
PRINTER DISPLAY Associate a print unit with the display
PRINTER FILE Associate a file with a print unit
PRINTER NAME Associate a print device with a print unit
PRINTER OFF Disable print unit zero
PRINTER ON Enable print unit zero
PRINTER RESET Reset default print unit and display
PRINTER SETTING Set a print unit parameter
PRINTER SETTING() Set or retrieve a print unit parameter
PRINTCSV Print data in comma separated variable format
PRINTERR Display an error message
PROMPT Set the input prompt character
PTERM() Set/clear/query terminal settings
RESTORE.SCREEN Restore screen image data
SAVE.SCREEN() Save screen image data
SETPU Set a characteristic of a print unit
TERMINFO() Retrieve information from the terminfo database
TERMINFO.EX() Retrieve extended information from the terminfo database
TTYGET() Get current terminal mode settings
TTYSET Set terminal modes

Select Lists and Alternate Key Indices
BSSCAN Scan an alternate key index
CLEARSELECT Clear one or all select lists
DELETELIST Delete a saved select list
FORMLIST Create a numbered select list from a dynamic array
FORMLISTV Create a select list variable from a dynamic array
GETLIST Restore a saved select list to a numbered list
GETLISTV Restore a saved select list to a select list variable
INDICES() Return information about alternate key indices
KEEPSELECT Indicate that the default select list should not be cleared on return to the command processor
MERGELIST() Combine two field mark delimited sorted lists
READLIST Save a select list in a dynamic array
READNEXT Read a record id from a select list
SAVELIST Save a select list in the $SAVEDLISTS file
SELECT Build a numbered select list of all records in an open file
SELECTE Transfer select list 0 to a select list variable
SELECTINDEX Build a numbered select list from an alternate key index
SELECTINDEXV Build a select list variable from an alternate key index
SELECTINFO() Return information regarding a select list
SELECTLEFT Scan left through an alternate key index, creating a numbered select list
SELECTLEFTV Scan left through an alternate key index, creating a select list variable
SELECTN Build a numbered select list of all records in an open file
SELECTRIGHT Scan right through an alternate key index, creating a numbered select list
SELECTTN Build a numbered select list of all records in an open file
SELECTRIGHT
### SELECTRIGHTV
Scan right through an alternate key index, creating a select list variable.

### SELECTV
Build a select list variable of all records in an open file.

### SETLEFT
Set alternate key index scan position to leftmost.

### SETRIGHT
Set alternate key index scan position to rightmost.

### SSELECT
Build a sorted numbered select list of all records in an open file.

### SSELECTV
Build a sorted select list variable of all records in an open file.

### WRITELIST
Create a saved select list from a dynamic array.

### Sorting
- **SORT()**
  Initialise a sort session.
- **SORTADD**
  Add an item to a sort session.
- **SORTCLEAR**
  Terminate a sort session.
- **SORTDATA()**
  Retrieve sorted data from a sort session.
- **SORTNEXT()**
  Retrieve a single item from a sort session.

### Socket Interface
- **ACCEPT_SOCKET_CONNECTION()**
  Accept an incoming connection on a server socket.
- **CLOSE_SOCKET**
  Close a socket.
- **CREATE_SERVER_SOCKET()**
  Open a server socket.
- **OPEN_SOCKET()**
  Open a socket connection.
- **READ_SOCKET()**
  Read data from a socket.
- **SELECT_SOCKET()**
  Monitor events on multiple sockets.
- **SERVER_ADDR()**
  Find the IP address for a given server name.
- **SET_SOCKET_MODE()**
  Set mode of a socket.
- **SOCKET_INFO()**
  Retrieve information about a socket.
- **WRITE_SOCKET()**
  Write data to a socket.

### Miscellaneous
- **ARG()**
  Returns an argument variable based on its argument list position.
- **ARG.COUNT()**
  Returns the number of arguments passed into a subroutine.
- **ARG.PRESENT()**
  Test for presence of an argument in a subroutine or function.
- **ASSIGNED()**
  Test whether variable is assigned.
- **CATALOGUED()**
  Check catalogue entry.
- **CHECKSUM()**
  Calculate a checksum value for a supplied data item.
- **CHILD()**
  Test if a phantom process is still running.
- **COLLECTION()**
  Create or copy a data collection.
- **CONFIG()**
  Returns the value of a configuration parameter.
- **CONNECT.PORT()**
  Connect a serial port to a phantom process.
- **DEBUG**
  Enter debugger.
- **DTX()**
  Convert a number to hexadecimal.
- **ENV()**
  Retrieve an operating system environment variable.
- **EVALUATE**
  Evaluate an I-type expression, trapping errors.
- **EXPAND()**
  Expand a linked data collection.
- **GET**
  Read data from a device.
- **GET_MESSAGES()**
  Retrieve messages from the message queue.
- **GETNLS()**
  Get national language support parameter value.
- **INMAT()**
  Return status of matrix operations.
- **ITYPE()**
  Execute a compiled I-type.
- **LOCK**
  Set task lock.
- **LOGMSG**
  Add an entry to the system error log.
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAT()</td>
<td>Create or copy a data collection array</td>
</tr>
<tr>
<td>NULL</td>
<td>No operation</td>
</tr>
<tr>
<td>OBJECT()</td>
<td>Instantiates an object</td>
</tr>
<tr>
<td>OBJINFO()</td>
<td>Returns information about an object variable</td>
</tr>
<tr>
<td>OPTION()</td>
<td>Return setting of an option switch</td>
</tr>
<tr>
<td>OS.ERROR()</td>
<td>Return operating system error information</td>
</tr>
<tr>
<td>PDUMP</td>
<td>Generate a process dump file</td>
</tr>
<tr>
<td>PRECISION</td>
<td>Set number of decimal places in numeric conversion</td>
</tr>
<tr>
<td>PROCREAD</td>
<td>Read data from the PROC primary input buffer</td>
</tr>
<tr>
<td>PROCWRITE</td>
<td>Write data to the PROC primary input buffer</td>
</tr>
<tr>
<td>REMARK</td>
<td>Alternative syntax for comments</td>
</tr>
<tr>
<td>SCAN</td>
<td>Scan a data collection</td>
</tr>
<tr>
<td>SEND</td>
<td>Send data to a device</td>
</tr>
<tr>
<td>SENTENCE()</td>
<td>Returns the command line that started the current program</td>
</tr>
<tr>
<td>SET.ARG</td>
<td>Sets an argument variable based on its argument list position</td>
</tr>
<tr>
<td>SET.EXIT.STATUS</td>
<td>Set final exit status value</td>
</tr>
<tr>
<td>SETNLS()</td>
<td>Set national language support parameter value</td>
</tr>
<tr>
<td>SET.STATUS</td>
<td>Set the STATUS() value</td>
</tr>
<tr>
<td>STATUS()</td>
<td>Return status from previous operation</td>
</tr>
<tr>
<td>SYSMSG()</td>
<td>Retrieve a message from the MESSAGES library</td>
</tr>
<tr>
<td>SYSTEM()</td>
<td>Return system information</td>
</tr>
<tr>
<td>TCLREAD</td>
<td>Returns the sentence that started the current program</td>
</tr>
<tr>
<td>TESTLOCK()</td>
<td>Test state of a task lock</td>
</tr>
<tr>
<td>UNASSIGNED()</td>
<td>Test whether variable is unassigned</td>
</tr>
<tr>
<td>UNLOCK</td>
<td>Release task lock</td>
</tr>
<tr>
<td>VARTYPE()</td>
<td>Determine data type of a variable</td>
</tr>
<tr>
<td>XTD()</td>
<td>Convert a hexadecimal number</td>
</tr>
</tbody>
</table>
### 6.5 QMBasic Statements and Functions by Name

<table>
<thead>
<tr>
<th>Function/Keyword</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@ (x,y)</td>
<td>Terminal cursor movement and control</td>
</tr>
<tr>
<td>ABORT</td>
<td>Abort to command prompt</td>
</tr>
<tr>
<td>ABORTE</td>
<td>Abort to command prompt with Pick style message handling</td>
</tr>
<tr>
<td>ABORTM</td>
<td>Abort to command prompt with Information style message handling</td>
</tr>
<tr>
<td>ABS()</td>
<td>Absolute value</td>
</tr>
<tr>
<td>ABS()</td>
<td>Multivalued absolute value</td>
</tr>
<tr>
<td>ACCEPT_SOCKET_CONNECTION()</td>
<td>Accept an incoming connection on a server socket</td>
</tr>
<tr>
<td>ACCOUNT_PATH()</td>
<td>Return pathname for a given account</td>
</tr>
<tr>
<td>ACOS()</td>
<td>Arc-cosine</td>
</tr>
<tr>
<td>ALPHA()</td>
<td>Test if string holds only alphabetic characters</td>
</tr>
<tr>
<td>ANDS()</td>
<td>Multivalued logical AND</td>
</tr>
<tr>
<td>ARG()</td>
<td>Returns an argument variable based on its argument list position</td>
</tr>
<tr>
<td>ARG.COUNT()</td>
<td>Returns the number of arguments passed into a subroutine</td>
</tr>
<tr>
<td>ARG.PRESENT()</td>
<td>Test for presence of an argument in a subroutine or function</td>
</tr>
<tr>
<td>ASCII()</td>
<td>Convert an EBCDIC string to ASCII</td>
</tr>
<tr>
<td>ASIN()</td>
<td>Arc-sine</td>
</tr>
<tr>
<td>ASSIGNED()</td>
<td>Test whether variable is assigned</td>
</tr>
<tr>
<td>ATAN()</td>
<td>Arc-tangent</td>
</tr>
<tr>
<td>BEGIN TRANSACTION</td>
<td>Start a new transaction</td>
</tr>
<tr>
<td>BINDKEY()</td>
<td>Set, remove, query, save or restore key bindings</td>
</tr>
<tr>
<td>BITAND()</td>
<td>Bitwise logical AND operation</td>
</tr>
<tr>
<td>BITNOT()</td>
<td>Bitwise logical NOT operation</td>
</tr>
<tr>
<td>BITOR()</td>
<td>Bitwise logical OR operation</td>
</tr>
<tr>
<td>BITRESET()</td>
<td>Turn off specified bit</td>
</tr>
<tr>
<td>BITSET()</td>
<td>Turn on specified bit</td>
</tr>
<tr>
<td>BITTEST()</td>
<td>Test specified bit</td>
</tr>
<tr>
<td>BITXOR()</td>
<td>Bitwise logical exclusive OR operation</td>
</tr>
<tr>
<td>BREAK</td>
<td>Enable or disable break key handling</td>
</tr>
<tr>
<td>BREAK.COUNT()</td>
<td>Return the value of the break inhibit counter</td>
</tr>
<tr>
<td>BSCAN</td>
<td>Scan an alternate key index</td>
</tr>
<tr>
<td>CALL</td>
<td>Call an external subroutine</td>
</tr>
<tr>
<td>CASE</td>
<td>Perform statements according to multiple conditions</td>
</tr>
<tr>
<td>CASING</td>
<td>Enable/disable case sensitivity in string comparisons</td>
</tr>
<tr>
<td>CATALOGUED()</td>
<td>Check catalogue entry</td>
</tr>
<tr>
<td>CATS()</td>
<td>Concatenate elements of a dynamic array</td>
</tr>
<tr>
<td>CAUGHT()</td>
<td>Test whether a handler exists for a named exception</td>
</tr>
<tr>
<td>CEIL()</td>
<td>Smallest integer not less than argument value</td>
</tr>
<tr>
<td>CHAIN</td>
<td>Terminate program and execute a command</td>
</tr>
<tr>
<td>CHANGE()</td>
<td>Replace substring in a string</td>
</tr>
<tr>
<td>CHAR()</td>
<td>Get ASCII character for a given collating sequence value</td>
</tr>
<tr>
<td>CHARS()</td>
<td>Evaluates the CHAR() function for each element of a dynamic array</td>
</tr>
<tr>
<td>CHECKSUM()</td>
<td>Calculate a checksum value for a supplied data item</td>
</tr>
<tr>
<td>CHILD()</td>
<td>Test if a phantom process is still running</td>
</tr>
<tr>
<td>CLASS</td>
<td>Declare a class module</td>
</tr>
<tr>
<td>CLEAR</td>
<td>Set all local variables to zero</td>
</tr>
<tr>
<td>CLEARCOMMON</td>
<td>Set all unnamed common variables to zero</td>
</tr>
<tr>
<td>CLEARDATA</td>
<td>Clear DATA queue</td>
</tr>
<tr>
<td>CLEARFILE</td>
<td>Clear a file, deleting all records and releasing disk space</td>
</tr>
<tr>
<td>CLEARINPUT</td>
<td>Clear keyboard type-ahead</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>CLEARSELECT</td>
<td>Clear one or all select lists</td>
</tr>
<tr>
<td>CLOSE</td>
<td>Close a file</td>
</tr>
<tr>
<td>CLOSESEQ</td>
<td>Close a record opened for sequential access</td>
</tr>
<tr>
<td>CLOSE_SOCKET</td>
<td>Close a socket</td>
</tr>
<tr>
<td>COL1()</td>
<td>Start of substring position from FIELD()</td>
</tr>
<tr>
<td>COL2()</td>
<td>End of substring position from FIELD()</td>
</tr>
<tr>
<td>COLLECTION()</td>
<td>Create or copy a data collection</td>
</tr>
<tr>
<td>COMMIT</td>
<td>Commit transaction updates</td>
</tr>
<tr>
<td>COMMON</td>
<td>Define a common block</td>
</tr>
<tr>
<td>COMPARE()</td>
<td>Compare strings</td>
</tr>
<tr>
<td>COMARES()</td>
<td>Multivalued variant of COMPARE()</td>
</tr>
<tr>
<td>CONFIU()</td>
<td>Returns the value of a configuration parameter</td>
</tr>
<tr>
<td>CONNECT_PORT()</td>
<td>Connect a serial port to a phantom process</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>Continue next iteration of a loop</td>
</tr>
<tr>
<td>CONVERT</td>
<td>Substitute characters with replacements</td>
</tr>
<tr>
<td>CONVERT()</td>
<td>Substitute characters with replacements</td>
</tr>
<tr>
<td>COS()</td>
<td>Cosine</td>
</tr>
<tr>
<td>COUNT()</td>
<td>Count occurrences of substring in string</td>
</tr>
<tr>
<td>COUNTS()</td>
<td>Multivalued variant of COUNT()</td>
</tr>
<tr>
<td>CREATE</td>
<td>Create an empty sequential file record</td>
</tr>
<tr>
<td>CREATE/File</td>
<td>Create a file</td>
</tr>
<tr>
<td>CREATE.SERVER.SOCKEL()</td>
<td>Open a server socket</td>
</tr>
<tr>
<td>CROP()</td>
<td>Remove redundant mark characters</td>
</tr>
<tr>
<td>CRT</td>
<td>Synonym for DISPLAY</td>
</tr>
<tr>
<td>CSV.DQ()</td>
<td>De-quote a CSV string</td>
</tr>
<tr>
<td>CSV.MODE</td>
<td>Set CSV conversion mode</td>
</tr>
<tr>
<td>DATA</td>
<td>Save text in DATA queue</td>
</tr>
<tr>
<td>DATE()</td>
<td>Return the current date as a day number</td>
</tr>
<tr>
<td>DCOUNT()</td>
<td>Count delimited substrings in string</td>
</tr>
<tr>
<td>DEBUG</td>
<td>Enter debugger</td>
</tr>
<tr>
<td>DECRYPT()</td>
<td>Decrypt text</td>
</tr>
<tr>
<td>DEFFUN</td>
<td>Define a function</td>
</tr>
<tr>
<td>DEL</td>
<td>Delete a field, value or subvalue</td>
</tr>
<tr>
<td>DELETE</td>
<td>Delete record from a file</td>
</tr>
<tr>
<td>DELETE()</td>
<td>Delete a field, value or subvalue</td>
</tr>
<tr>
<td>DELETELIST</td>
<td>Delete a saved select list</td>
</tr>
<tr>
<td>DELETESEQ</td>
<td>Delete an operating system file</td>
</tr>
<tr>
<td>DELETEU</td>
<td>Delete record from a file preserving locks</td>
</tr>
<tr>
<td>DFPART()</td>
<td>Access a part file within a distributed file</td>
</tr>
<tr>
<td>DIGEST()</td>
<td>Form a message digest value for a supplied string or file</td>
</tr>
<tr>
<td>DIM</td>
<td>Synonym for DIMENSION</td>
</tr>
<tr>
<td>DIMENSION</td>
<td>Set matrix dimensions</td>
</tr>
<tr>
<td>DISINHERIT</td>
<td>Disinherit an object</td>
</tr>
<tr>
<td>DISPLAY</td>
<td>Output to the display</td>
</tr>
<tr>
<td>DISPLAY.WIDTH()</td>
<td>Return the number of character positions required to display a string</td>
</tr>
<tr>
<td>DISPLAY.WIDTHS()</td>
<td>Evaluates the DISPLAY.WIDTH() function for each element of a dynamic array</td>
</tr>
<tr>
<td>DIR()</td>
<td>Return the contents of a directory</td>
</tr>
<tr>
<td>DISABLE.KEY</td>
<td>Deny access to a password protected encryption key</td>
</tr>
<tr>
<td>DIV()</td>
<td>Divide</td>
</tr>
<tr>
<td>DOWNCASE()</td>
<td>Convert string to lowercase</td>
</tr>
<tr>
<td>DPARSE</td>
<td>Split elements of a delimited string</td>
</tr>
<tr>
<td>DPARSE.CSV</td>
<td>Split elements of a CSV format delimited string</td>
</tr>
<tr>
<td>DQUOTE()</td>
<td>Synonym for QUOTE()</td>
</tr>
<tr>
<td>DQUOTES()</td>
<td>Synonym for QUOTES()</td>
</tr>
<tr>
<td>DTX()</td>
<td>Convert a number to hexadecimal</td>
</tr>
</tbody>
</table>
EBCDIC() Convert an EBCDIC string to ASCII
ECHAR() Returns the ECS character with a given codepoint value
ECHARS() Evaluates the ECHAR() function for each element of a
dynamic array
ECHO Enable or disable input echo
ELEMENT.EXISTS() Test for presence of a data collection element
ENABLE.KEY Enable access to a password protected encryption key
ENCRYPT() Encrypt data
END Terminate program or statement group
END TRANSACTION Terminate a transaction
ENTER Synonym for CALL
ENTER.PACKAGE() Enter a licensed package
ENUMERATE() Build a dynamic array of element names in a data
collection
ENV() Retrieve an operating system environment variable
EPOCH() Return time and date as an epoch value
EOS() Multivalued equality test
EQUATE Define a symbolic name for a constant or matrix element
EREPLACE() Replace substring in a string
ERRMSG Display a Pick style message from the ERRMSG file
EVALUATE Evaluate an I-type expression, trapping errors
EXECUTE Execute a command
EXIT Leave a loop
EXIT.PACKAGE() Exit from a licensed package
EXP() Exponential
EXPAND() Expand a linked data collection
EXTRACT() Extract a field, value or subvalue
FCONTROL() Perform control action on an open file
FIELD() Extract delimited fields
FIELDS() Multivalued variant of FIELD()
FIELDSTORE() Replace or insert delimited fields
FILE Open a file and access data by field name
FILE.EVENT() Create a file event monitoring variable
FILEINFO() Return information about an open file
FILELOCK Lock a file
FILEUNLOCK Unlock a file
FIND Find a string in a dynamic array element
FINDSTR Find a substring in a dynamic array element
FIX() Convert a floating point value to a string with a specified
precision
FLOOR() Largest integer not greater than argument value
FLUSH Flush sequential file data to disk
FLUSH.DH.CACHE Flush dynamic file cache
FMT() Format a string
FMTD() Format a string based on display width
FMTDWS() Format a dynamic array based on display width
FMTS() Format a dynamic array
FOLD() Break a string into sections, splitting at spaces where
possible
FOLDDW() Break a string into sections based on display width, splitting
at spaces where possible
FOLDS() Multivalued variant of FOLD()
FOLDDWS() Multivalued variant of FOLDDW()
FOOTING Set footing text
FOR / NEXT Iterative loop construct
FORMCSV() Transforms a string to a CSV standard compliant item
FORMLIST Create a numbered select list from a dynamic array
FORMLISTV
FUNCTION
GES()
GET
GET
GET(ARG.)
GET.MESSAGES()
GET.PORT.PARAMS()
GETLIST
GETLISTV
GETLOCKS
GETNL5()
GETPU()
GETREM()
GO / GOTO
GOSUB
GROUPSTORE
GTS()
HEADING
HUSH
ICONV()
ICONVS()
IDIV()
IF / THEN / ELSE
IFS()
IN
INDEX()
INDEXES()
INDICES()
INHERIT
INMAT()
INPUT
INPUT @
INPUTCLEAR
INPUTCSV
INPUTERR
INPUTFIELD
INPUTFIELDV
INPUTIF
INPUTIF @
INPUTNULL
INPUTTRAP
INS
INSERT()
INT()
IS.ALNUM()
IS.ALPHA()
IS.DIGIT()
IS.ECS()
IS.GRAPH()
IS.MARK()
IS.SPACe()
IS.USER.CHAR()
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IS.WIDESTR()</td>
<td>Determine whether a string contains any wide characters</td>
</tr>
<tr>
<td>ITYPE()</td>
<td>Execute a compiled I-type</td>
</tr>
<tr>
<td>JBUILD()</td>
<td>Build a JSON string from a data collection</td>
</tr>
<tr>
<td>JPARSE()</td>
<td>Parse a JSON string into a data collection</td>
</tr>
<tr>
<td>KEEP.SELECT</td>
<td>Indicate that the default select list should not be cleared on return to the command processor</td>
</tr>
<tr>
<td>KEYCODE()</td>
<td>Input a single keystroke from the keyboard with terminfo translation</td>
</tr>
<tr>
<td>KEYCODEV()</td>
<td>Input a single keystroke from the keyboard with terminfo translation to a code point value</td>
</tr>
<tr>
<td>KEYEYEDIT</td>
<td>Define editing keys for INPUT @</td>
</tr>
<tr>
<td>KEYEXIT</td>
<td>Define exit keys for INPUT @</td>
</tr>
<tr>
<td>KEYIN()</td>
<td>Input a single keystroke from the keyboard, returning the character</td>
</tr>
<tr>
<td>KEYINC()</td>
<td>Input a single keystroke from the keyboard with case inversion</td>
</tr>
<tr>
<td>KEYINCV()</td>
<td>Input a single keystroke from the keyboard with case inversion, returning a code point value</td>
</tr>
<tr>
<td>KEYINR()</td>
<td>Input a single keystroke from the keyboard in raw mode (no internal processing)</td>
</tr>
<tr>
<td>KEYINV()</td>
<td>Input a single keystroke from the keyboard, returning a code point value</td>
</tr>
<tr>
<td>KEYREADY()</td>
<td>Test for keyboard input</td>
</tr>
<tr>
<td>KEYTRAP</td>
<td>Define trap keys for INPUT @</td>
</tr>
<tr>
<td>LAST()</td>
<td>Extract the final substring from a delimited string</td>
</tr>
<tr>
<td>LASTS()</td>
<td>Multivalued equivalent of LAST()</td>
</tr>
<tr>
<td>LEFT()</td>
<td>Return leading substring</td>
</tr>
<tr>
<td>LEN()</td>
<td>Return length of a string</td>
</tr>
<tr>
<td>LENS()</td>
<td>Multivalued equivalent of LEN()</td>
</tr>
<tr>
<td>LES()</td>
<td>Multivalued less than test</td>
</tr>
<tr>
<td>LISTINDEX()</td>
<td>Return position of an item in a delimited list</td>
</tr>
<tr>
<td>LN()</td>
<td>Natural log</td>
</tr>
<tr>
<td>LOCAL</td>
<td>Declares an internal subroutine or function that has private local variables</td>
</tr>
<tr>
<td>LOCATE</td>
<td>Locate string in dynamic array</td>
</tr>
<tr>
<td>LOCATE()</td>
<td>Locate string in dynamic array</td>
</tr>
<tr>
<td>LOCK</td>
<td>Set task lock</td>
</tr>
<tr>
<td>LOGMSG</td>
<td>Add an entry to the system error log</td>
</tr>
<tr>
<td>LOOP / REPEAT</td>
<td>Define a loop to be repeated</td>
</tr>
<tr>
<td>LOWER()</td>
<td>Convert delimiters to lower level</td>
</tr>
<tr>
<td>LTS()</td>
<td>Multivalued less than or equal to test</td>
</tr>
<tr>
<td>MARK.MAPPING</td>
<td>Control field mark translation in directory files</td>
</tr>
<tr>
<td>MAT</td>
<td>Matrix initialisation or copy</td>
</tr>
<tr>
<td>MAT()</td>
<td>Create or copy a data collection array</td>
</tr>
<tr>
<td>MATEBUILD</td>
<td>Build a dynamic array from matrix elements</td>
</tr>
<tr>
<td>MATCH()</td>
<td>Matches each element of a dynamic array against a pattern template</td>
</tr>
<tr>
<td>MATCHFIELD()</td>
<td>Return portion of string matching pattern</td>
</tr>
<tr>
<td>MATCHFIELDS()</td>
<td>Return portion of string matching pattern for each element of a dynamic array</td>
</tr>
<tr>
<td>MATPARSE</td>
<td>Break a dynamic array into matrix elements</td>
</tr>
<tr>
<td>MATREAD</td>
<td>Read a record, parsing into a matrix</td>
</tr>
<tr>
<td>MATREADCSV</td>
<td>Read a CSV format text item into a matrix</td>
</tr>
<tr>
<td>MATREADL</td>
<td>Read a record setting a read lock, parsing into a matrix</td>
</tr>
<tr>
<td>MATREADU</td>
<td>Read a record setting an update lock, parsing into a matrix</td>
</tr>
<tr>
<td>MATSTR()</td>
<td>Convert a dimensioned matrix to a dynamic array</td>
</tr>
<tr>
<td>MATWRITE</td>
<td>Write a record from matrix elements</td>
</tr>
</tbody>
</table>
MATWRITECSV
Write CSV form data from matrix elements to a sequential file

MATWRITEU
Write a record from matrix elements, retaining any lock

MAX()  
Returns the greater of two values

MAXIMUM()  
Find the greatest value in a dynamic array

MD5()  
Convert a string to its 32 digit message digest value

MERGELIST()  
Combine two field mark delimited sorted lists

MIN()  
Returns the lesser of two values

MINIMUM()  
Find the lowest value in a dynamic array

MOD()  
Modulus value from division

MODS()  
Multivalued modulus value from division

MOUSE  
Enable or disable mouse input

MVDATE()  
Extract the multivalue style date from an epoch value

MVDATE.TIME()  
Extract the multivalue style date and time from an epoch value

MVEPOCH()  
Convert a multivalue style date to an epoch value

MVTIME()  
Extract the multivalue style time from an epoch value

NAP  
Suspend program for a short period

NEG()  
Arithmetic inverse

NEGS()  
Multivalued arithmetic inverse

NES()  
Multivalued inequality test

NOBUF  
Turn off buffering for a record opened using OPENSEQ

NOT()  
Logical NOT

NOTS()  
Multivalued logical NOT

NULL  
Return a dynamic array of subvalue positions for supplied data

NUM()  
No operation

NUMS()  
Test if string holds a numeric value

NV()  
Multivalued variant of NUM()

OBJECT()  
Return a dynamic array of value positions for supplied data

OBJINFO()  
Instantiate an object

OCONV()  
Perform output conversion

OCONVS()  
Perform output conversion on a dynamic array

ON GOSUB  
Jump to one of a list of labels selected by value

ON GOTO  
Enter one of a list of internal subroutines selected by value

OPEN  
Open a file

OPENPATH  
Open a file by pathname

OPENSEQ  
Open a record for sequential access

OPENTEMP  
Create and open a temporary file

OPEN.SOCKET()  
Open a socket connection

OPTION()  
Return setting of an option switch

ORS()  
Multivalued logical OR

OS.ERROR()  
Return operating system error information

OS.EXECUTE  
Execute an operating system command

OSDELETE  
Delete a file by pathname

OSPATH()  
Perform action on operating system file

OSREAD  
Read a file by pathname

OSRENAME()  
Rename an operating system file or directory

OSWRITE  
Write a file by pathname

OUTERJOIN()  
Fetch data from a file using an "outer join"

PAGE  
Start a new page

PARSE()  
Parse a string against a pattern template

PAUSE  
Pause execution until awoken by another process

PDUMP  
Generate a process dump file

PERFORM  
Synonym for EXECUTE

PHANTOM  
Start a phantom process
<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>POOL.IDLE</strong></td>
<td>Move a connection pooling phantom process into the idle state</td>
</tr>
<tr>
<td><strong>PRECISION</strong></td>
<td>Set number of decimal places in numeric conversion</td>
</tr>
<tr>
<td><strong>PRINT</strong></td>
<td>Output to a logical print unit</td>
</tr>
<tr>
<td><strong>PRINTCSV</strong></td>
<td>Print data in comma separated variable format</td>
</tr>
<tr>
<td><strong>PRINTER.CLOSE</strong></td>
<td>Close a print unit</td>
</tr>
<tr>
<td><strong>PRINTER.DISPLAY</strong></td>
<td>Associate a print unit with the display</td>
</tr>
<tr>
<td><strong>PRINTER.FILE</strong></td>
<td>Associate a file with a print unit</td>
</tr>
<tr>
<td><strong>PRINTER.NAME</strong></td>
<td>Associate a print device with a print unit</td>
</tr>
<tr>
<td><strong>PRINTER.OFF</strong></td>
<td>Disable print unit zero</td>
</tr>
<tr>
<td><strong>PRINTER.ON</strong></td>
<td>Enable print unit zero</td>
</tr>
<tr>
<td><strong>PRINTER.RESET</strong></td>
<td>Reset default print unit and display</td>
</tr>
<tr>
<td><strong>PRINTER.SETTING</strong></td>
<td>Set a print unit parameter</td>
</tr>
<tr>
<td><strong>PRINTER.SETTING()</strong></td>
<td>Set or retrieve a print unit parameter</td>
</tr>
<tr>
<td><strong>PRINTERR</strong></td>
<td>Display an error message</td>
</tr>
<tr>
<td><strong>PRIVATE</strong></td>
<td>Declare private variables in a local subroutine or a class module</td>
</tr>
<tr>
<td><strong>PROCREAD</strong></td>
<td>Read data from the PROC primary input buffer</td>
</tr>
<tr>
<td><strong>PROCWRITE</strong></td>
<td>Write data to the PROC primary input buffer</td>
</tr>
<tr>
<td><strong>PROGRAM</strong></td>
<td>Declare program name</td>
</tr>
<tr>
<td><strong>PROMPT</strong></td>
<td>Set the input prompt character</td>
</tr>
<tr>
<td><strong>PTERM()</strong></td>
<td>Set/clear/query terminal settings</td>
</tr>
<tr>
<td><strong>PUBLIC</strong></td>
<td>Declare public properties in a class module</td>
</tr>
<tr>
<td><strong>PUT</strong></td>
<td>Synonym for PUBLIC SUBROUTINE</td>
</tr>
<tr>
<td><strong>PWR()</strong></td>
<td>Raise value to power</td>
</tr>
<tr>
<td><strong>QUOTE()</strong></td>
<td>Enclose a string in quotes</td>
</tr>
<tr>
<td><strong>QUOTES()</strong></td>
<td>Enclose each element of a dynamic array in double quotes</td>
</tr>
<tr>
<td><strong>RAISE()</strong></td>
<td>Convert delimiters to higher level</td>
</tr>
<tr>
<td><strong>RANDOMIZE</strong></td>
<td>Set random number seed value</td>
</tr>
<tr>
<td><strong>RDIV()</strong></td>
<td>Rounded integer division</td>
</tr>
<tr>
<td><strong>READ</strong></td>
<td>Read a record from a file</td>
</tr>
<tr>
<td><strong>READ.SOCKET()</strong></td>
<td>Read data from a socket</td>
</tr>
<tr>
<td><strong>READBLK</strong></td>
<td>Read data from a sequential file</td>
</tr>
<tr>
<td><strong>READCSV</strong></td>
<td>Read a CSV format text item</td>
</tr>
<tr>
<td><strong>READL</strong></td>
<td>Read bytes from a sequential file</td>
</tr>
<tr>
<td><strong>READ.SOCKET()</strong></td>
<td>Read a record from a file</td>
</tr>
<tr>
<td><strong>READSEQ</strong></td>
<td>Save a select list in a dynamic array</td>
</tr>
<tr>
<td><strong>READNEXT</strong></td>
<td>Read a record id from a select list</td>
</tr>
<tr>
<td><strong>READU</strong></td>
<td>Read from a sequential file</td>
</tr>
<tr>
<td><strong>READV</strong></td>
<td>Read a record from a file, setting an update lock</td>
</tr>
<tr>
<td><strong>READVL</strong></td>
<td>Read a field from a record in a file</td>
</tr>
<tr>
<td><strong>READVU</strong></td>
<td>Read a field from a record in a file, setting a read lock</td>
</tr>
<tr>
<td><strong>RECORDLOCKED()</strong></td>
<td>Test if record is locked</td>
</tr>
<tr>
<td><strong>RECORDLOCKL</strong></td>
<td>Set a read lock on a record</td>
</tr>
<tr>
<td><strong>RECORDLOCKU</strong></td>
<td>Set an update lock on a record</td>
</tr>
<tr>
<td><strong>REDIM</strong></td>
<td>Re-dimension a matrix passed as an argument</td>
</tr>
<tr>
<td><strong>REDIMENSION</strong></td>
<td>Re-dimension a matrix passed as an argument</td>
</tr>
<tr>
<td><strong>REGEX()</strong></td>
<td>Test a string against a regular expression</td>
</tr>
<tr>
<td><strong>RELEASE</strong></td>
<td>Release record or file locks</td>
</tr>
<tr>
<td><strong>REM()</strong></td>
<td>Remainder value from division</td>
</tr>
<tr>
<td><strong>REMARK</strong></td>
<td>Alternative syntax for comments</td>
</tr>
<tr>
<td><strong>REMOVE</strong></td>
<td>Extract an item from a dynamic array</td>
</tr>
<tr>
<td><strong>REMOVE()</strong></td>
<td>Extrac an item from a dynamic array</td>
</tr>
<tr>
<td><strong>REMOVEF()</strong></td>
<td>Extract data from a delimited character string</td>
</tr>
<tr>
<td><strong>REMOVE.BREAK.HANDLER</strong></td>
<td>Deactivate a break handler subroutine</td>
</tr>
<tr>
<td><strong>REPLACE()</strong></td>
<td>Replace a field, value or subvalue</td>
</tr>
<tr>
<td><strong>RESTORE.SCREEN</strong></td>
<td>Restore screen image data</td>
</tr>
</tbody>
</table>
RETURN
RETURN FROM PROGRAM
RETURN TO
REUSE(
REUSING()
REVREMO
e
RIGHT(
RMVD(
RN()D
ROUNDUP()
ROLLE
ROM
RTRANS()
SADD(
SAVE.SCREEN(
SAVELIST
SCAN
SCMP() SDIV()
SEEK
SELECT
SELECT.SOCKE
SELECT.E
SELECT.INDEX
SELECT.INDEXV
SELECT.INFO
SELECT.LEFT
SELECTLEFT
SELECTLEFTV
SELECTN
SELECTRIGHT
SELECTRIGHTV
SELECTV
SEND
SENTENCE(
SEQ() SEQS()
SERVER.ADDR()
SERVER.WINDO
SET.ARG
SET.BREAK.HANDLER
SET.ECS.MAP()
SET.EXIT.STATUS
SET.PORT.PARAMS()
SET.SOCKE
SET.STATUS
SET.TIMEZONE
SE
SETNLS()
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETPU</td>
<td>Set a characteristic of a print unit</td>
</tr>
<tr>
<td>SETRIGHT</td>
<td>Set alternate key index scan position to rightmost</td>
</tr>
<tr>
<td>SETREM</td>
<td>Set remove pointer position</td>
</tr>
<tr>
<td>SHARED</td>
<td>Define shared variables in a class module</td>
</tr>
<tr>
<td>SHIFT()</td>
<td>Perform bit shift</td>
</tr>
<tr>
<td>SIGNAL()</td>
<td>Raise a cross-process application signal</td>
</tr>
<tr>
<td>SIN()</td>
<td>Sine</td>
</tr>
<tr>
<td>SLEEP</td>
<td>Suspend program to / for given time</td>
</tr>
<tr>
<td>SMUL()</td>
<td>Perform integer multiplication of two numeric strings of any length</td>
</tr>
<tr>
<td>SOCKET_INFO()</td>
<td>Retrieve information about a socket</td>
</tr>
<tr>
<td>SORT()</td>
<td>Initialise a sort session</td>
</tr>
<tr>
<td>SORTADD</td>
<td>Add an item to a sort session</td>
</tr>
<tr>
<td>SORTCLEAR</td>
<td>Terminate a sort session</td>
</tr>
<tr>
<td>SORTDATA()</td>
<td>Retrieve sorted data from a sort session</td>
</tr>
<tr>
<td>SORTNEXT()</td>
<td>Return next item from a sort session</td>
</tr>
<tr>
<td>SORT_COMPARE()</td>
<td>Compare items according to sort rules</td>
</tr>
<tr>
<td>SOUNDEX()</td>
<td>Form a Soundex code value for a string</td>
</tr>
<tr>
<td>SOUNDEXXS()</td>
<td>Multivalued variant of SOUNDEX()</td>
</tr>
<tr>
<td>SPACE()</td>
<td>Create a string of spaces</td>
</tr>
<tr>
<td>SPACES()</td>
<td>Multivalued variant of SPACE()</td>
</tr>
<tr>
<td>SPLICE()</td>
<td>Concatenates elements of two dynamic arrays, inserting a string between the items.</td>
</tr>
<tr>
<td>SQRT()</td>
<td>Square root</td>
</tr>
<tr>
<td>SQUOTE()</td>
<td>Enclose a string in single quotes</td>
</tr>
<tr>
<td>SQUOTES()</td>
<td>Enclose each element of a dynamic array in single quotes</td>
</tr>
<tr>
<td>SSELECT</td>
<td>Build a sorted select list of all records in an open file</td>
</tr>
<tr>
<td>SUB()</td>
<td>Perform integer subtraction of two numeric strings of any length</td>
</tr>
<tr>
<td>STATUS()</td>
<td>Return status from previous operation</td>
</tr>
<tr>
<td>STATUS</td>
<td>Returns a dynamic array of information about an open file</td>
</tr>
<tr>
<td>STOP</td>
<td>Terminate program</td>
</tr>
<tr>
<td>STOPE</td>
<td>Terminate program with Pick style message handling</td>
</tr>
<tr>
<td>STOPM</td>
<td>Terminate program with Information style message handling</td>
</tr>
<tr>
<td>STR()</td>
<td>Create a string from a repeated substring</td>
</tr>
<tr>
<td>STRS()</td>
<td>Multivalued variant of STR()</td>
</tr>
<tr>
<td>SUBR()</td>
<td>Call a subroutine as a function</td>
</tr>
<tr>
<td>SUBROUTINE</td>
<td>Declare subroutine name and arguments</td>
</tr>
<tr>
<td>SUBSTITUTE()</td>
<td>Multivalued substring replacement</td>
</tr>
<tr>
<td>SUBSTRDW()</td>
<td>Extract a substring based on its display width</td>
</tr>
<tr>
<td>SUBSTRDWS()</td>
<td>Multivalued equivalent of SUBSTRDW()</td>
</tr>
<tr>
<td>SUBSTRINGS()</td>
<td>Multivalued substring extraction</td>
</tr>
<tr>
<td>SUM()</td>
<td>Sum lowest level elements of a numeric array</td>
</tr>
<tr>
<td>SUMMATION()</td>
<td>Sum all elements of a numeric array</td>
</tr>
<tr>
<td>SWAP()</td>
<td>Synonym for CHANGE()</td>
</tr>
<tr>
<td>SWAPCASE()</td>
<td>Invert case of alphabetic characters in a string</td>
</tr>
<tr>
<td>SWAPMARKS()</td>
<td>Exchange mark characters with displaced Unicode accented characters</td>
</tr>
<tr>
<td>SYSMSG()</td>
<td>Retrieve a message from the MESSAGES library</td>
</tr>
<tr>
<td>SYSTEM()</td>
<td>Return system information</td>
</tr>
<tr>
<td>TAN()</td>
<td>Tangent</td>
</tr>
<tr>
<td>TCLREAD</td>
<td>Returns the sentence that started the current program</td>
</tr>
<tr>
<td>TERMINO()</td>
<td>Retrieve information from the terminfo database</td>
</tr>
<tr>
<td>TERMINFO.EX()</td>
<td>Retrieve extended information from the terminfo database</td>
</tr>
<tr>
<td>TESTLOCK()</td>
<td>Test state of a task lock</td>
</tr>
<tr>
<td>THROW</td>
<td>Throw an exception</td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>--------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TIME()</td>
<td>Return the current time</td>
</tr>
<tr>
<td>TIMEDATE()</td>
<td>Return the date and time as a string</td>
</tr>
<tr>
<td>TIMEOUT</td>
<td>Sets a timeout for READBLK and READSEQ</td>
</tr>
<tr>
<td>TRANS()</td>
<td>Fetch data from a file</td>
</tr>
<tr>
<td>TRANSACTION ABORT</td>
<td>Abort a transaction</td>
</tr>
<tr>
<td>TRANSACTION COMMIT</td>
<td>Commit a transaction</td>
</tr>
<tr>
<td>TRANSACTION START</td>
<td>Start a new transaction</td>
</tr>
<tr>
<td>TRANSLITERATE()</td>
<td>Replace characters by 8 bit transliteration substitutes</td>
</tr>
<tr>
<td>TRIM()</td>
<td>Trim characters from string</td>
</tr>
<tr>
<td>TRIMB()</td>
<td>Trim spaces from back of string</td>
</tr>
<tr>
<td>TRIMBS()</td>
<td>Multivalued variant of TRIMB()</td>
</tr>
<tr>
<td>TRIMF()</td>
<td>Trim spaces from front of string</td>
</tr>
<tr>
<td>TRIMFS()</td>
<td>Multivalued variant of TRIMF()</td>
</tr>
<tr>
<td>TRIMS()</td>
<td>Multivalued variant of TRIM()</td>
</tr>
<tr>
<td>TRIMW()</td>
<td>Trim whitespace characters from string</td>
</tr>
<tr>
<td>TRIMWS()</td>
<td>Multivalued variant of TRIMW()</td>
</tr>
<tr>
<td>TRY/CATCH</td>
<td>Execute statements with an exception handler</td>
</tr>
<tr>
<td>TTYGET()</td>
<td>Get current terminal mode settings</td>
</tr>
<tr>
<td>TTYSET</td>
<td>Set terminal modes</td>
</tr>
<tr>
<td>UNASSIGNED()</td>
<td>Test whether variable is unassigned</td>
</tr>
<tr>
<td>UNLOCK</td>
<td>Release task lock</td>
</tr>
<tr>
<td>UNTIL</td>
<td>Leave loop if condition is met</td>
</tr>
<tr>
<td>UPCASE()</td>
<td>Convert string to uppercase</td>
</tr>
<tr>
<td>VARTYPE()</td>
<td>Determine data type of a variable</td>
</tr>
<tr>
<td>VOCPATH()</td>
<td>Resolve a filename to its corresponding pathname</td>
</tr>
<tr>
<td>VOID</td>
<td>Discard the result of evaluating an expression</td>
</tr>
<tr>
<td>VSLICE()</td>
<td>Extract a value or subvalue slice from a dynamic array</td>
</tr>
<tr>
<td>WAIT.FILE.EVENT()</td>
<td>Wait for a file monitoring event to occur</td>
</tr>
<tr>
<td>WAKE</td>
<td>Restart execution of a process on a PAUSE</td>
</tr>
<tr>
<td>WEOFSEQ</td>
<td>Write end of file position to sequential file</td>
</tr>
<tr>
<td>WHILE</td>
<td>Leave loop unless condition is met</td>
</tr>
<tr>
<td>WRITE</td>
<td>Write a record to a file</td>
</tr>
<tr>
<td>WRITE(Socket)</td>
<td>Write data to a socket</td>
</tr>
<tr>
<td>WRITEBLK</td>
<td>Write bytes to a sequential file</td>
</tr>
<tr>
<td>WRITECSV</td>
<td>Write CSV format data to a sequential file</td>
</tr>
<tr>
<td>WRITELIST</td>
<td>Create a saved select list from a dynamic array</td>
</tr>
<tr>
<td>WRITESEQ</td>
<td>Write to sequential file</td>
</tr>
<tr>
<td>WRITESEOF</td>
<td>Write to sequential file, flushing to disk</td>
</tr>
<tr>
<td>WRITEEU</td>
<td>Write a record to a file, retaining any lock</td>
</tr>
<tr>
<td>WRITEV</td>
<td>Write a field to a record in a file</td>
</tr>
<tr>
<td>WRITEVU</td>
<td>Write a field to a record in a file, retaining any lock</td>
</tr>
<tr>
<td>XLATE()</td>
<td>Synonym for TRANS()</td>
</tr>
<tr>
<td>XTD()</td>
<td>Convert a hexadecimal number</td>
</tr>
</tbody>
</table>
 @(x,y) Function

The @(x, y) function is used to control the format of displayed output.

Format

@ (col {, line})  Cursor movement
@ (mode {, arg})  Device control functions

where

col  evaluates to a display column position.
line  evaluates to a display line position.
mode  evaluates to a mode value as described below.
arg  provides qualifying information for use with some mode values.

The @(x, y) functions return string values which can be used in the same way as any other strings. They only take effect when the string is used in a CRT, DISPLAY or PRINT statements directed to the display. The actual value returned by the function is a control code to be sent to the terminal and is dependant on the type of terminal in use (see the TERM command). By using this function instead of hard coding control sequences into programs, applications can be portable across a wide range of terminal devices or emulations.

When output is directed to a printer or to a file for later printing rather than to the display, escape sequences relevant to the printer may be used for formatting, etc. Because the @(x,y) function returns codes specific for the terminal in use, it is unlikely that these codes are relevant for printers.

Some functions are extensions to the standard terminal capability definitions and may require use of AccuTerm with terminal type that have a -at suffix or QMterm in qterm mode.

Cursor Positioning

The @ (col {, line}) format specifies that subsequent output is to appear at the given column and, optionally, line position. Columns and lines are numbered from zero where the top left of the screen is line 0, column 0. The effect of attempting to move to a cursor position outside the display area is undefined.

Some terminal emulations (e.g. vpa2-at) may not handle cursor positions correctly if set to a large screen size such as 132 column width. This is not a defect in QM but is caused by limitations of the control codes used by these terminals.

Use of the @ (col {, line}) function disables screen pagination (automatic display of the "Press return to continue" prompt after each screen of output). Pagination remains disabled until the program executes the PRINTER RESET statement or returns to the command prompt.

Use of the EXECUTE statement enables screen pagination, executes the command(s) and then restores pagination to its state when the EXECUTE was performed.

Special Functions
@\((x, y)\) functions with negative values of \(x\) are used to provide a variety of control functions. These are largely in common with the functions defined for other systems. The token names shown in the table below are defined in the KEYS.H include record in the SYSCOM file.

Functions not supported by the terminal device in use return a null string and hence will be ignored. Note that individual terminal types may place restrictions on use of display attributes such as flashing, underline, colour, etc. For example, although each attribute has a corresponding start and end control code pair, many terminals can only apply a single visual attribute to any particular screen region.

Display attributes are also implemented in two fundamentally distinct ways. Most modern terminals maintain an attribute for each character position on the display and data is stored by the terminal using the currently active attributes regardless of any intervening cursor movements. On some other terminals such as the Wyse50, the code sent by an @() function to set or clear an attribute occupies a character position on the screen. Starting at the top left of the screen and working row by row through each character position, the attributes of any particular character are determined by the most recent attribute setting encountered.

For example,

\[
\text{DISPLAY } \text{@(-1):@(-15)}: \text{'ABC':@(-16):'DEF'}
\]

on a terminal that stores the attribute for each character position would result in a display of

```
ABC
```

whereas a terminal that uses a character position to store the attribute would display

```
ABC
```

Furthermore, if the program then executed

```
\text{DISPLAY } \text{@(3,0):'X'}
```

the first terminal would display

```
ABC
```

but the second would display

```
ABCX
```

with the underline extending to the end of the screen or the next character position holding an attribute setting.

<table>
<thead>
<tr>
<th>Value</th>
<th>Token</th>
<th>Function</th>
<th>Argument</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>IT$SCS</td>
<td>Clear screen</td>
<td></td>
</tr>
<tr>
<td>-2</td>
<td>IT$CAH</td>
<td>Cursor home</td>
<td></td>
</tr>
<tr>
<td>-3</td>
<td>IT$CLEOS</td>
<td>Clear to end of screen</td>
<td></td>
</tr>
<tr>
<td>-4</td>
<td>IT$CLEOL</td>
<td>Clear to end of line</td>
<td></td>
</tr>
<tr>
<td>-5</td>
<td>IT$SBBLINK</td>
<td>Start flashing text</td>
<td></td>
</tr>
<tr>
<td>-6</td>
<td>IT$EBLINK</td>
<td>End flashing text</td>
<td></td>
</tr>
<tr>
<td>-7</td>
<td>IT$SPA</td>
<td>Start protected area</td>
<td>No of characters (default 1)</td>
</tr>
<tr>
<td>-8</td>
<td>IT$EPA</td>
<td>End protected area</td>
<td></td>
</tr>
<tr>
<td>-9</td>
<td>IT$CUB</td>
<td>Backspace</td>
<td>No of characters (default 1)</td>
</tr>
<tr>
<td>-10</td>
<td>IT$CUU</td>
<td>Cursor up</td>
<td>No of lines (default 1)</td>
</tr>
<tr>
<td>-11</td>
<td>IT$SHALF</td>
<td>Start half brightness</td>
<td></td>
</tr>
<tr>
<td>-12</td>
<td>IT$EHALF</td>
<td>End half brightness</td>
<td></td>
</tr>
<tr>
<td>-13</td>
<td>IT$SREV</td>
<td>Start reverse video</td>
<td></td>
</tr>
<tr>
<td>-14</td>
<td>IT$EREV</td>
<td>End reverse video</td>
<td></td>
</tr>
</tbody>
</table>
-15 ITSSUL Start underline
-16 ITSEUL End underline
-17 ITSIL Insert line No of lines (default 1)
-18 ITSIDL Delete line No of lines (default 1)
-19 ITSICH Insert character No of characters (default 1)
-20 ITSSIRM Set insert/remove mode
-21 ITSRIRM Reset insert/remove mode
-22 ITSDCH Delete character No of characters (default 1)
-23 ITSAUXON Turn on printer
-24 ITSAUXOFF Turn off printer
-29 ITSE80 Set 80 column mode
-30 ITSE132 Set 132 column mode
-31 ITSRIC Reset inhibit cursor
-32 ITSSIC Inhibit cursor
-33 ITSCUD Cursor down No of lines (default 1)
-34 ITSCUF Cursor forward No of characters (default 1)
-37 ITSFGC Set foreground colour Colour
-38 ITSBGC Set background colour Colour
-54 ITSSLT Set line truncation
-55 ITSLRT Reset line truncation
-58 ITSSBOLD Set bold mode
-59 ITSEBOLD Reset bold mode
-60 ITSSAM Start automatic margin mode
-61 ITSEAM End automatic margin mode

User definable via the u0 to u7 terminfo keys

-108 IT SACMD Asynchronous command Command to execute
-109 ITSSCMCMD Synchronous command Command to execute
-250 IT$STYLUS Enable/disable stylus taps 0 = disable, 1 = enable (PDA, obsolete)
-251 IT$KEYS Display/hide screen keyboard 0 = hide, 1 = display (PDA, obsolete)

Descriptions

Each of the descriptions below expands on the preceding table. There is also a reference to the terminfo element that is used to determine the control code sent to the terminal. Unless otherwise stated, where more than one terminfo element is shown, the first one defined for the current terminal.
device is used. The function will only operate correctly if the corresponding terminfo element is correctly defined.

**IT$SCS (Clear screen)**
The screen is cleared to the current background colour. The cursor is positioned at the top left of the screen (position 0,0). Terminfo "clear".

**IT$CAH (Cursor home)**
The cursor is positioned at the top left of the screen (position 0,0). Terminfo "home".

**IT$CLEOS (Clear to end of screen)**
All screen positions between the current cursor position and the end of the screen are cleared. Some terminal devices/emulators set the cleared character positions to have the current background colour. Terminfo "ed".

**IT$CLEOL (Clear to end of line)**
All screen positions between the current cursor position and the end of the line are cleared. Some terminal devices/emulators set the cleared character positions to have the current background colour. Terminfo "el".

**IT$SBLINK (Start flashing text)**
Displays subsequent text in flashing mode. Terminfo "blink".

**IT$EBLINK (End flashing text)**
Terminates flashing mode. Terminfo "eblink" or "sgr0".

**IT$SPA (Start protected area)**
Displays subsequent text in protected mode. Characters displayed in this mode can only be updated by erasing them from the screen with modes IT$CS, IT$CLEOS or IT$CLEOL. Terminfo "prot".

**IT$EPA (End protected area)**
Terminates protected mode. Terminfo "eprot" or "sgr0".

**IT$CUB (Backspace)**
The cursor moves left by the number of positions specified in the second argument which defaults to one or until the left edge of the screen is reached. Terminfo "cub1" repeated.

**IT$CUU (Cursor up)**
The cursor moves up by the number of lines specified in the second argument which defaults to one or until the top line of the screen is reached. Terminfo "cuu1" repeated.

**IT$SHALF (Start half brightness)**
Displays subsequent data in half brightness (dim) mode. Terminfo "dim".

**IT$EHALF (End half brightness)**
Terminates half brightness (dim) mode. Terminfo "edim" or "sgr0".

**IT$SREV (Start reverse video)**
If not already in reverse video mode, the foreground and background colours are interchanged for subsequent output. Selecting this mode does not directly affect any data which is already displayed. Terminfo "rev".

**IT$EREV (End reverse video)**
If in reverse video mode, this operation reverts to the normal display colours for subsequent text output. Terminfo "erev" or "sgr0".

**IT$SUL (Start underline)**
Displays subsequent data in underline mode. Terminfo "smul".

**IT$EUL (End underline)**
Terminates underline mode. Terminfo "rmul".

**IT$IL (Insert line)**
The number of lines specified in the second argument (default value one) are inserted at the current cursor position. Data at the bottom of the screen will be lost. The newly inserted lines are set to the background colour. Terminfo "ill" if inserting one line, otherwise "il" with qualifying line count.

**IT$DL (Delete line)**
The number of lines specified in the second argument (default value one) are deleted at the current cursor position. Blank lines are inserted at the bottom of the screen. Terminfo "dll" if inserting one line, otherwise "dl" with qualifying line count.

**IT$ICH (Insert character)**
The number of characters specified in the second argument (default value one) are inserted at the current cursor position. Data at the right of the screen will be lost. Terminfo "ichl" repeated.

**IT$DCH (Delete character)**
The number of characters specified in the second argument (default value one) are deleted at the current cursor position. Blanks are inserted at the right edge of the screen. Terminfo "dchl" repeated.

**IT$AUXON (Turn on printer)**
For terminals with attached printers, this mode directs output to the printer. Terminfo "mc5".

**IT$AUXOFF (Turn off printer)**
For terminals with attached printers, this mode turns off output to the printer. Terminfo "mc4".

**IT$E80 (Set 80 column mode)**
The display window is set to be 80 characters wide. Terminfo "set80".

**IT$E132 (Set 132 column mode)**
The display window is set to be 132 characters wide. Terminfo "set132".

**IT$RIC (Reset inhibit cursor)**
The cursor is displayed if it was previously inhibited. Terminfo "cvvis".

**IT$SIC (Inhibit cursor)**
Display of the cursor is inhibited. All cursor positioning functions continue to work whilst the cursor is not visible. Terminfo "civis".

**IT$CUD (Cursor down)**
The cursor moves down by the number of lines specified in the second argument which defaults to one or until the bottom line of the screen is reached. Terminfo "cud1" repeated.

**IT$CUF (Cursor forward)**
The cursor moves right by the number of positions specified in the second argument which defaults to one or until the right edge of the screen is reached. Terminfo "cuf1" repeated.

**IT$FGC (Set foreground colour)**
The foreground colour is set according to the value of the second argument. This may be set using the tokens listed below from the KEYS.H record of the SYSCOM file. Terminfo "setf" with colour value.

<table>
<thead>
<tr>
<th>Token</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>IT$BLACK</td>
<td>0</td>
</tr>
<tr>
<td>IT$BLUE</td>
<td>1</td>
</tr>
</tbody>
</table>
Some terminal emulators provide the ability to map these colour values to an alternative colour palette. The terminfo COLOURMAP setting can be used to translate the internal QM values listed above to an alternative set relevant to a specific terminal emulator.

**IT$SBGC (Set background colour)**
The background colour is set according to the value of the second argument. This may be set using the tokens from the KEYS.H record of the SYSCOM file as listed above. Terminfo "setb" with colour value.

**IT$SLT (Set line truncation)**
With this mode enabled, the cursor does not automatically move to a new line when data is displayed in the final column of the screen. Any further output on the line will overwrite the final character. Terminfo "slt".

**IT$RLT (Reset line truncation)**
Clears line truncation mode so that the cursor automatically moves to a new line when data is displayed in the final column of the screen. Terminfo "rlt".

**IT$SBOLD (Start bold)**
Displays subsequent data in bold mode. Terminfo "bold".

**IT$EBOLD (End bold)**
Terminates bold mode. Terminfo "ebold" or "sgr0".

**IT$SAM (Start automatic margin)**
In this mode, writing data to the final character position of the line automatically moves to the start of the next line. Terminfo "smam".

**IT$EAM (End automatic margin)**
With this mode disabled, writing data to the final character position of the line does not automatically move to the start of the next line. Terminfo "rmam".

**IT$ACMD (Asynchronous command)**
Executes the given command on the client system without suspending the QM session. Terminfo "u8" with qualifying data.

**IT$SCMD (Synchronous command)**
Executes the given command on the client system, suspending the QM session until the command completes. Terminfo "u9" with qualifying data.

**IT$STYLUS (Enable/disable stylus taps)**
Applicable only to QM on a PDA, this operation is equivalent to use of the MOUSE statement and allows an application to receive stylus taps via input operations. When enabled, a stylus tap appears
as char(200) (K$MOUSE) followed by the column and row coordinates separated by a comma and terminated with a carriage return. Set the second argument to a non-zero value to enable stylus taps, zero to disable them. Stylus input is disabled by default when QM starts. Terminfo "stylus" with qualifier. Use of the KEYCODE() or KEYCODEV() functions will automatically process the positional data related to the stylus tap, leaving the screen coordinates in @MCOL and @MROW.

**IT$KEYS (Display/hide screen keyboard)**
Applicable only to QM on a PDA, this operation displays or hides the on screen keyboard. Set the second argument to a non-zero value to show the keyboard, zero to hide it. Terminfo "pdakeys" with qualifier.

**Examples**

```
DISPLAY @(IT$CS) : @(34,10) : "Please wait" :
```

This statement clears the screen and displays "Please wait".

```
DISPLAY @(IT$FGC, IT$BRIGHT.RED) : "Error " : STATUS()
```

This statement displays the value of the STATUS() function in bright red. Further output to the display will continue to be in this colour until the foreground colour is reset.
ABORT

The **ABORT** statement terminates the current program, returning to the command prompt. **ABORTE** and **ABORTM** provide compatibility with other multivalue database products.

**Format**

\[ \text{ABORT}\{message\} \]

where

message evaluates to the message to be displayed. Although this is optional, an **ABORT** with no message is likely to be hard to debug.

If an **ON.ABORT** paragraph is defined in the VOC, this will be executed before the command prompt is issued.

The program location at which the abort was generated will be reported unless the SUPPRESS.ABORT.MSG option has been set using the **OPTION** command.

Because **ABORT** terminates all active programs, menus, paragraphs, etc., it should only be used to handle error conditions.

The Pick syntax of **ABORT** can be enabled by including a line

\[ $\text{MODE}\ P\text{ICK.ER RMSG} \]

in the program before the first **ABORT** statement. This mode is also selected automatically if there is a comma after message.

In this syntax, the **ABORT** statement becomes

\[ \text{ABORT}\{message\{,arg\...}\} \]

where

message evaluates to the id of a record in the ERRMSG file which holds the message to be displayed. If this id is numeric, it will be copied to @SYSTEM.RETURN.CODE.

arg... is an optional comma separated list of arguments to be substituted into the message.

See the **ERRMSG** statement for a description of the ERRMSG file message format.

The **ABORTE** statement always uses Pick style message handling and the **ABORTM** statement always uses Information style message handling, regardless of the setting of the PICK.ER RMSG option.

**Examples**

\[ \text{IF NO.OF.ENTRIES = 0 THEN ABORT} \]
This statement aborts to the command prompt if the value of the variable NO.OF.ENTRIES is zero. No error message is printed. **ABORT** statements without error text messages can result in difficult diagnostic work to locate faults.

```
OPEN "STOCK.FILE" TO STOCK ELSE
   ABORT "Cannot open STOCK.FILE - Error " : STATUS()
END
```

This program fragment attempts to open a file named STOCK.FILE. If the open fails, the program displays an error message and aborts to the command prompt.

**See also:**

ERRMSG, STOP
ABS()

The ABS() function returns the absolute (positive) value of a numeric expression.

The ABSS() function is similar to ABS() but operates on successive elements of a dynamic array, returning a similarly structured dynamic array of results.

Format

\texttt{ABS(expr)}

where

\texttt{expr} evaluates to a number or a numeric array.

The ABS() function returns the absolute value of \texttt{expr}. If \texttt{expr} is positive or zero, the value of \texttt{ABS(expr)} is \texttt{expr}. If \texttt{expr} is negative, the value of \texttt{ABS(expr)} is \texttt{-expr}.

If \texttt{expr} is a numeric array (a dynamic array where all elements are numeric), the ABS() function operates on each element in turn and returns a numeric array with the same structure as \texttt{expr}.

Examples

\texttt{DIFF = ABS(A - B)}

This statement assigns DIFF to the difference in value of A and B. The result is positive regardless of which value is the greater.

A contains 11FM-2VM-8VM4

\texttt{B = ABSS(A)}

B now contains 11FM2VM8VM4
ACCEPT.SOCKET.CONNECTION()

The ACCEPT.SOCKET.CONNECTION() function opens a data socket on a server to handle an incoming stream connection.

Format

\[ \text{ACCEPT.SOCKET.CONNECTION}(srvr.sk, \text{timeout}) \]

where

\[ srvr.sk \] is the server socket created by an earlier use of CREATE.SERVER.SOCKET.
\[ \text{timeout} \] is the timeout period in milliseconds. A value of zero implies no timeout.

The ACCEPT.SOCKET.CONNECTION() function waits for an incoming connection on a previously created server socket and returns a new data socket for this connection.

If the action is successful, the function returns a socket variable that can be used to read and write data using the READ.SOCKET() and WRITE.SOCKET() functions. The STATUS() function will return zero. The socket will initially be in blocking mode. This can be modified by use of the SET.SOCKET.MODE() function.

If the socket cannot be opened, the STATUS() function will return an error code that can be used to identify the cause of the error. If no connection arrives before the timeout period expires, the error code will be ER$TIMEOUT as defined in the SYSCOM ERR.H include record.

Example

\[
\begin{align*}
\text{SRVR.sk} & = \text{CREATE.SERVER.SOCKET}('', 4000, \text{SKT$STREAM + SKT$TCP}) \\
\text{IF STATUS()} & \text{ THEN STOP 'Cannot initialise server socket'} \\
\text{SKT} & = \text{ACCEPT.SOCKET.CONNECTION}(\text{SRVR.sk}, 0) \\
\text{IF STATUS()} & \text{ THEN STOP 'Error accepting connection'} \\
\text{DATA} & = \text{READ.SOCKET}(\text{SKT}, 100, \text{SKT$BLOCKING}, 0) \\
\text{CLOSE.SOCKET} & \text{ SKT} \\
\text{CLOSE.SOCKET} & \text{ SRVR.sk}
\end{align*}
\]

This program fragment creates a server socket, waits for an incoming connection, reads a single data packet from this connection and then closes the sockets.

See also:
Using Socket Connections, CLOSE.SOCKET, CREATE.SERVER.SOCKET(), OPEN.SOCKET(), READ.SOCKET(), SELECT.SOCKET(), SERVER.ADDR(), SET.SOCKET.MODE(), SOCKET.INFO(), WRITE.SOCKET()
ACCOUNT.PATH()

The ACCOUNT.PATH() function returns the pathname corresponding to the supplied account name.

Format

    ACCOUNT.PATH(account)

where

    account

is the account name for which the pathname is required.

The ACCOUNT.PATH() function allows an application to find the pathname of an account. The account name will be converted to uppercase internally. If the account does not exist, a null string is returned.

Example

    ACC.PATH = ACCOUNT.PATH("SALES")
    IF ACC.PATH = "" THEN STOP "Sales account not found"

This program fragment looks up the pathname of the SALES account. If it cannot be found, the program terminates with an error message.
ACOS()

The ACOS() function returns the arc-cosine (inverse cosine) of a value.

Format

ACOS(expr)

where

expr evaluates to a number or a numeric array.

The ACOS() function returns the arc-cosine of expr. Angles are measured in degrees.

If expr is a numeric array (a dynamic array where all elements are numeric), the ACOS() function operates on each element in turn and returns a numeric array with the same structure as expr.

Examples

ANGLE = ACOS(ADJ / HYP)

This statement finds the angle with cosine equal to the value of ADJ / HYP and assigns this to variable ANGLE.

See also:

ASIN(), ATAN(), COS(), SIN(), TAN()
**ALPHA()**

The **ALPHA()** function tests whether a string contains only alphabetic characters.

**Format**

```
ALPHA(string)
```

where

`string` evaluates to the string to be tested.

The **ALPHA()** function returns True if `string` contains only alphabetic characters (A to Z, a to z). The function returns False for a null string or a string that contains non-alphabetic characters.

Use of **ALPHA()** is a faster equivalent to using pattern matching against a pattern of "1A0A".

**Example**

```
LOOP
   DISPLAY "Enter surname ":
   INPUT NAME
   WHILE NOT (ALPHA(NAME))
      PRINTERR "Name is invalid"
   REPEAT
```

This program fragment prompts for and inputs a name. If the name is null or contains non-alphabetic characters, an error message is displayed and the prompt is repeated.

**See also:**

Pattern matching, **NUM()**
The **ANDS()** function performs a logical AND operation on successive elements of a dynamic array, returning a similarly structured dynamic array of results.

**Format**

```
ANDS(expr1, expr2)
```

where

- `expr1`, `expr2` are the dynamic arrays to be processed.

The **ANDS()** function performs the logical AND operation between corresponding elements of the two dynamic arrays and constructs a similarly structured dynamic array of results as its return value. An element of the returned dynamic array is 1 if both of the corresponding elements of `expr1` and `expr2` are True.

The **REUSE()** function can be applied to either or both expressions. Without this function, any absent trailing values are taken as False.

**Examples**

A contains 1VM1SM0VM0VM1FM0VM1
B contains 1VM0SM1VM0VM1FM1VM0

```
C = ANDS(A, B)
```

C now contains 1VM0SM0VM0VM1FM0VM0

**See also:**

- **EQS()**, **GES()**, **GTS()**, **IFS()**, **LES()**, **LTS()**, **NES()**, **NOTS()**, **ORS()**, **REUSE()**
The ARG() function returns and argument value based on its position in the argument list. It is intended for use with subroutines declared with the VAR.ARGS option.

**Format**

ARG(n)

where

n is the argument list position, numbered from one.

Subroutines declared with the VAR.ARGS option may have a variable number of arguments. Although each argument must have a name assigned to it in the SUBROUTINE statement, it is often useful to be able to process a series of arguments by indexing this list.

The ARG() function returns the value of argument n. The actual number of arguments passed may be determined using the ARG.COUNT() function. Use of an argument position value less than one or greater than the number of arguments causes the program to abort.

**Example**

FUNCTION AVG(A,B,C,D,E) VAR.ARGS
   TOTAL = 0
   FOR I = 1 TO ARG.COUNT()
      TOTAL += ARG(I)
   NEXT I
   RETURN TOTAL / ARG.COUNT()
END

The above function returns the average of the supplied arguments. Because this function is declared with the VAR.ARGS option, the ARG.COUNT() function is used to determine the actual number of arguments and the ARG() function is used to access each argument by its position.

**See also:**

ARG.COUNT(), ARG.PRESENT(), SET.ARG
ARG.COUNT()

The **ARG.COUNT()** function returns the number of arguments passed into the current subroutine. It is intended for use with subroutines declared with the **VAR.ARGS** option.

**Format**

```
ARG.COUNT()
```

When a program calls a subroutine that has been declared with the **VAR.ARGS** option, the actual number of arguments passed may be fewer than the number of argument variables in the subroutine definition. The unused argument variables will be left unassigned. The **ARG.COUNT()** function allows a subroutine to determine the number of arguments that have been passed. See the **ARG()** function for a way to access arguments by their position in the argument list.

When used in a function, the value returned by **ARG.COUNT()** excludes the hidden return argument.

Note that the **ARG.COUNT()** function relates to the arguments passed into a **SUBROUTINE**, **FUNCTION** or **CLASS** module and does not reflect the number of arguments passed into a local subroutine or function.

**Example**

```
FUNCTION AVG(A,B,C,D,E) VAR.ARGS
   TOTAL = 0
   FOR I = 1 TO ARG.COUNT()
      TOTAL += ARG(I)
   NEXT I
   RETURN TOTAL / ARG.COUNT()
END
```

The above function returns the average of the supplied arguments. Because this function is declared with the **VAR.ARGS** option, the **ARG.COUNT()** function is used to determine the actual number of arguments and the **ARG()** function is used to access each argument by its position.

**See also:**

**ARG()**, **ARG.PRESENT()**, **SET.ARG**
**ARG.PRESENT()**

The **ARG.PRESENT()** function tests whether an argument variable was passed by the caller of a subroutine or function declared with the **VAR.ARGS** option.

**Format**

```
ARG.PRESENT(var)
```

where

```
var
```

is the name of an argument variable.

When a program calls a subroutine or function that has been declared with the **VAR.ARGS** option, the actual number of arguments passed may be fewer than the number of argument variables in the subroutine definition. The unused argument variables will be left unassigned. The **ARG.PRESENT()** function allows the subroutine or function to test whether the argument was passed by the caller. Use of default values for missing arguments does not affect the value returned by this function.

Note that the **ARG.PRESENT()** function relates to the arguments passed into a **SUBROUTINE**, **FUNCTION** or **CLASS** module and does not reflect the number of arguments passed into a local subroutine or function.

**Example**

```
SUBROUTINE INCREMENT.COUNTER(CT) VAR.ARGS
   READU REC FROM @VOC, 'COUNTER' ELSE STOP 'COUNTER missing'
   IF ARG.PRESENT(CT) THEN CT = REC<2>
   REC<2> += 1
   WRITE REC TO @VOC, 'COUNTER'
RETURN
END
```

The above subroutine increments a counter stored in field 2 of a VOC X-type record. If the CT argument is present in the call to this subroutine, the previous value of the counter is returned via this argument.

**See also:**

**ARG(), ARG.COUNT(), SET.ARG**
ASCII()

The ASCII() function converts an EBCDIC string to ASCII.

Format

\[ \text{ASCII}(expr) \]

where

\[ expr \] evaluates to the string to be converted.

The ASCII() function returns the ASCII equivalent of the supplied EBCDIC string. Characters that have no ASCII equivalent are returned as question marks.

See also:
EBCDIC()
ASIN()

The ASIN() function returns the arc-sine (inverse sine) of a value.

Format

```
ASIN(expr)
```

where

```
expr evaluates to a number or a numeric array.
```

The ASIN() function returns the arc-sine of expr. Angles are measured in degrees.

If expr is a numeric array (a dynamic array where all elements are numeric), the ASIN() function operates on each element in turn and returns a numeric array with the same structure as expr.

Example

```
ANGLE = ASIN(OPP / HYP)
```

This statement finds the angle with sine equal to the value of OPP / HYP and assigns this to variable ANGLE.

See also:
ACOS(), ATAN(), COS(), SIN(), TAN()
ASSIGNED()

The ASSIGNED() function tests whether a variable is assigned.

Format

ASSIGNED(var)

where

\[ var \] is the variable to be tested.

All QMBasic variables except those in common blocks are initially unassigned. Any attempt to use the contents of the variable in an expression would cause a run time error until such time as a value has been stored in it. The ASSIGNED() function allows a program to test whether a variable has been assigned, returning True if it is assigned or False if it is unassigned.

Example

SUBROUTINE VALIDATE(ACCOUNT.CODE, ERROR)
    IF ASSIGNED(ACCOUNT.CODE) THEN
        ERROR = 0
        ...processing code...
    END ELSE
        ERROR = 1
    END
    RETURN
END

This program fragment validates an account code. The use of the ASSIGNED() function prevents an abort if the variable has not been assigned.

See also:

UNASSIGNED()
**ATAN()**

The ATAN() function returns the arc-tangent (inverse tangent) of a value.

**Format**

```
ATAN(expr)
```

where

```
expr evaluates to a number or a numeric array.
```

The ATAN() function returns the arc-tangent of `expr`. Angles are measured in degrees.

If `expr` is a numeric array (a dynamic array where all elements are numeric), the ATAN() function operates on each element in turn and returns a numeric array with the same structure as `expr`.

**Example**

```
ANGLE = ATAN(OPP / ADJ)
```

This statement finds the angle with tangent equal to the value of OPP / ADJ and assigns this to variable ANGLE.

**See also:**

`ACOS()`, `ASIN()`, `COS()`, `SIN()`, `TAN()`
BEGIN CASE

The **CASE** statement provides conditional execution dependant on the result of expression evaluation.

**Format**

```qmbasic
BEGIN CASE
  CASE expr
    statement(s)
  CASE expr
    statement(s)
END CASE
```

where

- `expr` is an expression which can be resolved to a Boolean value
- `statement(s)` is a group of QMBasic statements. There may be any number of statements in the group (including zero).

The expressions are evaluated in turn until one evaluates to True. The statements associated with that test expression are then executed and control passes to the statement following the **END CASE**. Only one of the statement groups is executed. If none of the test expressions evaluates to True, no statements are executed.

It is frequently useful to execute a default set of statements where no specific test expression results in non-zero result. This can be achieved by a case of the form

```qmbasic
CASE 1
```

which makes use of the fact that 1 is the numeric representation of the Boolean True. This must be the final element of the **CASE** statement.

**Example**

```qmbasic
N = DCOUNT(ITEMS, @VM)
BEGIN CASE
  CASE N = 0
    DISPLAY "There are no items"
  CASE N = 1
    DISPLAY "There is 1 item"
  CASE 1
    DISPLAY "There are " : N : " items"
END CASE
```

This program fragment displays a message indicating the number of values in the ITEMS variable. Note the use of the **CASE 1** construct.
BEGIN TRANSACTION

The BEGIN TRANSACTION statement marks the start of a new transaction.

Format

BEGIN TRANSACTION
{ statements }
COMMIT / ROLLBACK
...
END TRANSACTION

A transaction is a group of updates that must either be performed in their entirety or not at all. The BEGIN TRANSACTION statement starts a new transaction. All updates until a corresponding END TRANSACTION are cached and only applied to the database when a COMMIT statement is executed. Execution of the program then continues at the statement following the END TRANSACTION.

The ROLLBACK statement discards any cached updates and continues at the statement following the END TRANSACTION. A roll-back is implied if the program executes the END TRANSACTION directly.

Testing the value of the STATUS() function immediately after the END TRANSACTION statement will return 0 if the transaction committed successfully, ER_COMMIT_FAIL if an error occurred during the commit action, or ER_ROLLBACK if the transaction was rolled back.

Deletes and writes inside a transaction will fail unless the program holds an update lock on the record or the file. All locks obtained inside the transaction are retained until the transaction terminates and are then released. Locks already owned when the transaction begins will still be present after the transaction terminates, even if the record is updated or deleted within the transaction.

Closing a file inside a transaction appears to work in that the file variable is destroyed though the actual close is deferred until the transaction terminates and any updates have been applied to the file. Rolling back the transaction will not reinstate the file variable.

Access to indices using SELECTINDEX, SELECTLEFT or SELECTRIGHT inside a transaction will not reflect any updates within the transaction as these have not been committed.

Updates to sequential records opened using OPENSEQ are not affected by transactions.

Transactions may be nested. If the BEGIN TRANSACTION statement is executed inside an active transaction, the active transaction is stacked and a new transaction commences. Termination of the new transaction reverts to the stacked transaction. The default behaviour of QM is that transactions are durable such that updates in a child transaction are applied to the data files on use of COMMIT. The NON.DURABLE.TXN setting of the QMBasic $MODE compiler directive, makes transactions non-durable such that updates in a child transaction are inherited by the parent transaction on COMMIT.

The following operations are banned inside transactions:

- CLEARFILE
- PHANTOM

Example
BEGIN TRANSACTION
    READU CUST1.REC FROM CUST.F, CUST1.ID ELSE ROLLBACK
    CUST1.REC<C.BALANCE> -= TRANSFER.VALUE
    WRITE CUST1.REC TO CUST.F, CUST1.ID

    READU CUST2.REC FROM CUST.F, CUST2.ID ELSE ROLLBACK
    CUST2.REC<C.BALANCE> += TRANSFER.VALUE
    WRITE CUST2.REC TO CUST.F, CUST2.ID
    COMMIT
END TRANSACTION

The above program fragment transfers money between two customer accounts. The updates are only committed if the entire transaction is successful.

See also:
Transactions, TRANSACTION
**BINDKEY()**

The **BINDKEY()** function sets, removes, queries, saves or restores key bindings.

**Format**

```
BINDKEY(key.string, action)
```

where

- **key.string**  The character sequence for the key to be bound, unbound or queried.
- **action**  identifies the action to be performed:
  - >= 0  Bind key to this code (0 to 65535)
  - -1  Remove binding for `key.string`.
  - -2  Query binding for `key.string`.
  - -3  Save bindings.
  - -4  Restore bindings from `key.string`.
  - -5  Disables lone Esc key handling in **KEYCODE()**.
  - -6  Re-enables lone Esc key handling in **KEYCODE()**.

The **BINDKEY()** function used with an **action** value in the range 0 to 65535 (0xFFFF) binds the key sequence in `key.string` to the given **action** value. This is the underlying mechanism of the **KEYEDIT**, **KEYEXIT** and **KEYTRAP** statements used to set up keys for special handling by **INPUT@** and **INPUTFIELD**. If successful, the function returns True and the **STATUS()** function would return zero. If an error occurs, the function returns False and the **STATUS()** function can be used to find the cause of the error:

1. Invalid `key.string`
2. Invalid **action**
3. `Key.string` conflicts with an existing binding

Action values for many keys are defined in the KEYIN.H include record in the SYSCOM file. It is recommended that when binding special keys such as the function keys and cursor keys, the action values defined with the KV$ prefix should be used in preference to those with the K$ prefix. If this is done, the **KEYCODE()** function will return the corresponding 8-bit character associated with this key but the **KEYCODEV()** function will return the Unicode BMP Private Use Area value. If the K$ prefix values are used, both functions return the 8-bit value which may cause programs to fail on ECS mode systems.

An **action** value of -1 removes any defined binding for `key.string`. Used in this mode, the function always returns True even if there was no binding for this `key.string`.

An **action** value of -2 returns the action number bound to the given `key.string`. If there is no binding, the function returns -1.

An **action** value of -3 returns a string that contains all defined key bindings. The value of `key.string` is ignored. Programs should make no assumption about the format of this string as it may change between releases of QM.

An **action** value of -4 restores the bindings define in a `key.string` that was returned by a previous call to **BINDKEY()** with an action of -3. This action also restores the state of lone Esc key handling to its setting at the time when the key bindings were saved.

Actions -5 and -6 control whether the **KEYCODE()** function returns char(27) on detection of an incoming Escape character that is not closely followed by further characters. When this mode is
enabled (which is the default), the Esc is returned by `KEYCODE()` if no further characters are received within 200mS. When disabled, the Esc is always treated as the start of a control sequence and `KEYCODE()` will wait for further characters. These two action codes return True if lone Esc key handling was previously enabled, or False if it was previously disabled.

Because retrieval of a key binding returns -1 if the key is not bound, it is easy to save and restore a single key binding:

```
OLD.BINDING = BINDKEY(KEY.STRING, -2)
IF BINDKEY(KEY.STRING, NEW.ACTION) THEN ... 
```

To restore the original binding, unbinding the key if there was no previous binding:

```
X = BINDKEY(KEY.STRING, OLD.BINDING)
```

To save and subsequently restore all bindings:

```
SAVED.KEYS = BINDKEY('', -3)
...rebind some keys and do some processing...
X = BINDKEY(SAVED.KEYS, -4)
```

Note that key bindings set using `BINDKEY()` are lost on return to the command processor. This ensures that private bindings do not interfere with programs run later in the process. Persistent bindings can be set by modifying the `terminfo` database.

See also: `INPUT@`, `INPUTFIELD`, `KEYCODE()`, `KEYCODEV()`, `KEYEDIT`, `KEYEXIT`, `KEYTRAP`
BITAND()

The BITAND() function forms the bitwise logical AND of two integer values.

Format

    BITAND(expr1, expr2)

where

    expr1 and expr2 evaluate to integers

The BITAND() function converts expr1 and expr2 to 32 bit integers and performs a bit-by-bit logical AND to form a new integer value as the result.

The value of each bit in the result is 1 if same bit position in both of expr1 and expr2 is 1.

Example

    IF BITAND(N, 1) THEN N += 1

This statement adds one to N if the least significant bit is 1. The effect is to round N to an even integer.

See also:
BITNOT(), BITOR(), BITRESET(), BITSET(), BITTEST(), BITXOR(), SHIFT()
BITNOT()

The BITNOT() function forms the bitwise logical NOT of an integer value.

**Format**

```
BITNOT(expr)
```

where

```
expr    evaluates to an integer.
```

The BITNOT() function converts `expr` to a 32 bit integer and forms a result value by inverting each bit.

**Example**

```
N = BITNOT(A)
```

This statement sets N to the bitwise logical inverse of A.

**See also:**

BITAND(), BITOR(), BITRESET(), BITSET(), BITTEST(), BITXOR(), SHIFT()
The **BITOR()** function forms the bitwise logical OR of two integer values.

**Format**

```
BITOR(expr1, expr2)
```

where

- `expr1` and `expr2` evaluate to integers

The **BITOR()** function converts `expr1` and `expr2` to 32 bit integers and performs a bit-by-bit logical OR to form a new integer value as the result.

The value of each bit in the result is 1 if same bit position in one or both of `expr1` and `expr2` is 1.

**Example**

```
FLAGS = BITOR(FLAGS, 8)
```

This statement sets the bit with integer value 8 in the `FLAGS` variable.

**See also:**
- **BITAND()**, **BITNOT()**, **BITRESET()**, **BITSET()**, **BITTEST()**, **BITXOR()**, **SHIFT()**
BITRESET()

The BITRESET() function turns off a specified bit in an integer value.

Format

BITRESET(expr, bit)

where

expr evaluates to the value in which the bit is to be reset.

bit evaluates to the bit position (0 to 31).

The BITRESET() function converts expr to a 32 bit integer and turns off (sets to 0) the bit identified by bit to form a new integer value as the result. Bits are numbered from 0 to 31 from the least significant end of the value. The effect of this function with a bit value outside this range is undefined.

Example

FLAGS = BITRESET(FLAGS, 2)

This statement turns off bit 2 in the FLAGS variable.

See also: BITAND(), BITNOT(), BITOR(), BITSET(), BITTEST(), BITXOR(), SHIFT()
BITSET()

The BITSET() function turns on a specified bit in an integer value.

Format

BITSET(expr, bit)

where

expr evaluates to the value in which the bit is to be set.

bit evaluates to the bit position (0 to 31).

The BITSET() function converts expr to a 32 bit integer and turns on (sets to 1) the bit identified by bit to form a new integer value as the result. Bits are numbered from 0 to 31 from the least significant end of the value. The effect of this function with a bit value outside this range is undefined.

Example

FLAGS = BITSET(FLAGS, 2)

This statement turns on bit 2 in the FLAGS variable.

See also:
BITAND(), BITNOT(), BITOR(), BITRESET(), BITTEST(), BITXOR(), SHIFT()
BITTEST()

The BITTEST() function tests the state of a specified bit in an integer value.

Format

BITTEST(expr, bit)

where

expr evaluates to the value in which the bit is to be tested.
bit evaluates to the bit position (0 to 31).

The BITTEST() function converts expr to a 32 bit integer and tests the state of the bit identified by bit, returning True if it is set and False if it is reset. Bits are numbered from 0 to 31 from the least significant end of the value. The effect of this function with a bit value outside this range is undefined.

Example

IF BITTEST(FLAGS, 2) THEN DISPLAY(IT$CS) :

This statement clears the screen if bit 2 is set in the FLAGS variable.

See also:
BITAND(), BITNOT(), BITOR(), BITRESET(), BITSET(), BITXOR(), SHIFT()
BITXOR()

The BITXOR() function forms the bitwise logical exclusive-OR of two integer values.

Format

    BITXOR(expr1, expr2)

where

    expr1 and expr2 evaluate to integers

The BITXOR() function converts expr1 and expr2 to 32 bit integers and performs a bit-by-bit logical exclusive-OR to form a new integer value as the result.

The value of each bit in the result is 1 if same bit position in one and only one of expr1 and expr2 is 1.

Example

    FLAGS = BITXOR(FLAGS, 8)

This statement inverts the bit with integer value 8 in the FLAGS variable.

See also:
BITAND(), BITNOT(), BITOR(), BITRESET(), BITSET(), BITTEST(), SHIFT()
**BOOL()**

The **BOOL()** function converts a value to Boolean form.

**Format**

\[
\text{BOOL}(expr)
\]

where

\[
expr \quad \text{is the value to be converted. This must be a numeric value or a string.}
\]

The **BOOL()** function returns False if `expr` is zero or a null string. True for all other values.

This function is rarely needed because all QMBasic operations requiring a Boolean value perform the conversion automatically. It might be useful, for example, when constructing a data collection that will be converted to JSON using the **JBUILD()** function as JSON differentiates Boolean and numeric values.

**Example**

\[
N = \text{BOOL}(X)
\]

This statement converts X to Boolean form using the rules described above.

**See also:**

**INT(), NUMERIC(), STR()**
BREAK

The BREAK statement allows the action of the break key to be disabled during program execution.

Format

\[
\begin{align*}
\text{BREAK (KEY) OFF} \\
\text{BREAK (KEY) ON} \\
\text{BREAK (KEY) CLEAR} \\
\text{BREAK (KEY) RESET} \\
\text{BREAK (KEY) expr}
\end{align*}
\]

where

\[
\text{expr} \quad \text{evaluates to a number.}
\]

QM maintains a break inhibit counter which is set to zero before the command prompt is first displayed. This counter is incremented by the BREAK OFF statement and decremented by BREAK ON though it cannot become negative. Use of the break key whilst the counter is non-zero will not cause a break action to occur. Instead, the break is remembered and will be handled when the counter returns to zero. Multiple use of the break key will not result in more than one break event being handled.

The BREAK RESET statement sets the break inhibit counter to zero.

The BREAK CLEAR statement cancels any deferred break event.

The BREAK expr format of this statement is equivalent to BREAK OFF if the value of expr is zero, BREAK ON if expr is positive and BREAK CLEAR if expr is negative.

Example

\[
\begin{align*}
\text{BREAK OFF} \\
\text{GOSUB UPDATE.FILES} \\
\text{BREAK ON}
\end{align*}
\]

This program fragment inhibits use of the break key while the internal subroutine UPDATE.FILES is executed.

See also:
BREAK command, BREAK.COUNT(), SET.BREAK.HANDLER, REMOVE.BREAK.HANDLER
BREAK.COUNT()

The BREAK.COUNT() function returns the current value of the break inhibit counter.

Format

BREAK.COUNT()

The BREAK.COUNT() function returns the value of the break inhibit counter. This counter is incremented by the BREAK OFF statement and decremented (never below zero) by the BREAK ON statement.

Example

```
LOOP
  WHILE BREAK.COUNT()
    BREAK.ON
  REPEAT
```

This program fragment unwinds the break counter to ensure that breaks are enabled. Use of BREAK RESET would be better.
**BSCAN**

The **BSCAN** statement provides a method to scan an alternate key index. It is provided for compatibility with another multivalue product and is best replaced by use of **SETLEFT**, **SETRIGHT**, **SELECTLEFT** and **SELECTRIGHT**.

**Format**

```plaintext
BSCAN id {, data} FROM fvar {, value} USING index {RESET} {BY seq}
{THEN statement(s)}
{ELSE statement(s)}
```

where

- **id** is the variable to receive the id of the record found.
- **data** is the variable to receive the record data.
- **fvar** is the file variable for the data file.
- **value** is the indexed value to be located. If omitted, the BSCAN statement moves one indexed value in the direction specified in the BY clause.
- **index** is the name of the index to be scanned.
- **RESET** causes BSCAN to position at the extreme left or right of the index depending on the BY clause.
- **seq** is "A" (ascending) to scan left to right or "D" (descending) to scan right to left. If omitted, ascending, ascending is used by default.

At least one of the **THEN** and **ELSE** clauses must be present. The **THEN** clause will be executed if the action is successful. The **ELSE** clause will be executed at an error and the value of the **STATUS()** function will provide further information regarding the cause of the failure.

- **0** The scan has reached the end of the index. **id** and **data** are returned as null strings.
- **1** An index entry was found. The indexed value will be in **id** and **data** will be a list of data file record ids for that indexed value.
- **2** No index entry with the given **value** was found.
- **3** Inappropriate file type.
- **4** Index does not exist.
- **5** **seq** is not A or D.
- **10** Some other unspecified error has occurred.

If **value** is specified, the BSCAN operation positions at that value in the index or, if there is no index entry for the value, at the position where it would have been. The nearest index entry in the scan order is returned.
CALL, ENTER

The CALL statement calls a catalogued subroutine. The ENTER statement is a synonym for CALL unless the PICK.ENTER option of the $MODE directive is used.

Format

CALL name {(arg.list)}
CALL @var {(arg.list)}
CALL "file name" {(arg.list)}

where

name is the name of the subroutine to be called.
@var is the name of a variable holding the name of the subroutine to be called.
arg.list is the list of arguments to the subroutine.
file is the name of the file containing the subroutine to be called.

A subroutine with no arguments is equivalent to a program. A whole matrix can be passed as an argument by prefixing it with MAT.

Direct Calls

Placing the subroutine name in the CALL statement as in the first syntax shown above is referred to as a direct call. QM will search for the subroutine as described below when any CALL statement referencing the subroutine is first executed in the program or subroutine. For CALL statements which occur within catalogued subroutines the search will take place every time the calling subroutine itself is called. QM includes an object code caching mechanism to minimise the performance impact of this repeated search.

Indirect Calls

The second syntax executes a CALL statement using a variable to hold the subroutine name and is referred to as an indirect call. In this case, QM will search for the subroutine as described below when the first CALL statement is executed. Indirect calls allow an application to call a subroutine where the name of the routine was not known at compile time. This might be of use, for example, in menu systems.

When an indirect call is executed, the variable containing the subroutine name is modified to become a subroutine reference. This can still be used as a string in the program but also contains a pointer to the memory resident copy of the subroutine. The subroutine will remain in memory so long as one or more subroutine references point to it. Overwriting the variable will destroy the subroutine link and may make the subroutine a candidate for removal from the object code cache.

One advantage of indirect calls is that, by placing the variable in a common block where it is accessible by all modules of the application and will not be discarded, the catalogue search need only be performed once even when the CALL is in a subroutine which itself may be called many times. A direct call works in a similar way but the variable in which the subroutine reference is placed is local to the program containing the CALL and is thus lost when the program terminates.
If the string stored in the variable used in an indirect call contains a space, this is taken as being a Pick style subroutine reference described below.

**Searching for the Subroutine**

Subroutines to be executed using `CALL` must be placed in the catalogue using the `CATALOGUE` command or the equivalent automated cataloguing from within the QMBasic compiler.

Subroutine names must conform to the QMBasic name formats except that two special prefix characters are allow. An exclamation mark prefix character is used on all standard globally catalogued subroutines provided as part of QM that are intended for user use. An asterisk prefix may be used on user written globally catalogued subroutines for compatibility with other products.

Unless the subroutine name commences with one of the global catalogue prefix characters, QM goes through a series of steps when a `CALL` statement searches for a subroutine:

- The local catalogue is checked. This consists of a VOC record of the form:
  
  Field 1 V
  Field 2 CS
  Field 3 Runfile pathname

- The private catalogue file is checked.

- The global catalogue is checked.

Note that subsequent calls to the same subroutine where the subroutine reference has not been reset will continue to use the original catalogued routine even if it has been deleted from the catalogue or replaced.

**Pick Style Call with a File Name**

The third syntax of `CALL`,

```plaintext
CALL "file name" {(arg.list)}
```

allows a program to specify subroutine to be called by its file and program name. The `file` element of this syntax is the case sensitive name of the file containing the compiled program. A suffix of `.OUT` is added to this name automatically. Extended file name syntaxes are supported where permitted by the value of the `FILERULE` configuration parameter. The `name` element is the name of the program within the specified file. On platforms with case sensitive file systems, the casing of this name must match that of the stored program. The program is loaded directly from the specified location instead of following the catalogue search process described above.

**Arguments**

The argument list may contain up to 255 items. If a subroutine has no arguments, the brackets may be omitted.

Each argument is

- A constant: `CALL SUB("MY.FILE")`
- An expression: `CALL SUB(X + 7)`
- A variable name: `CALL SUB(X)`
- An indexed matrix element name: `CALL SUB(A(5, 2))`
- A matrix name prefixed by MAT: `CALL SUB(MAT A)`
Where the argument is a reference to a variable, a matrix element or a whole matrix, the subroutine may update the values in these variables. Except when passing a whole matrix, the calling program can effectively prevent this by forcing the argument to be passed by value rather than by reference by enclosing it in brackets, thus making the argument into an expression.

**Forced Reload of Object Code**

Whenever a program is compiled or catalogued, an event is signalled to all QM processes to cause them to unload any inactive programs from the object code cache. In this context, an inactive program is one that is not the currently executing program, an earlier one in the hierarchy of subroutine calls, or one that has a subroutine reference variable pointing to it.

If the FORCE.RELOAD mode of the **OPTION** command is in effect, this mechanism is extended such that all subroutine link variables are reset. The next call to the subroutine will reload the object code. This allows dynamic modification of an application in a production environment but carries the risk that a simple loop such as

```fortran
FOR I = 1 TO 5
  CALL MYSUB
NEXT I
```

might call a different version of MYSUB from one iteration of the loop to the next.

Note that the event causes all subroutine links to be reset, not just those for the program that has been compiled or catalogued, however, it does not apply to instantiated objects. Note also that the FORCE.RELOAD option must be set in the QM session that is to perform the reload, not necessarily in the session that compiles or catalogues the program.

**Pick Style ENTER**

By default, the **ENTER** statement is a synonym for **CALL**. Use of the PICK.ENTER option of the **SMODE** compiler directive causes **ENTER** to behave in the same way as its equivalent in Pick style multivalue database products.

In this mode, use of **ENTER** terminates the current program and replaces it with the named program. This new program may not take arguments. If the **ENTER** statement is performed in a program that was started using the **EXECUTE** statement, or a subroutine called from such a program, the **ENTER** does not discard the program containing the **EXECUTE**.

**Examples**

```fortran
COMMON /COM1/ INITIALISED, SUB1
IF NOT(INITIALISED) THEN
  SUB1 = "SUBR1"
  INITIALISED = @TRUE
END
```

This program fragment declares a common block to hold subroutine call references. When the program is first executed, the conditional statements will be performed as common block variables are initially zero. This path sets the name of the subroutine SUBR1 into common variable SUB1.

Later in the program, perhaps in a different subroutine from that in which the common was initialised, a statement of the form

```fortran
CALL @SUB1(ARG1, ARG2)
```
will call the SUBR1, changing the common variable to be a subroutine reference for fast access on subsequent calls.

A statement of the form

    CALL SUBR1(ARG1, ARG2)

would call the same subroutine but does not use the common block variable. If this call was in a subroutine, the catalogue search would be performed for the first call each time the calling subroutine is entered.

See also:

CATALOGUE
The CASING statement enables or disables case sensitivity for string comparisons.

**Format**

```
CASING OFF
CASING ON
CASING expr
```

where

```
expr evaluates to a Boolean value.
```

By default, string comparisons in QMBasic programs are case sensitive. Case sensitivity can be selected by use of the NOCASE.STRINGS setting of the `$MODE` compiler directive.

The CASING statement allows a program to dynamically enable or disable case sensitivity at runtime. Use of the ON qualifier or an expr that evaluates to True enables case sensitivity. Use of the OFF qualifier or an expr that evaluates to False disables case sensitivity. Note that unlike the equivalent Basic operation in some other multivalue products, the setting only affects the program in which the CASING statement is used and is not inherited by subroutines called from that program.

**Example**

```
CASING ON
IF 'ABC' = 'abc' THEN DISPLAY 'Strings match' ELSE DISPLAY 'No match'
CASING OFF
IF 'ABC' = 'abc' THEN DISPLAY 'Strings match' ELSE DISPLAY 'No match'
```

This program fragment displays "No match" for the first test and "Strings match" for the second.
CATALOGUED()

The CATALOGUED() function determines whether a subroutine can be found using the search process described for the CALL statement.

Format

   CATALOGUED(name)

where

   name      is the calling name of the program

The CATALOGUED() function returns

0   the subroutine is not catalogued
1   the subroutine is catalogued locally using a V-type VOC entry
2   the subroutine is catalogued privately
3   the subroutine is catalogued globally

Where a subroutine name appears in more than one catalogue form, the search order is as in the list above and the value returned reflects the first entry found.

The return value from the CATALOGUED() function can be treated as a Boolean (True/False) value if the application merely wants to determine if a subroutine is catalogued and does not need to know in which format.

Example

   IF NOT(CATALOGUED('MYPROG')) THEN DISPLAY 'Not catalogued'

This statement displays a message if MYPROG is not in the system catalogue.

See also:
CATALOGUE, CALL
CATS()

The CATS() function concatenates corresponding elements of a dynamic array.

Format

\[
\text{CATS}(\text{string1}, \text{string2})
\]

where

- \text{string1} is the string to which \text{string2} is to be concatenated.
- \text{string2} is the concatenation string.

The CATS() function returns the result of concatenating corresponding dynamic array components (fields, values and subvalues) from the supplied strings.

The REUSE() function can be applied to either or both expressions. Without this function, any absent trailing values are taken as null strings.

Examples

\[
\begin{align*}
S1 &= "ABC":@fm:"DEF" \\
S2 &= "123":@vm:"456":@fm:"789" \\
X &= \text{CATS}(S1, S2)
\end{align*}
\]

Note that this example has both field and value marks. The above code fragment concatenates elements of the two strings yielding a result in X of "ABC123\text{VM}456\text{FM}DEF789"

\[
\begin{align*}
S1 &= "ABC":@fm:"DEF" \\
S2 &= "123":@fm:"456":@fm:"789" \\
X &= \text{CATS}(\text{REUSE}(S1), S2)
\end{align*}
\]

This example concatenates elements of the two strings yielding a result in X of "ABC123\text{FM}DEF456\text{FM}DEF789"

\[
\begin{align*}
S1 &= "ABC":@fm:"DEF" \\
S2 &= "123":@fm:"456":@fm:"789" \\
X &= \text{CATS}(\text{REUSE}(S1), S2)
\end{align*}
\]

Adding the REUSE() function to the previous example gives a result in X of "ABC123\text{FM}DEF456\text{FM}DEF789"
CAUGHT()

The CAUGHT() function allows an application to test whether there is a handler established for a named exception.

Format

   CAUGHT(exception, modes)

where

   exception     evaluates to an exception name.
   modes         identifies which special exception handlers are to be included.

An exception is a named event, typically an error, raised within an application by use of the THROW statement. The CAUGHT() function returns True or False to indicate whether there is an active exception handler for a given exception name. Exception names are case insensitive and may be up to 32 characters.

The modes argument determines whether special exception handlers are included in the scan. This is an additive bit significant value formed from

   1 Include the SYSSANY handler
   2 Include the SYSSUNHANDLED handler

A mode value of zero tests strictly for the named handler.

See also:
Exception handling, THROW, TRY/CATCH
CEIL(), FLOOR()

The CEIL() function returns the smallest integer not less than the argument value. The FLOOR() function returns the largest integer not greater than the argument value.

Format

\[
\text{CEIL}(\text{value}) \\
\text{FLOOR}(\text{value})
\]

where

\text{value} \quad \text{is a numeric value to be rounded.}

If value is a numeric array, a similarly structured array of result values is returned.

Examples

<table>
<thead>
<tr>
<th>Value</th>
<th>CEIL(value)</th>
<th>FLOOR(value)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>3.1</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3.7</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>-3</td>
<td>-3</td>
<td>-3</td>
</tr>
<tr>
<td>-3.1</td>
<td>-3</td>
<td>-4</td>
</tr>
<tr>
<td>-3.7</td>
<td>-3</td>
<td>-4</td>
</tr>
<tr>
<td>3.1\text{VM}-3.7</td>
<td>4\text{VM}-3</td>
<td>3\text{VM}-4</td>
</tr>
</tbody>
</table>

See also:

ROUNDDOWN(), ROUNDUP()
**CHAIN**

The **CHAIN** statement terminates the current program and executes a command.

**Format**

```plaintext
CHAIN expr
```

where

*expr* evaluates to a command.

The current program terminates immediately, discarding local variables but retaining named common variables. The command defined by *expr* is executed as though it replaced the sentence which invoked the program in which the **CHAIN** statement occurs. If this sentence is in a paragraph, the remainder of the paragraph will be executed when the **CHAIN**ed program terminates.

The exact behaviour of **CHAIN** when executed from a program started by a PROC depends on the VOC record type of the target item referenced by *expr*. If this is a further PROC, any stacked PROCs leading up to the program that performed the **CHAIN** are discarded. If the target is not a PROC, control will return to the PROC that executed the program performing the **CHAIN** when the chained command terminates.

The unnamed common block is discarded on execution of a **CHAIN** unless the **CHAIN.KEEP.COMMON** mode of the **OPTION** command is active.

**CHAIN** is provided primarily for compatibility with other systems. The same effect can usually be better achieved using **EXECUTE**.

**Examples**

```plaintext
CHAIN "RUN PROGRAM2"
```

This program fragment terminates the current program and executes PROGRAM2 in its place.
The `CHANGE()` function replaces occurrences of a substring within a string by another substring. The synonym `SWAP()` can be used.

**Format**

```
CHANGE(string, old, new[, occurrence[, start]]
```

where

- `string` is the string in which the replacement is to occur.
- `old` evaluates to the substring to be replaced.
- `new` evaluates to the substring with which `old` is to be replaced.
- `occurrence` evaluates to the number of occurrences of `old` to be replaced. If omitted or specified as a value of less than one, all occurrences are replaced.
- `start` specifies the first occurrence to be replaced. If omitted or specified as a value of less than one it defaults to one.

The `CHANGE()` function replaces the specified occurrences of `old` within `string` by `new`.

If `old` is a null string, the function returns `string` unchanged. The `CHANGE()` function is identical to `EREPLACE()` if `old` is not a null string.

If `new` is a null string, all occurrences of `old` are removed.

If the `SMODE NOCASE.STRINGS` compiler directive is used, matching of `old` against `string` is case insensitive.

The default behaviour of the `CHANGE()` and `SWAP()` functions is that overlapping substrings are allowed when searching `string` for the desired occurrence of `old`. Thus

```
X = CHANGE("aaaaaaaaaa", "aa", "xx", 0, 3)
```

yields a result of "aaxxxxxxa"

The `CHANGE.NO.OVERLAP` setting of the `SMODE` compiler directive can be used to select an alternative behaviour in which substrings do not overlap. The same example would then yield a result of "aaaaxxxa".

**Examples**

```
PRINT CHANGE("ABRACADABRA", "A", "a", 3, 2)
```

This statement results in printing the string "ABRaCaDaBRA".

**See also:**

`CONVERT()`, `EREPLACE()`
CHAR()

The **CHAR()** function returns the character with a given ASCII value.

**Format**

```
CHAR(seq)
```

where

`seq` evaluates to an integer in the range 0 to 255.

The **CHAR()** function returns a single character string containing the ASCII character with value `seq`. It is the inverse of the **SEQ()** function.

Only the least significant 8 bits of the integer value of `seq` are used. Values outside the range 0 to 255 may behave differently on other systems and should not be relied on.

The **CHARS()** function takes a dynamic array `seq` and returns a similarly structured dynamic array in which each element contains the character with the ASCII value from the corresponding element of `seq`.

**Example**

```
DISPLAY CHAR(7):
```

This statement outputs character 7 of the ASCII character set to the display. Character 7 is the BELL character and causes the audible warning to sound. This is similar to use of the **@SYS.BELL** variable except that CHAR(7) is not affected by use of the **BELL OFF** verb and will always work.

**See also:**

**ECHAR()**, **SEQ()**
CHECKSUM()

The CHECKSUM() function returns a checksum value for supplied data.

Format

CHECKSUM(data {, mode})

where

data is the character string for which a checksum is to be calculated.

mode specifies the algorithm to be used. Defaults to 0 if omitted.

The CHECKSUM() function calculates a cyclic redundancy checksum (CRC) value for the supplied data item. Note that the various multivalue products implement this function using different algorithms, not all of which are available in QM. The default algorithm used by QM is a simple rotate and merge but several alternative algorithms used by other software packages are available by use of the optional mode argument.

The checksum value produced by an ECS mode system for data that contains only characters from the 8-bit set is identical to that produced for the same data on a non-ECS mode system. The CRC algorithms are usually based on byte structured data. Applications using ECS mode may sometimes need to use the BSL or BSH conversion code to form a hardware byte order specific representation of the data prior to use of the CHECKSUM() function.

If data is a null string or mode is invalid, the CHECKSUM() function returns zero.

Algorithms

0  CRC$QM  QM default rotate and merge algorithm
1  CRC$XMODCCITT XModem (as used by some other multivalue products, e.g. UniVerse) EM
2  CRC$CCITTCCITT with 0xFFFF initialiser .FALSE
3  CRC$CCITTCCITT with 0x1D0F initialiser .AUG
4  CRC$KERMCCITT Kermit IT
5  CRC$16  CRC-16
6  CRC$MODBCCITT Modbus US
7  CRC$BYTE  Simple byte addition
8  CRC$DNP  CRC-DNP (IEEE 1815)
9  CRC$32  CRC-32
10 CRC$16.SIC CRC-16 "Sick" variant K
11 CRC$PARIT Longitudinal byte parity (Bitwise exclusive OR of all bytes) Y

Examples
DISPLAY CHECKSUM(REC)

This statement displays a checksum value for the data in the REC variable using the default algorithm.

DISPLAY CHECKSUM(REC, CRC$CMODEM)

This statement displays a checksum value for the data in the REC variable using the CCITT-XModem algorithm.

See also:
MD5()
CHILD()

The CHILD() function tests whether a phantom process started from the current session is still running.

Format

CHILD(userno)

where

userno is the user number of the phantom process to be examined.

The CHILD() function allows a process that started a phantom process to determine whether the phantom is still running. The userno argument is the user number of the phantom process as returned by the PHANTOM command via the @SYSTEM.RETURN.CODE variable.

The CHILD() function returns True if there is a phantom process running with the given user number that was started by the process in which the CHILD() function is executed. In all other cases, the function returns False.

The CHILD() function cannot be used to test the state of phantoms started by other sessions.

Example

EXECUTE "PHANTOM RUN END.OF.MONTH"
UID = @SYSTEM.RETURN.CODE
...further processing...
IF CHILD(UID) THEN DISPLAY "End of month process is still active"

This program fragment starts a phantom to run the END.OF.MONTH program, does some further processing, and then tests whether the phantom is still active.

See also:
LIST.PHANTOMS, PHANTOM, !PHLOG(), STATUS
CLASS

The CLASS statement declares a class module.

Format

`CLASS name [MAX.ARGS limit] [INHERITS class.list]`

where

- `name` is the name of the module.
- `limit` is the maximum number of arguments allowed in public function and subroutines within the class module. This defaults to 32 and cannot exceed 255.
- `class.list` is a comma separated list of the catalogue names of classes to be inherited by this class. These should not be enclosed in quotes.

QMBasic programs should commence with a PROGRAM, SUBROUTINE, FUNCTION or CLASS statement. If none of these is present, the compiler behaves as though a PROGRAM statement had been used with `name` as the name of the source record.

The CLASS statement must appear before any executable statements. For more details, see Object Oriented Programming.

The `name` need not be related to the name of the source record though it is recommended that they are the same as this eases program maintenance. The name must comply with the QMBasic name format rules.

A class module contains the components of a QMBasic object. The general structure of this is

```
CLASS name
  PUBLIC A, B(3), C
  PRIVATE X, Y, Z

  PUBLIC SUBROUTINE SUB1(ARG1, ARG2) {VAR.ARGS}
     ...processing...
  END

  PUBLIC FUNCTION FUNC1(ARG1, ARG2) {VAR.ARGS}
     ...processing...
     RETURN RESULT
  END

...Other QMBasic subroutines...
END
```

The MAX.ARGS option can be used to increase the default limit on the number of arguments permitted in a public function or subroutine within the class module. This has a small effect on performance and should only be used where the default value of 32 needs to be exceeded.

The INHERITS option causes the named class or classes to be inherited automatically by this class when it is instantiated. Each name in the list can optionally be followed by initialisation values that
will be passed to the CREATE.OBJECT subroutine of the inherited class. These values must be numeric or string constants. For example,

```
CLASS MYCLASS INHERITS OTHERCLASS("RATE", 25)
```

which is equivalent to use of

```
OTHERCLASS = OBJECT("OTHERCLASS", "RATE", 25)
INHERIT OTHERCLASS
```

If the inherited class is globally catalogued with a prefix character, the prefix is dropped when forming the object variable name. For example,

```
CLASS MYCLASS INHERITS *OTHERCLASS
```

creates an object variable named OTHERCLASS that references the inherited *OTHERCLASS object.

Alternatively, the name of the object variable can be set explicitly using the AS qualifier. For example,

```
CLASS MYCLASS INHERITS *OTHERCLASS AS MYOTHERCLASS(1)
```

Note that the initialisation values, if present, appear after the object variable name, not the catalogue name.

The second form shown above with explicit use of the OBJECT() function and the INHERIT statement allows the initialisation values to be expressions.

See also:
Object oriented programming, DISINHERIT, INHERIT, OBJECT(), OBJINFO(), PRIVATE, PUBLIC.
CLEAR

The CLEAR statement sets all local variables to zero.

Format

  CLEAR

All local variables, including all elements of matrices, are set to zero. Files associated with local file variables will be closed. The value of variables in common areas are not affected.

Subroutine arguments that are passed by reference will set the corresponding variable in the calling program to zero.

See also:
CLEARCOMMON
CLEARCOMMON

The CLEARCOMMON statement sets all variables in the unnamed common area to zero. Used with a common block name, it clears the named common.

Format

CLEARCOMMON \{name\}

CLEAR COMMON \{name\}

If \textit{name} is omitted, all variables in the unnamed common area are set to zero. Other variables are not affected.

If \textit{name} is present, it must be the name of a common block defined in the program. All variables in this common block will be cleared.

Examples

COMMON A, B, C
CLEARCOMMON

All variables in the unnamed common area (A, B and C) are set to zero.

COMMON /MYDATA/P, Q, R
CLEARCOMMON MYDATA

All variables in the MYDATA common block (P, Q and R) are set to zero.

See also:

CLEAR, COMMON
**CLEARDATA**

The **CLEARDATA** statement clears any data stored by previous **DATA** statements or **DATA** verbs and not yet processed by **INPUT** statements.

**Format**

```
CLEARDATA
```

The data queue is cleared. Any keyboard type-ahead is not affected by this statement. The **CLEARDATA** statement is most useful when recovering from error situations where the data queue could cause problems.

Stored data is cleared automatically on return to command prompt.

**Example**

```
ERROR.LABEL:
    CLEARDATA
    ABORT "A fatal error has occurred"
```

This program fragment could be used to ensure that the data queue is empty when aborting at some error condition.

**See also:**

**CLEAR.DATA** command
CLEARFILE

The **CLEARFILE** statement clears a file previously opened using the **OPEN** statement, deleting all records.

**Format**

```
CLEARFILE file.var { ON ERROR statement(s) }
```

where

- `file.var` is a file variable for an open file.

The file associated with the file variable will be cleared, deleting all records. In the case of a dynamic file, all overflow space is released and, unless the file has the **NO.RESIZE** mode enabled, the file returns to its minimum modulus value.

The **ON ERROR** clause is executed if the file cannot be cleared for any reason. The **STATUS()** function may be used to find the cause of such an error.

Note that the **CLEARFILE** statement executes the clear file trigger function, not the delete trigger function if one is defined.

This statement may not be used inside a transaction.

**Example**

```
OPEN "STOCK.FILE" TO STOCK THEN
  CLEARFILE STOCK
  CLOSE STOCK
END
ELSE ABORT "Cannot open file"
```

This program fragment opens a file, clears it and then closes the file.
CLEARINPUT

The CLEARINPUT statement clears any keyboard data that has been entered but not yet processed by INPUT or KEYIN() statements.

The synonym INPUTCLEAR may be used in place of CLEARINPUT.

Format

    CLEARINPUT
    CLEAR INPUT

Any type-ahead data is cleared. Data stored by the DATA statement is not affected.

Example

    ERROR.LABEL:   CLEARINPUT   ABORT "A fatal error has occurred"

This program fragment could be used to ensure that data entered at the keyboard is cleared when aborting at some error condition.
**CLEARSELECT**

The **CLEARSELECT** statement clears one or all select lists.

**Format**

```
CLEARSELECT {list.no}
CLEARSELECT ALL
CLEARSELECT var
```

where

- `list.no` evaluates to the number of the select list to be cleared. If omitted, select list zero is cleared.
- `var` is a Pick style select list variable.

Select lists are numbered from 0 to 10. Where no list number is specified, the default is to use select list 0.

The **CLEARSELECT** statement clears the specified numbered select list if it was active. Use of the **ALL** keyword causes all select lists to be cleared.

**Example**

```
CLEARSELECT 2
```

This statement clears select list number 2.
CLOSE

The CLOSE statement closes a file previously opened using the OPEN or OPENPATH statement.

Format

CLOSE file.var [ON ERROR statement(s)]

where

    file.var     is a file variable for an open file.

The file associated with the given file variable will be closed and file.var converted to integer zero. Any other file variable which refers to the same file, either from a separate OPEN or from copying the file variable, will be unaffected.

The ON ERROR clause is provided for source program compatibility with other systems and will never be executed by QMBasic programs.

Files do not always need to be closed explicitly. Local variables are released when a program or subroutine returns and files associated with local file variables are closed automatically. File variables in common areas will not be affected.

Closing a file inside a transaction destroys the file variable but defers the actual close until the transaction ends. Rolling back the transaction will not reinstate the file variable.

Example

OPEN "STOCK.FILE" TO STOCK ELSE ABORT "Cannot open file"
...further statements...
CLOSE STOCK

This program fragment opens a file, processes it and then closes the file.
CLOSE.SOCKE

The **CLOSE.SOCKE** statement closes a socket.

**Format**

```
CLOSE.SOCKE skt
```

where

- **skt** is the socket variable corresponding to the socket to be closed.

The socket previously opened using **ACCEPT.SOCKE.CONNECTION()**, **CREATE.SERVER.SOCKE()** or **OPEN.SOCKE()** is closed. The **skt** variable is set to unassigned.

**See also:**

- Using Socket Connections, **ACCEPT.SOCKE.CONNECTION**,
- **CREATE.SERVER.SOCKE()**, **OPEN.SOCKE()**, **READ.SOCKE()**, **SELECT.SOCKE()**, **SERVER.ADDR()**, **SET.SOCKE.MODE()**, **SOCKET.INFO()**, **WRITE.SOCKE()**
CLOSESEQ

The CLOSESEQ statement closes an item previously opened using OPENSEQ.

Format

```
CLOSESEQ file.var [ON ERROR statement(s)]
```

where

- `file.var` is the file variable previously associated with the directory file record or device by use of the OPENSEQ statement.
- `statement(s)` are statement(s) to be executed if the close action fails.

The directory file record or device is closed and `file.var` is converted to integer zero.

The ON ERROR clause is provided for source program compatibility with other systems and will never be executed by QMBasic programs.

The CLOSESEQ and CLOSE statements can be interchanged without adverse effect in QMBasic programs, however, care should be taken to use the correct statement if portability to other systems is required.

Example

```
CLOSESEQ STOCK.LIST
```

This statement closes a directory file record previously opened using OPENSEQ and associating it with STOCK.LIST as the file variable.

See also:
CREATE, NOBUF, OPENSEQ, READBLK, READSEQ, SEEK, WEOFSEQ, WRITEBLK, WRITESEQ, WRITESEQF
COL1()

The COL1() function returns the character position immediately preceding the substring extracted by the last FIELD() or LAST() function.

Format

COL1()

The COL1() function is used after a FIELD() or LAST() function to find the character position of the character immediately preceding the extracted substring.

The value of the COL1() function is maintained on a per-program basis. If an external subroutine is called between extracting the data and use of the COL1() function, the positional data is not lost by any use of the FIELD() or LAST() function in the subroutine.

COL1() returns zero if

- No FIELD() or LAST() function has been executed by this program
- The last field extracted was at the start of the string
- The delimiter to the last FIELD() function was a null string
- The field number of the last FIELD() function was beyond the end of the string

Example

S = "A*BB*CCC*DDDD*EEEEE"
X = FIELD(S, "*", 3, 2)
N = COL1()

This program fragment extracts the string "CCC*DDDD" to variable X. The COL1() function returns 5 and assigns this to N.

See also:
COL2(), FIELD(), LAST()
COL2()

The COL2() function returns the character position immediately following the substring extracted by the last FIELD() or LAST() function.

Format

COL2()

The COL2() function is used after a FIELD() or LAST() function to find the character position of the character immediately following the extracted substring.

The value of the COL2() function is maintained on a per-program basis. If an external subroutine is called between extracting the data and use of the COL2() function, the positional data is not lost by any use of the FIELD() or LAST() function in the subroutine.

COL2() returns zero if

No FIELD() or LAST() function has been executed by this program

The field number of the last FIELD() function was beyond the end of the string

Example

S = "A*BB*CCC*DDDD*EEEEE"
X = FIELD(S, "*", 3, 2)
N = COL2()

This program fragment extracts the string "CCC*DDDD" to variable X. The COL2() function returns 14 and assigns this to N.

See also:
COL1(), FIELD(), LAST()
COLLECTION()

The **COLLECTION()** function creates a data collection variable.

**Format**

```
COLLECTION()
COLLECTION(var)
```

The first form of the **COLLECTION()** function creates an empty data collection variable.

The second form of the **COLLECTION()** function copies an existing data collection to create a new collection variable. Note that simply copying the collection variable with a statement such as

```
NEW.COLLECTION = OLD.COLLECTION
```

creates a second reference to the same data collection, not an independent copy. This is very similar in concept to copying file variables or socket variables.

**Examples**

```
CLI = COLLECTION()
CLI['name'] = CLIENT.NAME
```

This program fragment creates an empty data collection in variable CLI and then adds an element to hold a client name.

```
NEW.CLI = COLLECTION(CLI)
```

This statement copies the CLI collection to a new data collection named NEW.CLI.

**See also:**

[Data collections]
COMMON

The COMMON statement declares variables in a common block.

Format

    COMMON {/name} var1 {,var...}

where

    name           is the name of the common block
    var1, etc      are variable names

The COMMON statement is used to define variables as being in common blocks, that is, memory areas that may be used to pass data between different programs and subroutines.

The variable names may extend over multiple lines by splitting the statement after a comma. For example

    COMMON /VARS/ VAR1, VAR2,              VAR3, VAR4

or after the block name

    COMMON /VARS/
    VAR1, VAR2,       VAR3, VAR4

The same common block could be defined as

    COMMON/VARS/ VAR1
    COMMON/VARS/ VAR2
    COMMON/VARS/ VAR3
    COMMON/VARS/ VAR4

The compiler assumes that definitions of variables with the same common block name are a continuation of previous definitions in the same block.

Common blocks are identified by the name that is used in the COMMON statement. The name of a common block must conform to the same rules as a variable name. Multiple programs in the same process using the same name share the same variables. Named common blocks are created on the first reference to the name and remain in existence until exit from QM. There is also a common block with no name (unnamed common) which may be referred to by a COMMON statement of the form

    COMMON var1, var2

or

    COMMON // var1, var2

The unnamed common block is associated with a single command and is discarded on termination of the command that created it. Use of the EXECUTE statement saves and removes any current unnamed common and restores it on completion of the executed command.

A common block may contain any number of variables but the number of variables may not be increased by later redefinition. It is valid for a program to define fewer variables than in the original
common block declaration. This is useful if a new item has been added at the end of a common block but not all programs have yet been recompiled.

Variables within a common block are referenced internally by position, not by name. Thus it would be possible (though not recommended) for different programs to use different names when accessing the same common block. Normally, the structure of a common block is best defined in an include record so that the same definition is used by all parts of the application.

By default, the variables in a common block are initialised to integer zero when the block is created. It is thus possible to include QMBasic code to perform further initialisation just once by statements of the form

```qmbasic
COMMON /MYCOMMON/ INITIALISED, VAR1, VAR2, VAR3,...etc...
IF NOT(INITIALISED) THEN
   ...do initialisation tasks...
   INITIALISED = @TRUE
END
```

For compatibility with some other systems, the UNASSIGNED.COMMON option of the $MODE compiler directive can be used to specify that common blocks are to be created with their component variables left unassigned instead of being set to zero. This directive only affects the program that actually creates the common block (i.e. the first program executed that references the common). It has no effect if the common has already been created. It is possible for an application to mix assigned and unassigned common by careful placement of the $MODE directive.

Common blocks may contain matrices. These are defined by including the row and column bounds in the COMMON statement, for example

```qmbasic
COMMON /MYCOMMON/ MAT1(5, 3)
```

QM supports two styles of matrix with different characteristics. The $MODE compiler directive can be used to select Pick style matrices.

The default style of matrix includes a zero element and is resizeable. A common block declared as

```qmbasic
COMMON A, B(3), C
```

has three variables, one of which is a matrix:

```
A    B    C
```

```
B(0) B(1) B(2) B(3)
```

A matrix of this type in a common block can be re-dimensioned by a later DIM statement.

Pick style matrices do not have a zero element and cannot be resized. A matrix of this type in a common block is equivalent to a series of simple variables:

```
A    B(1)    B(2)    B(3)    C
```
Although not recommended, three programs could use very different views of the same five element common block when using Pick style matrices:

- COMMON A, B(3), C
- COMMON X(2), Y, Z(2)
- COMMON P, Q, R, S, T

See also:
CLEARCOMMON
COMPARE(), COMPARES()

The COMPARE() function compares two strings using the same rules as the LOCATE statement. The COMPARES() function is similar but operates on a multivalued list of strings.

Format

COMPARE(string1, string2 {, justification})
COMPARES(string1, string2 {, justification})

where

string1, string2 evaluate to the strings to be compared.
justification evaluates to a string where the first character is "L" for left justified comparison or "R" for right justified comparison. If omitted or invalid, left justification is used. An optional second character of "I" causes the function to compare strings in a case insensitive manner.

The COMPARE() function compares the two strings and returns

1 string1 is greater than string2
0 string1 is equal to string2
-1 string1 is less than string2

For a left justified comparison, characters are compared one by one and the function return value is determined by the relative ASCII character set positions of the characters at which the first mismatch occurs. If the strings are of different lengths and match up to the end of the shorter, the longer string is treated as the greater.

For a right justified comparison, the COMPARE() function behaves as though sufficient spaces were inserted at the start of the shorter string to match the length of the longer string. Characters are then compared one by one and the function return value is determined by the relative ASCII character set positions of the characters at which the first mismatch occurs.

The COMPARE() statement always compares the two items as character strings unlike a simple relational operator that will perform a numeric comparison if both items can be treated as numbers:

X = COMPARE("0", "00")

will indicate that the two items are not equal whereas

X = ("0" = "00")

will indicate that the two items are equal.

The COMPARE() function is not affected by the setting of the $MODE NOCASE, STRINGS compiler directive and can therefore be used to force a case sensitive comparison in otherwise case insensitive programs. The optional I code in the justification argument causes case insensitive comparison.

The COMPARES() function treats string1 as a dynamic array, comparing each element in turn with string2 and returning a similarly structured dynamic array of results.
Use of the == operator

IF A == B THEN

is equivalent to

IF COMPARE(A, B) = 0 THEN

Examples

```qbasic
DIM ITEM(100)
ITEMS = 0
LOOP
    INPUT NEW.ITEM
    WHILE LEN(NEW.ITEM)
        I = 1
        LOOP
        WHILE I <= ITEMS
            IF COMPARE(NEW.ITEM, ITEM(I)) < 0 THEN EXIT
            I += 1
        REPEAT
    * Insert item at position I
    FOR J = ITEMS TO I STEP - 1
        ITEM(J + 1) = ITEM(J)
    NEXT J
    ITEM(I) = NEW.ITEM
    ITEMS += 1
    REPEAT
```

This program fragment creates a matrix, ITEMS, and then enters a loop to read NEW.ITEM values from the keyboard until a blank line is entered. Each item read is inserted into the matrix in its correct position to maintain the matrix in ascending collating sequence order. Additional statements to detect and handle matrix overflow would be useful in a full application.

```qbasic
A = 'AAA':@VM:'BBB':@FM:'CCC'
X = COMPARES(A, 'BBB')
```

The above program fragment returns X as "-1\vm0\fm1". Note how the mark characters used to delimit A have been replicated in X.

See also: SORT.COMPARE(), == and ~= operators.
CONFIG()

The CONFIG() function returns the value of a configuration parameter.

Format

    CONFIG(param)

where

    param is the name of the parameter to be retrieved.

The CONFIG() function can be used to retrieve the value of configuration parameters defined in the QM configuration file. The STATUS() function will return zero if successful.

Configuration parameters that may be repeated within the configuration file (e.g. STARTUP) are returned as a dynamic array with each entry as a separate field. This does not apply to parameters that are defined as additive numeric values (e.g. NETFILES) where the composite result is returned.

If param is not recognised as a configuration parameter name, the function returns a null string and the STATUS() function will return ER$NOT.FOUND.

Example

    GS = CONFIG("GRPSIZE")

This statement returns the value of the GRPSIZE configuration parameter, the default group size used when creating a dynamic file.
CONNECT.PORT()

The CONNECT.PORT() function converts a phantom process into an interactive session, using a serial port as its terminal device.

This function is only available on Windows.

Format

CONNECT.PORT(port, baud, parity, bits, stop)

where

- **port** is the name of the serial port to be used (e.g. COM1).
- **baud** evaluates to the data rate (e.g. 9600).
- **parity** specifies the parity setting for the connection (0 = none, 1 = odd, 2 = even).
- **bits** specifies the number of bits per byte.
- **stop** specifies the number of stop bits (1 or 2).

The CONNECT.PORT() function enables an application to start a phantom process that then uses a serial port as though it were a terminal device. The function returns True if successful, False if it fails. The STATUS() function can be used to determine the cause of failure.

Once the connection has been created, the process changes from a phantom to an interactive session and can use the normal QMBasic terminal i/o statements such as INPUT and PRINT to access the port. If the program exits to the command processor, the connection can be used in exactly the same way as if the user had logged in over the serial port. To terminate the session from within a program, execute the QUIT command.

Because this function converts the phantom process into an interactive user, the process consumes a licence. The CONNECT.PORT() function will fail if the user limit has been reached.

Example

IF NOT(CONNECT.PORT('COM1', 9600, 0, 8, 1)) THEN   STOP 'Cannot open COMO1 port'END

This program fragment, used in a phantom process, connects to the device on the COM1 port as the command source, converting the process into an interactive session.
CONTINUE

The CONTINUE statement continues execution of the next cycle of a LOOP/REPEAT or FOR/NEXT structure.

Format

    CONTINUE

The CONTINUE statement is equivalent to a jump to the REPEAT or NEXT statement of the innermost loop structure.

Example

    LOOP
        REMOVE ITEM FROM STOCK SETTING DELIM
        ...processing statements...
        WHILE DELIM
            IF ITEM[1,1] = "A" THEN CONTINUE
            ...further statements...
        ENDW
    REPEAT

This program fragment processes items extracted from the STOCK dynamic array. If the value of ITEM commences with "A", the section marked ...further statements... is omitted.

See also:
EXIT, FOR/NEXT, LOOP/REPEAT
CONVERT

The CONVERT statement and CONVERT() function replace selected characters by others in a string. The CONVERT statement performs this conversion in-situ; the CONVERT() function leaves the source string unchanged and returns the modified value.

Format

CONVERT from.chars TO to.string IN var
CONVERT(from.chars, to.string, source.string) See below for argument sequence

where

from.chars evaluates to a string containing the characters to be replaced.
to.chars evaluates to a string containing the replacement characters.
var is the variable in which the in-situ replacement is to occur.
source.string is the string to be modified.

The statement

S = CONVERT(X, Y, S)

is equivalent to

CONVERT X TO Y IN S

Characters taken from from.chars and to.chars define character translations to be performed. Each occurrence of a character from from.chars in var (or source.string) is replaced by the character in the same position in to.chars. If to.chars is shorter than from.chars, characters for which there is no replacement character are deleted. If to.chars is longer than from.chars the surplus characters are ignored.

If a character appears more than once in from.chars only the first occurrence is used.

If the SMODE NOCASE.STRINGS compiler directive is used, matching of from.chars against var is case insensitive.

Migration note: The default behaviour has the function argument order as in Information style systems which is different the equivalent function in Pick style systems. Use of the PICK.CONVERT setting of the SMODE compiler directive changes the argument order to be

CONVERT(source.string, from.chars, to.chars)

This alternative argument ordering is not available in dictionary I-type expressions.

Examples

S = "ABCDEFGHIJK"
CONVERT "CGAGJ" TO "123" IN S
This program fragment replaces all occurrences of the letter "C" in S by "1", "G" by "2" and "A" by "3". The second occurrence of "G" in the from.chars is ignored. The letter "J" is deleted from S. The result of this operation is to set S to "3B1DEF2HIK".

```
PRINT CONVERT(" ", ":", S)
```

This statement prints the string S with all spaces replaced by # characters.

```
LOOP
  INPUT ISBN,13_; 
  UNTIL CONVERT('0123456789X-', '', ISBN) = ''
  INPUTERR 'Invalid ISBN'
REPEAT
```

The loop above verifies that the data entered by the user contains only digits, the letter X and hyphens. The CONVERT() function is used to return a copy of the input data with all valid characters removed. If the result string is not null, it must contain an invalid character.

See also:

CHANGE()
COS()

The COS() function returns the cosine of a value.

Format

\[
\text{COS}(expr)
\]

where

\[
expr \quad \text{evaluates to a number or a numeric array.}
\]

The COS() function returns the cosine of \( expr \). Angles are measured in degrees.

If \( expr \) is a numeric array (a dynamic array where all elements are numeric), the COS() function operates on each element in turn and returns a numeric array with the same structure as \( expr \).

Example

\[
\text{ADJ} = \text{HYP} \times \text{COS(ANGLE)}
\]

This statement finds the length of the adjacent side of a right angled triangle from the length of the hypotenuse and the angle between these two sides.

See also:
\[
\text{ACOS(), ASIN(), ATAN(), SIN(), TAN()}
\]
COUNT()

The COUNT() function counts occurrences of a substring within a string. The COUNTS() function is similar to COUNT() but operates on successive elements of a dynamic array, returning a similarly structured dynamic array of results.

Format

\[
\text{COUNT}(\text{string}, \text{substring})
\]
\[
\text{COUNTS}(\text{string}, \text{substring})
\]

where

- \text{string} evaluates to the string in which substrings are to be counted.
- \text{substring} evaluates to the substring to count.

The COUNT() function counts occurrences of \text{substring} within \text{string}. Substrings may not overlap, thus

\[
S = "ABABABABABAB"
N = \text{COUNT}(S, "ABA")
\]

sets \(N\) to 3.

If \text{substring} is null, COUNT() returns the length of \text{string}.

Programs compiled with the \$MODE NOCASE.STRINGS compiler directive use case insensitive string comparisons in the COUNT() and COUNTS() functions.

The default behaviour of the COUNT() function is that overlapping substrings are not allowed. Thus

\[
X = \text{COUNT}("aaaaaaaaa", "aa")
\]
yields a result of 4.

The COUNT.OVERLAP setting of the \$MODE compiler directive can be used to select an alternative behaviour in which substrings may overlap. The same example would then yield a result of 8.

Example

\[
\text{MARKS} = \text{COUNT}(\text{REC}, @FM)
\]

This statement counts the field marks in REC.

See also: DCOUNT()
CREATE

The CREATE statement creates an empty directory file record after a previous OPENSEQ has reported that the record did not exist.

Format

```
CREATE file.var
{ON ERROR statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

where

- `file.var` is the file variable associated with the record by a previous OPENSEQ statement.
- `statement(s)` are statement(s) to be executed depending on the outcome of the CREATE.

The ON ERROR clause is taken in the event of a fatal error that would otherwise cause the program to abort.

At least one of the THEN and ELSE clauses must be present. The THEN clause is executed if the operation is successful. The ELSE clause is executed if the CREATE operation fails.

Example

```
OPENSEQ 'AUDIT', DATE() TO SEQ.F THEN
    SEEK SEQ.F, 0, 2 ELSE STOP 'Seek error'
END ELSE
    IF STATUS() THEN ABORT 'Error opening audit record'
    CREATE SEQ.F ELSE ABORT 'Error creating audit record'
END
```

This program fragment attempts to open a sequential file record. If the OPENSEQ fails because the record does not exist, an empty record is created.

See also:
CLOSESEQ, NOBUF, OPENSEQ, READBLK, READCSV, READSEQ, SEEK, WEOFSEQ, WRITEBLK, WRITECSV, WRITESEQ, WRITESEQF
CREATE.FILE

The **CREATE.FILE** statement creates the operating system representation of a directory or dynamic hash file.

**Format**

```
CREATE.FILE  path  {DIRECTORY | DYNAMIC}
{GROUP.SIZE  grpsz}
{BIG.REC.SIZE  bigrec}
{MIN.MODULUS  minmod}
{SPLIT.LOAD  split}
{MERGE.LOAD  merge}
{VERSION  ver}
{ON ERROR  statement(s)}
```

where

- `path` evaluates to the pathname of the file to be created. The **DIRECTORY** or **DYNAMIC** keywords determine the type of file to be created. One and only one of these keywords must be present. The remaining options apply only to creation of a dynamic file and, where included, must appear in the order shown above.
- `grpsz` is the group size (1 - 8) in multiples of 1024 bytes.
- `bigrec` is the large record size in bytes.
- `minmod` is the minimum modulus value. For compatibility with the **CREATE.FILE** command, the synonym **MINIMUM.MODULUS** may be used.
- `split` is the split load percentage.
- `merge` is the merge load percentage.
- `ver` is the file version.

The **CREATE.FILE** statement creates the operating system representation of a directory or dynamic hash file using the configuration information supplied via its optional parameters. Omitted parameters take their system defined default values. Note that this statement does not create a corresponding VOC entry.

**Example**

```
CREATE.FILE 'C:\MYFILE' DYNAMIC GROUP.SIZE 4
```

This statement creates a dynamic hash file with group size 4 (4096 bytes) as the C\MYFILE directory.
CREATE.SERVER.SOCKETH

The CREATE.SERVER.SOCKETH() function creates a server socket on which a program may wait for incoming connections.

**Format**

CREATE.SERVER.SOCKETH(addr, port, flags)

where

- **addr** is the address on which to listen for incoming connections.
- **port** is the port number on which to listen for incoming connections.
- **flags** is a flag value formed from the additive values below. For compatibility with earlier releases, this may be omitted and defaults to a TCP stream connection. The token values are defined in the SYSCOM KEYS.H include record.

Socket type, one of:

- SKTSSTREAM A stream connection (default)
- SKTSDGRM A datagram type connection
- SKTSUSTREAM A named Unix domain socket connection

Protocol, one of:

- SKTSTCP Transmission Control Protocol (default)
- SKT$UDP User Datagram Protocol

The **addr** argument may be

- An IPV4 network IP address. Only incoming connections from other IPV4 addresses will be accepted.
- An IPV6 network IP address. Only incoming connections from other IPV6 addresses will be accepted.
- A server name that will be translated to a network address
- A null string. Connections may be made from either IPV4 or IPV6 addresses.
- For Unix domain sockets, the **addr** argument is the pathname of the socket connection and the **port** argument is ignored.

When the IPADDR configuration parameter is used to cause QM to listen for incoming telnet, QMClient, QMNet or replication connections on a specific IP address, the SYSTEM(1071) function may be useful as the **addr** argument so that the same IP address is used by CREATE.SERVER.SOCKETH.

Use of IPV6 must be enabled using the IPV6 configuration parameter.

If the action is successful, this function returns the socket variable associated with the new server port and the STATUS() function returns zero.

If unsuccessful, the STATUS() function returns an error code that can be used to determine the cause of failure.

Use of CREATE.SERVER.SOCKETH() within a phantom process makes that process consume a licence.
Example

SRVR.SKT = CREATE.SERVER.SOCKET("", 4000, SKT$STREAM + SKT$TCP)
IF STATUS() THEN STOP 'Cannot initialise server socket'
SKT = ACCEPT.SOCKET.CONNECTION(SRVR.SKT, 0)
IF STATUS() THEN STOP 'Error accepting connection'
DATA = READ.SOCKET(SKT, 100, SKT$BLOCKING, 0)
CLOSE.SOCKET SKT
CLOSE.SOCKET SRVR.SKT

This program fragment creates a server socket, waits for an incoming connection, reads a single data packet from this connection and then closes the sockets.

See also:
Using Socket Connections, ACCEPT.SOCKET.CONNECTION, CLOSE.SOCKET, OPEN.SOCKEIT(), READ.SOCKEIT(), SELECT.SOCKEIT(), SERVER.ADDR(), SET.SOCKEIT.MODE(), SOCKET.INFO(), WRITE.SOCKEIT()
CROP()

The CROP() function removes redundant mark characters from a string.

Format

CROP(string)

where

string evaluates to the string from which mark characters are to be removed.

The CROP() function removes redundant mark characters from string. These are all mark characters after the final non-mark character, trailing subvalue marks within each value and trailing value marks within each field.

Example

REC = CROP(REC)

This statement removes all redundant mark characters from REC.
The **CSVDQ()** function de-quotes a CSV (comma separated variable) string.

### Format

\[
\text{CSVDQ(} \text{string} \{, \text{delimiter}\})
\]

where

- **string** is the string to be processed.
- **delimiter** is the delimiter character separating the elements of the string. This defaults to a comma if omitted.

The **CSVDQ()** function de-quotes a CSV string, removing outer double quotes, handling embedded quotes and returning the result as a dynamic array where each element of the original string is represented by a separate field. This is in accordance with the CSV format specification (RFC 4180).

### Examples

```plaintext
S = 'ABC,"DEF","GHI,JKL","MN"O"'
DISPLAY CSVDQ(S)
```

The above program fragment would display

```
ABCDEF,GHIJKL,MN"O
```

ABC was unquoted and remains unchanged.
DEF was quoted. The quotes have been removed.
GHIJKL contains a comma that has been preserved after removal of the quotes.
MN"O contains an embedded quote that has been preserved.

```plaintext
READSEQ DATA FROM SEQ.F THEN REC = CSVDQ(DATA)
```

The above program fragment reads a line of CSV format data into variable DATA and then uses **CSVDQ()** to parse the elements into a field mark delimited dynamic array.

**See also:**
- **DPARSE.CSV, FORMCSV0, READCSV, WRITECSV**
CSV.MODE

The CSV.MODE statement sets the conversion mode for CSV (comma separated variable) construction.

Format

CSV.MODE mode {, delim}

where

mode is the desired mode value.

delim is the delimiter to appear between items. If omitted or a null string, this defaults to a comma. If more than one character in length, only the first character is used.

The CSV mode setting determines the quoting rules and delimiter applied by the FORMCSV(), PRINTCSV and WRITECSV operations.

Three modes of conversion are supported:

1. Output conforms to the CSV format specification (RFC 4180). Items containing double quotes or the delimiter character are enclosed in double quotes with embedded double quotes replace by two adjacent double quotes.
2. Encloses all non-null values in double quotes except for numeric items that do not contain a comma. Embedded double quotes are replaced by two adjacent double quotes.
3. Encloses all values in double quotes. Embedded double quotes are replaced by two adjacent double quotes.

Mode 1 is used by default.

The optional delim component of this statement sets the delimiter character to be used, defaulting to a comma.

Any change to the mode or delimiter persists through subroutine calls. If changed within a subroutine, the previous value is restored on return. Use of EXECUTE to enter a new command processor level will reset the mode and delimiter to their default states. The previous mode and delimiter will be restored on return from the executed command.

An application can find the current CSV mode and delimiter by using the SYSTEM() function with key values 1043 and 1046 respectively. Tokens SYSSCSV.MODE and SYSSCSV.DELIM defined in the SYSCOM KEYS.H record can be used for these values.

Examples

S = 'ABC':@FM:'123':@FM:'A"B':@FM:'A,B'
CSV.MODE 1 ; DISPLAY FORMCSV(S)
CSV.MODE 2 ; DISPLAY FORMCSV(S)
CSV.MODE 3 ; DISPLAY FORMCSV(S)

The above program fragment would display

ABC,123,"A""B","A,B"
"ABC",123,"A""B","A,B"
"ABC","123","A"","B","A,B"

CSV.MODE 1, '*' ; PRINTCSV 'ABC', 'DEF', 'GHI'

The above program fragment would display
ABC*DEF*GHI

See also:
FORMCSV(), READCSV,WRITECSV
DATA

The DATA statement adds one or more items to the input data queue.

Format

```
DATA expr{, expr...}
```

where

```
expr
```
evaluates to the value to be added to the input data queue.

Where multiple `expr` clauses are present, they may be spread over successive lines by inserting a newline between a comma and the subsequent item. Any number of `expr` clauses may be present.

The INPUT statement takes data provided by DATA statements in preference to reading from the keyboard. Keyboard input is only used if there is no data from DATA statements remaining to be processed. The KEYIN() function always takes its input from the keyboard.

The data stream generated by successive DATA statements is held in the @DATA.PENDING variable which may be read by programs. This variable contains the individual data items each followed by an item mark. For this reason, DATA statement items should not include item marks as these will be taken as separators.

Example

```
DATA "123", "456"
DATA "789"
LOOP
   INPUT S
   WHILE LEN(S)
      DISPLAY "'" : S : "'
   REPEAT
```

This program fragment would result in display of

```
123
456
789
```

and then echo data typed at the keyboard until a blank line is entered.

See also:
CLEARDATA, DATA command
DATE()

The DATE() function returns the internal value of the current date.

Format

DATE()

The DATE() function returns the day number of the current date within the user's time zone. Day numbers are defined such that 31 December 1967 is day zero. Date values representing dates earlier than this are negative.

Example

INVOICE.REC<7> = DATE()

This statement assigns field 7 of INVOICE.REC with the internal date value of the current day.

See also: @DATE, EPOCH(), TIME()
DCOUNT()

The DCOUNT() function counts delimited substrings within a string.

Format

\[
\text{DCOUNT}(\text{string, delimiter})
\]

where

- \text{string} evaluates to the string in which delimited substrings are to be counted.
- \text{delimiter} evaluates to the substring separating items in \text{string}.

The DCOUNT() function counts substrings delimited by \text{delimiter} within \text{string}.

If \text{string} is null, DCOUNT() returns zero. In all other cases, DCOUNT() returns a value one greater than the COUNT() function applied to the same \text{string}.

If \text{substring} is null, DCOUNT() returns the length of \text{string}.

Programs compiled with the \texttt{MODE NOCASE.STRINGS} compiler directive use case insensitive string comparisons in the DCOUNT() function.

Examples

\begin{verbatim}
S = "172-34-876"
N = DCOUNT(S, "-")
\end{verbatim}

sets \text{N} to 3.

\begin{verbatim}
S = 'ABABABABABABA'
N = DCOUNT(S, "BAB")
\end{verbatim}

sets \text{N} to 4.

\begin{verbatim}
FIELDS = DCOUNT(REC, @FM)
\end{verbatim}

This statement counts the fields in \text{REC}.

See also:
COUNT()
DEBUG

The DEBUG statement enters the interactive debugger.

Format

```
DEBUG
```

The DEBUG statement enters debug mode for the program in which it was executed and all programs called by it. This statement causes a compile time warning but is otherwise ignored if the program is not complied with the DEBUGGING option or use of the $DEBUG compiler directive.

Example

```
IF @LOGNAME = 'mjones' THEN DEBUG
```

The above statement would enter the debugger if the user running the program is logged in as mjones.

See also:
QMBasic debugger
DECRYPT()

The **DECRYPT()** function decrypts data that has been encrypted for secure storage or transmission.

**Format**

```
DECRYPT(data, key)
```

where

- `data` is the string to be decrypted.
- `key` is the encryption key to be used.

The **DECRYPT()** function applies the AES 128 bit encryption algorithm to the supplied data and returns the decrypted text. The key string may be up to 64 characters in length and may contain any character. It is automatically transformed into a form that is usable by the AES algorithm. For optimum data security, the key should be about 16 characters.

The encrypted data is structured so that it can never contain characters from the C0 control group (characters 0 to 31) or the mark characters. As a result of this operation, the encrypted data is slightly longer than the resultant decrypted data.

The **DECRYPT()** function works with data that has been encrypted using either **ENCRYPT()** or **ENCRYPTX()**.

On an ECS mode system, the encrypted data must have been converted to a byte string prior to encryption if it may contain ECS characters. This is because encryption is a byte level operation. There is an implication that a program decrypting the data must know that it will need to convert the byte string back to ECS characters using the **BS** conversion code.

**Example**

```qbasic
FUNCTION LOGIN()
    OPEN 'USERS' TO USR.F ELSE
    DISPLAY 'Cannot open USERS file'
    RETURN @FALSE
END
    DISPLAY 'User name: '
    INPUT USERNAME, 20_: 
    READ USER.REC FROM USR.F THEN
    FOR I = 1 TO 3
        DISPLAY 'Password: '
        INPUT PW,20_: HIDDEN
        IF PW = DECRYPT(USR.REC<1>, 'MySecretKey') THEN RETURN @TRUE
        DISPLAY 'Password incorrect'
    NEXT I
END
RETURN @FALSE
END
```

The above function prompts for a user name and password, validating these against a record in the USERS file. The password field of this file is encrypted.
See also:
Data encryption, ENCRYPT()
DEFFUN

The DEFFUN statement defines a function.

Format

```
DEFFUN name \{(arg1 [, arg2 ...])\} \{CALLING "func" \| expr\} \{VAR.ARGS\} \{KEY key\}
DEFFUN name \{(arg1 [, arg2 ...])\} LOCAL
DEFFUN name \{(arg1 [, arg2 ...])\} EXTERNAL \{CALLING "libname"\}
DEFFUN name \{(arg1 [, arg2 ...])\} EXTERNAL \{CALLING "libname:funcname"\}
```

where

- `name` is the name of the function.
- `arg1, arg2...` are the function arguments. Up to 254 arguments can be supplied except for external functions where this limit is 31.
- `func` is the catalogue name of the function if different from `name`.
- `expr` is an expression that evaluates to the catalogue name of the function to be called.
- `key` is a key value to be passed into the subroutine as described below.
- `libname` is the name of the program providing the function. This is case sensitive on Linux systems. The .exe suffix is provided automatically on Windows. If no `CALLING` clause is present, the default name "qmextcall" is used.
- `funcname` is the case sensitive name of the function to be called from `libname`. name of the program providing the function.

The DEFFUN statement defines a function that may be called from within the program. The DEFFUN statement must appear before the first reference to the function. If the function `name` matches the name of a built in function, any references to `name` before the DEFFUN will call the intrinsic function and references after the DEFFUN will call the declared function.

If the LOCAL and EXTERNAL keywords are not present, the function must correspond to a catalogued item. A call to this function call is effectively translated to a call to a subroutine with an additional hidden first argument through which the result is returned. The optional CALLING component of the DEFFUN statement allows the catalogue name of the function to be different from the `name` of the function itself. This may be specified as a literal string or as an expression that evaluates to the catalogue name. When using an expression, this is evaluated at the point where the function is used, not when the DEFFUN statement is processed.

Use of the LOCAL keyword indicates that this function is internal to the program module and will be defined later in the source by use of the LOCAL FUNCTION statement.

Use of the EXTERNAL keyword defines reference to a subroutine that is written in some other language such as C and is accessed via the External Call Interface. See External Functions for more details. When used without the CALLING clause, use of the QMBasic function will be passed to the qmextcall handler with the uppercase form of `name` used as the function to be called. If the CALLING clause is present, the qualifying information may be either just an alternative handler...
name or it may be a handler name and case sensitive function name separated by a colon. In all
cases, the name used within the QMBasic program is case insensitive unless QMBasic case
sensitivity has been enabled (see the CASESENSITIVE setting of the $MODE
compiler directive).

The argument names used in the DEFFUN statement are for documentation purposes only and have
no significance within the program except that the compiler counts them to verify correct use of the
function. The variables used in the actual call to the function are determined by the use of the
function. Except when defining an external function, an argument may be defined to be a whole
matrix in which case it should be prefixed by the keyword MAT in the DEFFUN argument list.

The function is used in the same way as the intrinsic functions described in this manual. Although it
is not recommended, a function can update its argument variables.

The VARARGS option indicates that compiler should not check the number of arguments in calls
to the function. It is of use with functions that take variable length argument lists. This option cannot
be used with a local or external function.

The KEY option passes the key value as an additional argument before the first one named in calls to
the function. This enables construction of multiple functions that call a single catalogued item with a
mode key as the first argument. The !PCL() function provided in the BP file of the QMSYS account
uses this feature to implement the various PCL functions defined in the SYSCOM PCL.H include
record.

Example

    DEFFUN MATMAX(MAT A)
    DIM VALUES(100)
    ...
    MAX = MATMAX(MAT VALUES)

The program fragment above uses a user supplied MATMAX() function to find the maximum value
of all the elements of matrix VALUES.

See also:
FUNCTION, LOCAL
DEL

The **DEL** statement and **DELETE()** function delete a field, value or subvalue from a dynamic array. Used with a data collection, the **DEL** statement deletes an element.

**Format**

```
DEL dyn.array{field {}, value {}, subvalue}>
DELETE(dyn.array, field {, value {, subvalue}})
DEL collection{path}
```

where

- `dyn.array` is the dynamic array from which the item is to be deleted.
- `field` evaluates to the number of the field to be deleted.
- `value` evaluates to the number of the value to be deleted. If omitted or zero, the entire field is deleted.
- `subvalue` evaluates to the number of the subvalue to be deleted. If omitted or zero, the entire value is deleted.
- `collection` is a data collection variable.
- `path` is the element path of the data collection element to be deleted.

**Dynamic Arrays**

The specified field, value or subvalue is deleted from the dynamic array. The **DEL** statement assigns the result to the `dyn.array` variable. The **DELETE()** function returns the result without modifying `dyn.array`.

**Data Collections**

The collection element identified by the element `path` is deleted. If this is an array element, subsequent elements move back, reducing the array size. Note that when used in this way, the curly brackets surrounding the `path` are part of the syntax, not markers for optional items.

**Examples**

```
DEL ITEMS<1,N>
```

This statement deletes field 1, value N from dynamic array ITEMS.

```
S = DELETE(ITEMS, 1, N)
```

This statement is similar to the previous example except that the result is assigned to S, leaving ITEMS unchanged.
DEL CLIENT.DATA('CONTACT.NAME', N)

The above statement deletes element N of the CONTACT.NAME array in the CLIENT.DATA collection.

See also:
Data collections, EXTRACT(), FIND, FINDSTR, INS, INSERT(), LISTINDEX(), LOCATE, LOCATE(), REPLACE()
**DELETE**

The `DELETE` statement deletes a record from an open file. The `DELETEU` statement is similar but it preserves locks.

**Format**

```plaintext
DELETE file.var, record.id {ON ERROR statement(s)}
DELETEU file.var, record.id {ON ERROR statement(s)}
```

where

- `file.var` is a file variable for an open file.
- `record.id` evaluates to the id of the record to be deleted.
- `statement(s)` are statements to be executed if the delete fails.

The specified record is deleted from the file. No error occurs if the record does not exist.

If the process performing the `DELETE` had a read or update lock on the record, the lock is released. The `DELETEU` statement preserves any lock. Within a transaction, the lock is retained until the transaction terminates and then released regardless of which statement is used. Attempting to delete a record in a transaction will fail if the process does not hold an update lock on the record or the file.

The `ON ERROR` clause is executed for serious fault conditions such as errors in a file's internal control structures. It is also executed if the delete operation is disallowed by a pre-delete trigger function associated with the file. The `STATUS()` function will return an error number. If no `ON ERROR` clause is present, an abort would occur.

**Use of a THEN/ELSE Clause**

For compatibility with the way in which triggers operate in some other multivalue products, the `DELETE` statement has an optional `THEN/ELSE` clause. Because this would otherwise lead to a syntactic ambiguity, compilation of programs that use this clause requires the `WRITE.DELETE.THEN.ELSE` option of the `$MODE` compiler directive to be enabled. Once this is done, the optional `THEN/ELSE` clauses can be included in their usual position, after the `ON ERROR` clause.

When a `DELETE` statement has a `THEN/ELSE` clause, a failure returned from a trigger function, typically as a result of a pre-delete data validation error, will cause the `ELSE` clause to be executed instead of the `ON ERROR` clause.

**Example**

```plaintext
DELETE STOCK, ITEM.ID
```

This statement deletes the record whose id is in `ITEM.ID` from the file associated with file variable `STOCK`.
**DELETELIST**

The **DELETELIST** statement deletes a select list from the $SAVEDLISTS file.

**Format**

```
DELETELIST name
```

where

```
name
```

is the name of the $SAVEDLISTS entry to be deleted.

The **DELETELIST** statement deletes the previously saved select list identified by `name` from the $SAVEDLISTS file. No error occurs if the list does not exist.

**See also:**

**DELETE.LIST**
DELETESEQ

The **DELETESEQ** statement deletes an operating system file.

**Format**

```
DELETESEQ filename, id [ NO.MAP ] { ON ERROR statement(s) } { THEN statement(s) } { ELSE statement(s) }
```

or

```
DELETESEQ pathname [ NO.MAP ] { ON ERROR statement(s) } { THEN statement(s) } { ELSE statement(s) }
```

where

- `filename` evaluates to the VOC name of the directory file holding the record to be deleted.
- `id` evaluates to the name of the record to be deleted.
- `pathname` evaluates to the operating system pathname of the record to be deleted.
- `statement(s)` are statement(s) to be executed depending on the outcome of the **DELETESEQ** statement.

At least one of the **THEN** and **ELSE** clauses must be present.

The **DELETESEQ** statement deletes the operating system file identified by `filename` and `id` or by `pathname`. It is primarily intended as a counterpart to **OPENSEQ** but can be used to delete any operating system file.

The **THEN** clause will be executed if the action is successful.

The **ELSE** clause will be executed for conditions that most likely relate to user or programming errors such as the item to delete not existing or not having access rights to delete it. The **STATUS()** function can be used to determine the cause of the error.

The **ON ERROR** clause will be executed if an internal error occurs during the delete and should only be included if the program needs to continue execution rather than taking the default action of aborting at such an error. The **STATUS()** function can be used to determine the cause of the error.

**DELETESEQ** cannot be used to delete a directory.

The **NO.MAP** option is relevant only to directory files and suppresses the normal translation of restricted characters in record ids. See directory files for more information.

Note that **DELETESEQ** takes no part in the locking system. If locking is required, the directory containing the file to be deleted must be opened as a directory type file and standard file processing statements used.

**See also:**

**OSDELETE**
DFPART()

The DFPART() function returns a file variable that references an individual part of a distributed file.

Format

    DFPART(dist.file, part)

where

    dist.file is the file variable referencing a distributed file.
    part is the part number of the part to be accessed.

The DFPART() function allows a QMBasic program to refer to a specific part within a distributed file. It is of use, for example, with FILEINFO() to obtain details of a specific part or when constructing programs that need to merge index entries from multiple part files.

A list of part numbers can be obtained using the FILEINFO() function with key FL$PARTS. These part numbers can then be used with DFPART() to access the individual part files. If the part argument does not reference a valid part number, the program will abort with a run time error.

Examples

```qmb
OPEN 'ORDERS' TO ORD.F ELSE STOP 'Cannot open file'
PARTS = FILEINFO(ORD.F, FL$PARTS)
FOR EACH PT IN PARTS
    DISPLAY 'Part ' : PT : ' modulus ' : FILEINFO(DFPART(ORD.F,PT), FL$MODULUS)
NEXT PT
```

The above program reports the modulo values for each part file in the ORDERS distributed file.

```qmb
OPEN 'ORDERS' TO ORD.F ELSE STOP 'Cannot open file'
PARTS = FILEINFO(ORD.F, FL$PARTS)
LIST = ''
FOR EACH PT IN PARTS
    AK.F = DFPART(ORD.F,PT)
    SETLEFT 'VALUE' FROM AK.F
    LOOP
        SELECTRIGHT 'VALUE' FROM AK.F SETTING AK.KEY TO 1
        UNTIL STATUS()
        UNTIL AK.KEY > 100
        READLIST S FROM 1 THEN LIST<-1> = S
    REPEAT
NEXT PT
```

The index scanning operations are not available for distributed files because the indices are separate for each part file. The above program effectively merges index data from each part file within the ORDERS distributed file. In this example, LIST is created as a list ids for records where the VALUE item is less than 100.
**DIGEST()**

The `DIGEST()` function returns the message digest string for a supplied string or file.

**Format**

```
DIGEST(string, is.file, mode)
```

where

- `string` is either the string to be encoded or the pathname of a file containing data to be encoded.
- `is.file` is a Boolean value indicating whether `string` is the actual data string or a pathname.
- `mode` evaluates to the name of the digest algorithm to be used. Supported algorithms are MD4, MD5, SHA1, SHA256, SHA384 and SHA512.

If successful, the `DIGEST()` returns the result of applying the specified digest algorithm to the supplied data and the `STATUS()` function will return zero. All error conditions return a null string and the `STATUS()` function can be used to determine the cause.

**Example**

```
DISPLAY 'Password: ' : INPUT PASSWORD
IF DIGEST(PASSWORD, @FALSE, 'SHA256') # ENCRYPTED.PASSWORD THEN
   DISPLAY 'Access denied'
   RETURN
END
```

The above example compares a password entered by the user with a stored message digest value for the correct password. Because message digests are not reversible, there is no way to determine the actual password from the stored value.

**See also:**

`MD5()`
DIMENSION

The DIMENSION statement is used to set the dimensions of a matrix. The short form DIM may be used in place of DIMENSION. The REDIMENSION or REDIM statement is identical except where the matrix is a subroutine argument.

Format

    DIMENSION mat(rows {, cols})
    REDIMENSION mat(rows {, cols})
    DIMENSION var

where

    mat          is the name of the matrix.
    rows         evaluates to the number of rows in the matrix.
    cols         evaluates to the number of columns in a two dimensional matrix.
    var          is the name of a scalar variable.

In all cases, multiple comma separated items may be included in a single statement.

A matrix variable is a one or two dimensional array of values. Matrices must be declared by use of the DIMENSION statement and this statement must be executed at program run time before the variable is used in any other way.

A one dimensional matrix of ten elements is defined by a statement of the form

    DIMENSION A(10)

For a two dimensional matrix with 5 rows of 8 columns this becomes

    DIMENSION B(5,8)

By default, all matrices have an additional element, the zero element, which is used by some QMBasic statements. This is referred to as A(0) or B(0,0).

The elements of a matrix may hold data of differing types (numeric, string, file variable, etc).

A matrix may be re-dimensioned at any time by a further DIMENSION statement, including changing the number of dimensions. Existing values of matrix elements will be retained in the re-dimensioned matrix by effectively copying elements on a row by row basis. Thus, increasing the number of rows in a two dimensional matrix does not affect the previously existing items whereas increasing the number of columns will reorganise the existing data. When enlarging a matrix, the newly created elements will be unassigned. When decreasing the size of a matrix, any values in the excess elements are discarded.

A matrix passed into an external subroutine or function as an argument variable cannot be re-dimensioned using DIMENSION. Instead, the REDIMENSION statement must be used. See the SUBROUTINE statement for more detail.
The `$MODE` compiler directive can be used to select Pick style matrices which do not have a zero element and cannot be re-dimensioned.

The `INMAT()` function may be used to check on the success of a `DIMENSION` statement. A sequence such as

```
DIMENSION A(N)
IF INMAT() THEN AVOID "Insufficient memory"
```

will cause the program to abort if there is insufficient memory to hold the matrix. The `INMAT()` function used in this way returns 0 if the `DIMENSION` statement was successful, 1 if it failed. With the memory sizes found on modern systems, this test is probably totally unnecessary and usually omitted.

The `INMAT()` function can also be used to find the current dimensions of a matrix. For a single dimensional matrix, this function returns the number of elements excluding the zero element. For a two dimensional matrix, the function returns the row and column dimensions separated by a value mark.

If the `$MODE EXPLICIT` compiler directive is in force, scalar variables must also be declared in a `DIMENSION` statement before any other reference to the variable name in the program. This mode allows improved error checking by trapping errors such as misspelled variable names that might otherwise simply appear to be a separate variable.

### Examples

```
N = DCOUNT(REC, @FM)
DIM A(N)
MATPARSE A FROM REC, @FM
```

This program fragment creates an array with the correct number of elements to receive the result of a `MATPARSE` operation on dynamic array `REC` and then performs the `MATPARSE`.

```
DIM A(3), B(2,3)
DISPLAY INMAT(A)
DISPLAY CHANGE(INMAT(B), @VM, ',')
```

This program fragment creates two matrices and reports their sizes. For the second use of `INMAT()`, the `CHANGE()` function has been use to replace the value mark in the returned data with a comma. The program output would be `3,2,3`

```
DIM CLIENT, DATA.REC
```

This statement declares two scalar variables. This is only needed in conjunction with the `$MODE EXPLICIT` compiler directive.

### See also:
- `INMAT()`, `MAT`, `MATBUILD`, `MATREAD`, `MATREADCSV`, `MATPARSE`, `MATWRITE`
**DIR()**

The **DIR()** function returns the contents of an operating system directory.

**Format**

```
DIR(pathname)
```

where

*pathname* identifies the operating system directory to be processed.

The **DIR()** function returns a dynamic array with one field per entry in the specified directory. Each field contains the following items separated by value marks:

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Item name. On platforms with case insensitive file names, the text returned preserves the casing used by the underlying operating system.</td>
</tr>
<tr>
<td>2</td>
<td>Item type. This is <em>F</em> for a file or <em>D</em> for a directory.</td>
</tr>
</tbody>
</table>
| 3     | File modes. On Windows systems, this is a collection of letters representing the file attributes:
  - *A* = archive
  - *C* = compressed
  - *H* = hidden
  - *R* = read only
  - *S* = system
  - *T* = temporary
  On other platforms, this value contains the file permissions as a decimal value. |
| 4     | Date/time modified for the item as an epoch value. |
| 5     | File size in bytes. On Windows systems, this is zero for a directory. |

The standard . and .. directory entries are not returned.

Applications should not assume that this structure will remain unchanged. Additional values may appear in future releases.

Because the **DIR()** function returns data for the entire directory as a single string, it could fail for very large directories.

**Example**

```
FILES = DIR('C:\MYDIR')
NUM.FILES = DCOUNT(FILES, @FM)
FOR I = 1 TO NUM.FILES
   IF FILES<I,2> = 'F' THEN DISPLAY FILES<I,1>
NEXT I
```

The above example lists all the files in C:MYDIR, omitting subdirectories.
DISINHERIT

The DISINHERIT statement used in a class module removes an inherited object.

Format

DISINHERIT object

where

object is the object variable previously used in an INHERIT statement.

The DISINHERIT statement removes a previously inherited object from the name search used when locating the variable or public function/subroutine for an object reference.

See also:
Object oriented programming, CLASS, INHERIT, OBJECT(), OBJINFO(), PRIVATE, PUBLIC.
DISPLAY

The DISPLAY statement sends data to the display. The synonym CRT can be used in place of DISPLAY.

Format

DISPLAY {print.list}

where

print.list is a list of items to be displayed.

The DISPLAY statement is equivalent to a PRINT statement directed to the display.

The print.list contains any number of items (including zero) but must all appear on a single source program line. The individual items are expressions which can be evaluated and represented as strings.

Where multiple items are present, they are separated by commas. The DISPLAY statement will replace the comma by a TAB character, causing display to align to the next horizontal tabulation column, initially set to columns 0, 10, 20, 30, etc. The same effect can be achieved by inserting TAB characters in the data to be displayed.

A colon as the final token on the source line is not treated as a concatenation operator but as a special symbol which causes the normal line feed and carriage return at the end of the DISPLAY action to be suppressed.

When data is output to the display, QM will pause at the end of the screen and ask for confirmation to continue.

Press RETURN to continue, A to abort, Q to Quit, S to suppress pagination

Entering A will cause an abort to occur. Entering Q will quit from the current program or command. Entering S will continue display and suppress this prompt at the end of subsequent pages. Any other character causes display to continue up to the next prompt.

Pagination may be suppressed by use of any @(x, y) cursor positioning function. Note that this does not have to be sent to the screen; use of the function is sufficient. Thus a statement such as

DUMMY = @(0,0)

will suppress pagination even though nothing is displayed by this statement. Pagination may be turned on again by use of the PRINTER RESET statement.

Examples

DISPLAY "Error code " : STATUS()

This statement displays the literal string "Error code " and the value of the STATUS() function. The cursor is then positioned at the start of the next line.

DISPLAY "Enter product code " :
INPUT PRODUCT
This program fragment displays a prompt for a product code to be entered. Note the use of the trailing colon to suppress the line feed so that the cursor is left after the prompt ready for the `INPUT` statement.

```qmbasic
FOR I = 1 TO 10
    DISPLAY I, RATE(I), TOTAL(I)
NEXT I
```

This example displays a three column table, lining up the columns with horizontal tabulation positions.

See also:

- `@`(x,y), `PRINT`
DISPLAY.WIDTH

The DISPLAY.WIDTH() function returns the number of character positions required to display a string. The DISPLAY.WIDTHS() function is similar to DISPLAY.WIDTH() but operates on successive elements of a dynamic array, returning a similarly structured dynamic array of results.

Format

```
DISPLAY.WIDTH(string)
```

where

```
string    is the string to be examined.
```

In ECS mode, some Unicode characters are defined to have a double width glyph (a displayed character that requires two positions on the screen or printer). The DISPLAY.WIDTH() function takes a string and returns the number of display positions required. Note that the presence of control characters or device specific escape sequences within string may cause a misleading result. This function assumes that all characters occupy a single position with the exception of those defined in the currently active character map as occupying two positions.

In a non-ECS mode system, the DISPLAY.WIDTH() function behaves the same as the LEN() function, returning the length of the string that is passed as its argument.

See also:
LEN(), SUBSTRDW()
DISABLE.KEY

The **DISABLE.KEY** statement is used to disable access to a secure encryption key previously enabled with **ENABLE.KEY**.

**Format**

```
DISABLE.KEY keyname
```

where

```
keyname
```

is the name of the encryption key to be disabled.

An encryption key created using the **CREATE.SECURE.KEY** command can only be accessed in a QM session after it has been enabled by use of the **ENABLE.KEY** QMBasic statement, or corresponding **ENABLE.KEY** command. Access to the key can later be denied by use of **DISABLE.KEY**.

**See also:**
- Data encryption, **CHANGE.KEY.PASSWORD**, **CREATE.FILE**, **CREATE.KEY**, **CREATE.SECURE.KEY**, **DELETE.KEY**, **DISABLE.KEY**, **ENABLE.KEY**, **ENABLE.KEY (QMBasic)**, **ENCRYPT.FILE**, **GRANT.KEY**, **LIST.KEYS**, **RESET.MASTER.KEY**, **REVOKE.KEY**, **SET.ENCRIPTION.KEY.NAME**, **UNLOCK.KEY.VAULT**
DIV()

The **DIV()** function returns the quotient from a division operation.

**Format**

\[
\text{DIV}(\text{dividend}, \text{divisor})
\]

where

- **dividend** evaluates to a number.
- **divisor** evaluates to a number.

The **DIV()** function returns the quotient (integer part) from dividing \textit{dividend} by \textit{divisor}. Any remainder is ignored.

See also;

- **IDIV()**, **RDIV()**
DOWNCASE()

The DOWNCASE() function returns a string with all letters converted to lower case.

Format

    DOWNCASE(string)

where

    string evaluates to the string in which substitution is to occur.

The DOWNCASE() function returns the value of string with all letters converted to lower case. If string is a variable rather than an expression, the value of the variable is not affected.

On an ECS mode system, the lowercase equivalent of each character is defined by the currently selected character map. On other systems, the standard letter pairings of the ASCII character set are used.

Example

    REF.NO = "A12F635"
    PRINT DOWNCASE(REF.NO)

This program fragment prints the string "a12f635".

See also:

    UPCASE()
DPARSE and DPARSE.CSV

The DPARSE statement splits the elements of a delimited string into other variables. DPARSE.CSV is similar but unquotes strings according to the rules in the CSV standard.

Format

DPARSE string, delimiter, var1, var2,...

DPARSE.CSV string, delimiter, var1, var2,...

where

string evaluates to the string to be processed.

delimiter evaluates to the substring delimiter. If this is more than one character, only the first character is used.

var1, var2,... is a list of variables to receive the elements extracted from string. These may be simple variables, matrix elements or field, value or subvalue references. If string has more elements than the number of variables, the excess is lost. If it has fewer elements than the number of variables, the trailing unused ones are set to null strings.

The DPARSE statement extracts successive elements delimited by delimiter within string into the listed variables. A statement such as

DPARSE S, ",", A, B, C

is equivalent to

A = FIELD(S, ",", 1)
B = FIELD(S, ",", 2)
C = FIELD(S, ",", 3)

DPARSE.CSV extends this action by removing quotes that have been applied to meet the CSV standard. CSV format allows optional double quotes around items and specifies that these must be present when the item contains the delimiting character or a double quote. In the latter case, the embedded double quote will be represented by two adjacent double quotes.

Examples

LOOP
    READSEQ LINE FROM SEQ.F ELSE EXIT
    STOCK.REC = ""
    DPARSE LINE, ",", PART.NO, STOCK.REC<1>, STOCK.REC<2>
    ...Processing...
REPEAT

This loop dismantles a comma separated string to extract three elements.

S = 'abc,"de""f"'
DPARSE S, ",", A, B
DISPLAY A, B
DPARSE.CSV S, ",", A, B
DISPLAY A,B

This program will print:

- abc "de" "f"
- abc de"f"

showing the difference between the **DPARSE** and **DPARSE.CSV** statements.
The **DTX()** function converts a number to hexadecimal.

### Format

```
DTX(expr {, min.width})
```

where

- `expr` evaluates to the number to be converted.
- `min.width` specifies the minimum number of digits in the converted result.

The **DTX()** function converts the supplied `expr` value to hexadecimal. If the converted value is shorter than `min.width`, leading zeros are added. Note that **DTX()** treats the value as an unsigned quantity.

### Examples

In each example, the **DISPLAY** statement is followed by the output that it would produce.

```
DISPLAY DTX(87)
57
```

```
DISPLAY DTX(87,4)
0057
```

### See also:

**XTD()**
**EBCDIC()**

The **EBCDIC()** function converts an ASCII string to EBCDIC.

**Format**

```
EBCDIC(expr)
```

where

```
expr       evaluates to the string to be converted.
```

The **EBCDIC()** function returns the EBCDIC equivalent of the supplied ASCII string. The action of this function with non-ASCII characters is undefined.

**See also:**

**ASCII()**
ECHAR()

The ECHAR() function returns the ECS character with a given code point value. The ECHARS() function is similar to ECHAR() but operates on successive elements of a dynamic array, returning a similarly structured dynamic array of results.

Format

\[ \text{ECHAR}(\text{seq}) \]

where

\[ \text{seq} \] evaluates to an integer in the range 0 to 65535.

The ECHAR() function returns a single character string containing the ECS character with value seq. It is the inverse of the SEQ() function.

Only the least significant 16 bits of the integer value of seq are used.

Used on a non-ECS system, the ECHAR() function is equivalent to the CHAR() function and restricted to character values in the range 0 to 255.

Example

\[ S = \text{ECHAR}(0x03A3) \]

This statement sets variable S to contain the character with Unicode code point value hexadecimal 03A3, the uppercase Greek Sigma. Note the use of the hexadecimal numeric constant notation because codepoints are normally written in this form. This example is exactly equivalent to use of the corresponding decimal value:

\[ S = \text{ECHAR}(913) \]

See also:
CHAR(), SEQ()
ECHO

The ECHO statement enables or disables echoing of keyboard input.

Format

    ECHO OFF
    ECHO ON
    ECHO expr

where

    expr  evaluates to a number.

Keyboard input is normally echoed to the display when it is processed by an INPUT statement. Use of ECHO OFF will suppress this echo until a subsequent ECHO ON statement. Typically, ECHO OFF is used to prevent display of passwords, etc. Turning off echo does not suppress display of the input prompt character, if enabled (see PROMPT).

The ECHO expr format of this statement is equivalent to ECHO ON if the value of expr is non-zero and ECHO OFF if expr is zero.

Example

    DISPLAY "Enter password " :
    ECHO OFF
    INPUT PASSWORD
    ECHO ON

This program fragment requests entry of a password with echoing inhibited. The HIDDEN option of the INPUT statement may be a better way to do this as it echoes asterisks back to the terminal.

See also:

    INPUT, INPUT @
**ELEMENT.EXISTS()**

The `ELEMENT.EXISTS()` function tests whether an element exists in a data collection.

**Format**

```
ELEMENT.EXISTS(var{{path}})
```

where

- `var` is a data collection variable.
- `path` is the element path to be tested. The `{}` characters are part of the function syntax.

The `ELEMENT.EXISTS()` function scans `var` to test whether element `path` exists. The function returns True if it is found, False if not.

**Example**

```
COL = JPARSE(JSON.STRING)
IF NOT(ELEMENT.EXISTS(COL{'CLIENT', 'AGENT.CODE'}) THEN
    STOP 'Agent code not found'
END
```

The above code fragment parses a JSON string into a data collection and then checks whether the `CLIENT/AGENT.CODE` element is present.
ENABLE.KEY

The ENABLE.KEY statement is used to enter the password for a secure encryption key.

Format

ENABLE.KEY keyname, password {THEN statement(s)} {ELSE statement(s)}

where

keyname is the name of the encryption key to be enabled.
password is the password protecting this key.

At least one of the THEN and ELSE clauses must be present.

An encryption key created using the CREATE.SECURE.KEY command can only be accessed in a QM session after it has been enabled by use of the ENABLE.KEY statement of the corresponding ENABLE.KEY command.

If successful, the THEN clause is executed. If unsuccessful, the ELSE clause is executed after a short pause.

Access to the key can be denied again by use of DISABLE.KEY.

See also:
Data encryption, CHANGE.KEYPASSWORD, CREATE.FILE, CREATE.KEY.
CREATE.SECURE.KEY, DELETE.KEY, DISABLE.KEY, DISABLE.KEY (QMBasic),
ENABLE.KEY, ENCRYPT.FILE, GRANT.KEY, LIST.KEYS, RESET.MASTER.KEY,
REVOC.KEY, SET.ENCRYPTION.KEY.NAME, UNLOCK.KEY.VAULT
ENCRYPT(), ENCRYPTX()

The ENCRYPT() and ENCRYPTX() functions encrypt data for secure storage or transmission.

Format

ENCRYPT(data, key)

where

- **data** is the string to be encrypted.
- **key** is the encryption key to be used. Note that this is the actual key string, not a reference to a key established using the CREATE.KEY command.

The ENCRYPT() function applies the AES 128 bit encryption algorithm to the supplied data and returns the encrypted text. The key string may be up to 64 characters in length and may contain any character. It is automatically transformed into a form that is usable by the AES algorithm. For optimum data security, the key should be about 16 characters.

The encrypted data is post-processed so that it can never contain characters from the C0 control group (characters 0 to 31) or the mark characters. As a result of this operation, the encrypted data is slightly longer than the original source data.

The ENCRYPTX() function is similar but uses a random initialisation vector that it included in the returned encrypted string. This results in greater security.

Both encrypted data formats are decrypted using the DECRYPT() function.

On an ECS mode system, the data to be encrypted must be converted to a byte string using the BS conversion code prior to encryption if it may contain ECS characters. This is because encryption is a byte level operation. The effect of encrypting an ECS string directly is undefined. There is an implication that a program decrypting the data must know that it will need to convert the byte string back to ECS characters.

Also note that encrypting the same string in both ECS and non-ECS modes will yield different results as the underlying representation of the data is different.

Examples

FUNCTION LOGIN()
OPEN 'USERS' TO USR.F ELSE
   DISPLAY 'Cannot open USERS file'
   RETURN @FALSE
END
DISPLAY 'User name: ':
INPUT USERNAME, 20_:
READ USER.REC FROM USR.F THEN
   FOR I = 1 TO 3
      DISPLAY 'Password: ':
      INPUT PW,20_: HIDDEN
      IF ENCRYPT(PW, 'MySecretKey') = USR.REC<1> THEN RETURN
   @TRUE
      DISPLAY 'Password incorrect'
The above function prompts for a user name and password, validating these against a record in the USERS file. The password field of this file is encrypted.

Changing the encryption line in the above example to be

```basic
IF ENCRYPT(OCONV(PW, 'BSH'), 'MySecretKey') = USR.REC<1> THEN RETURN @TRUE
```

would allow use of ECS characters in the password.

**See also:**
Data encryption, **DECRIPT()**
END

The **END** statement terminates a program, subroutine or a block of statements conditioned by the **THEN**, **ELSE**, **LOCKED** or **ON ERROR** keywords.

**Format**

```
END
```

When used to terminate a program or subroutine, the **END** statement may not be followed by any further executable statements; only comments and blank lines are allowed. An **END** statement is not mandatory at the end of a program, subroutine or function but lack of an **END** will cause a warning message to be displayed as this may be symptomatic of a structural problem elsewhere in the program.

The compiler will generate a **RETURN** statement at the end of the program or subroutine which will be executed if there is no **RETURN** or other program flow control statement immediately prior to the **END**. For compatibility with other multivalue database products, the **IMPLIED.STOP** option of the **$MODE** compiler directive can be used to insert a **STOP** in place of a return in program, subroutine or function modules. Class modules always insert an implied **RETURN** regardless of the setting of this mode.

The **END** statement is also used to terminate a group of conditional statements. For details of the use of **END** in this way, see the description of the QMBasic statement to which the conditional element applies.

**Example**

```
READ REC FROM STOCK.FILE, ITEM THEN
   DISPLAY "Stock item " : ITEM
   GOSUB PROCESS.ITEM
END
```

This program fragment reads a record from a file, displays its record id and calls an internal subroutine to process the record. The two indented statements are both conditioned so that they only occur if the read was successful.
**ENTER.PACKAGE()**

The **ENTER.PACKAGE**() function attempts to enter a licensed software package.

**Format**

```
ENTER.PACKAGE(package.name)
```

where

- `package.name` is the name of the package to be entered.

The **ENTER.PACKAGE**() function behaves differently according to the package type.

For a **registered package**, it confirms that the named package is installed, has not expired, and that the user limit has not been reached. If all is ok, the user is recorded as being active in the package and the function returns True. If an error occurs, the function returns False and the error code can be found using the **STATUS**() function.

For a **private package**, the function returns the number of users in the package, including the one attempting to enter. An error will return zero and the error code can be found using the **STATUS**() function. It is the application's responsibility to apply any concurrent user limits or other constraints.

**See also:**
- Package licensing, **EXIT.PACKAGE()**, **PACKAGE.LICENCE**, 'PACKAGE()'
ENUMERATE()

The ENUMERATE() function identifies the elements in a data collection.

Format

    ENUMERATE(var)

where

    var is the data collection to be processed.

The ENUMERATE() function returns a field mark delimited sorted list of the element names in a data collection. Where an element is itself a further data collection, the function does not recurse down into the lower level item. Instead, if an application needs to create a complete map of an arbitrarily multi-dimensional data collection, it must handle the recursion internally.

If var references a non-existent data collection element, the ENUMERATE() function returns a null string.

The VARTYPE() function can be used to determine the type of a variable within a data collection.

Example

    VAR = JPARSE(JSON.STRING)
    NAMES = ENUMERATE(VAR)

The above code fragment parses a JSON string into a data collection and then creates a list of the top level elements within the collection.

See also:

    Data collections
ENV()

The ENV() function retrieves an operating system environment variable.

Format

\[ \text{ENV}(\text{var.name}) \]

where

\[ \text{var.name} \]

is the name of the variable to be retrieved.

The ENV() function retrieves the named operating system environment variable, returning its value. If the variable is not defined or var.name is invalid a null string is returned.

In general, environment variables do not change during the life of a QM session. For example, on a Linux system, the PWD environment variable stores the current working directory pathname. Within QM, use of ENV(PWD) will return the value of the environment variable passed to the QM process when it started. Using LOGTO, which effectively changes the current working directory, will not affect the value of the PWD environment variable within QM. A shell process started using SH after the LOGTO will see the new value of PWD as environment variables are maintained separately for each Linux process.
EPOCH()

The EPOCH() function returns the internal value of the current date and time in epoch format.

Format

    EPOCH()

The EPOCH() function returns the current date and time in time zone independent epoch format, as a number of seconds since 1 January 1970 UTC. This is equivalent to use of the SYSTEM(1035) function.

Because epoch values are defined to be 32 bit integers, there is a limitation that they can only represent times between 13 December 1901 and 19 January 2038. This is a widely recognised issue, potentially more widespread than the "millennium bug", that will require changes to operating systems and run time libraries.

Example

    TIMESTAMP = EPOCH()

This statement assigns the TIMESTAMP variable with the current date and time in epoch format.

See also:
DATE(), TIME(), Date, Times and Epoch Values, EPOCH(), MVDATE(), MVDATE.TIME(), MVEPOCH(), MVTIME(), SET.TIMEZONE
**EQS()**

The **EQS()** function processes two dynamic arrays, returning a similarly structured result array indicating whether corresponding elements are equal.

**Format**

```
EQS(expr1, expr2)
```

where

```
expr1 and expr2 are the dynamic arrays to be compared.
```

The **EQS()** function compares corresponding elements of the dynamic arrays `expr1` and `expr2`, returning a similarly structured dynamic array of True / False values indicating the results of the comparison.

The **REUSE()** function can be applied to either or both expressions. Without this function, any absent trailing values are taken as False.

**Example**

A contains 11FM0VM14VMABCFM2  
B contains 12FM0VM14VMACBFM2

```
C = EQS(A, B)
```

C now contains 0FM1VM1VM0FM1

**See also:**  
ANDS(), GES(), GTS(), IFS(), LES(), LTS(), NES(), NOTS(), ORS(), REUSE()
EQUATE

The EQUATE statement, often abbreviated to EQU, defines a symbolic name to represent a QMBasic expression.

Format

\[
\begin{align*}
\text{EQUATE} & \quad \text{name LITERALLY "string"} \\
\text{EQUATE} & \quad \text{name TO expression}
\end{align*}
\]

where

- **name** is the name to be attached to the value or matrix reference.
- **string** is a quoted string consisting of QMBasic language elements.
- **expression** is any valid QMBasic expression.

The EQUATE statement defines a name that can be used later within the program as a reference to the string or expression.

In most cases, the two forms of this statement are interchangeable but the first form will be needed if the **string** contains unpaired brackets. The LITERALLY keyword may be abbreviated to LIT.

The second form of the EQUATE statement can alternatively be written using the SDEFINE compiler directive.

EQUATE statements are often placed in include records so that the same values can be incorporated into many programs without duplication. Common uses of EQUATE include:

- **EQUATE name TO value**
  - This form of the EQUATE statement creates a symbolic **name** that can be used in place of the constant **value**. The EQUATE statement can be used to eliminate constants for state variables, field positions, etc from the main body of a program. Subsequent changes to the **value** thus only require a single amendment and recompilation of all programs using **name**.

- **EQUATE name TO variable**
  - This form of EQUATE defines a synonym for a variable name.

- **EQUATE name TO matrix(index1 {, index2])**
  - This form of EQUATE defines a name by which an element of a dimensioned array can be referenced. The index values must be numeric constants.

- **EQUATE name TO dyn.array<field {, value {, subvalue}}}>**
  - This form of EQUATE defines a name by which an element of a dynamic array can be referenced. The index values must be numeric constants. Declaration and dimensionality of the **matrix** are only checked when a reference to **name** is encountered during program compilation.

- **EQUATE name TO CHAR(seq)**
This form of **EQUATE** defines a name to represent a character value. It is often used with non-printing characters.

Multiple tokens may be equated on a single line by separating each definition by a comma. For example:

```
EQUATE LOW TO 12, HIGH TO 20
```

An equated token may not be used within the same source line as its definition. For example

```
EQUATE NAME TO CLIENT.REC(1) ; NAME = ""
```

will not recognise the use of NAME in the second statement as being a reference to CLIENT.REC(1)

Although it is valid for *name* to correspond to a built-in function, statement or reserved word, use in this way is discouraged as the function, statement or reserved word becomes inaccessible. As an aid to detecting this situation, particularly when migrating an existing application to QM, use of the **STRICT.EQUATE** setting of the **$MODE** compiler directive will generate a compiler warning for name clashes.

### Examples

```
EQUATE ADDRESS TO 1
EQUATE TEL.NO TO 2
...
READ REC FROM DATA.FILE, ID THEN
   PRINT "Address: ": REC<ADDRESS>
   PRINT "Telephone: ": REC<TEL.NO>
END
```

The above program fragment attaches name to two fields of a data record and then uses these when the data is extracted for printing.

```
EQUATE ADDRESS TO REC(1)
EQUATE TEL.NO TO REC(2)
DIM REC(10)
...
MATREAD REC FROM DATA.FILE, ID THEN
   PRINT "Address: ": ADDRESS
   PRINT "Telephone: ": TEL.NO
END
```

This program fragment achieves the same as the previous example but uses the **MATREAD** statement to separate the fields of the record read from the file into the elements of matrix REC. The names defined in the **EQUATE** statements are then used to reference elements of this matrix

```
EQUATE VALUE LIT "COST * QTY"
COST = 14
QTY = 2
DISPLAY VALUE
```

The above example uses an equated token, VALUE, to represent the calculation of COST * QTY. When executed, this example would display the value 28.

**See also:**
$DEFINE, GENERATE
**EREPLACE()**

The **EREPLACE()** function replaces occurrences of a substring within a string by another substring.

**Format**

```plaintext
EREPLACE(string, old, new[, occurrence[, start]])
```

where

- `string` is the string in which the replacement is to occur.
- `old` evaluates to the substring to be replaced.
- `new` evaluates to the substring with which `old` is to be replaced.
- `occurrence` evaluates to the number of occurrences of `old` to be replaced. If omitted or specified as a value of less than one, all occurrences are replaced.
- `start` specifies the first occurrence to be replaced. If omitted or specified as a value of less than one it defaults to one.

The **EREPLACE()** function replaces the specified occurrences of `old` within `string` by `new`.

If `old` is a null string, the function prepends `new` to `string`. The **EREPLACE()** function is identical to **CHANGE()** if `old` is not a null string.

If `new` is a null string, all occurrences of `old` are removed.

If the **SMODE NOCASE.STRINGS** compiler directive is used, matching of `old` against `string` is case insensitive.

The default behaviour of the **EREPLACE()** function is that overlapping substrings are allowed when searching `string` for the desired occurrence of `old`. Thus

```plaintext
X = EREPLACE("aaaaaaaaa", "aa", "xx", 0, 3)
```

yields a result of "aaxxxxxxa"

The **CHANGE.NO.OVERLAP** setting of the **SMODE** compiler directive can be used to select an alternative behaviour in which substrings do not overlap. The same example would then yield a result of "aaaaxxxa".

**Examples**

```plaintext
PRINT CHANGE("ABRACADABRA", "A", "a", 3, 2)
```

This statement results in printing the string "ABRaCaDaBRA".

**See also:**

**CHANGE()**, **CONVERT()**
**ERRMSG**

The **ERRMSG** statement displays a Pick style message from the ERRMSG file.

**Format**

```
ERRMSG msg.id [, arg...]
```

where

- `msg.id` evaluates to the id of a record in the ERRMSG file which holds the message to be displayed. If this id is numeric, it will be copied to `@SYSTEM.RETURN.CODE`.
- `arg...` is an optional comma separated list of arguments to be substituted into the message.

A standard Pick ERRMSG file is supplied with QM. Many of the messages in this file are irrelevant on QM. Users may modify this to add new messages or to change existing ones. QM only uses this file through **ERRMSG** and Pick style **STOP** or **ABORT** statements in user written programs.

The ERRMSG file entry consists of one or more fields, each prefixed by an action code. The message is built up and displayed by processing each code in turn. The codes are:

- `A{n}` Display the next argument left aligned in a field of `n` characters. If `n` is omitted, the argument is displayed without any additional spaces.
- `B` Sound the terminal "bell".
- `D` Outputs the system date in the form `dd mmm yyyy`.
- `E` Outputs the `msg.id` enclosed in square brackets.
- `H{text}` Outputs the given `text`.
- `L{n}` Outputs `n` newlines. The value of `n` defaults to 1 if omitted.
- `R{n}` Display the next argument right aligned in a field of `n` characters. If `n` is omitted, the argument is displayed without any additional spaces.
- `Sn` Displays `n` spaces.
- `T` Outputs the system time in the form `hh:mm:ss`.

The component parts of the message are output with no insertion of newlines except as explicitly specified in the ERRMSG entry.

**See also:**

**ABORT, STOP**
EVALUATE

The **EVALUATE** statement evaluates an I-type expression, trapping errors.

**Format**

```
EVALUATE var FROM i.type {THEN statement(s)} {ELSE statement(s)}
```

where

- `var` is the variable to receive the result.
- `i.type` is a dictionary record corresponding to a previously compiled I-type expression.

At least one of the **THEN** and **ELSE** clauses must be present.

The **EVALUATE** statement evaluates an I-type or correlative expression in much the same way as the **ITYPE()** function but allows the program using it to trap some error conditions. The statement takes the **THEN** clause if successful. It takes the **ELSE** clause if `i.type` is not a valid compiled expression. Note that this statement does not trap errors that occur in evaluating the expression.

**Example**

```
READ DICT.REC FROM DICT.F, 'SALE.VALUE' THEN
   EVALUATE SALE FROM DICT.REC ELSE
   DISPLAY 'SALE.VALUE is not a compiled expression'
END
```

**See also:**

- **ITYPE()**
EXECUTE

The EXECUTE statement enables a QMBasic program to execute any command. The synonym PERFORM can be used in place of EXECUTE.

Format

EXECUTE expr {TRAPPING ABORTS}
{CAPTURING var}
{NO.TTY}
{PASSLIST {src.list}}
{RTNLIST tgt.list}
{SILENT}
{STACKLIST}
{SETTING status.var} or {RETURNING status.var}

where

expr evaluates to the command(s) to be executed. Multiple commands are separated by field marks.

var is a variable to receive captured output.

src.list is a select list variable holding a list to be passed into the executed command as list 0.

tgt.list is a variable to receive a returned select list. This variable may be used in a subsequent READNEXT to extract items from the list.

status.var is a variable to receive the value of @SYSTEM.RETURN.CODE after the command is executed.

The EXECUTE statement creates a new command level by starting a new version of the command processor and passing it the command line to be executed. On completion of the command, the command processor returns control to the calling program.

The value in expr may contain several commands delimited by field marks. These will be processed as a paragraph and may include DATA and LOOP constructs, etc.

An abort occurring in the command(s) processed by the EXECUTE statement is normally trapped by the command processor. The TRAPPING ABORTS qualifier to the EXECUTE statement causes aborts to return to the program issuing the EXECUTE without execution of the optional ON.ABORT paragraph. The @ABORT.CODE variable may be used to determine whether the command caused an abort to occur. This variable is initially zero and is reset to zero only by the EXECUTE statement.

The CAPTURING clause captures output that would otherwise have gone to the terminal or phantom log file, saving it in the named variable with field marks in place of newlines. Alternatively, all command output can be suppressed by use of the SILENT clause which is equivalent to using HUSH ON before the executed command and restoring the previous hush state afterwards.

The NO.TTY clause suppresses all @() function expansion in the executed command. Used with CAPTURING, this ensures that no control sequences will appear in the captured data.

The PASSLIST clause passes the list in the src.list select list variable into the executed command as list 0. If src.list is omitted, the default select list is passed to the executed command. If neither form
of **PASSLIST** is present, the behaviour depends on the setting of the **EXECUTE.CLEARLIST** option of the **$MODE** compiler directive. If this mode is not set, the default select list is passed into the executed command. If the mode is set, the default select list is cleared prior to execution of the command.

The **RTNLIST** clause returns the named variable as the content of the default select list after execution of the command.

The **STACKLIST** option saves the current state of the default select list (list 0) before executing the command and restores the list on return to the program, thus allowing the executed command to use the default select list internally. This option cannot be used with either **PASSLIST** or **RTNLIST**.

The **SETTING** or **RETURNING** clause copies the value of **@SYSTEM.RETURN.CODE** to the named variable after the command has been executed.

The unnamed common area is saved by the **EXECUTE** statement and the new command level may define a new structure for this area. On return from the **EXECUTE** statement, the original unnamed common area is restored. Named common areas are not affected by the **EXECUTE** statement. Except as influenced by the **PASSLIST** and **RTNLIST** options described above, the numbered select lists are carried into the executed command and any changes will be visible on return.

Application designers should carefully consider the possible impact of **EXECUTE** inside a transaction.

The value of the **STATUS()** function after an **EXECUTE** is undefined.

If the executed command directly or indirectly changes account by performing a **LOGTO**, the process will continue to run in the new account on return from the **EXECUTE**. Setting the **STACKED.ACCOUNT** mode of the **OPTION** command causes the system to revert to the previous account on completion of the **EXECUTE**.

**Examples**

```qmb
EXECUTE "LIST STOCK.FILE ITEMS QUANTITY"
```

This statement performs the **LIST** command from within the calling program. Control is returned to the program once the **LIST** command is complete.

```qmb
EXECUTE "SELECT CLIENTS WITH BALANCE > 1000" : @FM : RUN PAYMENT.LETTER
```

This statement executes two related commands in a single step.
EXIT

The **EXIT** statement terminates execution of a **LOOP/REPEAT** or **FOR/NEXT** structure.

**Format**

```
EXIT
```

The **EXIT** statement is equivalent to a jump to the statement following the **REPEAT** or **NEXT** statement of the innermost loop structure.

**Example**

```
LOOP
    REMOVE ITEM FROM STOCK SETTING DELIM
    ...processing statements...
    WHILE DELIM
        IF ITEM[1,1] = "A" THEN EXIT
        ...further statements...
    REPEAT
```

This program fragment processes items extracted from the STOCK dynamic array. If the value of ITEM commences with "A", the program exits from the loop.

**See also:**

**CONTINUE, FOR/NEXT, LOOP/REPEAT, WHILE, UNTIL**
EXIT.PACKAGE()

The **EXIT.PACKAGE()** function exits from a software package.

**Format**

```
EXIT.PACKAGE(package.name)
```

where

*package.name* is the name of the package.

This function returns True if it is successful, False if it fails. In the event of failure, the **STATUS()** function can be used to determine the cause.

**See also:**

Package licensing, **ENTER.PACKAGE()**, **PACKAGE.LICENCE**, **!PACKAGE()**
**EXP()**

The `EXP()` function returns the exponential (the natural anti-log) of a value.

**Format**

```
EXP(expr)
```

where

```
expr 
```
evaluates to a number or a numeric array.

The `EXP()` function returns the exponential of `expr`, that is, the mathematical constant e raised to the power `expr`. The value of e is about 2.71828.

If `expr` is a numeric array (a dynamic array where all elements are numeric), the `EXP()` function operates on each element in turn and returns a numeric array with the same structure as `expr`.

**Example**

```
N = EXP(X)
```

This statement finds the natural anti-log of X and assigns this to N.

**See also:**

`LN()`
EXPAND()

The **EXPAND()** function merges a data collection with a linked item.

**Format**

\[
\text{EXPAND}(\text{link.item}) \\
\text{EXPAND}(\text{link.item}, \text{fvar})
\]

where

- \( \text{link.item} \) is a reference to a data collection string element containing the link.
- \( \text{fvar} \) is the file variable to be used when resolving a link that does not specify a filename.

A data collection stored in a collection file may contain links to other collections in the same file or a different file. The links are stored as a string variable that contains a reference to the linked item as

\[
\text{filename:}id
\]

The \( \text{filename} \) (but not the colon) can be omitted if the link is to a record in a default file (perhaps the same file) but this reduced syntax has implications for the application as described below. There is nothing about this string item that defines it as a link except for how it is used by the application.

Where a data collection containing a link has been read into a QMBasic variable, the linked item can be joined onto its parent by use of the **EXPAND()** function

\[
\text{OK} = \text{EXPAND(VAR('link'))}
\]

where \( \text{link} \) is the element path of the string that defines the link.

If the link string uses the reduced syntax that has no \( \text{filename} \), the extended form of the **EXPAND()** function must be used where \( \text{fvar} \) is a file variable that references the file containing the linked item. This optional function argument is ignored when using the full syntax of the link string.

The **EXPAND()** function returns True if successful.

**See also:**

[Data collections](#)
**EXTRACT()**

The **EXTRACT()** function extracts a field, value or subvalue from a dynamic array.

**Format**

```
EXTRACT(dyn.array, field {, value {, subvalue}})
```

where

- `dyn.array` is the dynamic array from which the item is to be extracted.
- `field` evaluates to the number of the field to be extracted.
- `value` evaluates to the number of the value to be extracted. If omitted or zero, the entire field is extracted.
- `subvalue` evaluates to the number of the subvalue to be extracted. If omitted or zero, the entire value is extracted.

The specified field, value or subvalue is extracted from the dynamic array and returned as the result of the **EXTRACT()** function. This function is identical in effect to the alternative syntax of

```
dyn.array<field {, value {, subvalue}}>
```

**Example**

```
AMOUNT.DUE = EXTRACT(INVOICE.REC, 3)
```

This statement extracts field 3 from INVOICE.REC, assigning the result to variable AMOUNT.DUE.

**See also:**  
**DEL, DELETE(), FIND, FINDSTR, INS, INSERT(), LISTINDEX(), LOCATE, LOCATE(), REPLACE()**
**FCONTROL()**

The **FCONTROL()** function performs control actions on an open file.

**Format**

\[ \text{FCONTROL}(fvar, \text{action} \ {, \ qualifier}) \]

where

- **fvar** is the file variable associated with an open file.
- **action** identifies the action to be performed.
- **qualifier** is additional information relevant to the action.

The **FCONTROL()** function performs the action defined by **action** against the file open as **fvar**.

The **action** values are listed below using the tokens defined in the SYSCOM KEYS.H include record:

**FC$SYNC** (1) Controls or retrieves the state of forced writes for a hashed file. See synchronous (forced write) mode in the section that discusses dynamic files for more details.

If **qualifier** is False, forced writes are disabled. If qualifier is True, forced writes are enabled. For all values of **qualifier**, the previous state of forced writes is returned as the result of the function. Use a **qualifier** value of -1 to return the current state without updating it.

Note that, if the file is open to multiple file variables within the one process, this action affects all writes to the file regardless of which file variable is used.

**FC$SET.SEQ.KEY** (2) Sets the sequential number to be used as the record id of the next record written using the CREATING.SEQKEY option to the **WRITE** statement. Applies only to hashed files. Returns True if successful.

**FC$GET.SEQ.KEY** (3) Retrieves the sequential number to be used as the record id of the next record written, incrementing this value. This allows a program to reserve a sequential id without using the CREATING.SEQKEY option to the **WRITE** statement. Returns a null string if **fvar** does not reference a hashed file.

**FC$CRLF** (4) Controls or retrieves the line terminator mode to be used for a directory file or sequential file. If **qualifier** is a valid mode setting, the file is switched into the specified mode. For all values of **qualifier**, the previous state of the line terminator selection is returned as the result of the function. Use a **qualifier** value of -1 to return the current state without updating it. See directory files for a discussion of the mode values. The SYSCOM KEYS.H record defines tokens for the mode settings:

- 0  **FC$CRLF.LF**  LF terminator, CR is data character
- 1  **FC$CRLF.CRLF**  CRLF pair terminator, keep lone CR or LF
- 2  **FC$CRLF.BOTH**  CRLF or LF terminator, keep lone CR
- 3  **FC$CRLF.NOCR**  CRLF or LF terminator, discard all CR
FC$ACCESS (5)
Checks whether the user has access to the field position specified by \textit{qualifier}. If the file does not use data encryption, this action always returns True.

FC$VALID.ID (6)
Checks whether \textit{qualifier} is a valid record id for the file, returning a Boolean value.

Example

\begin{verbatim}
OPEN "AUDIT" TO AUD.F ELSE STOP "Cannot open AUDIT file"
VOID FCONTROL(AUD.F, FC$SYNC, @TRUE)
\end{verbatim}

This program fragment opens the AUDIT file and sets forced write mode. The same effect could have been achieved with the SYNC option to the \texttt{OPEN} statement.

See also:
\texttt{OPEN, OPENPATH}
FIELD()

The **FIELD()** function returns one or more delimited substrings from a string. The **FIELDS()** function is similar to **FIELD()** but operates on a multivalued *string*, returning a similarly structured dynamic array of results.

**Format**

```
FIELD(string, delimiter, occurrence {, count})
```

where

- **string**: is the string from which substrings are to be extracted.
- **delimiter**: evaluates to the delimiter character.
- **occurrence**: evaluates to the position of the substring to be extracted. If less than one, the first substring is extracted.
- **count**: evaluates to the number of substrings to be extracted. If omitted or less than one, one substring is extracted. Specifying the **count** as an asterisk extracts all elements of the string from the given start position.

The **FIELD()** function extracts **count** substrings starting at substring **occurrence** from **string**. Substrings within **string** are delimited by the first character of **delimiter**. If **delimiter** is a null string, the entire **string** is returned.

If the value of **occurrence** is greater than the number of delimited substrings in **string**, a null string is returned.

If the value of **count** is greater than the number of delimited substrings in **string** starting at substring **occurrence**, the remainder of **string** is returned. Additional delimiters are not inserted.

There is an alternative syntax,

```
string[delimiter, occurrence, count]
```

that performs exactly the same operation. In this for, the **count** is not optional. See [QMBasic Expressions and Operators](#).

The **COL1()** and **COL2()** functions can be used to find the character positions of the extracted substring. When using the **FIELDS()** function, the values returned by the **COL1()** and **COL2()** functions relate to the final extracted item.

Use of the **$MODE NOCASE.STRINGS** compiler directive makes the **delimiter** case insensitive.

**Examples**

```
A = "1*2*3*4*5"
S = FIELD(A, '*', 2, 3)
```

This program fragment assigns the string "2*3*4" to variable S.
A = "1*2*3*4*5"
S = FIELD(A, '*', 2, *)

This program fragment assigns the string "2*3*4*5" to variable S.

See also:
COL1(), COL2(), FIELDSTORE(), LAST()
FIELDSTORE()

The FIELDSTORE() function provides delimited substring assignment. It is closely related to the GROUPSTORE statement.

Format

FIELDSTORE(string, delimiter, i, n, rep.string)

where

- **string** evaluates to the string in which replacement is to occur. If string is a variable name, the contents of this variable are not changed unless it also appears on the left hand side of the assignment statement in which the FIELDSTORE() function appears.
- **delimiter** evaluates to a string, the first character of which is used as the delimiter separating the substrings within string. A null delimiter string will cause a run time error.
- **i** evaluates to the position of the first substring to be replaced. Substring positions are numbered from one.
- **n** evaluates to the number of substrings to be replaced.
- **rep.string** evaluates to the new data to be inserted in string.

The value returned by the FIELDSTORE() function is the result of the replacement.

A statement of the form

\[ S = \text{FIELDSTORE}(S, d, i, n, \text{rep.string}) \]

is equivalent to the delimited substring assignment operation

\[ S[d, i, n] = \text{rep.string} \]

The action of FIELDSTORE() depends on the values of **i** and **n** and the number of substrings within **rep.string**.

If the value of the position expression, **i**, is less than one, a value of one is assumed. If there are fewer than **i** delimited substrings present in **string**, additional delimiters are added to reach the required position.

If the value of the number of substrings expression, **n**, is positive, **n** substrings are replaced by the same number of substrings from **rep.string**. If **rep.string** contains fewer than **n** substrings, additional delimiters are inserted.

If the value of the number of substrings expression, **n**, is zero or negative, **n** substrings are deleted from **string** and the whole of **rep.string** is inserted regardless of the number of substrings that it contains.

Use of the $MODE NOCASE.STRINGS compiler directive makes the delimiter case insensitive.

Example
S = 1*2*3*4*5
A = FIELDSTORE(S, "*", 2, 3, "A*B")
B = FIELDSTORE(S, "*", 2, 3, "A*B*C")
C = FIELDSTORE(S, "*", 2, 3, "A*B*C*D")
D = FIELDSTORE(S, "*", 2, 0, "A*B")
E = FIELDSTORE(S, "*", 2, -3, "A*B")

This program fragment performs the `FIELDSTORE()` function on the string S using different values for `rep.string` and `n`. The results are

A = 1*A*B**5 Note inserted delimiter as `rep.string` has only 2 substrings
B = 1*A*B*C*5 `rep.string` replaces substrings 2 to 5
C = 1*A*B*C*5 Final substring of `rep.string` is not inserted
D = 1*A*B*2*3*4*5 No substrings are deleted as n is zero
E = 1*A*B*5 Three substrings are deleted and `rep.string` is inserted

See also: `FIELD()`, `GROUPSTORE`
**FILE**

The **FILE** statement provides a way to reference data in files using field names defined in the dictionary.

This statement is provided for compatibility with other systems and its use is discouraged.

**Format**

```
FILE name, ...
```

where

```
name    is a comma separated list of files to be processed
```

The **FILE** statement opens the named files for processing in the current program. The QMBasic compiler will process the associated dictionary to allow use of field names as described below. Note that any change to the dictionary may require the program to be recompiled.

Note that the **FILE** statement has no error handling clause. If the file cannot be opened, any subsequent access to it will fail.

After a file has been opened using the **FILE** statement, a record can be read using a modified form of the **READ** statement:

```
READ name FROM id
```

where **name** is the file name in the **FILE** statement and **id** is the key of the record to be read. This form of the **READ** statement take the same optional clauses as a normal **READ**. The locking variants (**READL** and **READU**) are also supported.

After a record has been read, data from that record can be accessed using a construct:

```
name(field)
```

where **name** is the file name and **field** is a field name defined in the dictionary. Note that QMBasic restricts this construct such that **field** must be a D-type item or an A or S-type item with no correlative. Calculated values are not supported.

Records may be written to the file using a modified form of the **WRITE** (or **WRITEU**) statement:

```
WRITE name TO id
```

where **name** is the file name in the **FILE** statement and **id** is the key of the record to be written.

Apart from the above special cases, use of **name** in a file handling statement or function refers to the file variable associated with the file. Use of **name** in any other context refers to the dynamic array that will be used implicitly by **READ** or **WRITE** statements.

Use of the **FILE** statement is not recommended in new developments as it requires that the file is opened separately in every program that will access it rather than using a file variable in a common block for best performance. The syntax of the associated statements is also likely to be confusing as the same name references either the file variable or the data depending on its context.

**Example**

```
FILE 'ORDERS'
```
READ ORDERS FROM ORDER.NO THEN
   DISPLAY ORDERS(CUSTOMER.NO)
END

This program fragment opens the ORDERS file, reads the record for a given order number and displays the customer number from this record.
FILE.EVENT

The FILE.EVENT() function creates a file event monitoring variable (Windows only).

Format

    FILE.EVENT(path, event)

where

    path is the pathname of the item to be monitored.

    event is the event to be trapped.

The FILE.EVENT() function allows a program to wait for a change within the Windows file system. It can be used, for example, to wait until a new file is created within a directory without needing to write a loop that periodically compares the directory content with a previously recorded list of items.

Path is the pathname of the directory to which the monitoring is to be applied. Event is any combination of the following additive tokens defined in the SYSCOM KEYS.H record:

- FES$FILE.NAME Any file name change in the watched directory or sub-tree. Changes include renaming, creating, or deleting a file name.
- FES$DIR.NAME Any directory-name change in the watched directory or sub-tree. Changes include creating or deleting a directory.
- FES$ATTRIBUTES Any attribute change in the watched directory or sub-tree.
- FES$SIZE Any file-size change in the watched directory or sub-tree. The operating system detects a change in file size only when the file is written to the disk. For operating systems that use extensive caching, detection occurs only when the cache is sufficiently flushed.
- FES$LAST.WRITE Any change to the last write time of files in the watched directory or sub-tree. The operating system detects a change to the last write-time only when the file is written to the disk. For operating systems that use extensive caching, detection occurs only when the cache is sufficiently flushed.
- FES$LAST.ACCESS Any change to the last access time of files in the watched directory or sub-tree. The operating system detects a change to the last write-time only when the file is written to the disk. For operating systems that use extensive caching, detection occurs only when the cache is sufficiently flushed.
- FES$CREATION Any change to the creation time of file in the watched directory or sub-tree.
- FES$SECURITY Any security-descriptor change in the watched directory or sub-tree causes a change notification wait operation to return.
- FES$SUBTREE Apply monitoring to the specified directory and all sub-directories.
If the `FILE.EVENT()` function is successful, it returns a value that can be stored in a variable as a file event monitoring action for use with `WAIT.FILE.EVENT()` and `STATUS()` will be zero. Use of the returned value in any other way will appear to be a number.

If the function is unsuccessful, it returns -1 and `STATUS()` will be `ER$FAILED`.

See `WAIT.FILE.EVENT()` for an example of use of this function.
FILEINFO()

The FILEINFO() function returns information about an open file.

Format

FILEINFO(file.var, key)

where

file.var is the file variable associated with the file.

key identifies the action to be performed.

Values for the key to the FILEINFO() function are defined in the KEYS.H record in the SYSCOM file. These are

0 FLSOPEN Check if file is open. Returns True if file.var is associated with an open file, False if it is not.
1 FLSVOCNAME Returns the VOC name used to open the file. For a file opened via QMNet, this is the server:account:file form of the file name.
2 FLSPATH Returns pathname of open file.
3 FLSTYPE File type. Returns one of
  FLSTYPE.DH (3) - Dynamic file, including data collections
  FLSTYPE.DIR (4) - Directory file
  FLSTYPE.SEQ (5) - Sequential file
  FLSTYPE.VFS (6) - Virtual file system
  FLSTYPE.DIST (7) - Distributed file
5 FLSMODULUS File modulus (dynamic files only)
6 FLSMINMOD Minimum modulus (dynamic files only)
7 FLSGRPSIZE Group size in multiples of 1024 bytes (dynamic files only)
8 FLSLARGEREC Large record size (dynamic files only)
9 FLSMERGE Merge load percentage (dynamic files only)
10 FLSPLIT Split load percentage (dynamic files only)
11 FLSLOAD Current load percentage (dynamic files only)
13 FLSAK File has AK indices (dynamic files only)
14 FLSLINE Line to read or write next (sequential files only)
15 FLSPARTS Field mark delimited list of part file numbers (distributed files)
1000 FLSLOADBYTES Current load bytes (dynamic files only)
1001 FLSREADONLY Returns True is file is read-only
1002 FLSTRIGGER Returns trigger function name, null if none
1003 FLSPHYSBYTES Returns total size of file, excluding indices
1004 FLSVERSION Internal file version (dynamic files only)
1005 FLSTATS.QUERY Returns True if file statistics gathering is enabled
1006  FL$SEQPOS  File position (sequential files only)
1007  FL$TRG.MODES  Returns mode flags for trigger function
1008  FL$NOCASE  Returns True if the file uses case insensitive ids.
1009  FL$FILENAME  Returns the internal file number for file.var. This may change each time the file is opened.
1011  FL$AKPATH  Returns the pathname of the alternate key index directory. This is a null string if the indices are in their default location.
1012  FL$ID  Returns the id of the last record read from the file. Used with a dynamic file that is configured for case insensitive record ids, this will return the actual id as stored in the file, which may differ in casing from that supplied in the associated READ statement.
1013  FL$STATUS  Returns a dynamic array as for the STATUS statement.
1014  FL$MARK.MAPPING  Returns True if mark mapping is enabled, False if not (directory files).
1015  FL$RECORD.COUNT  Returns a count of the number of records in the file (dynamic files only). This count may be incorrect if the file was not closed in the event of an abnormal process termination such as a system failure. The counter will be corrected by use of a select operation during which there were no updates to the file or by use of the QMFix utility. The value will be returned as -1 until the count is set or for non-dynamic files.
1016  FL$PRI.BYTES  Physical size of the primary subfile in bytes (dynamic files only). This figure will include space previously used by groups that have been discarded as the result of a merge operation.
1017  FL$OVF.BYTES  Physical size of the overflow subfile in bytes (dynamic files only). This figure will include space previously used by overflow blocks that are no longer active and are retained for future use.
1018  FL$NO.RESIZE  Is resizing inhibited on this file? See the description of dynamic files for more information.
1019  FL$UPDATE  Returns the file update counter. This counter, shared across all users of the file, is initially set to 1 when a file is first opened and is incremented by every write, delete or clear file operation. It can be used to detect whether a file has been updated by another process.
1020  FL$ENCRYPTED  Returns True if the file uses encryption, False otherwise.
1021  FL$WHO  Returns field mark delimited list of QM user numbers of users that have the file open. The process that executes the FILEINFO() function will not appear in this list if file.var is the only file variable referencing the file.
1022  FL$NETFILE  Returns True if this is a QMNet file, otherwise returns False.
1023  FL$SEQTYPE  Sequential file sub-type:
    FL$SEQTYPE.PORT (1) - Serial port
    FL$SEQTYPE.CHDEV (2) - Character device
    FL$SEQTYPE.FIFO (3) - FIFO
    FL$SEQTYPE.DRIVE (4) - Disk drive or other block device
1024  FL$REPLICATED  Returns a field mark delimited list of replication targets.
Returns True if this is a directory file with character translation of record ids suppressed.

Returns True if this is a hashed file created in ECS mode.

Returns the ECS map name associated with the file.

Returns the data encoding name for a directory file.

Returns the part number returned by the last evaluation of the partitioning algorithm.

File supports index scanning operations (SETLEFT, SETRIGHT, SELECTLEFT, SELECTRIGHT). Primarily intended for detection of VFS handlers that do not provide this functionality.

Returns True if the file is a data collection.

Returns the owner of the file (Windows only)

Returns true if file.var references a temporary file.

Returns True if file is updated in the current transaction.

Returns a field mark delimited list of locks owned by the current user in this file. Each entry is formed from two values; the lock type code and the record id.

File uses data integrity constraints?

Example

IF FILEINFO(FVAR, FL$TYPE) # FL$TYPE.DH THEN
   ABORT "Dynamic file required"
END

This program fragment checks whether a file variable is associated with a dynamic file and, if not, aborts.

See also:
STATUS
FILELOCK

The **FILELOCK** statement sets a file lock on a file.

**Format**

```
FILELOCK file.var [ON ERROR statement(s)]
[LOCKED statement(s)]
[THEN statement(s)]
[ELSE statement(s)]
```

where

- `file.var` is the file variable associated with the file.
- `statement(s)` are statements to be executed depending on the outcome of the operation.

The **FILELOCK** statement sets a file lock which prevents other users from obtaining read or update locks on records within the file or a file lock on the whole file. File access operations that do not obtain locks are can still be performed by other users. These are **CLEARFILE**, **DELETE**, **DELETEU**, **MATREAD**, **MATWRITE**, **MATWRITEU**, **READ**, **READY**, **WRITE**, **WRITEU**, **WRITEV**, and **WRITEVU**. Correct application design avoids accidental deletion or overwriting of locked records.

A file lock does not prevent access by the user that owns the lock.

The **LOCKED** clause is executed if another user holds a file lock or a read or update lock on any record in the file. The **STATUS()** function will return the user id of a process holding a lock. If the **LOCKED** clause is omitted, the program will wait for any lock to be released.

The **THEN** and **ELSE** clauses are both optional and neither need be present. The **THEN** clause will be executed if the file lock is successfully obtained. The **ELSE** clause will be executed if the lock cannot be obtained for a reason other than it being held by some other user. There are currently no conditions that would execute the **ELSE** clause.

**Example**

```
FILELOCK STOCK
SELECT STOCK
TOTAL = 0
LOOP
 _READNEXT ID ELSE EXIT
  READ REC FROM STOCK, ID ELSE ABORT "Cannot read " : ID
  TOTAL += REC<QTY>
_REPEAT
FILEUNLOCK STOCK
```

This program fragment obtains a file lock on the file open as STOCK and then reads all records from the file, forming a total of the values in the QTY field. The lock prevents other users obtaining update locks when they might be updating this field in some record. The lock ensures that the total value represents a true picture of the file when the file lock was obtained. The lock is released on leaving the main processing loop.
See also:
Locking, FILEUNLOCK
FILEUNLOCK

The FILEUNLOCK statement releases a file lock previously obtained using the FILELOCK statement.

Format

FILEUNLOCK file.var {ON ERROR statement(s)}
{THEN statement(s)}
{ELSE statement(s)}

where

file.var is the file variable associated with the file.
statement(s) are statements to be executed depending on the outcome of the operation.

The FILEUNLOCK statement releases a file lock, making the file available to other users. Read and update locks on records from the file are not affected.

The ON ERROR clause is executed in the event of a fatal error. The STATUS() function will return an error code giving the cause of the error.

Where the ON ERROR clause is not taken, the STATUS() function returns 0 if the file was previously locked by this user, ERSLCK if the lock is held by another user or ER$NLK if no user holds the lock.

The THEN and ELSE clauses are both optional and neither need be present. The THEN clause will be executed if the file lock is successfully released. The ELSE clause will be executed if the lock cannot be released. Currently, this will only occur if the lock is not owned by the process executing the FILEUNLOCK.

Example

FILELOCK STOCK
SELECT STOCK
TOTAL = 0
LOOP
   READNEXT ID ELSE EXIT
   READ REC FROM STOCK, ID ELSE ABORT "Cannot read " : ID
   TOTAL += REC<QTY>
REPEAT
FILEUNLOCK STOCK

This program fragment obtains a file lock on the file open as STOCK and then reads all records from the file, forming a total of the values in the QTY field. The lock prevents other users obtaining update locks when they might be updating this field in some record. The lock ensures that the total value represents a true picture of the file when the file lock was obtained. The lock is released on leaving the main processing loop.

See also:
Locking, FILELOCK
**FIND**

The **FIND** statement searches a dynamic array for a given string in any position.

**Format**

```
FIND string IN dyn.array {, occurrence} {SETTING field{, value {, subvalue}}} {THEN statement(s)} {ELSE statement(s)}
```

where

- `string` evaluates to the item to be located.
- `dyn.array` is the dynamic array in which searching is to occur.
- `occurrence` is the occurrence of `string` to be found. If omitted, the first occurrence is located.
- `field` is the variable to receive the field number at which `string` is found.
- `value` is the variable to receive the value number at which `string` is found. If omitted, only the field position is returned.
- `subvalue` is the variable to receive the subvalue number at which `string` is found. If omitted, only the field and, optionally, value positions are returned.
- `statement(s)` are statements to be executed depending on the outcome of the **FIND** action.

At least one of the **THEN** and **ELSE** clauses must be present.

The **FIND** statement searches `dyn.array` for a field, value or subvalue equal to `string`. If found, the position of `string` within `dyn.array` is returned in the `field`, `value`, and `subvalue` variables and the **THEN** clause is executed. If not found or `dyn.array` is a null string, the `field`, `value`, and `subvalue` variables are unchanged and the **ELSE** clause is executed.

Use of the **$MODE NOCASE.STRINGS** compiler directive makes the comparison case insensitive.

**Example**

Variable X contains A FM BV C VM D FM VM E FM G

```
FIND 'D' IN X SETTING F, V, S
```

The above FIND would return F = 2, V = 3, S = 1

**See also:**
**DEL, DELETE(), EXTRACT(), FINDSTR, INS, INSERT(), LISTINDEX(), LOCATE, LOCATE(), REPLACE()**
FINDSTR

The FINDSTR statement searches a dynamic array for a given substring in any position.

Format

```
FINDSTR string IN dyn.array {, occurrence} {SETTING field{, value {, subvalue}}} }
{THEN statement(s)}
{ELSE statement(s)}
```

where

- `string` evaluates to the item to be located.
- `dyn.array` is the dynamic array in which searching is to occur.
- `occurrence` is the occurrence of `string` to be found. If omitted, the first occurrence is located.
- `field` is the variable to receive the field number at which `string` is found.
- `value` is the variable to receive the value number at which `string` is found. If omitted, only the field position is returned.
- `subvalue` is the variable to receive the subvalue number at which `string` is found. If omitted, only the field and, optionally, value positions are returned.
- `statement(s)` are statements to be executed depending on the outcome of the FINDSTR action.

At least one of the THEN and ELSE clauses must be present.

The FINDSTR statement searches `dyn.array` for a field, value or subvalue containing `string`. This need not be the entire field, value or subvalue. If found, the position of `string` within `dyn.array` is returned in the `field`, `value`, and `subvalue` variables and the THEN clause is executed. If not found or `dyn.array` is a null string, the `field`, `value`, and `subvalue` variables are unchanged and the ELSE clause is executed.

Use of the $MODE NOCASE.STRINGS compiler directive makes the comparison case insensitive.

Example

Variable X contains ABCDEFVMGHVMJKLVMMNOVMPQRSTU

```
FINDSTR 'KL' IN X SETTING F, V, S
```

The above FINDSTR would return F = 2, V = 3, S = 1

See also:
DELE(), DELETE(), EXTRACT(), FIND, INS, INSERT(), LISTINDEX(), LOCATE, LOCATE(), REPLACE()
**FIX()**

The **FIX()** function returns a string representation of a floating point value with a specified precision and optional truncation.

**Format**

```
FIX(expr {, prec{, truncate} })
```

where

- `expr` evaluates to the data to be formatted
- `prec` evaluates to the required precision
- `truncate` A Boolean value that determines whether the value should be rounded or truncated.

The **FIX()** function converts a floating point value to a character string.

The `prec` argument determines the number of decimal places in the result. If omitted, the current value of the program **PRECISION** is used. A precision less than 0 is treated as 0. A precision greater than 14 is treated as 14.

The `truncate` argument controls rounding. A value of False (0 or a null string) or omitting the argument causes rounding to be applied. Any other value causes truncation.

**Examples**

```
V = 12.3456789
DISPLAY FIX(V, 3)    12.346
DISPLAY FIX(V, 3, 1)  12.345
```
FLUSH

The **FLUSH** statement flushes the internal buffers for a directory file record previously opened for sequential access.

**Format**

```
FLUSH file.var
{THEN statement(s)}
{ELSE statement(s)}
```

where

- **file.var** is the file variable associated with the record by a previous **OPENSEQ** statement.
- **statement(s)** are statement(s) to be executed depending on the outcome of the **FLUSH**.

At least one of the **THEN** and **ELSE** clauses must be present. The **THEN** clause is executed if the operation is successful. The **ELSE** clause is executed if the **FLUSH** operation fails.

The sequential file buffers are automatically flushed periodically during file usage and on closing the file. The **FLUSH** statement allows the programmer to ensure that data is flushed to disk at a given point in the program.

The **WRITESEQF** statement is the equivalent of a **WRITESEQ** followed by a **FLUSH**. Flushing the buffers after every write may result in poor performance.

**Example**

```
LOOP   LINE = REMOVE(LIST, CODE)   WRITESEQ LINE TO SEQ.F ELSE ABORT "Write error"
   WHILE CODE
   REPEAT
   FLUSH SEQ.F ELSE ABORT "Flush error"
```

This program fragment writes a series of text lines extracted from LIST to a sequential file and then flushes the buffers to verify that the data has been written to disk.

**See also:**

The FLUSH.DH.CACHE statement flushes the dynamic file cache.

Format

FLUSH.DH.CACHE [ LOCAL | NO.WAIT]

To improve performance of applications that repeatedly open and close the same files, closing a dynamic hashed file does not immediately close it at the operating system level. Instead, the file is moved into a cache of recently closed files from where it can be reopened at the application level very quickly. The size of this cache is determined by the DHCACHE configuration parameter. On reaching this limit, the oldest file in the cache is closed at the operating system level.

Sometimes an application may do something that will fail if the file is still open in the cache, perhaps in another process. Built-in QM operations such as using OSDELETE to delete a file will automatically flush the cache in all QM processes. Where the operation is not a built-in part of QM, the FLUSH.DH.CACHE statement can be used to cause all QM processes to close files that are in their cache.

Note that this action is asynchronous and it may take a few seconds for the request to be handled in all other QM processes. The FLUSH.DH.CACHE statement will wait for other processes to acknowledge that they have seen the request, with a timeout after four seconds. The NO.WAIT qualifier causes the program to continue without waiting.

When used with the LOCAL option, only the file cache of the process in which this statement appears is flushed.

Excessive use of the FLUSH.DH.CACHE statement can have a detrimental effect on performance.
FMT()

The FMT() function performs data formatting according to a format template. It is typically used to convert data for display or printing. The FMTS() function is identical to FMT() except that it works on each element of a dynamic array in turn, returning the result in a similarly delimited dynamic array.

Format

FMT(expr, fmt.spec)
FMTS(expr, fmt.spec)

where

expr evaluates to the data to be formatted
fmt.spec evaluates to the format specification.

The FMT() function sets the STATUS() function value to indicate whether the operation was successful. Possible values are

0 Successful formatting.
1 Data to format was too long or invalid for the format specification.
2 The format specification was invalid.

Operations that result in a non-zero STATUS() value return expr as the function result.

Shortform Notation

For compatibility with other systems, the FMT() function action can also be performed in QMBasic programs and I-type dictionary entries by use of a shortform notation in which the expr and fmt.spec are simply written next to each other with no operator in between.

Thus

X = FMT(A, '8R')
can be written as
X = A '8R'
FMTDW(), FMTDWS()

The FMTDW() function performs data formatting according to a format template based on display width. The FMTDWS() function is identical to FMTDW() except that it works on each element of a dynamic array in turn, returning the result in a similarly delimited dynamic array.

Format

\[
\text{FMTDW} (\text{expr}, \text{fmt.spec}) \\
\text{FMTDWS} (\text{expr}, \text{fmt.spec})
\]

where

- \( \text{expr} \) evaluates to the data to be formatted
- \( \text{fmt.spec} \) evaluates to the format specification.

The FMT() function sets the STATUS() function value to indicate whether the operation was successful. Possible values are

- 0 Successful formatting.
- 1 Data to format was too long or invalid for the format specification.
- 2 The format specification was invalid.

Operations that result in a non-zero STATUS() value return expr as the function result.

See also:
FMT(), FMTS(), format specifications
FOLD() and FOLDS()

The FOLD() function breaks a string into field mark delimited sections no longer than a given width, placing breaks on spaces where possible.

Format

\[
\text{FOLD(string, width \{, delim\})} \\
\text{FOLDS(string, width \{, delim\})}
\]

where

- \( string \) evaluates to the string to be formatted
- \( width \) evaluates to the maximum length for each fragment.
- \( delim \) evaluates to the delimiter character to appear between each fragment.

The FOLD() function breaks \( string \) into sections, placing the \( delim \) character between each section. Each section is at most \( width \) characters in length with the break from one section to the next occurring at a space where possible.

The \( width \) argument may be multivalued. In this case, the first value specifies the width for the first fragment of the result, the second value specifies the width for the second fragment of the result and so on. If there are more fragments in the result than there are width specifications, the final width is used for the remaining data.

The \( delim \) argument is optional. If omitted, a field mark is used by default. Specifying \( delim \) as a null string uses a value mark as the delimiter.

The FOLDS() function is similar to FOLD() but works on each field, value or subvalue of \( string \) separately, returning a similarly structured dynamic array of folded strings.

Example

\[
S = 'The quick brown fox jumps over the lazy dog' \\
X = FOLD(S, 10) \\
\text{LOOP} \\
\quad \text{CRT REMOVE(X, CODE)} \\
\quad \text{WHILE CODE} \\
\quad \text{REPEAT}
\]

The above program fragment prints

```
The quick  
  brown fox  
  jumps over 
  the lazy   
   dog
```

See also:

FOLDDW()
FOLDDW(), FOLDDWS()

The FOLDDW() function breaks a string into field mark delimited sections no longer than a given display width, placing breaks on spaces where possible.

Format

FOLDDW(string, width {, delim})
FOLDDWS(string, width {, delim})

where

string evaluates to the string to be formatted
width evaluates to the maximum length for each fragment as a display width.
delim evaluates to the delimiter character to appear between each fragment.

The FOLDDW() function breaks string into sections, placing the delim character between each section. Each section has a display width not exceeding width with the break from one section to the next occurring at a space where possible.

The width argument may be multivalued. In this case, the first value specifies the width for the first fragment of the result, the second value specifies the width for the second fragment of the result and so on. If there are more fragments in the result than there are width specifications, the final width is used for the remaining data.

The delim argument is optional. If omitted, a field mark is used by default. Specifying delim as a null string uses a value mark as the delimiter.

The FOLDDWS() function is similar to FOLDDW() but works on each field, value or subvalue of string separately, returning a similarly structured dynamic array of folded strings.

On a non-ECS system, the FOLDDW() and FOLDDWS() functions are identical to FOLD() and FOLDS().

Example

S = 'The quick brown fox jumps over the lazy dog'
X = FOLDDW(S, 10)
LOOP
   CRT REMOVE(X, CODE)
   WHILE CODE
      REPEAT

The above program fragment prints

The quick
brown fox
jumps over
the lazy
dog
See also:

DISPLAY.WIDTH(), FOLD()
FOOTING

The FOOTING statement defines text to be printed or displayed at the foot of each page of output.

Format

```
FOOTING {ON print.unit} text
```

where

- `print.unit` identifies the logical print unit in the range -1 to 255 to which the footing text is to be applied. If omitted, the default print unit (unit 0) is used.
- `text` is the footing text. This may include control tokens as described below.

The FOOTING statement defines the text of a page footing and, optionally, control information determining the manner in which the text is output. A page footing is output whenever the bottom of the page is reached or on execution of a PAGE statement to terminate the current page.

The footing `text` may include the following control tokens enclosed in single quotes. Multiple tokens may appear within a single set of quotes.

- **C** Centers text on the line.
- **D** Inserts the date. The default format is dd mmm yyyy (e.g. 24 Aug 2005) but can be changed using the DATE FORMAT command.
- **G** Insert a gap. Spaces are inserted in the footing line at the position of each G control token such that the overall length of the line is the same as the printer unit width. A single use of the G token will right justify the subsequent text. Multiple G tokens will distribute spaces as evenly as possible.

  When a footing line uses both G and C, the footing is considered as a number of elements separated by the G control options. The element that contains the C option will be centered. The items either side of the centered element are processed separately when calculating the number of spaces to be substituted for each G option.

- **Hn** Sets horizontal position (column) numbered from one. Use of H with C or with a preceding G token may have undesired results.
- **L** Start a new line
- **N** Inhibit automatic display pagination
- **O** Reverses the elements separated by G tokens in the current line on even numbered pages. This is of use when printing double sided reports.
- **Pn** Insert page number. The page number is right justified in \( n \) spaces, widening the field if necessary. If omitted, \( n \) defaults to four.
- **Sn** Insert page number. The page number is left justified in \( n \) spaces, widening the field if necessary. If omitted, \( n \) defaults to one.
- **T** Inserts the time and date in the form hh:mm:ss dd mmm yyyy. The format of the date component can be changed using the DATE FORMAT command.
Unrecognised control tokens are ignored. A quotation mark may be inserted in the printed text by using two adjacent quotation marks in the text string.

There is no limit to the length of a footing text. Each line will be truncated at the width of the print unit. The effect of using a footing which will not fit on to the physical page is undefined.

See the RUN command for a discussion of how the page end pause on output to the screen may be affected by setting a page footing.

See also:
HEADING, PAGE
FOR / NEXT

The **FOR / NEXT** statement defines a group of statements to be executed with an iterative control variable.

**Format**

```
FOR var = start.expr TO limit.expr {STEP step.expr}
{WHILE expr}
{UNTIL expr}
    ...statement(s)...
NEXT {var}
```

```
FOR var = value1, value2, value3...
{WHILE expr}
{UNTIL expr}
    ...statement(s)...
NEXT {var}
```

```
FOR EACH var IN string {DELIMITER delim}
{WHILE expr}
{UNTIL expr}
    ...statement(s)...
NEXT {var}
```

where

- **var** is the loop control variable.
- **start.expr** evaluates to the value to be placed in **var** for the first iteration of the loop.
- **limit.expr** evaluates to the value beyond which **var** must not pass.
- **step.expr** evaluates to the value by which **var** should be incremented between iterations of the loop. If omitted, **step.expr** defaults to one.
- **value1, value2...** are values to be assigned to **var**.
- **string** is an expression that evaluates to a mark character delimited list of values to be assigned to **var**.
- **delim** is the delimiting character separating entries in the list. If omitted, any mark character is treated as the delimiter.
- **statement(s)** are statement(s) to be executed within the loop.

The first form of the **FOR / NEXT** statement executes **statement(s)** for values of **var** from **start.expr** to **limit.expr** in increments of **step.expr**. If **step.expr** is positive, the loop continues while the value of **var** is less than or equal to **limit.expr**. If **step.expr** is negative, the loop continues while the value of **var** is greater than or equal to **limit.expr**. The value of **var** on leaving the loop is the last value for which the loop was executed or **start.expr** if the initial value was already out of range.
The value of \( \text{var} \) should not be changed within the loop as this may lead to unexpected results.

Use of non-integer values for \( \text{start.expr} \), \( \text{limit.expr} \) and \( \text{step.expr} \) is not recommended where rounding errors in incrementing \( \text{var} \) and the loop termination comparison may lead to unexpected effects.

For best performance, \( \text{limit.expr} \) and \( \text{step.expr} \) should be constants or simple variable references as they are evaluated for every iteration of the loop. In the unlikely case that \( \text{limit.expr} \) or \( \text{step.expr} \) use the value of \( \text{var} \), note that these expressions are evaluated before the first cycle of the loop assigns the start value to \( \text{var} \).

The second form of the \texttt{FOR / NEXT} statement executes the loop for each of the supplied values of \( \text{var} \) in turn. The value of \( \text{var} \) on leaving the loop will be the final value for which the loop was executed. The values to be assigned to \( \text{var} \) may be constants or expressions.

The third form of the \texttt{FOR / NEXT} statement executes the loop for each delimited value in \( \text{string} \). The optional \texttt{DELIMITER} clause specifies the character that separates items in the list. If omitted, any mark character will be treated as the delimiter. This form of \texttt{FOR / NEXT} is broadly equivalent to using the \texttt{REMOVE} or \texttt{REMOVEF} statement to extract successive values from \( \text{string} \). The value of \( \text{string} \) is copied to an internal variable at the start of the loop and hence changing it inside the loop will have no effect. The value of \( \text{var} \) on leaving the loop will be the final value for which the loop was executed.

If \( \text{var} \) is present in the \texttt{NEXT} statement it must be the same \( \text{var} \) as in the \texttt{FOR} statement at the head of the loop. Use of \( \text{var} \) in the \texttt{NEXT} statement aids program readability and is checked by the compiler for correct matching of \texttt{FOR} and \texttt{NEXT} statements.

The \texttt{WHILE}, \texttt{UNTIL} and \texttt{EXIT} statements can be used with the \texttt{FOR / NEXT} loop to provide another loop termination control. The \texttt{CONTINUE} statement causes a jump to the start of the next iteration. Any number of these may be included in the loop.

\texttt{FOR / NEXT} loops may be nested to any depth. Jumping to a label within a \texttt{FOR / NEXT} loop from outside may have undefined effects.

The \texttt{FOR / NEXT} construct is semantically different in the various multivalue database environments. Some, including QM, test the end condition of the loop before storing the updated control variable such that a loop

\begin{verbatim}
FOR I = 1 TO 10
\end{verbatim}

would exit with I set to 10. Others store the value before testing the end condition such that this same loop would exit with I set to 11.

The \texttt{FOR.STORE.BEFORE.TEST} option of the \texttt{\$MODE} compiler directive can be used to modify the behaviour of \texttt{FOR / NEXT} constructs to store the new value of the control variable before testing for the end condition. For ease of maintenance and portability, it is recommended that programs should not rely on the value of the control variable after the loop has terminated.

**Examples**

\begin{verbatim}
FOR I = 1 TO 10 STEP 2
  DISPLAY I
NEXT I
\end{verbatim}
This program fragment displays the values 1, 3, 5, 7 and 9. The final value of I on leaving the loop is 9.

```
FOR I = 1 TO 20
UNTIL A(I) < 0
    DISPLAY A(I)
NEXT I
```

This program fragment displays elements of matrix A. The loop terminates if an element is found with a negative value.

```
FOR I = 1, 7, 22
    DISPLAY I
NEXT I
```

This program fragment displays the numbers 1, 7 and 22.

```
FOR EACH X IN SALES.REC<S.ITEM>
    DISPLAY X
NEXT X
```

This program fragment walks though the values in SALES.REC<S.ITEM>, displaying each value. Note that the actual mark character separating the items in the list is not significant and cannot be determined from within the loop.

See also: 
CONTINUE, EXIT, LOOP/REPEAT, WHILE, UNTIL
FORMCSV()

The FORMCSV() function transforms a string to a CSV standard compliant item.

Format

\[
\text{FORMCSV}(\text{string})
\]

where

\[
\text{string} \quad \text{is the string to be transformed}
\]

The FORMCSV() function converts the supplied \textit{string} to comma separated variable format, treating each field as a separate item.

Three modes of conversion are supported:

1. Output conforms to the CSV format specification (RFC 4180). Items containing double quotes or the delimiter character are enclosed in double quotes with embedded double quotes replace by two adjacent double quotes.

2. Encloses all non-null values in double quotes except for numeric items that do not contain a comma. Embedded double quotes are replaced by two adjacent double quotes.

3. Encloses all values in double quotes. Embedded double quotes are replaced by two adjacent double quotes.

Mode 1 is used by default. This can be changed by use of the CSV.MODE statement. Any change persists through subroutine calls but the mode is reset to 1 in a new command processor level from use of \texttt{EXECUTE}, restoring the previous mode on return from the executed command.

See also: \texttt{CSV.MODE, CSVDQ()}, \texttt{DPARSE.CSV, READCSV, WRITECSV}
FORMLIST, FORMLISTV

The FORMLIST statement creates a numbered select list from a dynamic array. The FORMLISTV statement is similar but creates a select list variable.

Format

FORMLIST dyn.array {TO list.no}

FORMLISTV dyn.array TO var

where

dyn.array evaluates to the list of items to form the select list.
list.no evaluates to the select list number. If omitted, select list zero is created.
var is the select list variable to be created.

Any existing select list list.no or in var is cleared and a new list is created from the elements of dyn.array. This list may be separated by field marks, item marks or a mixture of both.

Examples

READ REC FROM ACCOUNTS, "OVERDUE" THEN FORMLIST REC

This program fragment reads a list from record OVERDUE of file ACCOUNTS and creates select list zero from the elements of this list.

READ REC FROM ACCOUNTS, "OVERDUE" THEN FORMLIST REC TO 2

This program fragment is similar but creates select list 2.

READ REC FROM ACCOUNTS, "OVERDUE" THEN FORMLISTV MYLIST

This program fragment is again similar but creates MYLIST as a select list variable.

See also:

READLIST
FUNCTION

The FUNCTION statement introduces a user written.

Format

```
FUNCTION name{(arg1 {}, arg2...}) {VAR.ARGS}
```

where

- `name` is the name of the function.
- `arg1`, etc are the names of the arguments to the function.

QMBasic programs should commence with a PROGRAM, SUBROUTINE, FUNCTION or CLASS statement. If none of these is present, the compiler behaves as though a PROGRAM statement had been used with `name` as the name of the source record.

The FUNCTION statement must appear before any executable statements. The brackets are optional if there are no arguments. The FUNCTION statement may be split over multiple lines by breaking after a comma.

The name used in a FUNCTION statement need not be related to the name of the source record though it is recommended that they are the same as this eases program maintenance. The name must comply with the QMBasic name format rules.

A function is a special form of subroutine. It returns a value to the calling program and can be referenced in a QMBasic statement in the same way as the intrinsic functions described in this manual, without the need for an explicit CALL statement. The FUNCTION statement is equivalent to a SUBROUTINE statement with an additional hidden argument at the start which is used to return the result.

Variable Length Argument Lists and Default Values

The number of arguments in calls to the function must be the same as in the FUNCTION statement unless the function is declared with the VAR.ARGS option. When VAR.ARGS is used, any arguments not passed by the caller will be unassigned. The ARG.COUNT() function can be used to determine the actual number of arguments passed, excluding the hidden return argument. The ARG.PRESENT() function can be used to test for the presence of an optional argument by name.

```
FUNCTION CREDIT.RATING(CLIENT, CLASS = 1, CODE = "Standard") VAR.ARGS
```

In this example, if the calling program supplies only one argument, the CLASS and CODE variables will be unassigned. If the calling program provides two arguments, the CODE variable will be unassigned.

When using VAR.ARGS, default values may be provided for any arguments by following the argument name with an = sign and the required numeric or string value or the @FALSE or @TRUE constants. For example,

```
FUNCTION CREDIT.RATING(CLIENT, CLASS = 1, CODE = "Standard") VAR.ARGS
```
In this example, if the calling program supplies only one argument, the CLASS variable will default to 1 and the CODE variable will default to "Standard". If the calling program provides two arguments, the default value for CLASS is ignored and the CODE variable defaults to "Standard".

Default argument values can also be provided when the DEFAULT.UNASS.ARGS option of the SMOKE compiler directive is enabled. In this case, the default value will be applied if the argument variable passed from the calling program is unassigned.

**Pass By Reference**

Function arguments are normally passed by reference such that changes made to the argument variable inside a subroutine will be visible in the caller’s variable referenced by that argument. A function call allows arguments to be passed by value by enclosing them in brackets. The `FUNCTION` statement also supports this dereferencing syntax. For example

```qbasic
FUNCTION CREDIT(P, (Q))
```

If dereferencing is used with the default argument syntax described above, the default value is placed inside the parenthesis. For example,

```qbasic
FUNCTION INVOICE(P, (Q = 7)) VAR.ARGS
```

**Matrix Arguments**

An argument may refer to a whole matrix. In this case the argument variable name must be preceded by the keyword `MAT` and there must be a `DIM` statement following the function declaration to indicate whether this is a one or two dimensional matrix. Alternatively, the dimensions may be given after the variable name in the `FUNCTION` statement. In either case, the actual dimension values are counted by the compiler but otherwise ignored. Use of a dimension value of one emphasises to readers of the program that the value is meaningless. A default style (not Pick style) matrix passed as an argument can be re-dimensioned in the subroutine by use of the `REDIMENSION` (or `REDIM`) statement. Used in any other context, this statement is identical to use of `DIM`.

Programs that use the function must include a `DEFFUN` statement to define the function template.

**Example**

```qbasic
FUNCTION MATMAX(MAT A)
    DIM A(1)
    MAX = A(1)
    N = INMAT(A)
    FOR I = 1 TO N
        IF A(I) > MAX THEN MAX = A(I)
    NEXT I
    RETURN MAX
END
```

This function scans a one dimensional matrix and passes back the value of the largest element. The first two lines could alternatively be written as

```qbasic
FUNCTION MATMAX(MAT A(1))
```

See also: `DEFFUN`
GES()

The GES() function processes two dynamic arrays, returning a similarly structured result array indicating whether elements of the first array are greater than or equal to corresponding elements of the second array.

Format

GES(expr1, expr2)

where

expr1 and expr2 are the dynamic arrays to be compared.

The GES() function compares corresponding elements of the dynamic arrays expr1 and expr2, returning a similarly structured dynamic array of True / False values indicating the results of the comparison.

The REUSE() function can be applied to either or both expressions. Without this function, any absent trailing values are taken as zero.

Example

A contains  11FM0VM14VMACBFM2
B contains  12FM0VM14VMACBFM2

C = GES(A, B)

C now contains  0FM1VM1VM1FM1

See also: ANDS(), EQS(), GTS(), IFS(), LES(), LTS(), NES(), NOTS(), ORS(), REUSE()
GET

The GET statement reads data from a device previously opened using OPENSEQ.

Format

```
GET var {, length} {SETTING count} FROM dev {UNTIL term} {RETURNING last} 
{WAITING timeout} 
{THEN statement(s)} {ELSE statement(s)}
```

where

- `var` is the variable to receive the data.
- `length` is the maximum number of characters to read.
- `count` is the actual number of characters read.
- `dev` is the file variable from OPENSEQ.
- `term` is a string of termination characters. Receiving any of these will terminate the read.
- `last` is the last character read.
- `timeout` is the timeout period in seconds.

The THEN and ELSE clauses are both optional. The THEN clause will be executed if the read action is successful having either read `length` characters or received a character that appears in `term`. The ELSE clause will be executed if the timeout period expired or if an i/o error occurs. The two situations can be distinguished by use of the STATUS() function which will return zero for a timeout and a non-zero value for any error.

See also:

OPENSEQ, SEND
GET(ARG.)

The GET(ARG.) statement fetches command line arguments.

Format

\[ \text{GET(ARG.} \{, \text{argno}\}\text{) var } \{\text{THEN statement(s)}\} \{\text{ELSE statement(s)}\} \]

The GET(ARG.) statement allows a program to retrieve command line arguments. This statement is provided for compatibility with other systems. The \texttt{PARSER()} subroutine provides a more powerful way to process command arguments.

The command line is considered to be formed from a number of tokens separated by spaces. Spaces within quoted strings are treated as part of the string. The GET(ARG.) statement, used without the \texttt{argno} element or with \texttt{argno} less than one, returns the value of the next command line argument in \texttt{var}. Specifying a positive and non-zero value for \texttt{argno} retrieves the specified argument.

The \texttt{THEN} and \texttt{ELSE} clauses are both optional. If present, the \texttt{THEN} clause will be executed if the argument is retrieved successfully. The \texttt{ELSE} clause is executed if the argument was not present in the command line.

If the program is started using the \texttt{RUN} or \texttt{DEBUG} commands, the argument data is each argument following the program record name. If the program is started by reference to its name within the catalogue, the argument data is everything after the catalogue name.

The command arguments and the current position within them are stacked on a per-command processor level basis.

Examples

```
PROGRAM TEST
    GET(ARG.) A
    GET(ARG.) B
    GET(ARG.) C
    DISPLAY A
    DISPLAY B
    DISPLAY C
END
```

If the above program is executed using the command

```
RUN TEST PPP "QQQ RRR" SSS
```

the output would be

```
PPP
QQQ RRR
SSS
```

The same output would be produced by

```
RUN BP TEST PPP "QQQ RRR" SSS
```

or, if the program was catalogued,

```
TEST PPP "QQQ RRR" SSS
```
Changing the program to be

```
PROGRAM TEST
  GET(ARG., 2) A
  GET(ARG.) B
  DISPLAY A
  DISPLAY B
END
```

any of the commands above would display

```text
QQQ RRR
SSS
```

Use of a construct such as

```
GET(ARG., 4) ELSE DISPLAY "Insufficient command arguments"
```

allows a program to check how many arguments have been supplied.
GET.MESSAGES()

The GET.MESSAGES() function returns any messages currently queued for display.

Format

GET.MESSAGES()

The GET.MESSAGES() function allows an application to retrieve messages queued by the MESSAGE command and display these in a convenient form. The returned string contains one field for each message, in chronological order of message generation. Each field has two values: the message text and the message origin header. The second value may not be present in all messages.

Use of GET.MESSAGES() removes the messages from the queue so that they are not displayed on return to the command prompt. Normally, the application that uses GET.MESSAGES() would handle display of the messages.

Messages submitted with the IMMEDIATE option of the MESSAGE command appear on the screen immediately and cannot be retrieved with the GET.MESSAGES() function.
GET.PORT.PARAMS()

The GET.PORT.PARAMS() function retrieves the communications parameters for a serial port.

Format

GET.PORT.PARAMS(fvar)

where

fvar is the file variable from the OPENSEQ statement that was used to open the port.

The GET.PORT.PARAMS() function returns a dynamic array containing the following data:

Field 1  Port name
Field 2  Baud rate
Field 3  Parity mode (0 = off, 1 = odd, 2 = even)
Field 4  Bits per byte
Field 5  Stop bits
Field 6  State of CTS (clear to send)
Field 7  State of DSR (data set ready)
Field 8  State of RING
Field 9  State of DCD (carrier detect)

Programs should be written to allow for the possibility of additional fields being added in future releases.

Example

DISPLAY 'Baud rate = ' : GET.PORT.PARAMS(port)<2>

See also:
SET.PORT.PARAMS()
GETLIST, GETLISTV

The GETLIST statement restores a select list from the $SAVEDLISTS file. The GETLISTV statement is similar but creates a select list variable.

Format

```
GETLIST name {TO list.no} {SETTING count.var} {THEN statement(s)} {ELSE statement(s)}
GETLISTV name TO list.var {SETTING count.var} {THEN statement(s)} {ELSE statement(s)}
```

where

- `name` is the name of the $SAVEDLISTS entry to be restored.
- `list.no` is the select list number to which it is to be restored. If omitted, this defaults to zero.
- `list.var` is the select list variable to be created.
- `count.var` is the variable to receive the item count.
- `statement(s)` are statements to be executed depending on the outcome of the operation.

The GETLIST and GETLISTV statements restore the previously saved select list identified by `name` from the $SAVEDLISTS file. The disk copy is not removed by this operation.

The @SELECTED variable can be examined to determine the number of items in the list. Alternatively, the SETTING clause can be used to set `count.var` to this value.

At least one of the THEN and ELSE clauses must be present. If the list is successfully restored, the THEN clause is executed. If the list does not exists or cannot be restored for any other reason, the ELSE clause is executed.

See also:
SAVELIST, WRITELIST
GETLOCKS()

The GETLOCKS() function returns information about file and record locks.

Format

GETLOCKS(file, user)

where

file is the file variable referencing the file for which locks are to be reported. All files are reported if this is not a valid file variable.

user is the user number for which locks are to be reported. All users are reported if this is zero.

The GETLOCKS() function returns a string where field 1 contains three values:

1. NUMLOCKS configuration parameter value
2. Current record lock count
3. Peak number of record locks

and the remaining fields each describe a record or file lock as seven values:

1. Internal file number
2. Pathname
3. User number
4. Lock type as below
5. Record id, null for a file lock
6. User name of lock owner
7. Time as an epoch value when the lock was acquired or a blocked process started waiting

The lock type code is:

FX  File exclusive lock
RL  Shareable read lock (READL)
RU  Exclusive read lock (READU)
SX  File lock for CLEARFILE
WAIT Process is waiting for the lock

Example

LOCK.INFO = GETLOCKS(0, 0)

This statement returns lock information for all files and all users.
**GETNLS()**

The `GETNLS()` function returns the value of a national language support parameter.

**Format**

```
GETNLS(key)
```

where

```
key
```
identifies the parameter to be returned.

The `GETNLS()` function returns the value of the named national language support parameter. NLS parameter name tokens are defined in the KEYS.H include record.

Available parameters are:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Key</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NLS$CURRENCY</td>
<td>Default currency symbol. Maximum 8 characters.</td>
</tr>
<tr>
<td>2</td>
<td>NLS$THOUSANDS</td>
<td>Default thousands separator character.</td>
</tr>
<tr>
<td>3</td>
<td>NLS$DECIMAL</td>
<td>Default decimal separator character.</td>
</tr>
<tr>
<td>4</td>
<td>NLS$IMPICIT.DECEIMAL</td>
<td>Default decimal separator character for implicit conversions. See SETNLS for details.</td>
</tr>
<tr>
<td>5</td>
<td>NLS$EUROPEAN.DATES</td>
<td>Retrieves European date format mode</td>
</tr>
</tbody>
</table>

See also:

Conversion codes, Format codes, NLS, OCONV(), SETNLS
GETPU()

The GETPU() function gets the characteristics of a print unit.

Format

GETPU(key, unit)

where

<table>
<thead>
<tr>
<th>key</th>
<th>identifies the parameter to retrieved. This may be:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>PUSDEFINED Is print unit defined?</td>
</tr>
<tr>
<td>1</td>
<td>PUSMODE Print unit mode</td>
</tr>
<tr>
<td>2</td>
<td>PUSWIDTH Characters per line</td>
</tr>
<tr>
<td>3</td>
<td>PUSLENGTH Lines per page</td>
</tr>
<tr>
<td>4</td>
<td>PUSTOPMARGIN Top margin size</td>
</tr>
<tr>
<td>5</td>
<td>PUSBOTMARGIN Bottom margin size</td>
</tr>
<tr>
<td>6</td>
<td>PUSLEFTMARGIN Left margin size</td>
</tr>
<tr>
<td>7</td>
<td>PUSSPOOLFLAGS Various print unit flags</td>
</tr>
<tr>
<td>9</td>
<td>PUSFORM Form name (not used by all spoolers)</td>
</tr>
<tr>
<td>10</td>
<td>PUSBANNER Banner page text</td>
</tr>
<tr>
<td>11</td>
<td>PUSLOCATION Printer / file name</td>
</tr>
<tr>
<td>12</td>
<td>PUSCOPIES Number of copies to print</td>
</tr>
<tr>
<td>15</td>
<td>USPAGENUMBER Current page number</td>
</tr>
<tr>
<td>1002</td>
<td>PUSLINESLEFT Lines left on page</td>
</tr>
<tr>
<td>1003</td>
<td>PUSHEADERLINES Lines occupied by header</td>
</tr>
<tr>
<td>1004</td>
<td>PUSSFOOTERLINES Lines occupied by footer</td>
</tr>
<tr>
<td>1005</td>
<td>PUSDATALINES Lines between header and footer</td>
</tr>
<tr>
<td>1006</td>
<td>PUSOPTIONS Options to be passed to the spooler</td>
</tr>
<tr>
<td>1007</td>
<td>PUSPREFIX Pathname of file holding prefix data to be added to the start of the output</td>
</tr>
<tr>
<td>1008</td>
<td>PUSSPOOLER Spooler to be used (ignored on Windows)</td>
</tr>
<tr>
<td>1009</td>
<td>PUSOVERLAY Catalogued overlay subroutine name (see SETPTR)</td>
</tr>
<tr>
<td>1010</td>
<td>PUS CPI Characters per inch (may be non-integer value)</td>
</tr>
<tr>
<td>1011</td>
<td>PUSPAPER.SIZE Paper size. See SYSCOM PCL.H</td>
</tr>
<tr>
<td>1012</td>
<td>PUSLPI Lines per inch</td>
</tr>
<tr>
<td>1013</td>
<td>PUSWEIGHT Font stroke weight. See SYSCOM PCL.H</td>
</tr>
<tr>
<td>1014</td>
<td>PUSSSYMBOL.SET Symbol set. See SYSCOM PCL.H</td>
</tr>
</tbody>
</table>
1015  PU$STYLE  Query processor style. See the Query processor
        STYLE option for details.
1016  PU$NEWLINE  Newline string for this print unit
1017  PU$PRINTER.NAME  Name of printer
1018  PU$FILENAME  File name for output directed to a file
1019  PU$SEQNO  Hold file sequence number. See also
        @$SEQNO.
1020  PU$FORM.QUEUE  Last assigned form queue number with
        SP.ASSIGN. This value will be meaningless if
        the characteristics of the print unit have
        subsequently been modified.
1021  PU$FORM.FEED  Get form feed string for this print unit
1022  PU$ENCODING  Get character encoding for this print unit
1023  PU$FONT  Get font name
1024  PU$PATHNAME  Get pathname of output file
1025  PU$FONT.SIZE  Get the GDI mode font size. Zero implies use
        of the default font.
2000  PU$LINENO  Current line number on page

unit evaluates to the print unit number.

The GETPU() function returns the print unit characteristic specified by key. It is closely related to the
!GETPU() subroutine.

The PU$DEFINED key should be used to determine if a print unit has been defined. Use of any other
key value for an undefined print unit will create it with all settings at their default values.

Example

    MODE = GETPU(PU$MODE, 3)

The above statement gets the mode of print unit 3, storing it in MODE.

See also:
SETPU
GETREM()

The GETREM() function returns the remove pointer position into a string.

Format

```
GETREM(string)
```

where

```
string
```

is the string for which the remove pointer position is to be returned.

The GETREM() function returns the offset of the remove pointer into string. It is typically used with SETREM to save and restore the remove pointer position. The remove pointer is positioned on the mark character preceding the next fragment to be extracted. It is reset to zero when a new value is assigned to the string.

Example

```
RMV.PTR = GETREM(S)
GOSUB PROCESS.DATA
SETREM RMV.PTR ON S
```

The above code fragment saves the remove pointer associated with string S and restores it after execution of subroutine PROCESS.DATA which might change this remove pointer.

See also:

REMOVE, SETREM
GOSUB

The GOSUB statement calls an internal subroutine.

Format

\[
\text{GOSUB } \text{label}\{;\}\]
\[
\text{GOSUB } \text{label}\{;\}(\text{args})
\]

where

- label is the label attached to the statement at the start of the internal subroutine.
- args is a comma separated list of arguments to a subroutine defined with the \text{LOCAL SUBROUTINE} statement.

The optional colon after the \text{label} has no effect on the action of the statement.

The program continues execution at the given \text{label}, saving the location of the GOSUB for a later \text{RETURN} which will resume execution at the statement following the GOSUB. See also the \text{RETURN TO} statement for details of alternate returns.

QMBasic defines two styles of internal subroutine. A conventional internal subroutine, as found in other multivalue database products, has no formal start or end. The \text{label} may be any label defined within the program or subroutine. It is the programmer's responsibility to ensure that internal subroutines return correctly. Variables referenced in the internal subroutine are accessible across the entire program module, requiring great care from the programmer to ensure that data in one part of the module is not accidentally altered elsewhere by use of the same name. Loop counters in \text{FOR/NEXT} loops are a good example of where this frequently happens. Calling this style of internal subroutine recursively is possible but of limited use because the variables in one invocation will be overwritten by the next.

The second style, referred to in QMBasic terminology as a \text{local subroutine}, is introduced by the \text{LOCAL} statement and is terminated by \text{END}. Local subroutines may have arguments and may have private local variables that are not visible outside the subroutine and which are stacked if the subroutine is called recursively. The \text{PRIVATE} statement(s) used to declare private variables must appear at the start of the local subroutine.

The argument list for a local subroutine may contain up to 255 items. If a subroutine has no arguments, the brackets may be omitted.

Each argument is

- A constant
  
  \[
  \text{GOSUB SUB("MY.FILE")}
  \]

- An expression
  
  \[
  \text{GOSUB SUB(X + 7)}
  \]

- A variable name
  
  \[
  \text{GOSUB SUB(X)}
  \]

- An indexed matrix element name
  
  \[
  \text{GOSUB SUB(A(5,2))}
  \]

- A matrix name prefixed by MAT
  
  \[
  \text{GOSUB SUB(MAT A)}
  \]

Where the argument is a reference to a variable, a matrix element or a whole matrix, the subroutine may update the values in these variables. Except when passing a whole matrix, the calling program can effectively prevent this by forcing the argument to be passed by value rather than by reference by enclosing it in brackets, thus making the argument into an expression.
**Examples**

```qbasic
IF STOCK.LEVEL <= REORDER.LEVEL THEN GOSUB REORDER
```

This program fragment checks if the value of `STOCK.LEVEL` has fallen to the `REORDER.LEVEL` and, if so, calls internal subroutine `REORDER`.

```qbasic
LOCAL SUBROUTINE UPDATE.STOCK(PROD.NO, CHANGE)
PRIVATE STOCK.REC
READU STOCK.REC FROM STOCK.F, PROD.NO THEN
  STOCK.REC<STK.QOH> += CHANGE
WRITE STOCK.REC TO STOCK.F, PROD.NO
END
RETURN
END
```

The above local subroutine takes a record id and the amount by which a field is to be updated, reads the corresponding record and applies the update. A real program would include statements to handle the case where the record is not found. The subroutine would be called using a statement such as

```qbasic
GOSUB UPDATE.STOCK(PART.ID, QTY)
```

See also:

- **LOCAL**, **PRIVATE**
GOTO

The GOTO statement continues program execution at a given label.

Format

GOTO label[:]
GO {TO} label[:]

where

label is the label attached to the statement at which execution is to continue.

The trailing colon is optional and has no effect on the action of the statement.

The program continues execution at the given label. The label may be any label defined within the program or subroutine. Excessive use of GOTO and labels in place of other language constructs (e.g. LOOP/REPEAT) can make programs difficult to maintain.

Example

IF REC[1,1] # "A" THEN GOTO ERROR

This program fragment checks if the first character of REC is "A". If not, it jumps to label ERROR.
GROUPSTORE

The GROUPSTORE statement inserts, deletes or replaces elements of a delimited string. It is closely related to the FIELDSTORE() function.

Format

GROUPSTORE rep.string IN string USING i, n {, delimiter}

where

rep.string evaluates to the new data to be inserted in string.
string is the string in which replacement is to occur.
i evaluates to the position of the first substring to be replaced. Substring positions are numbered from one.
n evaluates to the number of substrings to be replaced.
delimiter evaluates to a string, the first character of which is used as the delimiter separating the substrings within string. If omitted, a field mark is used.

The action of GROUPSTORE depends on the values of i and n and the number of substrings within rep.string.

If the value of the position expression, i, is less than one, a value of one is assumed. If there are fewer than i delimited substrings present in string, additional delimiters are added to reach the required position.

If the value of the number of substrings expression, n, is positive, n substrings are replaced by the same number of substrings from rep.string. If rep.string contains fewer than n substrings, additional delimiters are inserted.

If the value of the number of substrings expression, n, is zero or negative, n substrings are deleted from string and the whole of rep.string is inserted regardless of the number of substrings that it contains.

Use of the $NOCASE.STRINGS compiler directive makes the delimiter case insensitive.

Example

Each of the following examples assumes variable S is initialised with

S = "1*2*3*4*5"

before the GROUPSTORE is executed.

GROUPSTORE "A*B" IN S USING 2, 3, "*"
Variable S becomes "1*A*B*5". Note the inserted delimiter as rep.string has only 2 substrings.

GROUPSTORE "A*B*C" IN S USING 2, 3, "*"
Variable S becomes "1*A*B*C*5". rep.string replaces substrings 2 to 5.
GROUPSTORE "A*B*C*D" IN S USING 2, 3, "*"
Variable S becomes "1*A*B*C*5". Final substring of rep.string is not inserted.

GROUPSTORE "A*B" IN S USING 2, 0, "*"
Variable S becomes "1*A*B*2*3*4*5". No substrings are deleted as n is zero.
Note
GROUPSTORE "A*B" IN S USING 2, -3, "*"
Variable S becomes "1*A*B*5". Three substrings are deleted and rep.string is inserted

See also:
FIELD(), FIELDSTORE()
GTS()

The GTS() function processes two dynamic arrays, returning a similarly structured result array indicating whether elements of the first array are greater than corresponding elements of the second array.

Format

\[ \text{GTS}(\text{expr1}, \text{expr2}) \]

where

\[ \text{expr1} \text{ and expr2 are the dynamic arrays to be compared.} \]

The GTS() function compares corresponding elements of the dynamic arrays expr1 and expr2, returning a similarly structured dynamic array of True / False values indicating the results of the comparison.

The REUSE() function can be applied to either or both expressions. Without this function, any absent trailing values are taken as zero.

Example

A contains 11FM0VM14VMABC
B contains 12FM0VM14VMACB

\[ C = \text{GTS}(A, B) \]

C now contains 0FM0VM1VM1FM0

See also:
ANDS(), EQS(), GES(), IFS(), LES(), LTS(), NES(), NOTS(), ORS(), REUSE()
**HEADING**

The **HEADING** statement defines text to be printed or displayed at the top of each page of output.

**Format**

\[
\text{HEADING \{NO.EJECT\} \{ON print.unit\} text}
\]

where

- `print.unit` identifies the logical print unit in the range -1 to 255 to which the heading text is to be applied. If omitted, the default print unit (unit 0) is used.
- `text` is the heading text. This may include control tokens as described below.

The **HEADING** statement defines the text of a page heading and, optionally, control information determining the manner in which the text is output. A page heading is output whenever the first line of output on a page, including blank lines, is about to be printed or displayed. The **HEADING** statement normally causes subsequent output to appear on a new page. The **NO.EJECT** option defers the new heading until the start of the next page. The **HEADING.NO.EJECT** option of the **$MODE** compiler directive can be used to make **NO.EJECT** the default.

The heading `text` may include the following control tokens enclosed in single quotes. Multiple tokens may appear within a single set of quotes.

- **C** Centers text on the line.
- **D** Insert the current date in the form dd mmm yyyy (e.g. 25 MAR 2006)
- **G** Insert a gap. Spaces are inserted in the heading line at the position of each G control token such that the overall length of the line is the same as the printer unit width. A single use of the G token will right justify the subsequent text. Multiple G tokens will distribute spaces as evenly as possible.

  When a heading line uses both G and C, the heading is considered as a number of elements separated by the G control options. The element that contains the C option will be centered. The items either side of the centered element are processed separately when calculating the number of spaces to be substituted for each G option.

- **Hn** Sets horizontal position (column) numbered from one. Use of H with C or with a preceding G token may have undesired results.
- **L** Start a new line
- **N** Inhibit automatic display pagination
- **On** Reverses the elements separated by G tokens in the current line on even numbered pages. This is of use when printing double sided reports.
- **Pn** Insert page number. The page number is right justified in \(n\) spaces, widening the field if necessary. If omitted, \(n\) defaults to four.
- **Sn** Insert page number. The page number is left justified in \(n\) spaces, widening the field if necessary. If omitted, \(n\) defaults to one.
T Insert current date and time in the form hh:mm:ss mm/dd/yy

Unrecognised control tokens are ignored. A quotation mark may be inserted in the printed text by using two adjacent quotation marks in the text string.

There is no limit to the length of a heading text. Each line will be truncated at the width of the print unit. The effect of using a heading which does not leave sufficient space for at least one line of text is undefined.

See the RUN command for a discussion of how the page end pause on output to the screen may be affected by setting a page heading.

See also:
FOOTING, PAGE
HUSH

The **HUSH** statement enables or disables display output.

**Format**

\[
\text{HUSH OFF} \{ \text{SETTING } var \} \\
\text{HUSH ON} \{ \text{SETTING } var \} \\
\text{HUSH } expr \{ \text{SETTING } var \}
\]

where

\[
\begin{align*}
expr & \quad \text{evaluates to a number.} \\
var & \quad \text{is a variable to receive the previous state of display output control.}
\end{align*}
\]

The **HUSH ON** statement causes all output sent to the display by **CRT, DISPLAY** or **PRINT** statements to be suppressed. The **HUSH OFF** statement re-enables display.

The **HUSH expr** format of this statement is equivalent to **HUSH ON** if the value of \( expr \) is non-zero and **HUSH OFF** if \( expr \) is zero.

The optional **SETTING** clause saves the previous state of display output control in \( var \) which can be used later to revert to that state. Alternatively, the previous state can be obtained using the **STATUS()** function immediately after the **HUSH** statement. In either case, the value is 1 if output was suppressed or 0 if it was enabled.

**Example**

\[
\begin{align*}
\text{HUSH ON} \\
\text{EXECUTE } "\text{SELECT STOCK.FILE WITH QTY > 50}" \\
\text{HUSH OFF}
\end{align*}
\]

This program fragment suppresses display while the SELECT statement is executed. Use of the query processor **COUNT.SUP** option might be better as it would not hide any errors.
ICONV()

The ICONV() function performs input conversion. Data is converted from its external representation to the internal form. This function is typically used to convert data entered at the keyboard. The ICONVS() function is identical to ICONV() except that it works on each element of a dynamic array, returning the result in a similarly delimited dynamic array.

Format

ICONV(expr, conv.spec)
ICONVS(expr, conv.spec)

where

expr evaluates to the data to be converted.
conv.spec evaluates to the conversion specification. This may be a multivalued string containing more than one conversion code separated by value marks. Each conversion will be carried out in turn on the result of the previous conversion.

The ICONV() function converts the value of expr to its internal representation according to the conversion codes in conv.spec. The result of ICONV() is stored internally as a string regardless of whether the value is a number or not except for the MO and MX conversions which always produce an integer value.

The ICONV() function sets the STATUS() function value to indicate whether the conversion was successful. Possible values are

0 Successful conversion.
1 Data to convert was invalid for the conversion specification. A null string is returned for all codes except ML and MR which result of conversion of the leading numeric portion of the input data or the entire input data if there is no leading numeric portion.
2 The conversion code was invalid. A null string is returned.
3 The conversion has been performed but the result is possibly incorrect. Examples are:
   The day number in a date conversion was beyond the end of the month. The returned value will be extended into the following month (e.g. 31 June becomes 1 July). This feature can be suppressed using the NO.DATE.WRAPPING option.
   A character encoding conversion (X) on a non-ECS system has returned data in which characters outside the 8 bit set have been replaced by some substitute character.

See also:
Conversion codes, OCONV()
IDIV()

The IDIV() function divides one integer by another and returns an integer result.

Format

\[
\text{IDIV}(\text{dividend}, \text{divisor})
\]

where

- \text{dividend} evaluates to the integer to be divided.
- \text{divisor} evaluates to the integer by which it is to be divided.

Both arguments to the IDIV() function are converted to integers. The function returns the result of the division as an integer rounded towards zero.

A zero value of \text{divisor} will cause a run time error.

Integer division can also be performed using the // operator.

Example

\[
\begin{align*}
X &= 7 \\
Y &= 2 \\
Z1 &= X \div Y \\
Z2 &= \text{IDIV}(X, Y)
\end{align*}
\]

This program fragment shows the difference between the division operator and the IDIV() function. \(Z1\) will be set to 3.5 and \(Z2\) will be set to 3.

See also:
DIV(), RDIV()
RDIV()
**IF /THEN / ELSE**

The **IF** statement provides conditional execution of one or more statements.

**Format**

```
IF expr
{THEN statement(s)}
{ELSE statement(s)}
```

where

- `expr` is an expression which can be resolved to a numeric value
- `statement(s)` are statements to be executed depending on the value of `expr`.

At least one of the **THEN** and **ELSE** clauses must be present. If both are used, the **THEN** clause must be first. An **IF** statement with just an **ELSE** clause is syntactically valid but likely to be difficult to understand.

The newline before the **THEN** and **ELSE** clauses is optional.

The **statement(s)** under the **THEN** clause are executed if the value of `expr` is non-zero. The **statement(s)** under the **ELSE** clause are executed if the value of `expr` is zero.

Where the keyword **THEN** or **ELSE** is followed by an executable statement on the same line, the condition applies to that statement. If there is nothing else or only a comment on the line, the conditioned **statement(s)** must appear on subsequent lines terminated by an **END** statement. For example:

```
IF QTY > 99 THEN
    LARGE.ORDER = @TRUE
    DISCOUNT = 0.1
END
```

Alternatively, this could be written using semicolons to separate the conditioned statements:

```
IF QTY > 99 THEN LARGE.ORDER = @TRUE ; DISCOUNT = 0.1
```

Use of this format is discouraged as the semantics differ across multivalue database products.

```
IF QTY > QOH THEN
    DISPLAY 'Insufficient stock'
END ELSE
    QOH -= QTY
    DISPLAY 'Order confirmed'
END
```

The above program fragment might be used to compare the quantity in an order (QTY) with the quantity on hand (QOH), taking different paths dependent on whether there is sufficient stock. Note the need for the **END** to terminate the **THEN** clause as, unlike some other programming languages, the **END** pairs up with the **THEN** or **ELSE** and not with the **IF**.
IFS()

The IFS() function returns a dynamic array constructed from elements chosen from two other dynamic arrays depending on the content of a third dynamic array.

Format

    IFS(control.array, true.array, false.array)

where

control.array is a dynamic array of True / False values.
true.array holds values to be returned where the corresponding element of control.array is True.
false.array holds values to be returned where the corresponding element of control.array is False.

The IFS() function examines successive elements of control.array and constructs a result array where elements are selected from the corresponding elements of either true.array or false.array depending on the control.array value.

Note that if control.array has only one element but one or both of true.array and false.array have more than one element, the IFS() function has no way to determine whether it should be extracting a field, value or subvalue from the data. In this situation, appending the relevant mark character to control.array would avoid make the intention clear.

Example

A contains 1VM0VM0VM1VM1VM1VM0
B contains 11VM22VM3VM4VM91VM36VM7
C contains 14VM61VM2VM0VM3VM5VM18VM3

    D = IFS(A, B, C)

D now contains 11VM61VM2VM4VM91VM36VM3

See also: ANDS(), EQS(), GES(), GTS(), LES(), LTS(), NES(), NOTS(), ORS(), REUSE()
IN

The IN statement reads a single character from the terminal with an optional timeout.

Format

```
IN var {FOR timeout {THEN statement(s)} {ELSE statement(s)}}
```

where

- `var` is the variable to receive the input character value. This is the character code point value, not the character itself.
- `timeout` is the timeout period in tenths of a second.

The IN statement reads a single character from the terminal, returning the character value in `var`. Unless the character is a non-printing control code, it is echoed to the terminal. On Windows console sessions, the special keys return values in the upper half of the 8-bit character set as defined in the KEYS.H include record in the SYSCOM file. The KEYINV() function should be used instead of IN if it is necessary to distinguish these keys from the characters that are defined to have the same code point values.

If a `timeout` is specified, the program will continue execution if no input is received after this period. The `var` will be set to zero if a timeout occurs.

The optional THEN and ELSE clauses can be used with the timeout to determine whether input was received.

See also:

INPUT, KEYIN(), KEYINV(), KEYREADY()
INDEX()

The INDEX() function returns the position of a specified occurrence of a substring within a string. The INDEXS() function is similar to INDEX() but operates on each element of a dynamic array element separately, locating the required occurrence of substring and returning a similarly structured dynamic array of results.

Format

INDEX(string, substring {, occurrence})

where

- **string** is the string in which the search is to occur.
- **substring** evaluates to the substring to be located.
- **occurrence** evaluates to the position of the occurrence of the substring to be located. If omitted, this argument defaults to 1.

The INDEX() function locates the specified occurrence of substring within string and returns its character position.

If occurrence is less than one or the desired occurrence of substring is not found, the INDEX() function returns zero.

If substring is null, the value of occurrence is returned.

Use of the $MODE NOCASE.STRINGS compiler directive makes the comparison case insensitive.

The default behaviour of the INDEX() function is that overlapping substrings are not allowed. Thus

\[
X = INDEX("aaaaaaaaa", "aa", 3)
\]

yields a result of 5.

The INDEX.OVERLAP setting of the $MODE compiler directive can be used to select an alternative behaviour in which substrings may overlap. The same example would then yield a result of 3.

Examples

\[
N = INDEX(S, "*", 3)
\]

This statement assigns N with the character position of the third asterisk in variable S.

\[
S = "ABABABABABAB"
N = INDEX(S, "ABA", 2)
\]

Because substrings are defined not to overlap, this sets N to 5.
**INDICES()**

The **INDICES()** function returns information about alternate key indices.

**Format**

\[
\text{INDICES(file.var)} \quad \text{To retrieve a list of indices}
\]

\[
\text{INDICES(file.var, index.name)} \quad \text{To retrieve information for a specific index}
\]

where

- **file.var** is the file variable associated with an open file.
- **index.name** is the name of the index to be examined.

The first form of the **INDICES()** function returns a field mark delimited list of alternate key index names for the file referenced via **file.var**.

The second form of the **INDICES()** function returns a dynamic array resembling a dictionary record for the index named by the **index.name** argument. This dynamic array is broadly similar to the original dictionary record used to create the index except that field 1 is extended to include additional flags as a multivalued list and field 5 of a C, D or I-type item contains only L or R as the justification mode.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Index type (D, I, A, S or C)</td>
</tr>
<tr>
<td>2</td>
<td>Set to 1 if the index needs to be built, otherwise null</td>
</tr>
<tr>
<td>3</td>
<td>Set to 1 if the index is null-suppressed by use of the NO.NULLS option to <strong>CREATE.INDEX</strong>, otherwise null</td>
</tr>
<tr>
<td>4</td>
<td>Set to 1 if updates are enabled, otherwise null</td>
</tr>
<tr>
<td>5</td>
<td>Internal AK number</td>
</tr>
<tr>
<td>6</td>
<td>The key sort mode of record ids within each index entry (L or R), null for indices created prior to release 2.2-16 that were unsorted.</td>
</tr>
<tr>
<td>7</td>
<td>Set to 1 if the index is encrypted</td>
</tr>
<tr>
<td>8</td>
<td>Set to 1 if the index is case insensitive, otherwise null</td>
</tr>
<tr>
<td>9</td>
<td>Set to 1 if index is being built, otherwise null</td>
</tr>
</tbody>
</table>

Interpretation of fields 2 onwards is dependent on the record type. For an A/S-type record that has no correlative in field 8 or for a D-type record, field 2 holds the field number on which the index is based. For a C/I type record, field 2 holds the indexed expression. For an A/S-type record with a correlative, the cumulative expression is in field 8.

The sort order of indexed values in the index is determined by the justification code in the dictionary item as returned in field 5 for a C, D or I-type item and field 9 for an A or S-type item.

See the **SELECTINDEX** statement for an example that describes how an index is sorted.

**Example**

\[
\text{INDEX.NAMES = INDICES(FVAR)}
\]
NUM.INDICES = DCOUNT(INDEX.NAMES, @FM)
FOR I = 1 TO NUM.INDICES
    NAME = INDEX.NAMES<\textless I>\textgreater
    CRT NAME : ' Type ' : INDICES(FVAR, NAME)[1,1]
NEXT I

The above program displays a list of alternate key index names and their type.
INHERIT

The **INHERIT** statement used in a class module makes the public variables, functions and subroutines of another object visible as part of this object.

**Format**

```plaintext
INHERIT object
```

where

- `object` is an object variable returned from a previous use of the **OBJECT()** function.

The process of searching for a public variable, function or subroutine scans the object referenced in the statement that initiated the scan and then all inherited objects in the order in which they were inherited. Where an inherited object has itself inherited other objects, the scan treats these inherited names as part of the directly inherited object.

**See also:**
- Object oriented programming,
- **CLASS**, **DISINHERIT**, **OBJECT()**, **OBJINFO()**, **PRIVATE**, **PUBLIC**.
**INMAT()**

The **INMAT()** function provides qualifying information after certain statements are executed.

**Format**

\[
\text{INMAT}(\{mat\})
\]

where

\(mat\) is the name of a matrix.

Used without a \(mat\) matrix name, the **INMAT()** function returns information relating to the last use of the following statements:

- **DIMENSION**
  - 0 if successful, 1 if insufficient memory.
- **IPARSE()**
  - The character position at which parsing of a JSON string detected an error.
- **MATBUILD**
  - The index of the final non-null element, zero if the zero element was not null.
- **MATCHFIELD()**
  - The value within a multivalued pattern that matches the string.
- **MATPARSE**
  - The number of elements assigned. Zero if overflows.
- **MATREAD**
  - The number of elements assigned. Zero if overflows.
- **MATREADL**
  - The number of elements assigned. Zero if overflows.
- **MATREADU**
  - The number of elements assigned. Zero if overflows.
- **MATWRITE**
  - The index of the final non-null element, zero if the zero element was not null.
- **OPEN**
  - The modulus of a dynamic file.
- **PAUSE**
  - The user number of the process performing the **WAKE**.
- **Pattern matching**
  - The value position within a multivalued pattern that matches the string.

Further details of the information returned by **INMAT()** in each case is documented with the relevant statement.

Used with a matrix name, the **INMAT()** function returns the current dimensions of the matrix, including matrices in data collection variables. If \(mat\) is a single dimensional matrix, **INMAT()** returns the number of elements, excluding the zero element. If \(mat\) is a two dimensional matrix, **INMAT()** returns the number of rows and columns as two values separated by a value mark.

**Examples**

```
DIM A(N)
IF INMAT() THEN ABORT "Insufficient memory"
```
This program fragment dimensions matrix A and tests whether it was successful. If not, the program aborts. With memory sizes found on modern computer systems, dimensioning a matrix is very unlikely to fail and this test is usually omitted.

\[
\begin{align*}
N &= \text{INMAT}(A) \\
\text{FOR } I &= 1 \text{ TO } N \\
A(I) &= A(I) + 1 \\
\text{NEXT } I
\end{align*}
\]

This program fragment adds one to each element of matrix A. The \texttt{INMAT()} function is used because the matrix was dimensioned elsewhere and hence its size is not known.
INPUT, INPUTDW

The **INPUT** statement enables entry of data from the keyboard or from previously stored **DATA** statements. The **INPUTDW** statement is identical except that it limits data length based on display width.

**Format**

```
INPUT var {= default} {, length {, fill} {, _} {, } {modes}}
{THEN statement(s)}
{ELSE statement(s)}
```

where

- **var** is the variable in which the data is to be stored.
- **default** is the default value to be used if the user enters a null response. This option provides compatibility with other multi-value products. To avoid a syntactic ambiguity, the **default** must be enclosed in round brackets if it is an expression.
- **length** is the maximum number of characters to be allowed. In the **INPUTDW** statement, this is the maximum display width of the entered data.
- **fill** is a Pick style fill expression. The first character will be used to fill any spare space in the input field to make the field width clear to the user. The second character will be used to fill unused space after input is completed. If the **fill** string is only one character, spaces will be used. The third character, if present causes the cursor to be left at the end of the input field instead of the default action of leaving it after the last character of the entered data string. The **fill** option cannot be used with **HIDDEN**.
- **modes** are any combination of the following keywords:

  - **HIDDEN** echoes characters back to the screen as asterisks for password type fields.
  - **NO.ECHO** Suppresses echo of input data.
  - **NO.ENCODING** Suppresses decode of encoded input when the terminal is set to operate in UTF-8 mode. Use of this mode will also suppress encoding of data echoed back to the terminal.
  - **RAW** Causes **INPUT** to place input data into **var** without special interpretation of newline, backspace, etc. Used in this way, the **length** element of the statement must be present.
  - **TIMEOUT wait** Sets a timeout period in seconds. If input is not received in this time, the **INPUT** terminates, leaving **var** unchanged. The keywords **FOR** or **WAITING** can be used in place of
TIMEOUT for compatibility with other environments.

UPCASE converts input data to uppercase.

The optional THEN and ELSE clauses used with TIMEOUT allow a program to determine whether the input timed out. Successful input executes the THEN clause. A timeout will execute the ELSE clause.

The INPUT statement reads data from the DATA queue or, if there is no stored data, from the keyboard.

Keyboard Input

When taking input from the keyboard, the current prompt character will be displayed prior to reading data. The values stored for printing characters are the ASCII characters associated with the key. Non-printing characters result in other stored character values.

If no length expression is included, data characters are stored until the return key is pressed.

If length is specified, up to that number of characters may be entered after which input is automatically terminated as though the return key had been pressed, any subsequent key entries being retained for the next INPUT statement. The return key is not stored as part of the input data. For the INPUTDW statement, the length value specifies the maximum allowed display width of the data. If a double width character is entered when the remaining display width is one, the character is ignored and the operation continues to wait for a further character.

The optional underscore component of the statement suppresses the automatic input termination when length characters have been entered. Any number of characters may be entered but only length characters will be displayed.

The optional colon causes the carriage return and line feed output when the return key is used or on reaching the input length limit to be suppressed.

The INPUT statement recognises the backspace key, allowing this to be used to correct data entry errors. The terminfo system allows the code sent by the backspace key to be redefined from its default char(8). If an alternative, single byte definition is used, INPUT will honour this, otherwise char(8) is used as the backspace.

DATA Queue Input

Where the data queue is not empty, the INPUT statement reads the item at the head of this queue, copying it verbatim to var with no processing of any embedded control characters. The length expression is ignored. The prompt character is not displayed. Unless the NO.ECHO.DATA mode of the $MODE compiler directive or the NO.ECHO.DATA setting of the OPTION command is enabled, the item is displayed as though it had been typed.

Testing for Input

The INPUT statement may be used to test whether there are characters waiting to be read from the keyboard or the data queue by using a negative length value. For example, the statement

\[
\text{INPUT } S, -1
\]
will set $S$ to 1 if there is data waiting, 0 if no data is waiting. See the `KEYREADY()` function for an alternative way to detect keyboard input.

**Use of Pipes**

QM recognises input from pipes as a special case. Programs that process data from a pipe can read the data using the same QMBasic statements and functions as for keyboard input. If the end of the data is reached, a subsequent `INPUT` will return a null string. The `STATUS()` function will return ER$EOF$.

**Examples**

```qmb
INPUT ACCOUNT.NO, 10
```

This statement reads data into `ACCOUNT.NO` with a maximum length of 10 characters.

```qmb
DISPLAY @(0,24) :"Continue?" :
INPUT S:
```

This program fragment displays a query message on the bottom line of the screen and reads a response. Note the trailing colon in the `INPUT` statement to suppress the line feed which would cause the screen to roll up as output was to the bottom line of the display.

**See also:**

`KEYREADY()`, `INPUTIF`
**INPUT @**

The **INPUT @** statement enables entry of data from the keyboard at a specific screen position or from previously stored **DATA** statements.

**Format**

\[
\text{INPUT @} (x, y) \{ , \} \{ ; \} \text{ var } \{ , \text{ length } \{ , \text{ fill } \} \} \{ , \} \{ ; \} \{ \text{format} \} \{ \text{modes} \}
\{ \text{THEN statement(s)} \}
\{ \text{ELSE statement(s)} \}
\]

where

\( x, y \)

are the screen position (column and line) at which input is to occur.

\( \text{var} \)

is the variable in which the data is to be stored.

\( \text{length} \)

is the maximum length of data to be allowed. Because of a potential syntactic ambiguity in the language, this must be enclosed in brackets if it is an expression.

\( \text{fill} \)

is a Pick style fill expression. The first character will be used to fill any spare space in the input field to make the field width clear to the user. The second character will be used to fill unused space after input is completed. If the fill string is only one character, spaces will be used. The third character, if present causes the cursor to be left at the end of the input field instead of the default action of leaving it after the last character of the entered data string. The fill option cannot be used with a format mask, **PANNING** or **HIDDEN**.

\( \text{format} \)

is the format specification to be used for initial display of \( \text{var} \) and to redisplay the data on completion of input.

\( \text{modes} \)

are any combination of the following keywords:

- **APPEND**
  Position the cursor at the end of the data. Use of this keyword also implies EDIT mode.

- **EDIT**
  Starts in "edit" mode, suppressing the normal clearance of the input field if the first character entered by the user is a data character rather than an edit character.

- **HIDDEN**
  Echoes characters back to the screen as asterisks for password type fields.

- **NO.ECHO**
  Suppresses echo of input data.

- **NO.ENCODING**
  Suppresses decode of encoded input when the terminal is set to operate in UTF-8 mode. Use of this mode will also suppress encoding of data echoed back to the terminal.

- **OVERLAY**
  Starts in "overlay" mode where data entered by the user replaces the
PANNING

Allows entry of an unlimited number of characters in a field width of the given length by panning the data if it is longer than the display width. Use of this option requires length to be specified and implies the presence of the underscore.

TIMEOUT wait

Sets a timeout period in seconds. If input is not received in this time, the INPUT terminates, leaving var unchanged. The keywords FOR or WAITING can be used in place of TIMEOUT for compatibility with other environments.

UPCASE

converts the input data to uppercase.

The comma after the cursor position is optional and has no effect on the operation of the statement.

The optional THEN and ELSE clauses used with TIMEOUT allow a program to determine whether the input timed out. Successful input executes the THEN clause. A timeout will execute the ELSE clause.

The INPUT @ statement reads data from the DATA queue or, if there is no stored data, from the keyboard.

Keyboard Input

When reading from the keyboard, the current prompt character will be displayed to the left of the given input position. No prompt is displayed if the input column position, x, is zero or if the prompt has been disabled using the PROMPT statement. The prompt character will be removed from the screen on completion of the input.

If the colon character before var is present, the original contents or var are displayed in the input area and entry commences in overlay mode. If the colon character before var is not present, entry commences in insert mode with a blank field.

The user has three options:

- Pressing the return key retains the original content of var.
- Typing a data character replaces the original content of var, clearing any old displayed data (unless the EDIT option is used).
- Using an edit key (see below) allows the old data to be edited.

The values stored for printing characters are the ASCII characters associated with the key. Non-printing characters result in stored character values as listed under Character Values for Terminal Input.

If no length expression is included, data characters are stored until the return key is pressed.

If length is specified, up to that number of characters may be entered after which input is automatically terminated as though the return key had been pressed, any subsequent key entries being retained for the next INPUT statement. The return key is not stored as part of the input data.
The **INPUT @** statement may not behave correctly if the *length* of the input field causes it to extend over multiple lines and the terminal in use does not automatically wrap from one line to the next when displaying long text output.

The optional underscore component of the statement suppresses the automatic input termination when *length* characters have been entered. Any number of characters may be entered but only *length* characters will be displayed.

The optional colon causes the carriage return and line feed output when the return key is used or on reaching the input length limit to be suppressed.

In all cases, the following editing keys are available.

- **Ctrl-A** Home: Position the cursor at the start of the input data
- **Ctrl-B** Cursor left: Move the cursor left one character
- **Ctrl-D** Delete: Delete character under cursor
- **Ctrl-E** End: Position the cursor at the end of the input data
- **Ctrl-F** Cursor right: Move the cursor right one character
- **Ctrl-H** Backspace: Backspace one character
- **Ctrl-K** Insert: Delete all characters after the cursor

These editing keys can be modified using the **KEYEDIT** statement.

When the return key is pressed to terminate input, if a format is specified, the data is redisplayed using this mask to apply format rules such as right justification.

**DATA Queue Input**

Where the data queue is not empty, the **INPUT @** statement reads the item at the head of this queue, copying it verbatim to *var* with no processing of any embedded control characters. The *length* expression is ignored. The item is displayed as though it had been typed but the prompt character is not displayed. Unless the NO.ECHO.DATA mode of the **SMODE** compiler directive or the NO.ECHO.DATA setting of the **OPTION** command is enabled, the item is displayed as though it had been typed.

Use of the **STATUS()** function after an **INPUT @** statement returns zero unless input was terminated by a key defined using the **KEYEXIT** or **KEYTRAP** statements.

**Use of Pipes**

QM recognises input from pipes as a special case. Programs that process data from a pipe can read the data using the same QMBasic statements and functions as for keyboard input. If the end of the
data is reached, a subsequent INPUT @ will return a null string. The STATUS() function will return ER$EOF.

Examples

```
DISPLAY @(0,10) : "Account " :
PROMPT ""
INPUT @(8, 10) : ACCOUNT.NO, 16
```

This program fragment displays the current value of ACCOUNT.NO with a suitable annotation and accepts input. Pressing the RETURN key alone will retain the original value as displayed. The PROMPT statement has been used to suppress display of the prompt character.

```
INPUT @(10, 5) : PRICE, 10 '10R'
```

This program fragment inputs a value for the PRICE variable, redisplaying it right justified on completion of input.

See also: BINDKEY(), INPUTFIELD, INPUTIF, INPUTNULL, KEYCODE(), KEYEDIT, KEYEXIT, KEYTRAP
**INPUTCSV**

The `INPUTCSV` statement enables entry of CSV format data from the keyboard or from previously stored `DATA` statements.

**Format**

```
INPUTCSV var1, var2, ...
```

where

```
var1 var2, ...
```

are the variables to receive the input data.

The `INPUTCSV` statement reads CSV format data from the `DATA` queue or, if there is no stored data, from the keyboard. This data is then parsed into the named variables.

If there are insufficient data items entered to populate all the named variables, any unused variables are set to null strings. If there are more data items entered than the number of variables, the excess data is discarded.

**Keyboard Input**

When taking input from the keyboard, the current prompt character will be displayed prior to reading data. The values stored for printing characters are the ASCII characters associated with the key. Non-printing characters result in other stored character values.

The `INPUTCSV` statement recognises the backspace key, allowing this to be used to correct data entry errors. The terminfo system allows the code sent by the backspace key to be redefined from its default `char(8)`. If an alternative, single byte definition is used, `INPUTCSV` will honour this, otherwise `char(8)` is used as the backspace.

**DATA Queue Input**

Where the data queue is not empty, the `INPUTCSV` statement reads the item at the head of this queue, copying it verbatim to `var` with no processing of any embedded control characters. The item is displayed as though it had been typed.

**Example**

```
INPUTCSV PROD.NO, QTY
```

This statement parses the entered data into the PROD.NO and QTY variables.

**See also:**

`PRINTCSV`, `READCSV`, `WRITECSV`
The **INPUTFIELD** and **INPUTFIELDV** statements enable entry of data from the keyboard at a specific screen position or from previously stored **DATA** statements. They differ from **INPUT @** in that they terminate on entry of any control character not recognised as an editing key. This allows application software to capture and handle control and function keys.

**Format**

```plaintext
INPUTFIELD @(x, y) {,} {;} var, length {,} fill {,} {;} {format} {modes}
{THEN statement(s)}
{ELSE statement(s)}

INPUTFIELDV @(x, y) {,} {;} var, length {,} fill {,} {;} {format} {modes}
{THEN statement(s)}
{ELSE statement(s)}
```

where

- **x, y** are the screen position (column and line) at which input is to occur.
- **var** is the variable in which the data is to be stored.
- **length** is the maximum length of data to be allowed. Because of a potential syntactic ambiguity in the language, this must be enclosed in brackets if it is an expression.
- **fill** is a Pick style fill expression. The first character will be used to fill any spare space in the input field to make the field width clear to the user. The second character will be used to fill unused space after input is completed. If the fill string is only one character, spaces will be used. The third character, if present causes the cursor to be left at the end of the input field instead of the default action of leaving it after the last character of the entered data string. The fill option cannot be used with a format mask, **PANNING** or **HIDDEN**.
- **format** is the format specification to be used for initial display of var and to redisplay the data on completion of input.
- **modes** are any combination of the following keywords:

  - **APPEND** Position the cursor at the end of the data. Use of this keyword also implies **EDIT** mode.
  - **EDIT** Starts in "edit" mode, suppressing the normal clearance of the input field if the first character entered by the user is a data character rather than an edit character.
  - **HIDDEN** echoes characters back to the screen as asterisks for password type fields.
  - **NO.ECHO** Suppresses echo of input data.
  - **OVERLAY** Starts in "overlay" mode where data entered by the user replaces the
character under the cursor rather than being inserted.

**PANNING**
Allows entry of an unlimited number of characters in a field width of the given length by panning the data if it is longer than the display width. Use of this option requires length to be specified and implies the presence of the underscore.

**TIMEOUT** wait
Sets a timeout period in seconds. If input is not received in this time, the `INPUTFIELD` terminates, leaving var unchanged. The keywords FOR or WAITING can be used in place of TIMEOUT for compatibility with other environments.

**UPCASE**
converts the input data to uppercase.

The comma after the cursor position is optional and has no effect on the operation of the statement.

The optional THEN and ELSE clauses used with TIMEOUT allow a program to determine whether the input timed out. Successful input executes the THEN clause. A timeout will execute the ELSE clause.

The `INPUTFIELD` statement reads data from the DATA queue or, if there is no stored data, from the keyboard.

The `INPUTFIELD` statement works similarly to the `INPUT @` statement except that entry of any control character not recognised as an editing function (see `INPUT @`) terminates data entry. The `STATUS()` function can be used to determine the key that caused exit. This will return zero for the return key and the internal key code (128 to 227) for any other key. See the KEYIN.H include record in the SYSCOM file for key values. The tokens defining these special key values are prefixed with K$.

The `INPUTFIELDV` statement is identical except that the value returned via the `STATUS()` function will be the Unicode BMP Private Use Area code for the special key that terminated input. The token definitions for these values in the KEYIN.H include record are prefixed with KV$.

When the return key is pressed to terminate input, if a format is specified, the data is redisplayed using this mask to apply format rules such as right justification.

**Keyboard Input**

When reading from the keyboard, the current prompt character will be displayed to the left of the given input position. No prompt is displayed if the input column position, x, is zero or if the prompt has been disabled using the PROMPT statement. The prompt character will be removed from the screen on completion of the input.

If the colon character before var is present, the original contents or var are displayed in the input area and entry commences in overlay mode. If the colon character before var is not present, entry commences in insert mode with a blank field.

The user has three options:
- Pressing the return key retains the original content of var.
Typing a data character replaces the original content of var, clearing any old displayed data (unless the EDIT option is used).

Using an edit key (see below) allows the old data to be edited.

The values stored for printing characters are the ASCII characters associated with the key. Non-printing characters result in stored character values as listed under Character Values for Terminal Input.

If no length expression is included, data characters are stored until the return key is pressed.

If length is specified, up to that number of characters may be entered after which input is automatically terminated as though the return key had been pressed, any subsequent key entries being retained for the next INPUT statement. The return key is not stored as part of the input data.

The INPUTFIELD statement may not behave correctly if the length of the input field causes it to extend over multiple lines and the terminal in use does not automatically wrap from one line to the next when displaying long text output.

The optional underscore component of the statement suppresses the automatic input termination when length characters have been entered. Any number of characters may be entered but only length characters will be displayed.

The optional colon causes the carriage return and line feed output when the return key is used or on reaching the input length limit to be suppressed.

DATA Queue Input

Where the data queue is not empty, the INPUT @ statement reads the item at the head of this queue, copying it verbatim to var with no processing of any embedded control characters. The length expression is ignored. The item is displayed as though it had been typed.

See also: BINDKEY(), INPUT@, INPUTNULL, KEYCODE(), KEYEDIT, KEYEXIT, KEYTRAP
**INPUTIF**

The **INPUTIF** statement takes input from the keyboard type-ahead buffer or the data queue, if available.

**Format**

```
INPUTIF var {, length {, fill} } {_,} {;} {modes}
{THEN statement(s)}
{ELSE statement(s)}
```

```
INPUTIF @(x, y) {,} {;} var {, length {, fill} } {_,} {;} {modes}
{THEN statement(s)}
{ELSE statement(s)}
```

where

- **x, y** are the screen position (column and line) at which input is to occur.
- **var** is the variable in which the data is to be stored.
- **length** is the maximum number of characters to be allowed.
- **fill** is a Pick style fill expression. The first character will be used to fill any spare space in the input field to make the field width clear to the user. The second character will be used to fill unused space after input is completed. If the **fill** string is only one character, spaces will be used. The third character, if present causes the cursor to be left at the end of the input field instead of the default action of leaving it after the last character of the entered data string. The **fill** option cannot be used with **HIDDEN**.
- **modes** are any combination of the following keywords:
  - **HIDDEN** echoes characters back to the screen as asterisks for password type fields.
  - **NO.ECHO** Suppresses echo of input data.
  - **NO.ENCODING** Suppresses decode of encoded input when the terminal is set to operate in UTF-8 mode. Use of this mode will also suppress encoding of data echoed back to the terminal.
  - **RAW** Causes **INPUT** to place input data into **var** without special interpretation of newline, backspace, etc. Used in this way, the **length** element of the statement must be present.
  - **UPCASE** converts input data to uppercase.

At least one of the **THEN** and **ELSE** clauses must be present. If there is no keyboard type-ahead and the data queue is empty, the **ELSE** clause is executed.

See the **INPUT** and **INPUT@** statements for details of input editing.
**INPUTNULL**

The **INPUTNULL** statement sets a special character to cause **INPUT @** and **INPUTFIELD** to return a null string.

**Format**

```
INPUTNULL expr
```

where

```
expr  evaluates to the character to be set. If the expr string is more than one character, the first character is used.
```

The **INPUTNULL** statement is used with **INPUT @** and **INPUTFIELD**. It sets a character that, if supplied as a single character response to these statements, causes them to return a null string result. Any data echoed by the input action remains as typed.

Specifying `expr` as a null string cancels recognition of the special character. This also occurs automatically on return to the command prompt.

**See also:**

**INPUT@**, **INPUTFIELD**, **INPUTTRAP**
INPUTTRAP

The INPUTTRAP statement defines special actions for INPUT @ and INPUTFIELD.

Format

INPUTTRAP expr GOTO label1 {, label2...}
INPUTTRAP expr GOSUB label1 {, label2...}

where

expr evaluates to a string containing the characters to be trapped.
label1... are labels within the current program module.

The INPUTTRAP statement defines a set of characters that to be handled as special cases if used as a single character string entered for INPUT @ or INPUTFIELD. When the typed string is one of the characters defined in expr, the program performs a GOTO or GOSUB to the corresponding label from the list supplied in the INPUTTRAP statement. The variable into which input was being performed will be set to a null string.

If the number of characters in expr exceeds the number of labels, detection of a trapped character for which there is no corresponding label will jump to the last label in the list.

Specifying expr as a null string cancels any currently active INPUTTRAP action. Used in this way the syntax is unchanged and hence there must be at least one label present even though it will never be used.

When using the GOSUB form, returning from the subroutine will continue execution at the statement following the INPUTTRAP, not the statement following the INPUT @ or INPUTFIELD.

If both INPUTTRAP and INPUTNULL are used for the same character, INPUTNULL has precedence.

See also:
INPUT@, INPUTFIELD, INPUTNULL
The **INS** statement and **INSERT()** function insert a field, value or subvalue into a dynamic array. Used with a data collection, the **INS** statement inserts a name/value pair.

**Format**

```plaintext
INS string BEFORE dyn.array<field {, value {, subvalue}}>

INSERT(dyn.array, field, value, subvalue, string)

INSERT(dyn.array, field {, value {, subvalue}} ; string)

INS item AS collection{path}
```

where

- **string** is the string to be inserted.
- **dyn.array** is the dynamic array into which the item is to be inserted.
- **field** evaluates to the number of the field before which insertion is to occur.
- **value** evaluates to the number of the value before which insertion is to occur. If omitted or zero, value 1 is assumed.
- **subvalue** evaluates to the number of the subvalue before which insertion is to occur. If omitted or zero, subvalue 1 is assumed.
- **item** is a data item value to be inserted into a data collection.
- **collection** is a data collection variable.
- **path** is the element path to be used for the inserted item.

**Dynamic Arrays**

The **string** is inserted before the specified field, value or subvalue of the dynamic array. The **INS** statement assigns the result to the **dyn.array** variable. The **INSERT()** function returns the result without modifying **dyn.array**.

A negative value of **field, value** or **subvalue** causes the next item at this level to be appended. For example,

```plaintext
INS X BEFORE S<3, -1>
```

appends X as a new value at the end of field 3 of S. See the description of the **S<f,v,sv> assignment operator** for a discussion of how QM appends items.

Additional delimiters will be added to reach the specified field, value and subvalue unless the **string** to be inserted is null. Absent fields, values and subvalues are assumed to be null so there is no need to insert additional marks in this case.

Setting the `COMPATIBLE.APPEND` option of the `MODE` compiler directive modifies the behaviour such that a mark character is not inserted if the final element of the portion of the dynamic array to which data is being appended is null.
Data Collections

The INS statement inserts the supplied item value into the named data collection at the specified element path. Note that when used in this way, the curly brackets surrounding the path are part of the syntax, not markers for optional items.

If the element path corresponds to an array element, the final component of the element path must be numeric and is the array index position at which the item is to be inserted. Any elements at or beyond this position are moved to make space. The index value must not be greater than one more than the existing number of elements in the array.

If the element path does not correspond to an array element, the new item is inserted. If the path already exists, the named collection element is replaced.

Examples

LOCATE PART.NO IN PARTS<1> BY "AL" SETTING I ELSE
   INS PART.NO BEFORE PARTS<I>
END

This program fragment locates PART.NO in a sorted list PARTS and, if it is not already present, inserts it.

INS CLIENT.NAME AS CLIENT.DATA{"CONTACT.NAME"}

This statement inserts or updates the CONTACT.NAME element of the CLIENT.DATA collection.

See also:
Data collections, DEL, DELETE(), EXTRACT(), FIND, FINDSTR, LISTINDEX(), LOCATE, LOCATE(), REPLACE()
**INT()**

The **INT()** function returns the integer part of a value.

**Format**

```
INT(expr)
```

where

`expr` evaluates to a number or a numeric array.

The **INT()** function returns the integer part of `expr`. Any fractional part after rounding in accordance with the **INTPREC** configuration parameter is discarded such that **INT(1.9)**, for example, evaluates to 1.

If `expr` is a numeric array (a dynamic array where all elements are numeric), the **INT()** function operates on each element in turn and returns a numeric array with the same structure as `expr`.

**Example**

```
N = INT(A / B)
```

This statement finds the integer part of the quotient of `A / B` and assigns this to `N`. The **IDIV()** function provides an alternative way to achieve this but is specific to QMBasic.

**See also:**

**BOOL(), NUMERIC(), STR()**
IS.ALNUM(), IS.ALPHA(), IS.DIGIT(), IS.GRAPH(), IS.MARK(), IS.SPACE(), IS.WIDE()

This set of eight functions test whether a character is of a specific type.

**Format**

```
IS.ALNUM(char)
IS.ALPHA(char)
IS.DIGIT(char)
IS.GRAPH(char)
IS.MARK(char)
IS.SPACE(char)
IS.WIDE(char)
IS.USER.CHAR(char)
```

where

```
char
```

is the character to be tested.

If `char` is more than one character, only the first character is tested. If `char` is a null string, the function will return False.

The character maps assign type attributes to each character. These functions allow an application to test whether a character has a particular attribute set. The functions return True if the attribute is set, False if it is not set.

The **IS.ALNUM()** function tests whether a character is defined as being either a letter or a digit.

The **IS.ALPHA()** function tests whether a character is defined as being a letter.

The **IS.DIGIT()** function tests whether a character is defined as being a digit.

The **IS.GRAPH()** function tests whether a character is defined as being a graphical (printing) character. Note that the space is not included.

The **IS.MARK()** function tests whether a character is defined as being a mark character (251-255). The position of these in the character set and their meaning cannot be changed in user defined character maps.

The **IS.SPACE()** function tests whether a character is defined as being whitespace. Unicode defines several code points as being spaces of various types.

The **IS.WIDE()** function tests whether a character requires two display positions.

The **IS.USER.CHAR()** function tests whether a character has the user defined attribute set.

On a non-ECS mode system, the default behaviour for members of the 8-bit character set are fixed. The alphabetic characters are A-Z and a-z, the digits are 0-9, the graphical characters 33 to 126, the marks are the five mark characters and the space is only the space character. There is limited
support for user defined character maps created using the `EDIT.MAP` command and loaded either from use of the `CHARMAP` configuration parameter or by use of the `CHAR.MAP` command.

Example

```plaintext
S1 = ECHAR(0x03A3)
DISPLAY IS.ALPHA(S1)
```

The above program fragment tests whether `S1`, which contains a single character with Unicode code point value U+03A3 (uppercase Greek Sigma), is a letter. It will display 1 (True) when using a Greek character map.

See also:

`IS.WIDESTR()`
**IS.ECS()**

The **IS.ECS()** function tests whether a character string contains ECS characters.

**Format**

```
IS.ECS(string)
```

where

```
string   is the string to be tested.
```

The **IS.ECS()** function tests whether a character string contains characters outside the 8-bit set and hence requires an ECS mode file or an encoding such as UTF-8 to store it. The function returns True if any of the characters in `string` is outside the 8-bit set, otherwise it returns False.

Used on a non-ECS mode system, the **IS.ECS()** function always returns False.

**Examples**

```vbnet
S1 = 'ABC'
S2 = ECHAR(0x03A3)
DISPLAY IS.ECS(S1), IS.ECS(S2)
```

The above program fragment creates two strings, `S1` containing the ASCII characters "ABC" and `S2` containing the character with Unicode code point value U+03A3 (uppercase Greek Sigma). It then displays the result of testing each string with the **IS.ECS()** function,

```
0     1
```
IS.NULL()

The IS.NULL() function tests whether a data item holds the SQL style null value.

**Format**

    IS.NULL(var)

where

    var    is the variable to be tested.

QM provides very limited support for the SQL style null data value. This is intended for use only with the [JPARSE()](#) and [JBUILD()](#) functions where it is necessary to parse and rebuild a JSON string that contains null items.

The IS.NULL() function tests whether a variable contains the SQL style null value.

See also:

[@NULL](#)
IS.WIDESTR()

The **IS.WIDESTR()** function determines whether a string contains any wide characters.

**Format**

\[
\text{IS.WIDESTR}(\text{string})
\]

where

\[
\text{string} \quad \text{is the string to be tested.}
\]

On ECS mode systems, the character maps assign type attributes to each character. The "wide" (or double width) attribute identifies that the associated character occupies two positions on a terminal display or printer. The **IS.WIDESTR()** function returns True if the supplied *string* contains one or more wide characters, False if there are none.

On a non-ECS mode system, the attributes applicable to the members of the 8-bit character set are fixed and there is no concept of a wide character. This function always returns False on such systems.

**Example**

```
INPUT TITLE
IF IS.WIDESTR(TITLE) THEN TITLE = SUBSTRDW(TITLE, 1, WIDTH)
```

The above program fragment gets data from the user terminal and, if it contains wide characters, truncates it to fit in a specific width.

**See also:**

**IS.WIDE()**
**ITYPE()**

The ITYPE() function executes a compiled I or C-type dictionary record or an A or S-type with a correlative.

**Format**

\[
\text{ITYPE}(\text{itype})
\]

where

\[
\text{itype} \quad \text{is a previously compiled record read from a dictionary.}
\]

The ITYPE() function evaluates expression compiled as part of the given I-type dictionary record and returns its result. C-type and A/S-type records with correlatives can also be evaluated using this function.

The working environment for the ITYPE() function must be established by setting @ID to the key of the record being processed and @RECORD to the record data if these are used by the expression. These variables are also used internally by the query processor to store the id and data for the current record. If the expression evaluated by the ITYPE() function calls a subroutine that modifies these variables, the original values must be saved and reinstated if the dictionary item is to be usable within the query processor.

The dictionary record must have been compiled before the ITYPE() function is used.

If the byte ordering of the object code in the itype variable is not the same as that of the machine on which it is being executed, it will be converted automatically and the itype variable will be modified to contain the converted object code so that a subsequent call to the ITYPE() function will not repeat the conversion.

The ITYPE() function can also be used to evaluate D-type items and A or S type items that do not include a correlative. Although the performance of this use of the function will be significantly lower than simple field extraction in the calling program, it may allow some programs that do not require best performance to adopt a generalised interface for all dictionary record types that define data item values.

**Example**

```
READ IREC FROM DICT.FILE, "AGE" THEN
   @RECORD = REC
   AGE = ITYPE(IREC)
END
```

This program fragment reads dictionary record "AGE" to IREC. This might, perhaps, be an I-type to calculate a person’s age from their date of birth. The @RECORD variable is set to the data to be processed and the ITYPE() function is used to execute the I-type.

See also:

**EVALUATE**
**JBUILD()**

The **JBUILD()** function constructs a JSON string from a data collection.

**Format**

```
JBUILD(var)
```

```
JBUILD(var, mode)
```

where

*var* is a data collection variable.

*mode* is an additive value specifying the manner in which the JSON string is formed.

- **JBSNUM** Save numeric strings as numbers, where possible.
- **JBSINVALID** Save invalid data types (e.g. file variables) as null.
- **JBSNULL** Save empty strings as "null".

The **JBUILD()** function transforms the content of a QMBasic data collection variable into a JSON (JavaScript Object Notation) string. If successful, the JSON string is returned as the value of the function and the **STATUS()** function returns zero.

If the **JBUILD()** function fails, a null string is returned and the **STATUS()** function returns ER$BUILD.ERROR. Failure will occur if the data collection contains any elements that do not have a JSON representation such as a file variable.

**Example**

```
VAR = JPARSE(JSON.STRING)
VAR{'client/name'} = NEW.CLIENT.NAME
JSON.STRING = JBUILD(VAR)
```

The above program fragment parses a JSON string into a data collection variable, modifies the client name element and rebuilds the JSON representation of the modified data.

**See also:**

*Data collections, JPARSE()*
JPARSE()

The JPARSE() function parses a JSON string into a data collection.

Format

\[ \text{JPARSE}(\text{var}) \]

where

\[ \text{var} \] is a JSON string.

The JPARSE() function transforms a JSON (JavaScript Object Notation) string into a data collection variable. If successful, the return value of the function is a data collection and the \text{STATUS()} function returns zero.

If the JPARSE() function fails, a null string is returned and the \text{STATUS()} function returns a non-zero error code. The \text{INMAT()} function will return the character position in the JSON string at which the error was detected.

Example

\begin{verbatim}
VAR = JPARSE(JSON.STRING)
VAR['client/name'] = NEW_CLIENT.NAME
JSON.STRING = JBUILD(VAR)
\end{verbatim}

The above program fragment parses a JSON string into a data collection variable, modifies the client name element and rebuilds the JSON representation of the modified data.

See also:
Data collections, JBUILD()
**KEEP.SELECT**

The **KEEP.SELECT** statement indicates that the default select list should be retained on return to the command processor.

**Format**

```
KEEP.SELECT
```

The default behaviour of QM is that the default select list (list 0) is not cleared on return to the command processor, allowing a program to create a list that will be used by a subsequent command. As a side-effect of this behaviour, a program that creates a select list and only partially processes it before terminating will leave the remainder of the list active, potentially affecting later actions.

The CLEAR.SELECT mode of the **OPTION** command causes QM to clear the default select list on return to the command processor. Any program that intentionally leaves an active list on termination must use the **KEEP.SELECT** statement to prevent the command processor clearing the list. This statement has no effect if the CLEAR.SELECT option is not active.

**See also:**

Select lists
KEYCODE()

The KEYCODE() and KEYCODEV() functions read a single keystroke from the keyboard.

Format

    KEYCODE({timeout})
    KEYCODEV({timeout})

The KEYCODE() function pauses program execution until a key is pressed. The character associated with that key is returned as the value of the function. The character is not echoed to the display.

The KEYCODEV() function is similar but returns the code point value of the key rather than the character corresponding to the key.

KEYCODE() and KEYCODEV() do not take data from the DATA queue.

The optional timeout parameter specifies a period in seconds after which the function will return if no input is received. In this case the returned value is a null string and the STATUS() function will return ER$TIMEOUT.

A null string is also returned when taking input from a pipe if the piped data stream is exhausted. In this case the STATUS() function will return ER$EOF. The value returned by the STATUS() function is not significant unless KEYCODE() has returned a null string.

The KEYCODE() function differs from KEYIN() in that it uses the terminfo database to identify certain special keys and returns the character representing the internal representation of that key as defined in the KEYIN.H include record in the SYSCOM file. The special keys recognised by this function are:

- Function keys F1 to F12
- Left, right, up and down cursor keys
- Page up and Page down keys
- Home and End
- Insert and Delete
- Back-tab
- Mouse

Use of the Escape key will return char(27) if no further characters are received within 200mS. This mechanism can be disabled using the BINDKEY() function.

The KEYCODEV() function has a similar relationship with KEYINV(), returning the code point value. For the special keys recognised by the key bindings, the returned value will correspond to positions in the Unicode BMP Private Use Area that QM assigns to the keys. The function is valid on non-ECS systems but the returned values are usually not able to be stored as single characters.

Click here for a table of the values returned by these functions. These are also defined in the KEYIN.H include record in the SYSCOM file.

Handling Mouse Clicks

The mouse code is returned when the mouse control prefix sequence defined in the terminfo database is detected. The way in which mouse clicks are handled varies considerably between...
different terminal emulators. When using terminal types supported by standard QM terminfo definitions the `KEYCODE()` and `KEYCODEV()` functions additionally process the qualifying data sent by the terminal device, saving the button number in `@MBUTTON` and the screen coordinates at which the mouse was clicked in `@MCOL` and `@MROW`.

Mouse click detection is disabled by default. It can be enabled or disabled by use of the QMBasic `MOUSE` statement. See the `MOUSE` statement for more information.

**Example**

```
K = KEYCODEV()
BEGIN CASE
  CASE K = KV$LEFT ;* Cursor left key
    DISPLAY 'Seen cursor left key'
  CASE K = KV$RIGHT ;* Cursor right key
    DISPLAY 'Seen cursor right key'
  CASE K = KV$MOUSE ;* Mouse click
    DISPLAY 'Mouse clicked at ' : @MCOL : ',' : @MROW
    ...etc...
END CASE
```

The KV$xxx tokens are defined in the KEYIN.H include record in the SYSCOM file.

**See also:**
- Terminfo database
- `BINDKEY()`, `MOUSE`, `TERMINFO()`
KEYEDIT

The **KEYEDIT** statement defines editing keys for use with **INPUT @**.

**Format**

```
KEYEDIT (action, key), (action, key), ...
```

where

- **action** identifies the editing action to be performed when the key is pressed. This may be:
  - 2 Cursor left
  - 3 Return
  - 4 Backspace
  - 6 Cursor right
  - 7 Insert character (treated as action 13)
  - 8 Delete character
  - 13 Toggle insert mode
  - A negative action value removes the key binding specified by **key**.

- **key** identifies the key to be bound to the given **action**. This is specified as a numeric value:
  - 1 to 31 Use the control key with this character value. Ctrl-A is 1, Ctrl-B is 2, etc.
  - 32 to 159 Use the Escape key followed a character value of **key** - 32 (e.g. Esc-A is 97).
  - 160+ Use a sequence of up to four characters constructed from the bytes sent by the key to be trapped starting from the low order byte plus 160. See below for an example.

The **KEYEDIT** statement adds user defined alternative key bindings to the standard set used by the **INPUT @** statement. These may validly replace default bindings. The newly bound keys remain in effect until either they are rebound by a further **KEYEDIT** statement or the process returns to the command prompt.

The **INPUT @** statement checks for keys bound via the terminfo system or **KEYEDIT** before using the standard default bindings.

**Example of a multi-byte key sequence**

The F2 key of a vt100 terminal using the vt100-at definition with AccuTerm sends a three character sequence of "Esc O Q". The hexadecimal values of these characters are 1B, 4F, 51. The **key** value is formed by concatenating bytes with these character values in reverse order and adding 160. The simplest way to do this is

```
KEY = XTD('514F1B') + 160
```
See also:
BINDKEY(), INPUT@, INPUTFIELD, KEYCODE(), KEYEXIT, KEYTRAP
KEYEXIT

The **KEYEXIT** statement defines exit keys for use with **INPUT @**.

**Format**

```plaintext
KEYEXIT (action, key), (action, key), ...
```

where

- `action` is a user defined value in the range 1 to 255 to be returned by the **STATUS()** function following an **INPUT @** that is terminated by use of the key defined by `key`.

- A negative `action` value removes the key binding specified by `key`.

- `key` identifies the key to be bound to the given `action`. This is specified as a numeric value:
  - 1 to 31: Use the control key with this character value. Ctrl-A is 1, Ctrl-B is 2, etc.
  - 32 to 159: Use the Escape key followed a character value of `key` - 32 (e.g. Esc-A is 97).
  - 160+: Use a sequence of up to four characters constructed from the bytes sent by the key to be trapped starting from the low order byte plus 160. See below for an example.

The **KEYEXIT** statement defines one or more keys that will terminate an **INPUT @** statement. When any of these keys in pressed the **INPUT @** returns with the input data as entered up to the moment when this key was used. The **STATUS()** function will return the value defined by `action` for the key.

See the **KEYTRAP** statement for a method to return the original data.

**Example of a multi-byte key sequence**

The F2 key of a vt100 terminal using the vt100-at definition with AccuTerm sends a three character sequence of "Esc O Q". The hexadecimal values of these characters are 1B, 4F, 51. The `key` value is formed by concatenating bytes with these character values in reverse order and adding 160. The simplest way to do this is

```
KEY = XTD('514F1B') + 160
```

See also: **BINDKEY()**, **INPUT@**, **INPUTFIELD**, **KEYCODE()**, **KEYEDIT**, **KEYTRAP**
KEYIN()

The `KEYIN()`, `KEYINC()` and `KEYINR()` functions read a single keystroke from the keyboard.

Format

```
KEYIN({timeout})
KEYINC({timeout})
KEYINR({timeout})
```

The `KEYIN()` function pauses program execution until a key is pressed. The character associated with that key is returned as the value of the function. The character is not echoed to the display.

The `KEYINC()` function is identical except that the case of alphabetic characters is inverted if case inversion has been enabled.

The `KEYINR()` function reads a single character with no internal processing. In particular, null characters are not removed and `char(10)` and `char(13)` are passed through without any special handling.

*The KEYINR() function is redundant from release 2.1-8 as QM now honours the setting of the telnet binary mode parameter. The function will be retained as a synonym for KEYIN() for the foreseeable future.*

`KEYIN()`, `KEYINC()` and `KEYINR()` do not take data from the `DATA` statement queue.

The optional `timeout` parameter specifies a period in seconds after which the function will return if no input is received. In this case the returned value is a null string and the `STATUS()` function will return ER$TIMEOUT. The timeout value may be fractional to specify timeouts of less than one second. Values less than 10mS or greater than 24 hours may not behave correctly.

A null string is also returned when taking input from a pipe if the piped data stream is exhausted. In this case the `STATUS()` function will return ER$EOF. The value returned by the `STATUS()` function is not significant unless `KEYIN()` has returned a null string.

Character Values

On Windows console sessions, special keys such as the function keys, ALT sequences and special keys (Home, Delete, cursor moves, etc) are represented by characters with values in the range 128 to 227, effectively rendering the characters normally in these positions unusable with these functions. See `KEYINV()` for an alternative way to read a single keystroke. [Click here for a table of key code values.](#)

See also:

`IN`, `KEYCODE()`, `KEYINV()`, `KEYREADY()`
KEYINV()

The **KEYINV()** and **KEYINCV()** functions read a single keystroke from the keyboard.

**Format**

```
KEYINV({timeout})
KEYINCV({timeout})
```

The **KEYINV()** function pauses program execution until a key is pressed. The code point value of the character associated with that key is returned as the value of the function. The character is not echoed to the display.

The **KEYINCV()** function is identical except that the case of alphabetic characters is inverted if case inversion has been enabled.

**KEYINV()** and **KEYINCV()** do not take data from the **DATA** statement queue.

The optional **timeout** parameter specifies a period in seconds after which the function will return if no input is received. In this case the returned value is a null string and the **STATUS()** function will return ER$TIMEOUT. The timeout value may be fractional to specify timeouts of less than one second. Values less than 10mS or greater than 24 hours may not behave correctly.

A null string is also returned when taking input from a pipe if the piped data stream is exhausted. In this case the **STATUS()** function will return ER$EOF. The value returned by the **STATUS()** function is not significant unless **KEYIN()** has returned a null string.

**Character Values**

On Windows console sessions, special keys such as the function keys, ALT sequences and special keys (Home, Delete, cursor moves, etc) are represented by code point values that lie in the Unicode BMP Private Use Area. On non-ECS mode systems, it is not possible to encode these values as a single character. [Click here for a table of key code values.](#)

See also:

**IN, KEYCODE(), KEYIN(), KEYREADY()**
**KEYREADY()**

The **KEYREADY()** function tests for data entered at the keyboard.

**Format**

```
KEYREADY()
```

The **KEYREADY()** function tests whether characters have been typed at the keyboard, returning True if characters are waiting to be processed, False if not. It differs from use of the **INPUT** statement with a negative length value in that **KEYREADY()** checks the keyboard only whereas **INPUT** checks the keyboard and the **DATA** statement queue.

**KEYREADY()** always returns True when using piped input. The end of file condition can only be determined by a subsequent attempt to read data from the pipe.

Data detected by **KEYREADY()** may be read by a subsequent use of either **KEYIN()** or **INPUT**.

**See also:**

**IN**, **KEYIN()**
KEYTRAP

The **KEYTRAP** statement defines trap keys for use with **INPUT @**.

**Format**

```plaintext
KEYTRAP (action, key), (action, key), ...
```

where

- **action** is a user defined value in the range 1 to 255 to be returned by the **STATUS()** function following an **INPUT @** that is terminated by use of the key defined by **key**.
  
  A negative **action** value removes the key binding specified by **key**.

- **key** identifies the key to be bound to the given **action**. This is specified as a numeric value:
  
  - 1 to 31 Use the control key with this character value. Ctrl-A is 1, Ctrl-B is 2, etc.
  - 32 to 159 Use the Escape key followed a character value of **key** - 32 (e.g. Esc-A is 97).
  - 160+ Use a sequence of up to four characters constructed from the bytes sent by the key to be trapped starting from the low order byte plus 160. See below for an example.

The **KEYTRAP** statement defines one or more keys that will terminate an **INPUT @** statement. When any of these keys in pressed the **INPUT @** returns with the original value of the input variable. The **STATUS()** function will return the value defined by **action** for the key.

See the **KEYEXIT** statement for a method to return the input data as entered up to the moment when this key was used.

**Example of a multi-byte key sequence**

The F2 key of a vt100 terminal using the vt100-at definition with AccuTerm sends a three character sequence of "Esc O Q". The hexadecimal values of these characters are 1B, 4F, 51. The **key** value is formed by concatenating bytes with these character values in reverse order and adding 160. The simplest way to do this is

```
KEY = XTD('514F1B') + 160
```

See also: **BINDKEY()**, **INPUT @**, **INPUTFIELD**, **KEYCODE()**, **KEYEDIT**, **KEYEXIT**
LAST()

The LAST() function returns the final element of a string delimited by a specified character. The LASTS() function is similar to LAST() but operates on a multivalued string, returning a similarly structured dynamic array of results.

Format

LAST(string, delimiter)

where

string is the string from which the final substring is to be extracted.

delimiter evaluates to the delimiter character.

Multivalue applications frequently need to extract the final element of a delimited string. Traditionally, this is done using the FIELD() and DCOUNT() functions:

\[
\text{ITEM} = \text{FIELD}(\text{STR}, '\ast', \text{DCOUNT}(\text{STR}, '\ast'))
\]

The LAST() function in QM allows this to be done in a simpler and more efficient manner.

The COL1() and COL2() functions can be used to find the character positions of the extracted substring, returning the positions of the character immediately before and after the extracted item respectively. The COL2() value will always be one greater than the string length except when processing a null string in which case both COL1() and COL2() return zero. When using the LASTS() function, the values returned by the COL1() and COL2() functions relate to the final extracted item.

Use of the $MODE NOCASE.STRINGS compiler directive makes the delimiter case insensitive.

Examples

\[
\begin{align*}
A &= "1*2*3*4*5" \\
S &= \text{LAST}(A, '\ast')
\end{align*}
\]

This program fragment assigns the string "5" to variable S.

\[
\begin{align*}
A &= "1*2*3*4*5" : @VM : "A*BC" \\
S &= \text{LASTS}(A, '\ast')
\end{align*}
\]

This program fragment assigns the string "5VMC" to variable S.

See also:
COL1(), COL2(), FIELD(), FIELDSTORE()
LEFT()

The LEFT() function returns the leading part of a string.

Format

LEFT(string, len)

where

string is the string from which the leading substring is to be extracted.

len is the length of the substring to be extracted. A value less than 1 is replaced by 1.

The LEFT() function returns the leftmost length characters from the given string. If length is greater than the number of characters in the string, the entire string is returned.

This function is identical in effect to use of the substring assignment operator. Thus the following two statements are equivalent:

\[
\text{SUBSTR} = \text{LEFT} (\text{VAR}, \text{length})
\]

\[
\text{SUBSTR} = \text{VAR}[1, \text{length}]
\]

Example

\[
\begin{align*}
A & = "ABCDEFGHIJK" \\
S & = \text{LEFT} (A, 3)
\end{align*}
\]

This program fragment assigns the string "ABC" to variable S.

See also:

RIGHT()
LEN()

The **LEN()** function returns the length of a string. The **LENS()** function is similar to **LEN()** but operates on successive elements of a dynamic array, returning a similarly structured dynamic array of results.

**Format**

```plaintext
LEN(string)
```

where

- `string` is the string for which the length is to be returned.

The **LEN()** function returns the length of `string` (number of characters) including any trailing spaces. On ECS mode systems, this may not be the same as the number of screen positions required to display the string as some ECS characters are double width.

**Example**

```plaintext
LOOP
    DISPLAY "Enter account number: 
    INPUT ACCOUNT.NO
    WHILE LEN(ACCOUNT.NO) # 6
        PRINTERR "Invalid account number"
    REPEAT
```

This program fragment prompts for and inputs an account number. If it is not six characters in length, an error is displayed and the prompt is repeated.

See also:

**DISPLAY.WIDTH()**
LES()

The LES() function processes two dynamic arrays, returning a similarly structured result array indicating whether elements of the first array are less than or equal to corresponding elements of the second array.

Format

   LES(expr1, expr2)

where

   expr1 and expr2 are the dynamic arrays to be compared.

The LES() function compares corresponding elements of the dynamic arrays expr1 and expr2, returning a similarly structured dynamic array of True / False values indicating the results of the comparison.

The REUSE() function can be applied to either or both expressions. Without this function, any absent trailing values are taken as zero.

Example

   A contains 11FM0VM14VMABCFM2
   B contains 12FM0VM14VMACBFM2

   C = LES(A, B)

   C now contains 1FM1VM0VM0FM1

See also:

   ANDS(), EQS(), GES(), GTS(), IFS(), LTS(), NES(), NOTS(), ORS(), REUSE()
LISTINDEX()

The LISTINDEX() function returns the position of an item in a delimited list.

Format

LISTINDEX(list, delimiter, item)

where

- list is the list to search.
- delimiter is the single character delimiter separating items in the list.
- item is the item to find.

The LISTINDEX() function returns the position of item in the delimited list. If it is not found, the function returns zero.

See the LOCATE statement for a more powerful way of searching dynamic arrays.

Examples

SUFFIX = FIELD(DOC.NAME, ".", DCOUNT(DOC.NAME, "."))
IF LISTINDEX("TXT,DOC,PDF", ",", SUFFIX) THEN
  DISPLAY DOC.NAME
END

This program fragment extracts the suffix from a Windows style file name and checks whether it is TXT, DOC or PDF. If so, the document name is displayed.

LISTINDEX(PROD.NO, @VM, PART); IF @ THEN QTY<1,@> ELSE ""

Used as an expression in a dictionary I-type item, this example searches field PROD.NO for an entry containing PART and extracts the corresponding entry from the QTY field. If the item is not found, a null string is returned.

See also:
DEL, DELETE(), EXTRACT(), FIND, FINDSTR, INS, INSERT(), LOCATE, LOCATE(), REPLACE()
LN()

The LN() function returns the natural log of a value.

Format

    LN(expr)

where

    expr evaluates to a number or a numeric array.

The LN() function returns the natural log of expr. It is the inverse of the EXP() function.

If expr is a numeric array (a dynamic array where all elements are numeric), the LN() function operates on each element in turn and returns a numeric array with the same structure as expr.

Example

    N = LN(X)

This statement finds the natural log of X and assigns this to N.

See also:

    EXP()
LOCAL

The **LOCAL** statement introduces an internal function or subroutine that may have private local variables.

**Format**

```
LOCAL {FUNCTION | SUBROUTINE} name{(args)}
PRIVATE vars
...statements...
END
```

where

- **name** is the name of the function subroutine.
- **args** is the comma separated list of arguments to the function or subroutine. An argument may reference a whole matrix by prefixing it with **MAT**. The variable names used for the arguments are visible only to the one function or subroutine and do not prevent use of the same name in other parts of the program module to reference a different variable. Enclose the argument name in parenthesis to pass it by value.

When passing a whole matrix as an argument to a local function or subroutine, a **DIM** statement can be used within the body of the local function or subroutine to re dimension the matrix. This is not the same behaviour as applying **DIM** to a matrix passed as an argument to an external subroutine where the **DIM** statement merely specifies the number of dimensions, the actual dimension values being ignored.

Further local variables that are private to the function or subroutine may be defined by immediately following the **LOCAL** statement by one or more **PRIVATE** statements. These contain a comma separated list of variable names which may be simple scalar items or matrices where the dimension values are numeric constants. Any variables referenced in the local function or subroutine but not declared as private are considered as having scope across the entire program module.

Functions and subroutines declared using **LOCAL** must be terminated with an **END** statement. The private variables declared using the **PRIVATE** statement have scope from the **LOCAL** statement until the corresponding **END** statement. Variables referenced in the main body of the program and in conventional internal subroutines are accessible unless they have the same name as a locally defined variable.

A local function must be defined using the **DEFFUN** statement with the **LOCAL** option before its first use in the program.

All labels within the local subroutine are private. It is not possible to jump into a subroutine declared with **LOCAL** except by using a **GOSUB** or **ON GOSUB** to its unique entry point. Similarly, it is not possible to jump to a label outside the local subroutine. Use of the **RETURN TO** statement is prohibited within a local subroutine.

If a local subroutine is called recursively, either directly or indirectly via some other intermediate subroutine, the local variables are stacked and the new invocation has its own private local variables.

There is a small performance cost by comparison to use of conventional internal subroutines due to the dynamic allocation of variables but this should be negligible in most applications.
When using the QMBasic debugger, local variables have a name formed from the subroutine name and variable name separated by a colon.

Examples

LOCAL SUBROUTINE SCAN.LIST
    PRIVATE I, N, REC
    N = DCOUNT(LIST, @FM)
    FOR I = 1 TO N
        READV REC FROM STOCK.F, LIST<I>, 0 ELSE
            DISPLAY LIST<I> : ' is not in stock file'
        END
    NEXT I
    RETURN
END

The above program fragment represents a local subroutine that scans a list, checking that each entry corresponds to a record in a file. By using LOCAL and three local variables, all risk of overwriting valuable data in variables of the same names in the main body of the program is removed.

LOCAL FUNCTION NEXT.ID(FILENAME)
    PRIVATE DICT.F, ID
    OPEN 'DICT', FILENAME TO DICT.F THEN
        READU ID FROM DICT.F, 'NEXT.ID' THEN
            WRITE ID+1 TO DICT.F, 'NEXT.ID'
        RETURN ID
    END
END
RETURN ''
END

The above local function takes a file name as its argument and gets the next sequential record id from a record stored in the corresponding dictionary, returning this as the value of the function. The dictionary will automatically be closed when the DICT.F variable is discarded on return from the function.
LOCATE

The LOCATE statement searches a dynamic array for a given field, value or subvalue. The LOCATE() function provides similar capabilities and is particularly suited to use in dictionary I-type items.

Format

This statement has three different syntaxes / semantics for compatibility with other systems.

Default style

LOCATE string IN dyn.array<field {,value {, subvalue}}> {BY order} {SETTING var} {THEN statement(s)} {ELSE statement(s)}

UniVerse style (enabled by use of $MODE UV.LOCATE)

LOCATE string IN dyn.array<field {,value }>, { start} {BY order} SETTING var {THEN statement(s)} {ELSE statement(s)}

Pick style

LOCATE(string, dyn.array{, field {, start }}); var {; order}) {THEN statement(s)} {ELSE statement(s)}

Function style (returns position as function value)

LOCATE(string, dyn.array, field {,value {, subvalue } } {; order})

where

- string evaluates to the item to be located.
- dyn.array is the dynamic array in which searching is to occur.
- field is the field at which searching is to commence.
- value is the value at which searching is to commence. If omitted or zero, searching occurs at the field level.
- subvalue is the subvalue at which searching is to commence. If omitted or zero, searching occurs at the value level.
- order evaluates to the ordering string as described below. If omitted, no ordering is assumed.
- var is the variable to receive the position value.
- statement(s) are statements to be executed depending on the outcome of the LOCATE action.

At least one of the THEN and ELSE clauses must be present.

The LOCATE statement searches for fields, values or subvalues within a dynamic array. The semantics of this operation depend on the style selected as described below.
Default Style

The default style of **LOCATE** is as found in products such as Prime Information, PI/open and with certain mode settings in UniVerse and Unidata.

If only **field** is specified, **LOCATE** searches for a field that matches **string**. If **field** and **value** are specified but **subvalue** is omitted, **LOCATE** searches for a value within the specified field that matches **string**. If **field**, **value** and **subvalue** are specified, **LOCATE** searches for a subvalue within the specified field and value that matches **string**.

Searching commences at the starting position defined in the **IN** clause. If a match is found, **var** is set to its field, value or subvalue position as appropriate to the level of the search. If no match is found, **var** is set to the position at which a new item should be inserted. For an unordered **LOCATE** this will be such that it would be appended.

Note that the syntax actually reads incorrectly. A **LOCATE** statement such as

```
LOCATE DT IN DATES<1> SETTING POS THEN ...
```

is not searching in **DATES<1>** at all. It is searching starting at **DATES<1>**.

UniVerse Style

Pick and Reality systems and the UniVerse database running in Ideal, Pick, Reality or IN2 flavour uses the **IN** clause to identify the item to be searched, not the starting position. The starting position is assumed to be the first item in the data but may specified explicitly in the command by including the **start** item.

This format of **LOCATE** can be selected by including a line

```
$MODE UV.LOCATE
```

in the program on a line preceding the **LOCATE** statement.

Pick Style

The original Pick database used a very different syntax which can be used in QM without any special mode settings. Note that despite the presence of brackets, this is a statement and should not be confused with the **LOCATE()** function described below.

In all three styles, the **THEN** clause is executed if the **string** is found in **dyn.array**. The **ELSE** clause is found if the **string** is not found.

In QMBasic, unlike other multivalue environments, the **SETTING** clause is optional. If omitted, the **THEN** or **ELSE** clause is executed as described above but no positional information is returned.

Note that this syntax, as found in Information style multivalue products, is actually illogical. The **IN** clause does not specify the data within which to search. Instead it specifies the start position for the search. The alternative syntax enabled by the UV.**LOCATE** compiler mode and described below is more logical. This syntax is found in Pick, Reality and some flavours of UniVerse.

The optional **BY** clause allows selection of an ordering rule specified as a series of case insensitive letters. These are:

- **A** or **D** Ascending or descending sort order. On non-ECS mode systems, the sort weight of each character is determined by its ASCII character value. On ECS
mode systems, the sort weights are taken from the currently active character map. If neither code is present, no sorting is used.

**L or R**  
Left or right aligned sort. These codes are ignored if no sort mode is specified. A left aligned sort compares characters from left to right and determines the sort order when the first difference is encountered. In a right aligned sort, if the item being examined is not of the same length as the string being located, leading spaces are added to the shorter item prior to a left to right comparison. Unless the STRING.LOCATE setting of the $MODE compiler directive is enabled, integer numeric data that can be represented as a 64 bit value, including a negative value, is treated as a special case in a right aligned sort and is sorted into correct numeric sequence. In the Pick style syntax, fractional values are also allowed in numeric sorting.

**C or N**  
Overrides default case sensitivity of the program. C forces case sensitivity and N forces non-case sensitivity. If neither code is present, case sensitivity is applied unless the $MODE NOCASE.STRINGS compiler directive has been used to make string comparisons case insensitive.

Left aligned ordering is faster than right aligned and should be used for textual data. Right aligned ordering is useful for numeric data such as internal format dates where the left aligned ordering would lead to sequencing problems (for example, 17 May 1995 is day 9999, 18 May 1995 is day 10000. Use of a left aligned ordering would place these dates out of calendar order).

The LOCATE() function works like the Information style described above but returns the position at which the item was found as its result value, zero if it was not found. Although the order argument can be used to specify the expected ordering and has the impact described above for numeric data, this function does not provide a way to identify where an item should be inserted if it is not found. The LOCATE() function is particularly suited to use in dictionary I-type items.

The result of a LOCATE statement or LOCATE() function with a specific ordering when applied to a dynamic array which does not conform to that ordering is undefined and likely to lead to misbehaviour of the program at run time.

**Examples**

**Default style:**
```
LOCATE PART.NO IN PARTS<1> BY "AL" SETTING I ELSE
  INS PART.NO BEFORE PARTS<I>
END
```

This program fragment locates PART.NO in a sorted list PARTS and, if it is not already present, inserts it.

**UniVerse style:**
```
LOCATE PART.NO IN PARTS BY "AL" SETTING I ELSE
  INS PART.NO BEFORE PARTS<1>
END
```

This is the same as the first example but reworked to use UniVerse style.

**Function style:**
```
I = LOCATE(PART.NO, PARTS, 1; "AL")
```
This statement performs the same search as the previous example but can only be used to find the position of an existing item, not to determine where a new item should be inserted to maintain the correct sort order.

See also:
DEL, DELETE(), EXTRACT(), FIND, FINDSTR, INS, INSERT(), LISTINDEX(), REPLACE()
LOCK

The **LOCK** statement obtains one of 64 system wide task locks.

**Format**

```
LOCK lock.num {THEN statement(s)} {ELSE statement(s)}
```

where

- `lock.num` evaluates to the lock number in the range 0 to 63.
- `statement(s)` are statements to be performed depending on the outcome of the **LOCK** operation.

The **THEN** and **ELSE** clauses are both optional. Neither is required.

Task locks provide a means of synchronising the activities of multiple processes without using locks on records in data files. The **LOCK** statement attempts to acquire the lock identified by `lock.num`.

The **THEN** clause is executed if the lock was available or was already held by this process.

The **ELSE** clause is executed if the lock is held by another process. If no **ELSE** clause is present, the **LOCK** statement waits for the lock to become available. This wait can be interrupted by using the break key.

The **STATUS()** function returns zero if the lock was free when the **LOCK** statement was executed. Otherwise it returns the user number of the process that held the lock. This will be the user number of the current process if it already owned the lock.

There is no means for a program to determine which task locks are held by the user except by using **TESTLOCK()** against each lock in turn. Beware that unlike read, update and file locks, task locks are only automatically released on leaving QM, not on return to the command prompt.

**Examples**

```
LOCK 7 THEN
    ...processing statements...
UNLOCK 7
END
ELSE ABORT "Cannot obtain task lock"
```

This program fragment obtains task lock 7, performs some critical processing and then releases the lock. The program aborts if the lock is not available.

```
LOCK 7
```

This statement attempts to obtain task lock 7 but, unlike the previous example, waits for the lock to become free if it is owned by another user.

**See also:**

- CLEAR.LOCKS
- **LOCK (Command)**
- LIST.LOCKS
- TESTLOCK()
- UNLOCK
LOGMSG

The **LOGMSG** statement adds a line to the system error log.

**Format**

```
LOGMSG text
```

where

```
text       is the message to be logged.
```

QM includes the option to maintain a log of system error messages in a file named errlog in the QMSYS account. The **LOGMSG** statement can be used by application software to write messages into this file. If the error log is disabled by setting the **ERRLOG** configuration parameter to zero, the **LOGMSG** statement will be ignored.

Although programs can write to this file using the sequential file handling statements, the internal buffering mechanism used by these statements is likely to result in loss of messages. Programs should, therefore, use on the **LOGMSG** statement to write messages.

**Example**

```
READ ORDER.NO FROM @VOC, 'NEXT.ORDER' ELSE
    LOGMSG 'NEXT.ORDER record not found'
RETURN
END
```

The above program fragment logs a message in the system error log if the NEXT.ORDER record cannot be found in the VOC

**See also:**

**LOGMSG** command
LOOP / REPEAT

The **LOOP** statement introduces a sequence of statements to be executed repeatedly.

**Format**

```
LOOP
  {statement(s)}
  {WHILE expr}
  {UNTIL expr}
  {statement(s)}
REPEAT
```

where

- **statement(s)** are statements to be executed within the loop.
- **expr** is an expression which can be resolved to a numeric value

There may be any number of **WHILE** or **UNTIL** statements within the loop appearing at any position relative to other statements.

Execution of the statements within the loop continues repeatedly until either the expression associated with a **WHILE** statement evaluates to zero or the expression associated with an **UNTIL** statement evaluates to a non-zero value.

The loop may also be terminated by an **EXIT** statement as detailed in its own description.

The **CONTINUE** statement may be used to commence the next iteration of the loop without execution of any intervening statements.

**Example**

```
LOOP
  REMOVE ITEM FROM ITEM.LIST SETTING MORE
  DISPLAY "Item id " : ITEM
  WHILE MORE
    DISPLAY "Item id " : ITEM
  REPEAT
```

This program fragment displays the elements of the dynamic array ITEM.LIST.

**See also:**

**CONTINUE, EXIT, FOR/NEXT, WHILE, UNTIL**
LOWER()

The LOWER() function converts mark characters in a string to the next lower level mark.

Format

    LOWER(string)

where

    string      evaluates to the string in which mark characters are to be converted.

The LOWER() function replaces mark characters according to the following table:

<table>
<thead>
<tr>
<th>Original</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item mark</td>
<td>Field mark</td>
</tr>
<tr>
<td>Field mark</td>
<td>Value mark</td>
</tr>
<tr>
<td>Value mark</td>
<td>Subvalue mark</td>
</tr>
<tr>
<td>Subvalue mark</td>
<td>Text mark</td>
</tr>
<tr>
<td>Text mark</td>
<td>Text mark (unchanged)</td>
</tr>
</tbody>
</table>

Migration Note

Note that the behaviour in some other multivalue products may not be identical with regard to the effect of lowering a text mark.

Example

    READLIST LIST THEN REC<4> = LOWER(LIST)

This statement reads active select list zero to LIST and then saves it in field 4 of REC. Because a select list contains field marks, the LOWER() function is used to demote each mark character to the next lowest mark.

See also:

RAISE()
LTS()

The LTS() function processes two dynamic arrays, returning a similarly structured result array indicating whether elements of the first array are less than corresponding elements of the second array.

Format

\[
\text{LTS}(expr1, expr2)
\]

where

\(expr1\) and \(expr2\) are the dynamic arrays to be compared.

The LTS() function compares corresponding elements of the dynamic arrays \(expr1\) and \(expr2\), returning a similarly structured dynamic array of True / False values indicating the results of the comparison.

The REUSE() function can be applied to either or both expressions. Without this function, any absent trailing values are taken as zero.

Example

A contains 11FM0VM14VMABCFM2
B contains 12FM0VM14VMACBFM2

\[
C = \text{LTS}(A, B)
\]

C now contains 1FM0VM0VM0FM0

See also:

ANDS(), EQS(), GES(), GTS(), IFS(), LES(), NES(), NOTS(), ORS(), REUSE()
**MARK.MAPPING**

The **MARK.MAPPING** statement determines how field marks are handled when reading or writing from a directory file.

**Format**

- **MARK.MAPPING** `file.var, OFF`
- **MARK.MAPPING** `file.var, ON`
- **MARK.MAPPING** `file.var, expr`

where

- `file.var` is the file variable for a previously opened file.
- `expr` evaluates to a number.

Data written to directory files usually has field marks translated to newlines. On reading, the reverse translation is performed to recover the original data. Records storing binary information may contain bytes that appear to be field marks and these will be translated, possibly causing data corruption.

Use of **MARK.MAPPING** `file.var, OFF` will suppress this mark mapping process until a subsequent **MARK.MAPPING** `file.var, ON` statement or the file is closed.

The **MARK.MAPPING** `file.var, expr` format of this statement is equivalent to
- **MARK.MAPPING** `file.var, ON` if the value of `expr` is non-zero and
- **MARK.MAPPING** `file.var, OFF` if `expr` is zero.

**Example**

```
MARK.MAPPING FILE.VAR, OFF
READ REC FROM FILE.VAR, ID ELSE STOP 'NOT FOUND'
```

This program fragment reads a record with mark translation suppressed.
MAT

The MAT statement assigns a value to all elements of a matrix, copies one matrix to another, or tests for equivalent matrices.

Format

\[
\text{MAT} \quad \text{matrix} = \text{expr} \\
\text{MAT} \quad \text{matrix}2 = \text{MAT} \quad \text{matrix}1 \\
\text{IF} \quad \text{MAT} \quad \text{matrix}1 = \text{MAT} \quad \text{matrix}2 \quad \text{THEN} \ldots
\]

where

\[
\text{matrix} \quad \text{is the name of a previously dimensioned matrix.} \\
\text{expr} \quad \text{evaluates to the value to be stored in each matrix element.}
\]

The first format of this statement copies the value of \text{expr} into all elements of \text{matrix}. The zero element is set to a null string.

The second format copies elements from \text{matrix}1 to \text{matrix}2 row by row. If the number of columns differs, the copy behaves as depicted below.

<table>
<thead>
<tr>
<th>Source:</th>
<th>Target:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2 3</td>
<td>1 2 3</td>
</tr>
<tr>
<td>1 A B</td>
<td>1 A B C</td>
</tr>
<tr>
<td>2 C D</td>
<td>2 D E F</td>
</tr>
<tr>
<td>3 E F</td>
<td></td>
</tr>
</tbody>
</table>

The zero element of \text{src.matrix} is copied to the zero element of \text{matrix}.

If \text{src.matrix} has more elements than \text{matrix}, the excess elements are ignored. If \text{src.matrix} has fewer elements than \text{matrix}, the remaining elements of \text{matrix} are unchanged.

A single dimensional matrix can be copied to a two dimensional matrix and vice versa.

The third syntax tests whether the content of \text{matrix}1 is the same as the content of \text{matrix}2.

Although most likely to be used as part of an IF statement as shown above, the

\[
\text{MAT} \quad \text{matrix}1 = \text{MAT} \quad \text{matrix}2
\]

component of this statement may be used anywhere that a Boolean value is appropriate.

Examples

\[
\text{DIM} \quad A(25) \\
\text{MAT} \quad A = 0
\]

The above program fragment dimensions matrix A to have 25 elements and sets them all to zero.
DIM A(5,5), B(25)
... statements that set values in matrix A...
MAT B = MAT A

This program fragment dimensions two matrices, sets values into matrix A and then creates a single dimensional copy of A in matrix B.
MAT()

The **MAT()** function creates an array for insertion into a data collection.

**Format**

```
MAT()
MAT(array {, elements})
```

where

- `array` is a dimensioned array.
- `elements` is the maximum number of elements to copy. If omitted or zero, all elements are copied.

The first form of the **MAT()** function creates an empty array. The second form copies the content of an array variable previously dimensioned using the **DIM** statement. In both forms, the only valid use of the value returned by this function is to store it in a data collection.

Arrays in data collections are single dimensional. If the `array` argument is a two-dimensional array, elements are copied row by row to form a single-dimensional result.

**Example**

```
DIM EMAIL.ADDRESSES(10)
...statements that set values in the above array...
CLIENT('email') = MAT(EMAIL.ADDRESSES)
```

The above program fragment copies the values in the EMAIL.ADDRESSES dimensioned array to become an array element named email within the CLIENT data collection.

**See also:**

Data collections
MATBUILD

The MATBUILD statement constructs a dynamic array from the elements of a matrix.

Format

`MATBUILD var FROM mat [, start.expr [, end.expr] {USING delimiter}]`

where

- `var` is the variable to receive the dynamic array.
- `mat` is the matrix from which data is to be taken.
- `start.expr` evaluates to the index of the first matrix element to be used. If omitted or less than one, this defaults to one.
- `end.expr` evaluates to the index of the last matrix element to be used. If omitted or less than one, this defaults to the number of elements in the matrix.
- `delimiter` evaluates to the delimiter to be used between elements of `mat`. This may be more than one character. If omitted, this defaults to the field mark.

The MATBUILD statement behaves differently depending on the matrix style.

For a default style matrix, MATBUILD constructs a dynamic array by concatenating elements of `mat` in the range `start.expr` to `end.expr`, inserting `delimiter` between each element. If the zero element is null or unassigned, trailing null elements are ignored. If the zero element is not null, trailing null matrix elements are included and followed by a `delimiter` and the content of the zero element.

For a Pick style matrix, MATBUILD constructs a dynamic array by concatenating elements of `mat` in the range `start.expr` to `end.expr`, inserting `delimiter` between each element. Trailing null or unassigned elements are ignored. Pick style matrices do not have a zero element. See the COMMON and DIMENSION statements for more details.

In either style, use of the INMAT() function immediately after MATBUILD will return the index value of the final element included. If the zero element of a default style matrix was non-null, INMAT() will return zero.

Example

`MATBUILD REC FROM A USING @VM`

This statement constructs dynamic array REC from the elements of matrix A, separating each element by a value mark.

See also:
MATPARSE
MATCHESS()

The MATCHESS() function compares each element of a dynamic array with a pattern template, returning an equivalently structured dynamic array of True/False values. Note the spelling of this function name with the trailing S to "pluralise" the name in common with other multivalue function names.

Format

```
MATCHESS(dyn.arr, pattern)
```

where

- `dyn.arr` is the dynamic array to be processed.
- `pattern` evaluates to a template as described below.

The MATCHESS() function matches each element of `dyn.arr` against `pattern` and returns a dynamic array of True/False values indicating the result of each comparison.

The `pattern` string consists of one or more concatenated items from the following list.

- `...` Zero or more characters of any type
- `0X` Zero or more characters of any type
- `nX` Exactly `n` characters of any type
- `n-mX` Between `n` and `m` characters of any type
- `0A` Zero or more alphabetic characters
- `nA` Exactly `n` alphabetic characters
- `n-mA` Between `n` and `m` alphabetic characters
- `0N` Zero or more numeric characters
- `nN` Exactly `n` numeric characters
- `n-mN` Between `n` and `m` numeric characters
- "string" A literal string which must match exactly. Either single or double quotation marks may be used.

The values `n` and `m` are integers with any number of digits. `m` must be greater than or equal to `n`.

The `0A`, `nA`, `0N`, `nN` and "string" patterns may be preceded by a tilde (~) to invert the match condition. For example, ~4N matches four non-numeric characters such as ABCD (not a string which is not four numeric characters such as 12C4).

A null string matches patterns ..., `0A`, `0X`, `0N`, their inverses (~`0A`, etc) and "".

The `0X` and `n-mX` patterns match against as few characters as necessary before control passes to the next pattern. For example, the string ABC123DEF matched against the pattern 0X2N0X matches the pattern components as ABC, 12 and 3DEF.
The 0N, n-mN, 0A, and n-mA patterns match against as many characters as possible. For example, the string ABC123DEF matched against the pattern 0X2-3N0X matches the pattern components as ABC, 123 and DEF.

The *pattern* string may contain alternative templates separated by value marks. The **MATCHESS()** function tries each template in turn until one is a successful match against *string*. If a match is found, the **INMAT()** function can be used to retrieve the value position within the pattern that matched.

**Example**

```
VAR = "123":@VM:"ABC":@FM:"456"
X = MATCHESS(VAR, "3N")
```

The variable X will be returned as

```
1
```

**See also:**

- Pattern Matching
MATCHFIELD(), MATCHFIELDS()

The MATCHFIELD() function extracts a portion of a string that matches a pattern element. The MATCHFIELDS() function performs the same operation on each element of a dynamic array.

Format

MATCHFIELD(string, pattern, element)

where

string evaluates to the string in which the pattern is to be located.

pattern evaluates to a template as described below.

element evaluates to an integer indicating which pattern element of string is to be returned.

The MATCHFIELD() function matches string against pattern and returns the portion of string that matches the element'th component of pattern.

The pattern string consists of one or more concatenated items from the following list.

... Zero or more characters of any type

0X Zero or more characters of any type

nX Exactly n characters of any type

n-mX Between n and m characters of any type

0A Zero or more alphabetic characters

nA Exactly n alphabetic characters

n-mA Between n and m alphabetic characters

0N Zero or more numeric characters

nN Exactly n numeric characters

n-mN Between n and m numeric characters

"string" A literal string which must match exactly. Either single or double quotation marks may be used. Unlike the MATCHES operator, string must be enclosed in quotes otherwise each character is treated as a separate component.

The values n and m are integers with any number of digits. m must be greater than or equal to n.

The 0A, nA, 0N, nN and "string" patterns may be preceded by a tilde (~) to invert the match condition. For example, ~4N matches four non-numeric characters such as ABCD (not a string which is not four numeric characters such as 12C4).

A null string matches patterns ..., 0A, 0X, 0N, their inverses (~0A, etc) and "."

The 0X and n-mX patterns match against as few characters as necessary before control passes to the next pattern. For example, the string ABC123DEF matched against the pattern 0X2N0X matches the pattern components as ABC, 12 and 3DEF.
The 0N, $n$-$m$N, 0A, and $n$-$mA$ patterns match against as many characters as possible. For example, the string ABC123DEF matched against the pattern 0X2-3N0X matches the pattern components as ABC, 123 and DEF.

The pattern string may contain alternative templates separated by value marks. The MATCHFIELD() function tries each template in turn until one is a successful match against string. If a match is found, the INMAT() function can be used to retrieve the value position within the pattern that matched.

The element argument determines which component of string is returned as the MATCHFIELD() function result. For example,

MATCHFIELD("ABC123DEF", "0X2N0X", 2)

returns the string "12".

The MATCHFIELD() function returns a null string if no component of pattern matches string.

The MATCHFIELDS() function performs the same operation on each element of a dynamic array, returning a dynamic array of the same structure in which each element contains the data extracted from the corresponding element of the input string array.

Examples

TEL.NO = "01604-709200"
LOCAL.NO = MATCHFIELD(TEL.NO, "0N'-'0N", 3)

This program fragment extracts the local part of a telephone number (709200 in this case). Note that the literal element counts as a component of the string when identifying each element.

If the delimiter is multiple characters but not quoted, each character is counted as a separate element:

MATCHFIELD("123--456", "0N'--'0N", 1) returns 123
MATCHFIELD("123--456", "0N'--'0N", 2) returns --
MATCHFIELD("123--456", "0N'--'0N", 3) returns 456
MATCHFIELD("123--456", "0N--0N", 1) returns 123
MATCHFIELD("123--456", "0N--0N", 2) returns --
MATCHFIELD("123--456", "0N--0N", 3) returns --
MATCHFIELD("123--456", "0N--0N", 4) returns 456

S1 = 'AB12CD' : @VM : 'EF34GH'
S2 = MATCHFIELDS(S1, '0A0N0A', 2)

The above program fragment matches each element of dynamic array S1 against the '0A0N0A' pattern and returns S2 as

12vm34

See also:
Pattern Matching, \texttt{PARSE()}
MATPARSE

The **MATPARSE** statement breaks a delimited string into component substrings, assigning each to an element of a matrix.

**Format**

```
MATPARSE mat FROM string [, delimiter] {SETTING var}
MATPARSE mat FROM string {USING delimiter} {SETTING var}
```

where

- `mat` is the matrix into which the substrings are to be assigned. This matrix must already have been dimensioned.
- `string` is the string to be parsed.
- `delimiter` evaluates to the delimiter that separates substrings within `string`. If omitted, a field mark is used.
- `var` is a variable to receive the count of items.

The **MATPARSE** statement operates in one of three ways depending on the length of the `delimiter` string.

If `delimiter` is a null string, each character of `string` is assigned to a separate element of `mat`. If both `string` and `delimiter` are null, no elements are assigned.

If `delimiter` is a one character string, each substring of `string` delimited by `delimiter` is assigned to a separate element of `mat`. The delimiter character is not stored in `mat`.

If `delimiter` is more than one character long, each substring of `string` delimited by any character of `delimiter` is assigned to a separate element of `mat`. The next element is assigned the character that delimited the items. Where two or more occurrences of the same delimiter character occur within `string` with no other intervening characters, only a single element of `mat` is used to receive the multiple delimiters.

If `mat` is two dimensional, its elements are assigned row by row.

The optional **SETTING** clause sets `var` to the number of elements assigned. Alternatively, the **INMAT()** function after a **MATPARSE** returns the same value. Unused elements are set to null strings.

With the default style of matrix, where there are insufficient elements in `mat` to receive the parsed `string`, the remaining unparsed data is stored in the zero element of `mat`. In this case, the **INMAT()** function will return zero.

Pick style matrices do not have a zero element. Any excess data is stored in the final element of the matrix and the **INMAT()** function returns zero to indicate this condition. See the **COMMON** and **DIMENSION** statements for more details.

**Example**

```
DIM A(30)
```
MATPARSE A FROM S, @FM
IF INMAT() THEN DISPLAY INMAT() : 'elements used'
ELSE DISPLAY 'Data overflows available space'

This program fragment assigns successive fields of string S to elements of matrix A and displays the actual number of elements used. If there are more than 30 fields, the remaining fields and delimiting mark characters are stored in A(0).

See also:
MATBUILD
MATREAD, MATREADL, MATREADU

The **MATREAD** statement reads a record from a file, assigning each field to an element of a matrix.

The **MATREADL** statement is similar to **MATREAD** but sets a read lock on the record. The **MATREADU** statement sets an update lock on the record.

**Format**

```
MATREAD mat {ENCODING name} FROM file.var, record.id {ON ERROR statement(s)} {LOCKED statement(s)} {THEN statement(s)} {ELSE statement(s)}
```

where

- **mat** is the matrix into which fields are to be assigned. This matrix must already have been dimensioned.
- **file.var** is the file variable associated with the file. This may not be a data collection file.
- **record.id** evaluates to the key of the record to be read.
- **statement(s)** are statements to be executed depending on the outcome of the operation.

The **LOCKED** clause is not valid with the **MATREAD** statement. At least one of the **THEN** and **ELSE** clauses must be present unless the **OPTIONAL.THEN.ELSE** setting of the **$MODE** compiler directive is enabled.

The **MATREAD** statement is equivalent to a **READ** followed by a **MATPARSE**.

```
READ REC FROM file.var, record.id
ON ERROR statement(s)
LOCKED statement(s)
THEN MATPARSE mat FROM REC, @FM
ELSE statement(s)
```

The optional **ENCODING** element to this statement sets the character encoding to be used, overriding any encoding set in the **VOC F-type** record or when the file was opened. This is relevant only to directory files and is ignored for other file types.

Each field of the record is assigned to a separate element of **mat**. If **mat** is two dimensional, its elements are assigned row by row. The **INMAT()** function will return the number of elements assigned. Unused elements are set to null strings.

With the default style of matrix, where there are fewer elements in **mat** than the number of fields in the record, the remaining data is stored in the zero element of **mat**. In this case, the **INMAT()** function will return zero.
Pick style matrices do not have a zero element. Any excess data is stored in the final element of the matrix and the \texttt{INMAT()} function returns zero to indicate this condition. See the \texttt{COMMON} and \texttt{DIMENSION} statements for more details.

**Example**

```qbasic
DIM ITEMS(30)
MATREAD ITEMS FROM ITEM.FILE, "ITEM.LIST" ELSE
    ABORT "ITEM.LIST record not found"
END
IF INMAT() THEN DISPLAY INMAT() : " items read"
ELSE ABORT "Too many items"
```

This program fragment reads a record from a file, assigning fields to elements of matrix ITEMS. If there are more than 30 fields, the program aborts, otherwise it displays the number of items read.
MATREADCSV

The MATREADCSV statement reads a CSV format line of text from a directory file record previously opened for sequential access and parses it into the elements of a dimensioned matrix.

Format

```
MATREADCSV matrix {ENCODING name} FROM file.var {DELIMITER delim}
{THEN statement(s)}
{ELSE statement(s)}
```

where

- matrix is the dimensioned matrix to receive the data read from the file.
- encoding is the ECS character encoding to be used for the data.
- file.var is the file variable associated with the record by a previous OPENSEQ statement.
- delim is the delimiter separating data elements. This defaults to a comma if omitted. Only the first character of delim is significant.
- statement(s) are statement(s) to be executed depending on the outcome of the READSEQ.

At least one of the THEN and ELSE clauses must be present.

The optional ENCODING element to this statement sets the character encoding to be used, overriding any encoding set in the VOC F-type record or when the file was opened.

A line of text is read from the file. It is then parsed according to the CSV format rules, placing the elements into successive elements of the matrix. If successful, the THEN clause is executed and the STATUS() function would return zero.

If there are fewer data items in the line of text than the number of variables supplied, the remaining elements of the matrix will be set to null strings. If the line of text has more data items than the number of elements in the matrix, the excess data is placed in the zero element as for MATPARSE. The INMAT() function can be used to determine the number of data items in exactly the same way as for MATREAD.

If there are no further fields to be read, the ELSE clause is executed and the STATUS() function would return ER$RNF (record not found). The target matrix will be unchanged.

The CSV rules are described under the WRITECSV statement.

Example

```
DIM DETAILS(10)
LOOP
   MATREADCSV DETAILS FROM DELIVERY.F ELSE EXIT
   GOSUB PROCESS.DELIVERY.DETAILS
ENDLOOP
```
This program fragment reads CSV format lines of text from the record open for sequential access via the DELIVERY.F file variable into elements of the DETAILS matrix. It then calls the PROCESS.DELIVERY.DETAILS subroutine to process the new item. The loop terminates when the ELSE clause is executed when all fields have been processed.

See also:
CLOSESEQ, NOBUF, OPENSEQ, READBLK, READSEQ, SEEK, WEOFSEQ, WRITEBLK, WRITECSV, WRITESEQ, WRITESEQF
**MATSTR()**

The MATSTR() function builds a dynamic array from a matrix.

**Format**

```
MATSTR(array)
```

where

```
array is a dimensioned array.
```

The MATSTR() is intended for use with a two dimensional matrix and builds a dynamic array in which each row is represented by a field and the columns are represented by values in the field. Used with a single dimensional matrix, MATSTR() is closely related to the MATBUILD statement and returns a dynamic array with a field for each element of the matrix.

For arrays that include the zero element, this is ignored by MATSTR(). Also note that field or value mark characters appearing in the matrix data are not handled in any special way and are therefore likely to lead to ambiguous results.

**Example**

```
DIM DATA(100,4)
...statements that set values in the above array...
STRING = MATSTR(DATA)
```

The above program fragment sets STRING to be a dynamic array with 100 fields, each divided into 4 values corresponding to the rows and columns of the DATA matrix.

**See also:**

MATBUILD
MATWRITE

The MATWRITE statement builds a record from successive elements of a matrix and writes this to a file.

The MATWRITEU statement is similar but preserves any lock on the record being written.

Format

\[
\text{MATWRITE } \text{mat} \{\text{ENCODING name}\} \text{ TO } \text{file.var}, \text{record.id} \{\text{ON ERROR } \text{statement(s)}\}
\]

where

- \text{mat} is the matrix from which data is to be taken.
- \text{file.var} is the file variable associated with the file. This may not be a data collection file.
- \text{record.id} evaluates to the key of the record to be written.
- \text{statement(s)} are statements to be executed depending on the outcome of the operation.

The MATWRITE statement constructs a dynamic array from the elements of \text{mat}, writing this dynamic array to the file opened as \text{file.var} with the given \text{record.id}.

A MATWRITE statement is equivalent to a \text{MATBUILD} followed by a \text{WRITE}.

\[
\text{MATBUILD REC FROM } \text{mat} \\
\text{WRITE REC TO } \text{file.var}, \text{record.id} \text{ ON ERROR } \text{statement(s)}
\]

The way in which the dynamic array is constructed differs depending on the matrix style.

For a default style matrix, MATWRITE constructs the dynamic array by concatenating elements of \text{mat}, inserting a field mark between each element. If the zero element is null or unassigned, trailing null elements are ignored. If the zero element is not null, trailing null matrix elements are included and followed by a field mark and the content of the zero element.

For a Pick style matrix, MATWRITE constructs the dynamic array by concatenating elements of \text{mat}, inserting a field mark between each element. Trailing null elements are ignored. Pick style matrices do not have a zero element. See the COMMON and DIMENSION statements for more details.

In either style, use of the \text{INMAT()} function immediately after MATWRITE will return the index value of the final element included. If the zero element of a default style matrix was non-null, \text{INMAT()} will return zero.

The optional \text{ENCODING} element to this statement sets the character encoding to be used, overriding any encoding set in the VOC F-type record or when the file was opened. This is relevant only to directory files and is ignored for other file types.

If the \text{ON ERROR} clause is taken, the \text{STATUS()} function can be used to determine the cause of the error. Otherwise, for the MATWRITE statement, the \text{STATUS()} function returns 0 if the record was locked by this process prior to the MATWRITE or ER$NLK if it was not locked. The \text{STATUS()} function value is undefined after a successful MATWRITEU statement.
Use of a THEN/ELSE Clause

For compatibility with the way in which triggers operate in some other multivalue products, the MATWRITE statement has an optional THEN/ELSE clause. Because this would otherwise lead to a syntactic ambiguity, compilation of programs that use this clause requires the WRITE.DELETE.THEN.ELSE option of the $MODE compiler directive to be enabled. Once this is done, the optional THEN/ELSE clauses can be included in their usual position, after the ON ERROR clause.

When a MATWRITE statement has a THEN/ELSE clause, a failure returned from a trigger function, typically as a result of a pre-write data validation error, will cause the ELSE clause to be executed instead of the ON ERROR clause.

Example

```plaintext
MATWRITE ITEMS TO ITEM.FILE, "ITEM.LIST" ON ERROR
    ABORT "Error " : STATUS() : " writing item list"
END
IF STATUS() = 0 THEN DISPLAY "Data saved"
```

This program fragment writes a record built from elements of matrix ITEMS. If the MATWRITE is successful, a message is displayed.
MATWRITECSV

The **MATWRITECSV** statement builds a CSV form line of text from successive elements of a matrix and writes this to a file opened for sequential processing.

**Format**

```
MATWRITECSV mat [ENCODING name] TO file.var {ON ERROR statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

where

- **mat** is the matrix from which data is to be taken.
- **name** evaluates to the name of a character encoding.
- **file.var** is the file variable associated with the file previously opened with **OPENSEQ**.
- **statement(s)** are statements to be executed depending on the outcome of the operation.

The keyword **TO** may be replaced by **ON**. At least one of the **THEN** and **ELSE** clauses must be present.

The optional **ENCODING** clause sets the character encoding to be used, overriding any encoding set in the **VOC F-type** record or when the file was opened. A null string as the encoding name is equivalent to not having the **ENCODING** clause at all. To disable the default encoding, the encoding name should be specified as "NULL".

The data in **mat** is transformed to comma separated variable form and written to the file opened to **file.var**. The **THEN** clause is executed if the write is successful.

The **ELSE** clause is executed if the **MATWRITESEQ** operation fails. On ECS mode systems, **STATUS()** value ER$ECS.DATA will be returned if data containing ECS characters is written to a file with no encoding. The write will have completed but only the low order byte of each character will have been written.

If a fatal error occurs, the **ON ERROR** clause is executed. The **STATUS()** function can be used to establish the cause of the error. If no **ON ERROR** clause is present, a fatal error causes an abort.

The **FILEINFO()** function can be used with key FL$LINE to determine the field number that will be written by the next **WRITESEQ**. This information is not valid if the **SEEK**, **READBLK** or **WRITEBLK** statements have been used.

If the record was opened with the **SHARED** option of **OPENSEQ**, the data will be appended to any existing record content.

**Example**

```
OPENSEQ 'LOG', DATE() APPEND TO LOG.F ELSE
   IF STATUS() THEN STOP "Cannot open log file"
```
This program fragment opens a sequential file for appending and writes a line of text containing data from matrix LOGDATA in CSV format.
MAX()

The **MAX()** function returns the greater of two values.

**Format**

\[ \text{MAX}(a, b) \]

where

\[ a, b \] are the two values to be compared.

The **MAX()** function compares the values \( a \) and \( b \), returning the greater value. If either value cannot be treated as a number, a character by character comparison from the left end is performed until a difference is found, returning the "alphabetically last" item.

**Example**

\[
\begin{align*}
A &= 6 \\
B &= 8 \\
C &= \text{MAX}(A, B)
\end{align*}
\]

C will be set to the greater of the values in A and B, in this case 8.

**See also:**

**MIN()**
MAXIMUM()

The MAXIMUM() function returns the greatest value in a dynamic array.

Format

MAXIMUM(dyn.array)

where

dyn.array is the dynamic array to be scanned.

The MAXIMUM() function returns the greatest numeric value in dyn.array. Non-numeric and null elements of dyn.array are ignored. If the entire dynamic array is null, a null string is returned.

Example

S = '61':@VM:'42':@FM:'71':@VM:'57'
CRT MAXIMUM(S)

This program fragment searches S for the largest numeric value and displays the result (71).
MD5()

The MD5() function returns the 32 digit hexadecimal message digest value for a given string.

Format

    MD5(string)

where

    string is the string to be encoded.

The MD5() function provides a more comprehensive validation process than use of the default mode of the CHECKSUM() function. The returned value is a 32 digit hexadecimal string.

MD5 digest values are defined to work on byte level data. On an ECS mode system, it will be necessary to transform the data to a byte stream using the BS conversion code if it may contain characters outside the 8-bit set. Failure to do this will result in only the bottom 8 bits of each character in the source data being examined during encoding.

Example

DISPLAY MD5('The quick brown fox jumps over the lazy dog')

The above example would print '9e107d9d372bb6826bd81d3542a419d6'.

See also:
CHECKSUM(), DIGEST()
MERGELIST()

The MERGELIST() function combines two field mark delimited lists, returning the intersection, union or difference as a result list.

Format

MERGELIST(list1, list2, modes)

where

list1, list2 are the two lists to be combined. These may be strings or select list variables.

modes identifies the operation to be performed and the manner in which the lists are sorted.

The MERGELIST() function provides an efficient way to merge two field mark delimited lists. It is intended for use with lists of record ids such as select lists but can combine other lists where the elements are no more than 255 characters. Unless the MLSUNSORTED option is present, list1 and list2 must be in sorted order.

The modes argument must be one of the following token that are defined in the SYSCOM KEYS.H include record:

- ML$INTER Return a list of items in both list1 and list2.
- ML$UNION Return a list of items in either list1 and list2 but not duplicating items in both lists
- ML$DIFF Return a list of items in list1 but not in list2.

In addition, one of the following tokens may be added to the modes value to indicate how the data in list1 and list2 is sorted:

- SRT$LEFT A simple left aligned sort (default).
- SRT$RIGHT A simple right aligned sort, sorting integer numeric values as numbers.
- SRT$COMPOUND A compound sort.
- SRT$RIGHT.FLOAT A right aligned sort, sorting both integer and floating point values as numbers.

The additive MLSUNSORTED modes value indicates that the lists to be merged may not be in sorted order. In this case, the lists will be sorted in the manner specified by use of the SRT$xxx tokens prior to merging. If the MLSUNSORTED mode is not present and the supplied lists are not in the specified sort order, the effects of the MERGELIST() function are indeterminate.

For more information on the different sort modes, see the SORT() function.

The list to be sorted may be strings or select list variables. The result returned by the MERGELIST() function is always a field mark delimited dynamic array. It can be converted to a select list using FORMLIST().

Example

SELECTINDEXV 'CUSTNO', CUSTOMER FROM ORDERS.F TO CUST.ORDERS
SELECTINDEXV 'MONTH', ORDER.MONTH FROM ORDERS.F TO MONTHLY.ORDERS
JULY.ORDERS = MERGELIST(CUST.ORDERS, MONTHLY.ORDERS, ML$INTER)
The above example extracts data from two indices: one for all orders placed by a specific customer, the other for all orders placed in a specific month. It then uses `MERGELIST()` to combine these to give a list of all orders placed by that customer in the given month. Because `SELECTINDEXV` returns the index values in sorted order, no additional sorting is required.
MIN()

The MIN() function returns the lesser of two values.

**Format**

\[ \text{MIN}(a, b) \]

where

\[ a, b \]

are the two values to be compared.

The MIN() function compares the values \( a \) and \( b \), returning the lesser value. If either value cannot be treated as a number, a character by character comparison from the left end is performed until a difference is found, returning the "alphabetically first" item.

**Example**

\[
\begin{align*}
A &= 6 \\
B &= 8 \\
C &= \text{MIN}(A, B)
\end{align*}
\]

\( C \) will be set to the lesser of the values in \( A \) and \( B \), in this case 6.

**See also:**

MAX()
The `MINIMUM()` function returns the lowest value in a dynamic array.

**Format**

```
MINIMUM(dyn.array)
```

where

- `dyn.array` is the dynamic array to be scanned.

The `MINIMUM()` function returns the lowest numeric value in `dyn.array`. Non-numeric and null elements of `dyn.array` are ignored. If the entire dynamic array is null, a null string is returned.

**Example**

```
S = '61':@VM:'42':@FM:'71':@VM:'57'
CRT MINIMUM(S)
```

This program fragment searches `S` for the largest numeric value and displays the result (42).
MOD()

The MOD() function returns the modulus value of one value divided by another. The MODS() function is similar to MOD() but operates on successive elements of two dynamic arrays, returning a similarly structured dynamic array of results.

Format

\[
\text{MOD}(\text{dividend}, \text{divisor})
\]

where

- \text{dividend} evaluates to a number or a numeric array.
- \text{divisor} evaluates to a number or a numeric array.

The MOD() function returns the modulus value of dividing \text{dividend} by \text{divisor}. This is defined as

\[
\text{MOD}(x, y) = \text{IF } y = 0 \text{ THEN } x \text{ ELSE } x - (y * \text{FLOOR}(x / y))
\]

where the FLOOR() function returns the highest integer with value not greater than its argument. For example, FLOOR(-3.7) is -4. (FLOOR() is not part of QMBasic. It is used here only to explain the action of the MOD() function).

The MOD() function differs from the REM() function when one of its arguments is negative. The following table shows the result of the MOD() function.

<table>
<thead>
<tr>
<th>Dividend</th>
<th>Divisor</th>
<th>MOD()</th>
</tr>
</thead>
<tbody>
<tr>
<td>530</td>
<td>100</td>
<td>30</td>
</tr>
<tr>
<td>-530</td>
<td>100</td>
<td>70</td>
</tr>
<tr>
<td>530</td>
<td>-100</td>
<td>-70</td>
</tr>
<tr>
<td>-530</td>
<td>-100</td>
<td>-30</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>-100</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>-100</td>
<td>0</td>
<td>-100</td>
</tr>
</tbody>
</table>

The MODS() function operates on corresponding elements of two dynamic arrays, returning a similarly structured dynamic array of results. For arrays of differing structure, the structure of the result depends on whether the REUSE() function is used.

Example

\[
N = \text{MOD}(T, 30)
\]

This statement finds the modulus of dividing \(T\) by 30 and assigns this to \(N\).
See also:
REM(), ROUNDDOWN(), ROUNDUP()
The **MOUSE** statement enables or disables mouse input.

**Format**

```
MOUSE ON {SETTING var}
MOUSE OFF {SETTING var}
MOUSE expr {SETTING var}
```

where

- `expr` is an expression that evaluates to True or False.
- `var` is the variable to receive the previous mouse state.

Mouse input is initially off. Use of **MOUSE ON** enables mouse input until a subsequent **MOUSE OFF** statement.

The **MOUSE expr** format of this statement is equivalent to **MOUSE ON** if the value of `expr` is non-zero and **MOUSE OFF** if `expr` is zero or null.

The optional **SETTING** clause saves the previous mouse state in `var` which can be used later to revert to that state. Alternatively, the previous state can be obtained using the **STATUS()** function immediately after the **MOUSE** statement. In either case, the value is 1 if the mouse was enabled or 0 if it was disabled.

Mouse input is only available on console sessions, via QMTerm, and using AccuTerm with terminal types that have the -at suffix. When the mouse is enabled, clicking a mouse button sends a message that can be retrieved via the normal keyboard input statements and functions. This message is prefixed by char(200) on non-ECS systems, character echar(0x8F8C8) on ECS mode systems. The format of the qualifying data in the message varies between terminal types. Use of the **KEYCODE()** or **KEYCODEV()** function traps the mouse event prefix processes the qualifying data, leaving the button number in `@MBUTTON` and the screen position at which the mouse was clicked in `@MCOL` and `@MROW`.

The button state is the result of adding the values of each button that is depressed: 1 for the left button, 2 for the right button, 4 for the centre button. The x and y positions are based on character coordinates using the same conventions as for the cursor movement functions. Some terminal types only allow use of the left button.

**Example**

```
MOUSE ON
...
IF KEYCODEV() = KV$MOUSE THEN
   DISPLAY 'Mouse clicked at column ' : @MCOL : ', row ' : @MROW
END
```

This program fragment read mouse input, displaying the screen position.
**MVDATe()**

The **MVDATe()** function returns the multivalue style date value (days since 31 December 1967) for a supplied epoch value using the currently selected time zone.

Format

```
MVDATe(epoch.value)
```

where

```
epoch.value
```

is the epoch value to be converted.

An epoch value represents a moment in time. The **MVDATe()** function returns the date part of this as relevant to the currently selected time zone.

Example

```
T = 1234567890
SET.TIMEZONE 'GMT'
DISPLAY MVDATe(T)
SET.TIMEZONE 'Pacific/Wake'
DISPLAY MVDATe(T)
```

The above program fragment displays

```
15020
15021
```

The dates differ because this moment in time was 23:31:30 on 13 February 2009 in the GMT time zone but it was 11:31:30 on 14 February 2009 in the Pacific/Wake time zone.

See also:

*Date, Times and Epoch Values, EPOCH(), MVDATe.TIME(), MVEPOCH(), MVTIME(), SET.TIMEZONE*
MVDATE.TIME()

The MVDATE.TIME() function returns the multivalue style date and time values (days since 31 December 1967 and seconds since midnight) for a supplied epoch value using the currently selected time zone.

Format

```
MVDATE.TIME(epoch.value)
```

where

```
epoch.value
```

is the epoch value to be converted.

An epoch value represents a moment in time. The MVDATE.TIME() function returns the date and time parts of this as relevant to the currently selected time zone. The two parts are separated by an underscore.

Example

```
T = 1234567890
SET.TIMEZONE 'GMT'
DISPLAY MVDATE.TIME(T)
SET.TIMEZONE 'Pacific/Wake'
DISPLAY MVDATE.TIME(T)
```

The above program fragment displays

```
15020_84690
15021_41490
```

The dates differ because this moment in time was 23:31:30 on 13 February 2009 in the GMT time zone but it was 11:31:30 on 14 February 2009 in the Pacific/Wake time zone.

See also:

Date, Times and Epoch Values, EPOCH(), MVDATE(), MVEPOCH(), MVTIME(), SET.TIMEZONE
MVEPOCH()

The **MVEPOCH()** function returns the epoch value corresponding to a multivalue style date and time combination (days since 31 December 1967 and seconds since midnight) using the currently selected time zone.

**Format**

```
MVEPOCH(time.string)
MVEPOCH(date, time)
```

where

- `time.string` is the multivalue style date and time values separated by an underscore.
- `date` and `time` are separate date and time values.

An epoch value represents a moment in time. The **MVEPOCH()** function converts the supplied date and time values to the corresponding epoch value using the currently selected time zone.

If the `time.string` or `date` and `time` represents a moment in time that cannot be converted to an epoch value, the function returns a null string.

**Daylight Saving Time**

For most time values, the **MVEPOCH()** function will correctly handle daylight saving time. There is an one hour period when the clocks are moved backwards where the same external time value will correspond to two alternative epoch values. It is not defined which of the two will be returned by this function in this situation.

**Example**

```
S = '15021_41490'
SET.TIMEZONE 'Pacific/Wake'
DISPLAY MVEPOCH(S)
```

The above program fragment displays 1234567890.

**See also:**
Date, Times and Epoch Values, **EPOCH()**, **MVDATE()**, **MVDATE.TIME()**, **MVTIME()**, SET.TIMEZONE
The **MVTIME()** function returns the multivalue style time value (seconds since midnight) for a supplied epoch value using the currently selected time zone.

**Format**

\[
\text{MVTIME}(\text{epoch.value})
\]

where

\[
\text{epoch.value}
\]

is the epoch value to be converted.

An epoch value represents a moment in time. The **MVTIME()** function returns the time part of this as relevant to the currently selected time zone.

**Example**

\[
\begin{align*}
T &= 1234567890 \\
\text{SET.TIMEZONE} &\ 'GMT' \\
\text{DISPLAY} &\ \text{MVTIME}(T) \\
\text{SET.TIMEZONE} &\ 'Pacific/Wake' \\
\text{DISPLAY} &\ \text{MVTIME}(T)
\end{align*}
\]

The above program fragment displays

\[
84690 \\
41490
\]

The times differ because this moment in time was 23:31:30 on 13 February 2009 in the GMT timezone but it was 11:31:30 on 14 February 2009 in the Pacific/Wake time zone.

**See also:**

- [Date, Times and Epoch Values](#)
- **EPOCH()**
- **MVDATE()**
- **MVDATE.TIME()**
- **MVEPOCH()**
- **SET.TIMEZONE**
NAP

The NAP statement causes the program in which it is executed to pause for a given number of milliseconds.

Format

NAP time

where

time is the number of milliseconds for which the program is to sleep.

If the value of time is less than 5000, the program pauses for the given number of milliseconds. This sleep cannot be interrupted by the quit key.

If the value of time is 5000 or greater, the value is truncated to whole seconds and the program enters an interruptible sleep for that period.

The actual sleep time may vary from that specified due to process scheduling actions within the operating system.

See also:
SLEEP
NEG() function returns the arithmetic inverse of a value. The NEGS() function is similar to NEG() but operates on successive elements of a dynamic array, returning a similarly structured dynamic array of results.

**Format**

```
NEG(expr)
```

where

- `expr` evaluates to a number or a numeric array.

The NEG() function returns the negative of `expr`. It is equivalent to `-expr`.

If `expr` is a numeric array (a dynamic array where all elements are numeric), the NEG() function operates on each element in turn and returns a numeric array with the same structure as `expr`.

**Example**

```
N = NEG(A)
```

This statement finds the arithmetic inverse of A and assigns this to N.
NES()

The NES() function processes two dynamic arrays, returning a similarly structured result array indicating whether corresponding elements are not equal.

Format

\[
\text{NES(} expr1, \text{expr2) }
\]

where

\[
expr1 \text{ and expr2 are the dynamic arrays to be compared.}
\]

The NES() function compares corresponding elements of the dynamic arrays \( expr1 \) and \( expr2 \), returning a similarly structured dynamic array of True / False values indicating the results of the comparison.

The REUSE() function can be applied to either or both expressions. Without this function, any absent trailing values are taken as zero.

Example

A contains 1114ABC
B contains 1214ACB

\[
C = \text{NES (A, B)}
\]

C now contains 1000

See also: ANDS(), EQS(), GES(), GTS(), IFS(), LES(), LTS(), NOTS(), ORS(), REUSE()
**NOBUF**

The NOBUF statement turns off buffering for a record opened using **OPENSEQ**.

**Format**

```
NOBUF file.var
   {THEN statement(s)}
   {ELSE statement(s)}
```

where

- `file.var` is the file variable associated with the record by a previous **OPENSEQ** statement.
- `statement(s)` are statement(s) to be executed depending on the outcome of the NOBUF statement.

At least one of the **THEN** and **ELSE** clauses must be present.

Normally, QM buffers data for records opened using **OPENSEQ**. The NOBUF statement turns off this buffering such that **READBLK** will read the exact number of bytes specified and **READSEQ** will read byte by byte until a line feed character is found. **WRITEBLK**, **WRITESTEQ** and **WRITESTEQF** will write data without intermediate buffering.

Using unbuffered processing will result in lower performance than normal operation but may be useful, for example, when the item opened using **OPENSEQ** is actually a device or a pipe rather than a file system data record.

**See also:**

NOT()

The NOT() function returns the logical inverse of its argument. The NOTS() function is similar to NOT() but operates on successive elements of a dynamic array, returning a similarly structured dynamic array of results.

Format

    NOT(expr)

where

expr evaluates to a Boolean value

The NOT() function returns the logical inverse of expr. If expr is False (including a null string which equates to zero), the NOT() function returns True. If expr is True, the NOT() function returns False.

Example

    IF NOT(NUM(S)) THEN GOTO ERROR

This statement causes the program to jump to label ERROR if S is not numeric.

See also:

ANDS(), EQS(), GES(), GTS(), IFS(), LES(), LTS(), NES(), ORS(), REUSE()
NULL

The **NULL** statement performs no action.

**Format**

```
NULL
```

The **NULL** statement is useful to satisfy the requirements of QMBasic syntax where no specific action is required. No object code is generated by this statement.

**Example**

```
READ REC FROM CONTROL.FILE, "INVOICE.LIST" ELSE NULL
```

This statement reads the record "INVOICE.LIST" from the file open as file variable CONTROL.FILE. The **READ** statement must have either or both of the **THEN** and **ELSE** clauses. If the record is not found, REC will be set to a null string by the **READ** statement and the **ELSE** clause will be executed. As no further action is required, this clause is simply a **NULL** statement.
**NUM()**

The **NUM()** function tests whether a string can be converted to a number. The **NUMS()** function is similar to **NUM()** but operates on successive elements of a dynamic array, returning a similarly structured dynamic array of results.

**Format**

```
NUM(string)
```

where

```
string
evaluates to the string to be tested.
```

The **NUM()** function returns True if the *string* can be converted to a number. The function returns False for a string which cannot be converted to a number. A null string is a valid representation of zero and hence causes **NUM()** to return True.

If the NUMERIC.EXPONENT mode of the **OPTION** command is enabled, The **NUM()** function will accept exponent style values such as 123.45E-1 as being numeric.

**Example**

```
LOOP
    DISPLAY "Enter part number ":
    INPUT PART.NO
    UNTIL LEN(PART.NO) AND NUM(PART.NO)
    PRINTERR "Part number is invalid"
REPEAT
```

This program fragment prompts for and inputs a part number. If the data entered is null or cannot be converted to a number, an error message is displayed and the prompt is repeated.
**NUMERIC()**

The NUMERIC() function converts a value to Numeric form.

**Format**

\[
\text{NUMERIC}(expr)
\]

where

\[
expr \quad \text{is the value to be converted.}
\]

The NUMERIC() function converts \( expr \) to numeric form, integer or floating point.

This function is rarely needed because all QMBasic operations requiring a numeric value perform the conversion automatically, however, if the value is used many times in a loop, there may be some performance advantage in converting it just once.

Traditionally multivalue Basic programs have performed this conversion by a statement of the form

\[
N = X + 0
\]

where the addition operator will convert the value of \( X \) to numeric form. The NUMERIC() function provides a neater and marginally more efficient way to do this.

**Example**

\[
N = \text{NUMERIC}(X)
\]

This statement converts \( X \) to numeric form.

**See also:**

BOOL(), INT(), STR()
NV(), NS()

The NV() function returns a dynamic array containing ascending value position numbers corresponding to a supplied dynamic array. The NS() function is similar for subvalues.

Format

\[
\text{NV(string \{, repeat\})} \\
\text{NS(string)}
\]

where

- \text{string} evaluates to the string to be processed.
- \text{repeat} indicates whether the value position should be repeated for each subvalue. If omitted, it defaults to False.

The NV() and NS() functions are intended for use in I-type expressions but can be used in QMBasic programs.

The NV() function takes a dynamic array as its argument and returns a dynamic array where each value in the input array is replaced by its value position number. If the supplied string contains subvalues, the repeat argument determines how the function behaves. If this is True, the returned data will contain the value position for each subvalue in string. If it is False or omitted, subvalues are ignored, inserting just one value position number for each value.

The NS() function takes a dynamic array as its argument and returns a dynamic array where each subvalue in the input array is replaced by its subvalue position number.

Examples

\[ S = 11\text{VM}121\text{SM}122\text{SM}123\text{VM}131\text{SM}132\text{SM}211\text{SM}212\text{VM}221 \]

\[
\begin{align*}
\text{NV}(S,0) & \quad 1\text{VM}2\text{VM}3\text{FM}1\text{VM}2 \\
\text{NV}(S,1) & \quad 1\text{VM}2\text{SM}2\text{SM}2\text{VM}3\text{SM}3\text{FM}1\text{SM}1\text{VM}2 \\
\text{NS}(S) & \quad 1\text{VM}1\text{SM}2\text{SM}3\text{SM}1\text{SM}2\text{VM}1
\end{align*}
\]

The example below is based on the QM demonstration sales database and assumes that an I-type item named LINE.NO has been created as:

1: I
2: NV(ITEM)
3:
4: Line
5: 2R
6: M
7: LINE

A query

\[
\text{LIST SALES DATE CUST LINE.NO ITEM QTY}
\]
might produce a report containing

<table>
<thead>
<tr>
<th>SALES.....</th>
<th>Date.....</th>
<th>Cust</th>
<th>Line</th>
<th>Item</th>
<th>Qty</th>
</tr>
</thead>
<tbody>
<tr>
<td>12140</td>
<td>14 Oct 07</td>
<td>1056</td>
<td>1</td>
<td>003</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>122</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>234</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>121</td>
<td>6</td>
</tr>
<tr>
<td>12347</td>
<td>07 Feb 08</td>
<td>1087</td>
<td>1</td>
<td>004</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
<td>131</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td>232</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4</td>
<td>055</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>021</td>
<td>6</td>
</tr>
</tbody>
</table>
OBJECT()

The **OBJECT()** function instantiates an object for *object oriented programming*.

**Format**

\[
\text{OBJECT}(\text{cat.name}, \{\text{args}\})
\]

\[
\text{OBJECT}(\text{obj.var})
\]

where

- **cat.name** is the name of the catalogued **CLASS** module defining the object.
- **args** are optional arguments that will be passed into the CREATE.OBJECT subroutine.
- **obj.var** is a reference to an instantiated object.

The first form of the **OBJECT()** function loads the catalogued class module defined by **cat.name** and creates an object that references it. The function returns an object reference that should be stored in a program variable.

\[
\text{OBJ} = \text{OBJECT}("\text{MYOBJECT}")
\]

If the class module includes a public subroutine named CREATE.OBJECT this is executed as part of object instantiation.

Copying an object reference variable creates a second reference to the same object, not a new instantiation of the same object. The second form of the **OBJECT()** function creates a new object as a clone of an existing object. The CREATE.OBJECT subroutine is not executed.

\[
\text{NEWOBJ} = \text{OBJECT}(\text{OBJ})
\]

An object remains in existence until the last variable referencing it is overwritten or discarded. At this point, if the class module includes a public subroutine named DESTROY.OBJECT, it will be executed.

Attempting to instantiate an object that does not exist will result in an abort. Alternatively, the error can be trapped by use of an exception handler:

```
TRY
  OBJ = OBJECT("MYOBJECT")
CATCH SYS$ANY
    DISPLAY "Cannot load object"
END
```

**See also:**

*Object oriented programming*, **CLASS**, **DISINHERIT**, **INHERIT**, **OBJINFO()**, **PRIVATE**, **PUBLIC**.
OBJINFO()

The OBJINFO() function returns information about an object variable.

Format

OBJINFO(var, key)

where

var is the name of the object variable.

key identifies the information to be returned:

- 0 OISOBJ Returns True if var is an object, False if not.
- 1 OISCLASS Returns the class module catalogue name associated with the object.
- 2 OISNAMES Returns names of all public items in the object as a dynamic array, one field per class in inheritance name scan order divided into five values:
  V1 = Class catalogue name
  V2 = Subvalue mark delimited list of public functions in this class
  V3 = Subvalue mark delimited list of public subroutines in this class
  V4 = Subvalue mark delimited list of all public variables in this class
  V5 = Subvalue mark delimited list of directly inherited classes
  V6 = Subvalue mark delimited list of all read-only public variables in this class (also in V4).

The OBJINFO() function returns information about an object variable based on the key value supplied.

See also:
CLASS, OBJECT
OCONV()

The OCONV() function performs output conversion. Data is converted from its internal representation to the external form. This function is typically used to convert data for display or printing. The OCONVS() function is identical to OCONV() except that it works on each element of a dynamic array, returning the result in a similarly delimited dynamic array.

Format

\[
\begin{align*}
\text{OCONV}(\text{expr}, \text{conv.spec}) \\
\text{OCONVS}(\text{expr}, \text{conv.spec})
\end{align*}
\]

where

\[
\begin{align*}
\text{expr} & \quad \text{evaluates to the data to be converted.} \\
\text{conv.spec} & \quad \text{evaluates to the conversion specification. This may be a multivalued string containing more than one conversion code separated by value marks. Each conversion will be carried out in turn on the result of the previous conversion.}
\end{align*}
\]

The OCONV() function converts the value of expr to its external representation according to the conversion codes in conv.spec.

If conv.spec is a null string, OCONV() returns expr as its result.

The OCONV() function sets the STATUS() function value to indicate whether the conversion was successful. Possible values are

\[
\begin{align*}
0 & \quad \text{Successful conversion.} \\
1 & \quad \text{Data to convert was invalid for the conversion specification.} \\
2 & \quad \text{The conversion code was invalid.}
\end{align*}
\]

Conversions that result in a non-zero STATUS() value return the string that failed to convert as the function result. For an OCONV() function where conv.spec is not multivalued or where the first stage of a multiple conversion fails, the function would return expr. If one or more stages of a multivalued conv.spec have been completed, the returned value is the result of the last successful stage.

Examples

\[
\begin{align*}
\text{DT} & = 14992 \\
\text{DISPLAY OCONV(DT, 'D2')} \\
\text{DT} & = 14992 \\
\text{VM} & = 14000 \\
\text{S} & = \text{OCONVS(DT, 'D2')} \\
\end{align*}
\]

The above example converts a date from internal format as a value 14992 to its external form as 16 JAN 09.

\[
\begin{align*}
\text{DT} & = 14992_{\text{VM}}14000 \\
\text{S} & = \text{OCONVS(DT, 'D2')} \\
\end{align*}
\]

The above example converts a multivalued list of internal format dates to an equivalent multivalued list of external form dates. Variable S is set to 16 JAN 09, VM 30 APR 06
See also:
Conversion codes, ICONV()
ON GOSUB

The **ON GOSUB** statement enters one of a list of internal subroutines depending on the value of an expression.

**Format**

\[
\text{ON } expr \text{ GOSUB } label1\{;\}, \ label2\{;\}, \ label3\{;\}
\]

where

- \( expr \) is an expression which can be resolved to a numeric value
- \( label1... \) are statement labels. The trailing colons are optional and have no effect on the behaviour of the statement.

The **ON GOSUB** statement may be written over multiple lines by inserting a newline after the comma separating two labels.

Execution of the program continues at \( label1 \) if the value of \( expr \) (converted to an integer) is 1, \( label2 \) if it is 2 and so on. By default, a value less than one will use \( label1 \) and a value greater than the number of labels in the list will use the last label. The PICK.JUMP.RANGE option of the \$MODE directive can be used to invoke the Pick style behaviour where an out of range value continues execution at the statement following the **ON GOSUB**.

See the **GOSUB** statement for more details on internal subroutines.

**Example**

\[
\text{ON X GOSUB SUBR1, SUBR2, SUBR3}
\]

This program fragment enters one of three internal subroutines depending on the value of variable \( X \). If \( X \) could not be guaranteed to hold a valid value (1 to 3), error checking statements should be included to ease debugging of program errors.
ON GOTO

The ON GOTO statement jumps to one of a list of labels depending on the value of an expression.

Format

\[
\text{ON expr GOTO label1:, label2:, label3:} \\
\text{ON expr GO [TO] label1:, label2:, label3:}
\]

where

\[
\text{expr}\quad\text{is an expression which can be resolved to a numeric value}
\]

\[
\text{label1...}\quad\text{are statement labels. The trailing colons are optional and have no effect on the behaviour of the statement.}
\]

The ON GOTO statement may be written over multiple lines by inserting a newline after the comma separating two labels.

Execution of the program continues at label1 if the value of expr (converted to an integer) is 1, label2 if it is 2 and so on. By default, a value less than one will use label1 and a value greater than the number of labels in the list will use the last label. The PICK.JUMP.RANGE option of the $MODE$ directive can be used to invoke the Pick style behaviour where an out of range value continues execution at the statement following the ON GOTO.

Example

\[
\text{ON ACTION GOTO DISPLAY.REPORT,} \\
\text{PRINT.REPORT,} \\
\text{SAVE.REPORT}
\]

This program fragment jumps to one of three labels depending on the value of variable ACTION. If ACTION could not be guaranteed to hold a valid value (1 to 3), error checking statements should be included to ease debugging of program errors.
OPEN

The `OPEN` statement opens a directory file or dynamic file, associating it with a file variable.

**Format**

```
OPEN {dict.expr,} filename.expr {options} TO file.var
{ON ERROR statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

where

- `dict.expr` evaluates to DICT (case insensitive) to open the dictionary portion of the file or to a null string to open the data portion. If omitted or any other value, the data portion is opened. If given as a literal value, it must be enclosed in quotes. The `dict.expr` is followed by a comma.

- `filename.expr` evaluates to the VOC name of the file to be opened. If given as a literal value, it must be enclosed in quotes.

- `options` control the manner in which the file is opened and may be any combination of:
  - `ENCODING name` Specifies the default encoding mode for directory files. Ignored for other file types.
  - `NON.TRANSACTIONAL` Ignore transaction boundaries
  - `NO.MAP` Suppress id character translations in directory files
  - `READONLY` Disable file updates
  - `SYNC` Force write all file updates

- `file.var` is the name of the variable to hold the file reference for use in later operations on this file.

- `statement(s)` are statements to be executed depending on the outcome of the `OPEN` operation.

At least one of the `THEN` and `ELSE` clauses must be present.

A file opened by the `OPEN` statement may be referenced using the file variable in subsequent statements that operate on the file. The file remains open so long as the file variable remains intact. Overwriting this variable or discard of the variable on return from a subroutine will implicitly close the file. QM has limited support for the concept of a default file variable as found in some other multivalue products.

The `ENCODING` option, applicable only to directory files, sets the default character encoding to be used and overrides any encoding set in the VOC F-type record. The `name` may be a quoted literal string or an expression that evaluates to the encoding name. A null string as the encoding name is equivalent to not having the `ENCODING` clause at all, falling back on use of any encoding specified
in the VOC. To disable an encoding set via the VOC record, the encoding name should be specified as "NULL".

The optional \texttt{NON.TRANSACTIONAL} clause indicates that updates to the file are not to be treated as part of any transaction within which they occur.

The \texttt{NO.MAP} option is relevant only to directory files and suppresses the normal translation of restricted characters in record ids. See \texttt{directory files} for more information.

The optional \texttt{READONLY} clause opens the file for read only access. Any attempt to write will fail.

The \texttt{SYNC} option causes updates to the file to be flushed to disk after every write. This will have a severe impact on performance if the file is updated frequently. See synchronous (forced write) mode in the section that discusses \texttt{dynamic files} for more details.

If the file is opened successfully, the \texttt{THEN} clause is executed.

If the open fails the \texttt{ELSE} clause is executed and the \texttt{STATUS()} function may be used to determine the cause of the failure. The default action is to set the file variable as unassigned. Use of the \texttt{OPEN.FAIL.ZERO.FVAR} setting of the \texttt{$MODE} compiler directive changes this to set the file variable to zero.

The \texttt{ON ERROR} clause is taken only in the case of serious errors such as damage to the file's internal control structures. The \texttt{STATUS()} function will contain an error number. If no \texttt{ON ERROR} clause is present, a fatal error results in an abort.

QM allows more files to be open than the underlying operating system limit. This is achieved by automatically closing files at the operating system level if the limit is reached, retaining information to reopen them automatically when the next access to the file occurs. This process allows greater freedom of application design but has a performance penalty if a large number of files are used frequently.

For dynamic files, the \texttt{INMAT()} function used immediately after the \texttt{OPEN} returns the modulus of the file.

**Example**

```
OPEN "STOCK.FILE" TO STOCK ELSE ABORT "Cannot open file"
```

This statement opens a file with VOC name \texttt{STOCK.FILE}. If the open fails, the program aborts with an error message.

\textbf{See also:} \texttt{OPENPATH, OPENSEQ, OPENTEMP}
OPENPATH

The **OPENPATH** statement opens a directory file or dynamic file by pathname, associating it with a file variable.

**Format**

```
OPENPATH pathname {options} TO file.var
{ON ERROR statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

where

- **pathname** evaluates to the pathname of the file to be opened.
- **options** control the manner in which the file is opened and may be any combination of:
  - **ENCODING name** Specifies the default encoding mode for directory files. Ignored for other file types.
  - **NON.TRANSACTIONAL** Ignore transaction boundaries
  - **NO.MAP** Suppress id character translations
  - **READONLY** Disable file updates
  - **SYNC** Force write all file updates.

- **file.var** is the name of the variable to hold the file reference for use in later operations on this file.
- **statement(s)** are statements to be executed depending on the outcome of the **OPENPATH** operation.

At least one of the **THEN** and **ELSE** clauses must be present.

A file opened by the **OPENPATH** statement may be referenced using the file variable in subsequent statements that operate on the file. The file remains open so long as the file variable remains intact. Overwriting this variable or discard of the variable on return from a subroutine will implicitly close the file. QM has limited support for the concept of a default file variable as found in some other multivalue products.

The **ENCODING** option, applicable only to directory files, sets the default character encoding to be used. The **name** may be a quoted literal string or an expression that evaluates to the encoding name.

The optional **NON.TRANSACTIONAL** clause indicates that updates to the file are not to be treated as part of any transaction within which they occur.

The **NO.MAP** option is relevant only to directory files and suppresses the normal translation of restricted characters in record ids. See directory files for more information.

The optional **READONLY** clause opens the file for read only access. Any attempt to write to the file will fail.
The **SYNC** option causes updates to the file to be flushed to disk after every write. This will have a severe impact on performance if the file is updated frequently. See synchronous (forced write) mode in the section that discusses dynamic files for more details.

If the file is opened successfully, the **THEN** clause is executed.

If the open fails the **ELSE** clause is executed and the **STATUS()** function may be used to determine the cause of the failure. The default action is to set the file variable as unassigned. Use of the **OPEN.FAIL.ZERO.FVAR** setting of the **$MODE** compiler directive changes this to set the file variable to zero.

The **ON ERROR** clause is taken only in the case of serious errors such as damage to the file's internal control structures. The **STATUS()** function will contain an error number. If no **ON ERROR** clause is present, a fatal error results in an abort.

QM allows more files to be open than the underlying operating system limit. This is achieved by automatically closing files at the operating system level if the limit is reached, retaining information to reopen them automatically when the next access to the file occurs. This process allows greater freedom of application design but has a performance penalty if a large number of files are used frequently.

For dynamic files, the **INMAT()** function used immediately after the **OPENPATH** returns the modulus of the file.

**Example**

```q_macros
OPENPATH "\QMSYS\NEWVOC" TO NEWVOC ELSE ABORT "Cannot open NEWVOC"
```

This statement opens the skeleton NEWVOC file in the QMSYS directory using Windows pathname syntax. If the open fails, the program aborts with an error message.

**See also:**

**OPEN**, **OPENSEQ**, **OPENTEMP**
The **OPENSEQ** statement opens a record of a directory file, a device or a pipe for sequential access.

**Format**

```
OPENSEQ file.name, id {options} TO file.var
{ON ERROR statement(s)}
{LOCKED statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

or

```
OPENSEQ pathname {options} TO file.var
{ON ERROR statement(s)}
{LOCKED statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

where

- **file.name** evaluates to the VOC name of the directory file holding the record to be opened.
- **id** evaluates to the name of the record to be opened.
- **pathname** evaluates to the operating system pathname of the record to be opened. This may be a file, a device or a pipe.
- **options** are chosen from the following, each of which is described in detail below:
  - **ENCODING name**
  - **APPEND**
  - **CREATING**
  - **NOBUF**
  - **NO.MAP**
  - **OVERWRITE**
  - **READONLY**
  - **SHARED**
- **file.var** is the name of a variable to be used in later statements accessing this record.
- **statement(s)** are statement(s) to be executed depending on the outcome of the **OPENSEQ** statement.

At least one of the **THEN** and **ELSE** clauses must be present.

The named record is opened and associated with **file.var** for later operations.

The optional **ENCODING** clause sets the default character encoding to be used and overrides any encoding set in the VOC F-type record for the first form of **OPENSEQ**. Because the second form of this statement does not reference the VOC, any encoding to be applied must be specified using the **ENCODING** option. The **name** may be a quoted literal string or an expression that evaluates to the encoding name. A null string as the encoding name is equivalent to not having the **ENCODING** clause at all. To disable the default encoding, the encoding name should be specified as “NULL”.

Use of the **APPEND** option causes the **OPENSEQ** statement to position at the end of any existing data in the record such that subsequent write operations will append new data. Use of **OVERWRITE** truncates the record to remove any existing data. These two options may not be used together.

The **NOBUF** option causes the file, device or pipe opened by **OPENSEQ** to be accessed in an unbuffered mode. This is likely to degrade performance compared to use of the default buffered mode but allows, for example, two processes to read from the same pipe.

The **NO.MAP** option suppresses the normal translation of restricted characters in record ids. See directory files for more information.

The **READONLY** option opens the item for read only access. Any attempt to write will fail.

The **SHARED** option allows multiple QM processes to open the same record such that use of **WRITEBLK, WRITECSV, WRITELSEQ** or **WRITESEQF** will append to the record, allowing easy construction of audit trail logs or other similar items. Use of this option implies **APPEND** and **NOBUF**. Also, the record will be created by **OPENSEQ** if it does not already exist.

If the record already exists, the **THEN** clause is executed. Except when using the **SHARED** option, an update lock will be set on this record unless the record is read-only in which case a shared read lock is set.

If the record does not already exist, the action depends on whether the **CREATING** option has been used. If this option is present, the record is created and the **THEN** clause is executed. Otherwise the **ELSE** clause is executed and the **STATUS()** function returns zero. The record will have been locked and use of **WRITELSEQ, WRITELSEQF, WRITEBLK, WEOFSEQ** or **CREATE** with the returned **file.var** will create the record. Alternatively, the lock can be released using **RELEASE** or closing the **file.var**

The **ELSE** clause is also executed if the specified item cannot be opened due to an error. The **STATUS()** function will contain the error code.

The **LOCKED** clause is executed if the record is already locked by another process.

The **ON ERROR** clause is executed if a fatal error occurs when opening the record. The **STATUS()** function will return an error code relating to the problem.

A record open for sequential access may be read and written using **READSEQ** and **WRITELSEQ** respectively. The **WRITESEQF** statement provides a forced write and **WEOFSEQ** sets an end of file marker. The record should be closed using **CLOSESEQ** though it will be closed automatically when the program in which the file variable lies terminates.

The second form of **OPENSEQ** may be used to open a serial port by using the device name as **pathname**. On Windows, this name is COM1, COM2, etc optionally followed by a colon. On other platforms, it is the device driver name.

To open a floppy disk drive (e.g. a Pick style account save) specify the pathname as "A:" on Windows or use the device driver name (probably /dev/fd0) on Linux.

**Examples**

```
OPENSEQ "STOCKS", "STOCK.LIST" TO STOCK.LIST ELSE
   IF STATUS() THEN ABORT "Cannot open stocks list"
```
This program fragment opens the record STOCK.LIST of directory file STOCKS. If it fails to either open an existing record or to create a new record, the program aborts.

```
OPENSEQ "C:\TEMP\IMPORT.DATA" TO DAT.F ELSE
    IF STATUS() THEN ABORT "Cannot open import data file"
END
```

This program fragment opens the operating system file in C:\TEMP\IMPORT.DATA for sequential processing.

See also:
- CLOSESEQ
- CREATE
- NOBUF
- READBLK
- READCSV
- READSEQ
- SEEK
- WEOFSEQ
- WRITEBLK
- WRITECSV
- WRITESEQ
- WRITESEQF
OPENTEMP

The OPENTEMP statement creates and opens a temporary dynamic file by pathname, associating it with a file variable.

Format

```
OPENTEMP {pathname} {options} TO file.var
{ON ERROR statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

where

- **pathname** evaluates to the pathname of the file to be opened. If omitted or given as a null string, the temporary file is created in the QM temporary directory (see the TEMPDIR configuration parameter).

- **options** control the manner in which the file is opened. The available options are:
  - ECS
    - Create ECS mode file
    - (ignored on non-ECS systems)
  - NON.TRANSACTIONAL
    - Ignore transaction boundaries

- **file.var** is the name of the variable to hold the file reference for use in later operations on this file.

- **statement(s)** are statements to be executed depending on the outcome of the OPENTEMP operation.

At least one of the THEN and ELSE clauses must be present.

A file opened by the OPENTEMP statement may be referenced using the file variable in subsequent statements that operate on the file. The file is opened for exclusive use by the QM session in which the OPENPATH statement is executed and remains open so long as the file variable remains intact. Overwriting this variable or discard of the variable on return from a subroutine will implicitly close the file. The file will be deleted automatically when it is closed.

The optional NON.TRANSACTIONAL clause indicates that updates to the file are not to be treated as part of any transaction within which they occur.

If the file is opened successfully, the THEN clause is executed. If the open fails the ELSE clause is executed and the STATUS() function may be used to determine the cause of the failure.

The ON ERROR clause is taken only in the case of serious errors such as damage to the file's internal control structures. The STATUS() function will contain an error number. If no ON ERROR clause is present, a fatal error results in an abort.

Examples

```
OPENTEMP "\TEMP\MYFILE" TO TEMP.F ELSE ABORT "Cannot open temporary file"
```
This statement opens a temporary file as pathname \TEMP\MYFILE. If the open fails, the program aborts with an error message.

```
OPENTEMP TO TEMP.F ELSE ABORT "Cannot open temporary file"
```

This statement opens a temporary file in the QM temporary directory. If the open fails, the program aborts with an error message.

See also:
OPEN, OPENPATH
OPEN.SOCKETF()  

The OPEN.SOCKETF() function opens a data socket for an outgoing connection.

**Format**

\[
\text{OPEN.SOCKETF}(\text{addr, port, flags}, \text{timeout})
\]

where

- **addr** is the address of the system to which a connection is to be established. This may be an IPV4 or IPV6 address or a host name. Use of IPV6 must be enabled using the IPV6 configuration parameter.

- **port** is the port number on which the connection is to be established.

- **flags** is a flag value formed from the additive values below. The token values are defined in the SYSCOM KEYS.H include record.

  - Socket type, one of:
    - SKT$STREAM: A stream connection (default)
    - SKT$DGRAM: A datagram type connection
    - SKT$USTREAM: A named Unix domain socket connection

  - Protocol, one of:
    - SKT$TCP: Transmission Control Protocol (default)
    - SKT$UDP: User Datagram Protocol

  - Blocking mode, one of:
    - SKT$BLOCKING: Sets the default mode of data transfer as blocking.
    - SKT$NON.BLOCKING: Sets the default mode of data transfer as non-blocking.

- **timeout** is the maximum number of seconds to wait for connection.

The OPEN.SOCKETF() function opens a connection to the server with the given address and port number.

For Unix domain sockets, the **addr** argument is the pathname of the socket connection and the **port** and **timeout** arguments are ignored.

If the action is successful, the function returns a socket variable that can be used to read and write data using the READ.SOCKETF() and WRITE.SOCKETF() functions. The STATUS() function will return zero.

If the socket cannot be opened, the STATUS() function will return an error code that can be used to determine the cause of the error, possibly in conjunction with OS.ERROR().

**Example**

```plaintext
SKT = OPEN.SOCKETF("193.118.13.14", 3000, SKT$BLOCKING)
IF STATUS() THEN STOP 'Cannot open socket'
N = WRITE.SOCKETF(SKT, DATA, 0, 0)
CLOSE.SOCKETF SKT
```
This program fragment opens a connection to port 3000 of IP address 193.118.13.14, sends the data in DATA and then closes the socket.

See also:
Using Socket Connections, ACCEPT.SOCKET.CONNECTION, CLOSE.SOCKE
CREATE.SERVER.SOCKE(), READ.SOCKE(), SELECT.SOCKE(), SERVER.ADDR(),
SET.SOCKE.MODE(), SOCKET.INFO(), WRITE.SOCKE()
OPTION()

The OPTION() function returns or changes the setting of an option flag.

Format

\[
\text{OPTION}(\text{optno})
\]

\[
\text{OPTION}(\text{optno}, \text{value})
\]

where

- \text{optno} is the option flag number to be tested.
- \text{value} is the new state to be applied to the option.

The OPTION() function returns the state of the option identified by \text{optno}. If \text{optno} is out of range, the function returns False.

The two argument form of the function sets the option state and returns the old state.

Option key values are defined in the SYSCOM KEYS.H record.

A negative \text{optno} value can be used to retrieve or set all options. Used in the single argument form of the OPTION() function, this returns a character string that represents the complete set of option values. Using this string in the two argument form of the function reverts all options to the saved state, returning the previous state. A null \text{value} string turns off all options. Application programs must not make any assumptions about the structure of the returned option string and must not alter its value.

Examples

\[
\begin{align*}
\text{old.state} &= \text{option(OPT$DIR.SEL.Omit.HIDDEN, 1)} \\
\text{selectv} \text{ fv to files} \\
\text{void option(OPT$DIR.SEL.Omit.HIDDEN, old.state)}
\end{align*}
\]

The above code fragment sets the DIR.SEL.Omit.HIDDEN option for the following SELECTV operation and then restores the previous state of the option.

\[
\begin{align*}
\text{options} &= \text{option(-1)} \\
\text{void option(OPT$DIR.SEL.Omit.HIDDEN, 1)} \\
\text{selectv} \text{ fv to files} \\
\text{void option(-1, options)}
\end{align*}
\]

The above code fragment is an alternative way to achieve the same result. In this form, the code between the two uses of the negative \text{optno} could set or clear many options. The final line would restore the state as at the start of this example.

See also:

OPTION
**ORS()**

The **ORS**() function performs a logical OR operation on successive elements of a dynamic array, returning a similarly structured dynamic array of results.

**Format**

\[
\text{ORS}(\text{expr1, expr2})
\]

where

\[
\text{expr1, expr2}
\]

are the dynamic arrays to be processed.

The **ORS**() function performs the logical OR operation between corresponding elements of the two dynamic arrays and constructs a similarly structured dynamic array of results as its return value. An element of the returned dynamic array is 1 if either or both of the corresponding elements of \textit{expr1} and \textit{expr2} are True.

The **REUSE()** function can be applied to either or both expressions. Without this function, any absent trailing values are taken as False.

**Example**

A contains \(1\text{VM}1\text{SM}0\text{VM}0\text{VM}1\text{FM}0\text{VM}1\)
B contains \(1\text{VM}0\text{SM}1\text{VM}0\text{VM}1\text{FM}1\text{VM}0\)

\[C = \text{ORS}(A, B)\]

C now contains \(1\text{VM}1\text{SM}1\text{VM}0\text{VM}1\text{FM}1\text{VM}1\)

**See also:**

**ANDS()**, **EQS()**, **GES()**, **GTS()**, **IFS()**, **LES()**, **LTS()**, **NES()**, **NOTS()**, **REUSE()**
OS.ERROR()

The **OS.ERROR()** function returns the error number associated with the last recorded operating system level error.

**Format**

```
OS.ERROR()
```

Some actions that return errors via the **STATUS()** function are related to errors from operating system calls. The **OS.ERROR()** function returns the value of the most recent operating system error. The QM error codes for which this is valid are all marked with “os.error” in the SYSCOM ERR.H include record. The value returned by **OS.ERROR()** at other times is meaningless.

The values returned by **OS.ERROR()** are defined by the operating system and program development libraries. They are outside the control of QM and are documented with the operating system. In general users of Unix/Linux based systems can find these in the errno.h include record in /usr/include though this file then includes a variety of further include records. Windows users can find error numbers on the MSDN area of Microsoft's web site but they are scattered over several pages.

The **!ERRTEXT()** subroutine will automatically insert this value into relevant expanded error messages.
OS.EXECUTE

The OS.EXECUTE statement executes an operating system command.

Format

    OS.EXECUTE expr {CAPTURING var} [SILENT]

where

expr    evaluates to the command to be executed.
var     is a variable to receive captured output.

The OS.EXECUTE statement allows a QMBasic program to execute an operating system command. The program does not continue execution until the command terminates. QM attempts to redirect any output from the command back to the user's terminal but this is not always possible. Some commands may cause output to appear on the server system.

The CAPTURING clause captures output that would otherwise have gone to the terminal or phantom log file, saving it in the named variable with field marks in place of newlines. Alternatively, all command output can be suppressed by use of the SILENT clause.

The OS.EXECUTE statement returns two error codes. The STATUS() function returns a non-zero value if QM detected an error and was unable to execute the command. For a zero STATUS() value, the OS.ERROR() function returns the termination status of the executed command. The interpretation of this value will depend on the command being executed.

If the command to be executed on a Windows ECS mode system contains characters outside the 8-bit set, it is necessary to create a .bat file that starts with

    chcp 65001

to select Unicode mode followed by the commands to be executed. The OS.EXECUTE statement should then reference this .bat file.

Example

    OS.EXECUTE "MKDIR TEMPDIR"

This statement uses the operating system MKDIR command to create a directory named TEMPDIR.

Quoting on Windows systems

There is a well documented problem in Windows where command lines that need multiple sets of double quotes are not handled reliably. A command that references the Windows Program Files directory must include quotes around the pathname because it contains a space. Thus a command to be executed by OS.EXECUTE might be something like

    "C:\Program Files\XXX\YYY.EXE" AAA BBB

where the pathname of the executable needs quotes.

Conversely, a command might need to quote arguments that contain quotes
The problem occurs when there is a need to do both together

```
"C:\Program Files\XXX\YYY.EXE" "AAA BBB"
```

which looks as though it should work but actually fails with an error indicating that C:\Program is not a valid executable name.

The only reliable solution appears to be to write the command out to a batch (.bat) file and then use **OS.EXECUTE** to run that. The steps to do this might be something like:

```
OPEN '$ACC' TO ACC.F THEN
    SCRIPT.NAME = 'MYSCRIPT' : @USERNO : '.BAT'
    SCRIPT = '"C:\Program Files\XXX\YYY.EXE" "AAA BBB"'
    WRITE SCRIPT TO ACC.F, SCRIPT.NAME
    OS.EXECUTE SCRIPT.NAME
    DELETE ACC.F, SCRIPT.NAME
    CLOSE ACC.F
END
```

Note the use of **@USERNO** to ensure that the script name is unique across multiple QM sessions.
OSDELETE

The **OSDELETE** deletes an operating system file or directory by pathname.

**Format**

```
OSDELETE path
```

where

```
path
```

is the pathname of the file or directory to be deleted.

The **OSDELETE** statement provides a simple way for a program to delete an operating system file or directory by pathname. No error will be reported if the action fails.

**OSDELETE** takes no part in the locking system. It is up to the application developer to ensure that concurrency issues are handled appropriately.

**Example**

```
OSDELETE 'C:\FAXLOG'
```

This statement deletes an operating system file with pathname C:\FAXLOG.

**See also:**

**OSREAD, OSWRITE**
**OSPATH()**

The **OSPATH()** function performs actions on operating system files.

**Format**


```
OSPATH(path, key)
OSPATH(path, key, qualifier)
```

where

- **path** evaluates to a pathname.
- **key** identifies the action to be performed.
- **qualifier** provides additional information for the action to be performed.

The **OSPATH()** function returns information about **path**. The values of **key** and their associated actions are:

<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td><strong>OS$PATHNAME</strong> Test if <strong>path</strong> is a syntactically valid pathname (True/False).</td>
</tr>
<tr>
<td>1</td>
<td><strong>OS$FILENAME</strong> Test if <strong>path</strong> is a syntactically valid filename (True/False).</td>
</tr>
<tr>
<td>2</td>
<td><strong>OS$EXISTS</strong> Test if <strong>path</strong> exists as either file or directory (True/False).</td>
</tr>
<tr>
<td>3</td>
<td><strong>OS$UNIQUE</strong> Returns a unique file name.</td>
</tr>
<tr>
<td>4</td>
<td><strong>OS$FULLPATH</strong> Return full path name of <strong>path</strong>. On case insensitive file systems, the returned path will be upper case.</td>
</tr>
<tr>
<td>5</td>
<td><strong>OS$DELETE</strong> Deletes <strong>path</strong>. Returns success (True/False).</td>
</tr>
<tr>
<td>6</td>
<td><strong>OS$OPEN</strong> Check if file is open by pathname (True/False).</td>
</tr>
<tr>
<td>7</td>
<td><strong>OS$DTM</strong> Returns date/time modified of <strong>path</strong> as an epoch value, zero if path does not exist.</td>
</tr>
<tr>
<td>8</td>
<td><strong>OS$FLUSH.CACHE</strong> Flush dynamic file cache in local process. Always returns True.</td>
</tr>
<tr>
<td>9</td>
<td><strong>OS$FULLPATHX</strong> Return full path name of <strong>path</strong>, preserving casing.</td>
</tr>
<tr>
<td>10</td>
<td><strong>OS$PARENT</strong> Return parent directory of <strong>path</strong>. If <strong>path</strong> is a top level directory, the same pathname is returned.</td>
</tr>
<tr>
<td>11</td>
<td><strong>OS$UID</strong> Get UID of <strong>path</strong> (not Windows), zero if path does not exist.</td>
</tr>
<tr>
<td>12</td>
<td><strong>OS$GID</strong> Get GID of <strong>path</strong> (not Windows), zero if path does not exist.</td>
</tr>
<tr>
<td>13</td>
<td><strong>OS$MKDIR</strong> Creates directory <strong>path</strong>. Returns success (True/False).</td>
</tr>
<tr>
<td>14</td>
<td><strong>OS$MKPATH</strong> Creates directory <strong>path</strong> and any intermediate levels. Returns success (True/False).</td>
</tr>
<tr>
<td>15</td>
<td><strong>OS$MODES</strong> Returns file access modes of <strong>path</strong> (not Windows) as the decimal representation of a Linux mode value, zero if path does not exist. See below.</td>
</tr>
<tr>
<td>16</td>
<td><strong>OS$IS.FILE</strong> Test if <strong>path</strong> exists as a file (True/False)</td>
</tr>
<tr>
<td>17</td>
<td><strong>OS$IS.DIR</strong> Test if <strong>path</strong> exists as a directory (True/False)</td>
</tr>
</tbody>
</table>
OS$OWNER
Returns the owner of the file (Windows only)

OS$MKIFO
Create named pipe (Linux only)

OS$INFO
Returns a dynamic array of information similar to the STATUS statement.

OS$SPACE
Returns partition size and available space, separated by a field mark

OS$CHMOD
Sets file permissions from qualifier as the decimal representation of a Linux access mode value. See below. Returns success (True/False).

OS$SET.LINK
Create a hard link. The qualifier is the name of the link to be created.

OS$SET.UID.GID
Set uid and gid for named file. The qualifier is formed from the uid and gid, separated by a comma. These may be given as numeric values or as names. A null name indicates that the uid or gid is not to be changed. The return value is a Boolean value indicating the success of the action. (not Windows).

File Permission Values

Use of OSSMODES to retrieve the file access permissions or OS$CHMOD to set permissions is complicated by the conventional use of an octal representation of the bit significant mode value in Linux commands or in C programs. QMBasic has no syntax for an octal constant and hence these actions work with the decimal representation of the modes.

<table>
<thead>
<tr>
<th>Octal</th>
<th>Decimal</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>256</td>
<td>Owner, read access</td>
</tr>
<tr>
<td>200</td>
<td>128</td>
<td>Owner, write access</td>
</tr>
<tr>
<td>100</td>
<td>64</td>
<td>Owner, execute access</td>
</tr>
<tr>
<td>040</td>
<td>32</td>
<td>Group, read access</td>
</tr>
<tr>
<td>020</td>
<td>16</td>
<td>Group, write access</td>
</tr>
<tr>
<td>010</td>
<td>8</td>
<td>Group, execute access</td>
</tr>
<tr>
<td>004</td>
<td>4</td>
<td>Other, read access</td>
</tr>
<tr>
<td>002</td>
<td>2</td>
<td>Other, write access</td>
</tr>
<tr>
<td>001</td>
<td>1</td>
<td>Other, execute access</td>
</tr>
</tbody>
</table>

Examples

IF NOT(OSPATH(RID, OS$FILENAME)) THEN
  DISPLAY 'Not a valid filename'
END

The program fragment above validates a filename.

IF NOT(OSPATH('MYFILE', OS$SET.UID.GID, '100,staff')) THEN
  DISPLAY 'Error' : OS.ERROR() : ' setting uid/gid'
END

The program fragment above updates ownership of MYFILE.

OK = OSPATH('MYFILE', OSSCHMOD, ICONV('740', 'MO'))
The statement above sets modes octal 740 on MYFILE. This is more readable than using the decimal equivalent

OK = OSPATH('MYFILE', OSSCHMOD, 480)
OSREAD

The **OSREAD** statement reads an operating system file by pathname.

**Format**

```
OSREAD var FROM path [ON ERROR statement(s)]
{THEN statement(s)}
{ELSE statement(s)}
```

where

- `var` is the variable to receive the data.
- `path` is the pathname of the file to be read.

At least one of the **THEN** and **ELSE** clauses must be present.

The **OSREAD** statement provides a simple way for a program to read the content of an operating system file by pathname rather than having to open the parent directory with **OPENSEQ** and then using **READBLK** to read the data.

The data read from the specified file is transferred to `var` with no modification. In particular, note that the translation of newlines to field marks that occurs when reading from directory files does not happen with this statement.

**OSREAD** takes no part in the locking system. It is up to the application developer to ensure that concurrency issues are handled appropriately.

The **THEN** clause is executed if the read is successful. Use of **SYSTEM(1071)** after a successful read will return a dynamic array similar to that returned by the **STATUS** statement for a directory file item.

The **ELSE** clause is executed if the read fails because, for example, the pathname does not reference an existing file. Attempting to read an item larger than 1Gb will take the **ELSE** clause. The **STATUS()** function will indicate the cause of the error and the **OS.ERROR()** function will provide further information for operating system level errors.

The **ON ERROR** clause is executed for serious fault conditions such as the inability to read the data from the file for reasons outside of QM. The **STATUS()** function will return an error number and the **OS.ERROR()** function will provide further information about the operating system level error. If no **ON ERROR** clause is present, an abort would occur at these types of error.

**Example**

```
OSREAD FAXDATA FROM 'C:\FAXLOG' THEN CALL PROCESS.FAX(FAXDATA)
```

This statement reads the content of an operating system file with pathname C:\FAXLOG and, if found, executes a subroutine to process this data.

**See also:**

**OSDELETE**, **OSWRITE**
OSRENAME()

The OSRENAME() function renames an operating system file or directory.

Format

    OSRENAME(old.name, new.name)

where

    old.name    is the old name of the item.
    new.name    is the new name of the item.

The OSRENAME() function renames operating system item old.name to new.name. The items do not need to be in the same directory but, on Windows, must be on the same drive partition or, on Linux, must be in the same file system.

The function returns True if successful, False if it fails. The STATUS() function can be used to determine the cause of failure.
OSWRITE

The **OSWRITE** statement writes an operating system file by pathname.

**Format**

```
OSWRITE expr TO path {ON ERROR statement(s)}
```

where

- `expr` is the data to be written.
- `path` is the pathname of the file to be written.

The **OSWRITE** statement provides a simple way for a program to write an operating system file by pathname rather than having to open the parent directory with **OPENSEQ** and then using **WRITEBLK** to write the data.

The data is written to the specified file with no modification. In particular, note that the translation of field marks to newlines that occurs when writing to directory files does not happen with this statement. Any existing data in the file will be overwritten, truncating the file if the old content was larger than the new data.

**OSWRITE** takes no part in the locking system. It is up to the application developer to ensure that concurrency issues are handled appropriately.

The **ON ERROR** clause is executed for serious fault conditions such as the inability to write the data to the file. The **STATUS()** function will return an error number and the **OS.ERROR()** function will provide further information about the operating system level error. If no **ON ERROR** clause is present, an abort would occur at these types of error.

**Example**

```
OSWRITE FAXDATA TO 'C:\FAXLOG'
```

This statement writes the content variable FAXDATA to an operating system file with pathname C: \FAXLOG.

**See also:**

**OSDELETE, OSREAD**
OUTERJOIN()

The OUTERJOIN() function returns the record ids of records in a file where a field holds a specified value.

Format

\texttt{OUTERJOIN(file.name, field.name, value)}

where

- \texttt{file.name} evaluates to the name of the file from which data is to be retrieved. The evaluated \texttt{file.name} expression may include the uppercase word DICT before the actual file name and separated from it by a single space to specify that the dictionary portion of the file is to be used.

- \texttt{field.name} is the name of the field in \texttt{file.name} that determines the record ids to be returned. There must be an alternate key index on this field.

- \texttt{value} is the value that must appear in the specified field.

The OUTERJOIN() function uses an alternate key index on \texttt{field.name} to return a value mark delimited list of record ids of records in the specified file that contain the given value.

This function is mainly intended for use in dictionary I-type expressions where the equivalent programming built around \texttt{SELECTINDEX} cannot be used.

Examples

\texttt{OUTERJOIN('ORDERS', 'CUST.NO', CUST.NO)}

The above expression used in a dictionary I-type item retrieves a list of orders record ids for a given customer.

\begin{verbatim}
OPEN 'CUSTOMERS' TO CUS.F ELSE STOP 'Cannot open file'
SELECT CUS.F
LOOP
  READNEXT ID ELSE EXIT
  DISPLAY ID:': ':CHANGE(OUTERJOIN('ORDERS', 'CUST.NO', ID), @VM, ', ')
REPEAT
\end{verbatim}

The above program displays a list of customer numbers in the CUSTOMERS file and a list of the orders placed by each customer.
PAGE

The `PAGE` statement advances a print unit to a new page.

Format

\[ \text{PAGE} \{ \text{ON } \textit{print.unit} \} \{ \textit{page.no} \} \]

where

- \textit{print.unit} evaluates to the print unit number on which the action is to occur. If omitted, print unit zero is used.
- \textit{page.no} evaluates to the page number to be used for the new page. If omitted, the page number is incremented from its current value.

The `PAGE` statement causes the footing to be printed at the end of the current page and advances to the next page. The heading will be printed if further output is directed to the print unit. The `PAGE` statement can be used to complete printing of the final page of output from a program.

If a new page number is specified, this takes effect after print unit has been advanced to the new page. A \textit{page.no} of less than one causes the page number to be set to one.

A `PAGE` statement directed to the display causes the pagination prompt to be displayed unless it has been suppressed. The screen will be cleared to advance to the new page.

Example

```
PAGE ON PRINT.UNIT
```

This statement causes the print unit identified by PRINT.UNIT to advance to the next page.

See also:
- `FOOTING`
- `HEADING`
PARSE()

The PARSE() function matches a string against a pattern, inserting a delimiter between each pattern element.

Format

PARSE(string, pattern, delimiter)

where

- **string** evaluates to the string to be parsed against the supplied pattern.
- **pattern** evaluates to a template as described below.
- **delimiter** evaluates to the character to be inserted between each pattern element. If delimiter is a null string, a field mark is used. If delimiter is more than one character, only the first character is used.

The PARSE() function matches string against pattern and returns a copy of string with the delimiter character inserted between each pattern element. If the string does not completely match the pattern, a null string is returned.

The pattern string consists of one or more concatenated items from the following list.

- ... Zero or more characters of any type
- 0X Zero or more characters of any type
- nX Exactly n characters of any type
- n-mX Between n and m characters of any type
- 0A Zero or more alphabetic characters
- nA Exactly n alphabetic characters
- n-mA Between n and m alphabetic characters
- 0N Zero or more numeric characters
- nN Exactly n numeric characters
- n-mN Between n and m numeric characters
- "string" A literal string which must match exactly. Either single or double quotation marks may be used. Unlike the MATCHES operator, string must be enclosed in quotes otherwise each character is treated as a separate component.

The values n and m are integers with any number of digits. m must be greater than or equal to n.

The 0A, nA, 0N, nN and "string" patterns may be preceded by a tilde ( ~ ) to invert the match condition. For example, ~4N matches four non-numeric characters such as ABCD (not a string which is not four numeric characters such as 12C4).

A null string matches patterns ..., 0A, 0X, 0N, their inverses (~0A, etc) and "".
The 0X and \( n-mX \) patterns match against as few characters as necessary before control passes to the next pattern. For example, the string ABC123DEF matched against the pattern 0X2N0X matches the pattern components as ABC, 12 and 3DEF.

The 0N, \( n-mN \), 0A, and \( n-mA \) patterns match against as many characters as possible. For example, the string ABC123DEF matched against the pattern 0X2-3N0X matches the pattern components as ABC, 123 and DEF.

The *pattern* string may contain alternative templates separated by value marks. The `PARSE()` function tries each template in turn until one is a successful match against *string*. If a match is found, the `INMAT()` function can be used to retrieve the value position within the pattern that matched.

The *delimiter* argument determines the character to be inserted between the portions of *string* that match each part of the *pattern*.

The `PARSE()` function returns a null string if no component of *pattern* matches *string*.

**Example**

```plaintext
TEL.NO = "01604-709200"
DISPLAY PARSE(TEL.NO, "0N'-'0N", " | ")
```

This program fragment displays

```
01604 | - | 709200
```

**See also:**

- Pattern Matching
- `MATCHFIELD()`
PAUSE

The PAUSE statement pauses execution until awoken by another process.

Format

    PAUSE {timeout}

where

    timeout specifies the maximum time to wait in seconds. A value less than one indicates that an infinite timeout should be used.

The PAUSE statement suspends program execution until awoken by another process using the WAKE statement. The optional timeout specifies the maximum time in seconds for which the program can remain suspended.

If the PAUSE is terminated by detection of a WAKE event, the STATUS() function will return zero and the INMAT() function will return the user number of the process that performed the WAKE.

If the PAUSE is terminated by a timeout, the STATUS() function will return ER$TIMEOUT.

A WAKE request occurring before the PAUSE is executed is remembered and the program is not suspended. Note that under rare conditions, precise timing of the PAUSE/WAKE pair can cause a program to appear to wake spuriously. Programs should be written to allow for this possibility.
PDUMP

The **PDUMP** statement causes an immediate process dump of the process in which it is used.

**Format**

```
PDUMP
```

The **PDUMP** statement provides an easy way to creation of a diagnostic process dump file of the form generated at a run time error or by use of the **PDUMP** command.

**See also:**

*Process dump files, **PDUMP***
PHANTOM

The PHANTOM statement starts a phantom process from a QMBasic program without the overheads of using EXECUTE.

Format

PHANTOM command {options} {THEN statement(s)} {ELSE statement(s)}

where

command is the command to be executed by the phantom process.

options are:

GROUP The phantom process is grouped with its parent such that it will terminate if the parent terminates

LOGFILE name Specifies an alternative log file

NO.LOG Suppresses creation of a log file

NO.MSG Suppresses start and termination messages relating to this phantom in the parent process.

PASS.DATA Transfers the data queue from the parent process to the phantom.

POOL name Makes the phantom process a member of the specified connection pool.

The options may be in any order. At least one of the THEN and ELSE clauses must be present.

The PHANTOM statement is can be used to start a phantom process that does not need options supported only by the PHANTOM command. Removal of the need to use EXECUTE provides a small performance improvement.

If the LOGFILE option is used, name specifies the name of the record to be created in the $COMO file to save output from the phantom process. If this name contains a space, the portion before the space is used as the file name and the portion after the space is used as the record id within that file. Only a directory file can be used.

The GROUP keyword should be used with care. Termination of the parent process will cause all programs, paragraphs, etc running in the phantom process to be aborted. Unless the application uses transactions, this could result in only part of a linked sequence of updates being applied to data files.

When using the POOL option, the system will first look for an idle process in that pool that can be awakened. If no such process is found, a new phantom is started. The pool name may be followed by the keyword NEW.PROCESS to force creation of a new process even if an idle process exists. This allows an application to start a number of phantom processes that perform initialisation and then go into the idle pool waiting for work to do.

Any secure encryption keys enabled in the parent process are inherited by the phantom process.

The THEN clause is taken if the phantom process is started successfully. The STATUS() function will return the QM user number of the phantom process.
The **ELSE** clause is taken if the phantom fails to start. The `STATUS()` function will return the associated error code.

**Example**

```
PHANTOM 'RUN SALES.REPORT' LOGFILE 'SALES.LOG' NO.MSG
```

See also: `CHILD()`, `LIST.PHANTOMS`, `!PHLOG()`, `PHANTOM`, `POOL.IDLE`, `STATUS`
POOL.IDLE

The **POOL.IDLE** statement moves a phantom process into the idle state when using connection pooling.

**Format**

```
POOL.IDLE
```

The **POOL.IDLE** statement applies only to phantom processes that are members of a connection pool. It is ignored in all other process types.

Entering the idle state will close any socket referenced by the `@SOCKET` variable and then wait for new work. If the pool's idle process limit has been reached or the process remains idle for the timeout period of the pool, it will be terminated.

**See also:**
- Connection pooling, **PHANTOM** command, QMBasic **PHANTOM** statement
**PRECISION**

The **PRECISION** statement sets the maximum number of decimal places to appear when converting numeric values to strings.

**Format**

```
PRECISION expr

PRECISION INHERIT
```

where

```
expr
```

is an expression specifying the number of decimal places. This value must be between zero and fourteen. Negative values are treated as zero; values greater than fourteen are treated as fourteen.

Arithmetic operations performed by QM always work to the maximum precision of the computer system. The precision value determines the number of decimal places when numeric values are converted to strings, for example, when printing.

Values are converted with rounding on the last digit. Trailing zero digits are removed from the decimal places and, if the resultant value is an integer, the decimal point is also removed.

The precision value is associated with each program and subroutine and is initially set to 4. A program which sets a precision of 6 and calls a subroutine will use precision 6 up to the call, the subroutine will use precision 4 and, on return to the calling program, the precision reverts to 6.

Use of the INHERIT option to the **PRECISION** statement causes the program to inherit the precision value of the program from which it was called.

**Example**

```
X = 333.33333
Y = 666.66669
PRINT X, Y
PRECISION 4
PRINT X, Y
PRECISION 1
PRINT X, Y
PRECISION 0
PRINT X, Y
```

This program fragment would print

```
333.3333  666.6667
333.3     666.7
333       667
```
PRINT

The PRINT statement outputs data to a print unit.

Format

```
PRINT {NO.ENCODING} {ON print.unit} {print.list}
```

where

- `print.unit` identifies the print unit to which output is to be directed. If omitted, print unit zero is used.
- `print.list` is a list of items to print in the format described for the DISPLAY statement.

The data is output to the requested print unit. Print unit -1 is always associated with the display and cannot be changed. Print unit 0 can be switched between the display and the printer by use of the PRINTER statement. Print units 1 to 255 direct their output to the hold file by default but can be redirected using the SETPTR command.

The NO.ENCODING option suppresses encoding when the terminal or printer is set to use an encoding such as UTF-8. It is useful when the data to be output has already be encoded.

By using PRINT statements instead of DISPLAY in programs it is possible to select whether the output is directed to the display or to a printer. The LPTR option to the RUN command is equivalent to a PRINTER ON at the start of the program.

Use of the @(x,y) cursor movement function in a PRINT statement that sends output to the display will disable pagination. See DISPLAY for more details.

Example

```
N = DCOUNT(LINE, @FM)
FOR I = 1 TO N
   PRINT ON PU LINE<I>
NEXT I
PAGE ON PU
```

This program fragment emits each field of LINE to the print unit identified by PU and then advances to a new page.
PRINTCSV

The PRINTCSV statement outputs CSV format data to a print unit.

Format

PRINTCSV {ON print.unit} var1, var2, ...

where

print.unit identifies the print unit to which output is to be directed. If omitted, print unit zero is used.

var1, var2, ... is a list of items to be assembled as a CSV format text string. If any of the variables contains field marks, each field is treated as a separate item in the resultant CSV data.

The assembled CSV format data is output to the requested print unit. Print unit -1 is always associated with the display and cannot be changed. Print unit 0 can be switched between the display and the printer by use of the PRINTER statement. Print units 1 to 255 direct their output to the hold file by default but can be redirected using the SETPTR command.

The optional trailing colon suppresses the normal linefeed after the data has been output.

Example

PRINTCSV PROD.NO, QTY

This statement prints the contents of the PROD.NO and QTY variables as a CSV format text string.

See also:
CSV.MODE, INPUTCSV, READCSV, WRITECSV
PRINT ON
PRINT OFF

The PRINT ON statement causes subsequent output to print unit zero to be directed to the printer. A later PRINT OFF statement resumes output to the display.

The STATUS() function returns the previous state of the PRINT setting. A value of zero indicates that the printer was on. A value of one indicates that it was off.

By using PRINT statements instead of DISPLAY in programs it is possible to select whether the output is directed to the display or to a printer. The LPTR option to the RUN command is equivalent to a PRINT ON at the start of the program followed by a PRINT CLOSE on return to the command prompt.

Example

PRINT ON
PRINT "This is sent to the printer"
PRINT OFF
PRINT "This is sent to the display"
PRINTER CLOSE

The **PRINTER CLOSE** statement closes one or all print units.

**Format**

```plaintext
PRINTER CLOSE \{ ON print.unit \}
```

where

- `print.unit` identifies the print unit to be closed. If omitted, all print units are closed.

The **PRINTER CLOSE** statement terminates activity on a print unit. If this print unit was directed to a spool file, the data will be printed. Any heading and footing text is discarded. Subsequent data sent to the same print unit starts a new output stream. If this is for the default printer, it will be necessary to use **PRINTER ON** if the output is to be directed to a printer rather than the screen.

The implementation of **PRINTER CLOSE** with no `print.unit` specified differs on various multivalue database products. By default, in QM a **PRINTER CLOSE** with no `print.unit` causes all print units to be closed. The same effect can be achieved by using a `print.unit` value of -2 though this is not portable to other environments. The PRCLOSE.DEFAULT.0 option of the **$MODE** compiler directive can be used to modify this behaviour so that only printer zero is closed.

All print units are closed automatically on return to the command prompt.

The **PRINTER** and **SETPTR** commands have a **KEEP.OPEN** option which, when used, causes requests from programs to close printers only to terminate the page and discard any heading and footing text. This printer remains open so that subsequent output to the same print unit will be merged to form a single print job. The **PRINTER CLOSE** command must be used to cancel the **KEEP.OPEN** mode.
PRINTED DISPLAY

The PRINTED DISPLAY statement directs output sent to a print unit to the display.

Format

```
PRINTED DISPLAY { ON print.unit } 
{ ON ERROR statement(s) } 
{ THEN statement(s) } 
{ ELSE statement(s) } 
```

where

- `print.unit` evaluates to the print unit on which the action is to be performed. If omitted, the default print unit (unit 0) is used.
- `statement(s)` are statements to be executed depending on the outcome of the operation.

The **ON ERROR**, **THEN** and **ELSE** clauses are all optional.

The **ON ERROR** clause is executed in the event of a fatal internal error. The error code returned by the `STATUS()` function will indicate the cause of the error. If this clause is omitted, the program will abort in the event of a fatal error.

The **THEN** clause is executed if the operation is successful. The `STATUS()` function will return zero.

The **ELSE** clause is executed in the event of a non-fatal error. If this clause is omitted, program execution continues after an error.

Example

```
PRINTED DISPLAY ON 1 
```

This statement directs output from print unit 1 to the display.
PRINTER FILE

The **PRINTER FILE** statement directs printer output to a named record in a directory file.

**Format**

```
PRINTER FILE { ON print.unit } file.name, record.name
{ ON ERROR statement(s) }
{ THEN statement(s) }
{ ELSE statement(s) }
```

where

- `print.unit` evaluates to the print unit on which the action is to be performed. If omitted, the default print unit (unit 0) is used.
- `file.name` evaluates to the VOC name of an existing directory file.
- `record.name` evaluates to the name of the record within `file.name` to which output to `print.unit` is to be directed. If the record already exists, it will be overwritten.
- `statement(s)` are statements to be executed depending on the outcome of the operation.

The **ON ERROR**, **THEN** and **ELSE** clauses are all optional.

Output to print units 1 to 255 is directed to a hold file by default but can be redirected. The **PRINTER FILE** statement causes the named record to be created and output will be directed to this file until the print unit is closed. Reopening the print unit without changing the destination will overwrite the same record.

The **ON ERROR** clause is executed in the event of a fatal internal error while attempting to open the file. The error code returned by the **STATUS()** function will indicate the cause of the error. If this clause is omitted, the program will abort in the event of a fatal error.

The **THEN** clause is executed if the operation is successful. The **STATUS()** function will return zero.

The **ELSE** clause is executed if the file cannot be opened. The error code returned by the **STATUS()** function will indicate the cause of the error. If this clause is omitted, program execution continues after an error.

**Example**

```
PRINTER FILE ON 1 "MYFILE", "SAVED"
```

This statement directs output from print unit 1 to record SAVED in directory file MYFILE.
PRINT NAME

The PRINT NAME statement associates a named printer device with a print unit.

Format

```
PRINT NAME {ON print.unit} printer.name
{ON ERROR statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

where

- `print.unit` evaluates to the print unit on which the action is to be performed. If omitted, the default print unit (unit 0) is used.
- `printer.name` evaluates to a printer name.
- `statement(s)` are statements to be executed depending on the outcome of the operation.

The ON ERROR, THEN and ELSE clauses are all optional.

The ON ERROR clause is executed in the event of a fatal internal error. The error code returned by the STATUS() function will indicate the cause of the error. If this clause is omitted, the program will abort in the event of a fatal error.

The THEN clause is executed if the operation is successful. The STATUS() function will return zero.

The ELSE clause is executed if the printer does not exist. If this clause is omitted, program execution continues after an error.

Example

```
PRINT NAME ON 1 "LPT1"
```

This statement directs output from print unit 1 to a printer named LPT1.
PRINTER RESET

The PRINTER RESET statement resets the default print unit and display output.

Format

    PRINTER RESET

The PRINTER RESET statement performs the following actions:

    Output to the default print unit (unit 0) is directed to the display (similar to use of PRINTER OFF)
    Pagination is restarted on the display if it was previously suppressed.
    The page number is reset to 1.
    Any heading and footing set up for the display device are cancelled.

The PRINTER RESET statement is particularly useful in programs which have disabled line counting through use of cursor movement @() functions and subsequently want to restart pagination of line by line output.
PRINTER SETTING

The PRINTER SETTING statement sets a control parameter for a print unit.

This statement is obsolete. The SETPU statement should be used in its place.

Format

PRINTER SETTING { ON print.unit } param, new.value

where

print.unit evaluates to the print unit on which the action is to be performed. If omitted, the default print unit (unit 0) is used.

param identifies the parameter to be changed.

new.value is the value to be set. A new.value of -1 sets the parameter to its default value.

The parameters which may be set by this statement are identified by param numbers. Tokens for these are defined in the KEYS.H include record in the SYSCOM file.

<table>
<thead>
<tr>
<th>Key</th>
<th>Token</th>
<th>Default</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LPTR$WIDTH</td>
<td>80</td>
<td>Page width</td>
</tr>
<tr>
<td>2</td>
<td>LPTR$LINES</td>
<td>66</td>
<td>Lines per page for printer</td>
</tr>
<tr>
<td></td>
<td>or 24</td>
<td></td>
<td>Lines per page for display</td>
</tr>
<tr>
<td>3</td>
<td>LPTR$TOP.MARGIN</td>
<td>0</td>
<td>Top margin size (lines)</td>
</tr>
<tr>
<td>4</td>
<td>LPTR$BOTTOM.MARGIN</td>
<td>0</td>
<td>Bottom margin size (lines)</td>
</tr>
<tr>
<td>5</td>
<td>LPTR$LEFT.MARGIN</td>
<td>0</td>
<td>Left margin size (characters)</td>
</tr>
<tr>
<td>11</td>
<td>LPTR$FLAGS</td>
<td></td>
<td>Printer mode flags</td>
</tr>
</tbody>
</table>

The value of lines per page is best set to at least one less than the physical page size to prevent the automatic page throw of most printers after the final line of the page is printed.

Example

PRINTER SETTING ON 1 LPTR$LINES 60

This statement sets the number of lines per page on print unit 1 to 60.
PRINTERT.SETTING()

The PRINTERT.SETTING() function sets or retrieves a control parameter for a print unit.

*This function is obsolete. The SETPU statement or GETPU() function should be used in its place.*

Format

PRINTERT.SETTING(print.unit, param, new.value)

where

- **print.unit** evaluates to the print unit on which the action is to be performed.
- **param** identifies the parameter to be changed using the keys shown below.
- **new.value** is the value to be set. A new.value of -1 sets the parameter to its default value. A new.value of -2 returns the current value without changing it.

The PRINTERT.SETTING() function returns the new (or unchanged) value of the parameter.

The parameters which may be set or retrieved by this statement are identified by param numbers. Tokens for these are defined in the KEYS.H include record in the SYSCOM file.

<table>
<thead>
<tr>
<th>Key</th>
<th>Token</th>
<th>Default</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>LPTR$WIDTH</td>
<td>80</td>
<td>Page width</td>
</tr>
<tr>
<td>2</td>
<td>LPTR$LINES</td>
<td>66</td>
<td>Lines per page for printer or 24 Lines per page for display</td>
</tr>
<tr>
<td>3</td>
<td>LPTR$TOP.MARGIN</td>
<td>0</td>
<td>Top margin size (lines)</td>
</tr>
<tr>
<td>4</td>
<td>LPTR$BOTTOM.MARGIN</td>
<td>0</td>
<td>Bottom margin size (lines)</td>
</tr>
<tr>
<td>5</td>
<td>LPTR$LEFT.MARGIN</td>
<td>0</td>
<td>Left margin size (characters)</td>
</tr>
<tr>
<td>6 *</td>
<td>LPTR$DATA.LINES</td>
<td></td>
<td>Lines excluding page heading and footing</td>
</tr>
<tr>
<td>7 *</td>
<td>LPTR$HEADING.LINES</td>
<td></td>
<td>Heading lines per page</td>
</tr>
<tr>
<td>8 *</td>
<td>LPTR$FOOTING.LINES</td>
<td></td>
<td>Footing lines per page</td>
</tr>
<tr>
<td>9 *</td>
<td>LPTR$MODE</td>
<td></td>
<td>Printer mode number</td>
</tr>
<tr>
<td>10 *</td>
<td>LPTR$NAME</td>
<td></td>
<td>Printer or file name</td>
</tr>
<tr>
<td>11</td>
<td>LPTR$FLAGS</td>
<td></td>
<td>Printer mode flags</td>
</tr>
<tr>
<td>12 *</td>
<td>LPTR$LINE.NO</td>
<td></td>
<td>Current position on page within data area</td>
</tr>
<tr>
<td>13 *</td>
<td>LPTR$PAGE.NO</td>
<td></td>
<td>Current page number</td>
</tr>
<tr>
<td>14 *</td>
<td>LPTR$LINES.LEFT</td>
<td></td>
<td>Lines remaining on current page</td>
</tr>
<tr>
<td>15</td>
<td>LPTR$COPIES</td>
<td>1</td>
<td>Number of copies to print</td>
</tr>
</tbody>
</table>

Modes marked with an asterisk are query only.

The value of lines per page is best set to at least one less than the physical page size to prevent the automatic page throw of most printers after the final line of the page is printed.
Example

\[
\text{WIDTH} = \text{PRINTER.SETTING}(1, \text{LPTR$\text{WIDTH}$, -1})
\]

This statement sets the page width on print unit 1 to the default value and stores this value in WIDTH.
PRINTERR

The PRINTERR statement displays an error message which is removed from the screen when the next input is entered.

The synonym INPUTERR can be used in place of PRINTERR.

Format

PRINTERR expr

where

expr evaluates to the text to be displayed.

The expr text is displayed on the bottom line of the screen using the current foreground and background colours. This message will be removed after the first keystroke of the next INPUT @ statement. Input taken from the DATA queue will also clear the message.

Example

LOOP
   DISPLAY "Enter password " :
   PROMPT ""
   INPUT @ (5,10) PASSWORD : HIDDEN
   WHILE PASSWORD # "SECRET"
      PRINTERR "Incorrect password"
   REPEAT

This program fragment reads a password from the keyboard. If it is entered incorrectly, a message is displayed and the input is repeated. Note use of the HIDDEN qualifier in the INPUT statement so that data entered at the keyboard is displayed as asterisks.
PRIVATE

The **PRIVATE** statement defines private variables in a local subroutine or in a class module.

**Format**

```
PUBLIC var {READONLY} { = value}, mat(rows, cols), ...
```

where

- **var** is a simple scalar variable.
- **mat(rows, cols)** is a dimensioned matrix name. The **rows** and **cols** values must be numeric constants.

The **PRIVATE** statement has two uses:

Immediately after the **LOCAL** statement defining a local function or subroutine. It identifies variables that have scope only within the local routine and are discarded on exit. If the routine calls itself recursively, each invocation has its own private variables. See the **LOCAL** statement for more details.

Used in a **CLASS** module, it defines variables that are private to the object but persist between successive executions of components of the class module. See the **CLASS** statement and **Object Oriented Programming** for more details. Private variables are initially unassigned unless an initialisation **value** is present. If used, **value** must be a numeric or string constant, a **CHAR()** or **ECHAR()** function with a numeric constant argument, or a constant @-variable name. An initialisation value may be provided for a **mat** declaration and is equivalent to use of the **MAT** statement to set all elements to the given **value**. The initialisation is applied prior to execution of the **CREATE.OBJECT** subroutine, if present.

See also:
- **Object oriented programming**, **CLASS**, **DISINHERIT**, **INHERIT**, **OBJECT()**, **OBJINFO()**, **PUBLIC**, **SHARED**
**PROCREAD**

The **PROCREAD** statement reads data from the PROC primary input buffer.

**Format**

```
PROCREAD var {THEN statement(s)} {ELSE statement(s)}
```

where

- `var` is the variable to receive the data.
- `statement(s)` are statements to be executed dependant on the outcome of the operation.

At least one of the **THEN** and **ELSE** clauses must be present.

If the current program was called directly or indirectly from a PROC, the **PROCREAD** statement copies the content of the PROC primary input buffer to the named variable and executes the **THEN** clause.

If the current program was not called from a PROC, the variable is set to a null string and the **ELSE** clause is executed.

**See also:**

- [VOC PQ-type records](#)
PROCWRITE

The **PROCWRITE** statement writes data to the PROC primary input buffer.

**Format**

```
PROCWRITE expr
```

where

```
expr
```

is the data to be written.

The data specified by *expr* is copied to the PROC primary input buffer. The input pointer is set to the start of the data.

**See also:**

[VOC PQ-type records](#)
PROGRAM

The **PROGRAM** statement introduces a program.

**Format**

```
PROGRAM name
```

where

```
name
```

is the name of the program.

QMBasic programs should commence with a **PROGRAM**, **SUBROUTINE**, **FUNCTION** or **CLASS** statement. If none of these is present, the compiler behaves as though a **PROGRAM** statement had been used with `name` as the name of the source record.

The **PROGRAM** statement must appear before any executable statements.

The name need not be related to the name of the source record though it is recommended that they are the same as this eases program maintenance. The name must comply with the QMBasic **name format rules**.

A program module may be entered by referencing it a **RUN** command, by executing a command name that corresponds to the name of the program in the system catalogue, or by use of the QMBasic **CALL** statement in another program.

**Example**

```
PROGRAM SUM
    TOTAL = 0
    LOOP
        DISPLAY TOTAL
        INPUT S
        WHILE LEN(S)
            IF NUM(S) THEN TOTAL += S
            ELSE DISPLAY @SYS.BELL :
        REPEAT
    END
```

This program reads numbers from the keyboard and displays a running total until a blank line is entered.
PROMPT

The **PROMPT** statement sets the character to be used as the prompt in **INPUT** statements.

**Format**

```
PROMPT expr
```

where

```
expr
```
evaluates to the character to be used.

The first character of `expr` is used as the prompt character. If `expr` is a null string, the prompt is suppressed.

The default input prompt is the question mark. Changes to the prompt character remain in effect until the program returns to the command prompt.

On ECS mode systems, attempting to set the prompt to a double width character will be ignored.

Use of **EXECUTE** to start a new command processing layer resets the prompt to a question mark but it will be restored to its previous value on return from the executed command.

**Example**

```
DISPLAY "Enter account number " :
PROMPT ""
INPUT ACCOUNT.NO
PROMPT "?"
```

This program fragment suppresses the prompt for the **INPUT** statement and then restores the default prompt character. In normal usage, a program would use the **PROMPT** statement once at the start of the program to set the prompt character to be used for the entire program.
PTERM()

The PTERM() function sets, clears or queries a terminal setting.

Format

PTERM(action, value)

where

action specifies the terminal setting to be processed.

value specifies the new value. A negative value returns the current setting.

The PTERM() function can be used to set, clear or query the following terminal settings:

1. PT$BREAK Interprets break character as break key? (Boolean)
2. PT$INVERT Inverts case of alphabetic characters on input? (Boolean)
3. PT$BRKCH Sets break character to be char(value). The value must be in the range 1 to 31. For any other value, the current setting is returned.
4. PT$PAGE.PAUSE Set/clear page pause (Boolean)
5. PT$MARK Set/clear translation of characters 28-30 on keyboard input to field, value and subvalue marks. (Boolean)
6. PT$BINARY Set/clear telnet binary mode (Boolean)
7. PT$ENCODING Sets or returns the terminal encoding. The value associated with this action identifies the encoding to be set. This may be

   1. PT$ENCODING.8BIT
   2. PT$ENCODING.UTF8

   A negative value returns the currently active encoding value without change. The encoding value is ignored for QMConsole sessions on Windows.
8. PT$ERASE Sets erase (backspace) character to be char(value). The value must be in the range 0 to 31 where 0 disables erase key recognition. For any other value, the unchanged current setting is returned as a character number.
9. PT$BREAK.CT Sets break inhibit counter to value, if non-negative. Always returns old value.
10. PT$CTRL.FILTER Filter control characters in INPUT and INPUT@ other than those that are defined as edit keys? (Boolean). This setting is maintained separately for each command processor level.

Action values marked as Boolean enable the feature if value is True and disable it if value is False. The new state is returned as the value of the function. A negative value returns the current state without changing it.
Note that setting the erase character to 127 (the Delete key) also requires that the kdch1 element of the relevant `terminfo` definition is removed or modified so that this key does not clash with the internal processing of this key as a forward delete.

See also:

- PTERM
PUBLIC

The PUBLIC statement defines public property variables, subroutines and functions in a class module.

Format

PUBLIC var {READONLY} { = value}, mat(rows, cols), ...

PUBLIC SUBROUTINE name{(arg1, arg2)} {VAR.ARGS}
...statements...
END

PUBLIC FUNCTION name{(arg1, arg2)} {VAR.ARGS}
...statements...
END

where

var is a simple scalar variable. The variable name may be followed by READONLY to indicate that external references to the variable may not update it.

mat(rows, cols) is a dimensioned matrix name. The rows and cols values must be numeric constants. The dimension values may be followed by READONLY to indicate that external references to the variable may not update it.

name(arg1, arg2) is the subroutine or function name and an optional list of arguments. See the CLASS statement for the maximum number of arguments allowed in this list. Specifying the final argument name as three periods (...) effectively extends the argument list to the maximum permissible length with unnamed scalar arguments that may be accessed using the ARG() function. Use of this syntax automatically implies the VAR.ARGS option which must not also be present.

Note that the equivalence of a function to a subroutine with a hidden first argument as found with the SUBROUTINE and FUNCTION statements does not apply to public subroutines and functions.

The first form of the PUBLIC statement defines persistent variables that may be visible from outside the object in which they appear. Public variables are initially unassigned unless an initialisation value is present. If used, value must be a numeric or string constant, a CHAR() or ECHAR() function with a numeric constant argument, or a constant @-variable name. An initialisation value may be provided for a mat declaration and is equivalent to use of the MAT statement to set all elements to the given value. The initialisation is applied prior to execution of the CREATE.OBJECT subroutine, if present.

The second and third forms of the PUBLIC statement define a subroutine or function that can be referenced from outside the object. The synonyms GET and SET may be used for PUBLIC FUNCTION and PUBLIC SUBROUTINE respectively.

Arguments to public subroutines and functions may reference a whole matrix by following the matrix name by its dimensions. The actual values given are ignored; the compiler simply counts them to determine whether the matrix has one or two dimensions. For example:
PUBLIC SUBROUTINE CALC(CLIENT, CLI.REC(1), TOTVAL)

In this example, the dimension value has been shown as 1 to emphasise that the actual value is irrelevant. The compiler uses this purely to determine that CLI.REC is a single dimensional matrix, possibly representing a database record read using \texttt{MATREAD}. The alternative syntax used with \texttt{SUBROUTINE} statements by prefixing the matrix name with \texttt{MAT} and using a \texttt{DIMENSION} statement to set dimensionality is not available for public subroutines and functions.

The number of arguments in calls to the subroutine or function must be the same as in the declaration unless the \texttt{VAR.ARGS} option is used in which case any arguments not passed by the caller will be unassigned. The \texttt{ARG.COUNT()} function can be used to determine the actual number of arguments passed, excluding the hidden return argument. The \texttt{ARG.PRESENT()} function can be used to test for the presence of an optional argument by name.

PUBLIC FUNCTION CREDIT.RATING(CLIENT, CLASS, CODE) VAR.ARGS

In this example, if the calling program supplies only one argument, the CLASS and CODE variables will be unassigned. If the calling program provides two arguments, the CODE variable will be unassigned.

When using \texttt{VAR.ARGS}, default values may be provided for any arguments by following the argument name with an \texttt{=} sign and the required numeric or string value or the \texttt{@FALSE} or \texttt{@TRUE} constants. For example,

\begin{verbatim}
PUBLIC FUNCTION CREDIT.RATING(CLIENT, CLASS = 1, CODE = "Standard") VAR.ARGS
\end{verbatim}

In this example, if the calling program supplies only one argument, the CLASS variable will default to 1 and the CODE variable will default to "Standard". If the calling program provides two arguments, the default value for CLASS is ignored and the CODE variable defaults to "Standard".

Examples

\begin{verbatim}
PUBLIC FUNCTION CONNECT(SERVER, PORT)   SKT = OPEN.SOCKET(SERVER, PORT, SKT$BLOCKING)   RETURN STATUS() = 0END
\end{verbatim}

The above function takes a fixed length list of two arguments and uses the supplied values to open a socket connection to a remote server. The SKT variable in this example would be a private variable within the class module.

\begin{verbatim}
PUBLIC FUNCTION CONNECT(SERVER, PORT) VAR.ARGS   IF UNASSIGNED(PORT) THEN PORT = 4000   SKT = OPEN.SOCKET(SERVER, PORT, SKT$BLOCKING)   RETURN STATUS() = 0END
\end{verbatim}

This example extends the previous one by making the PORT argument optional and, if it is not supplied by the caller, defaulting it to 4000.

\begin{verbatim}
PUBLIC SUBROUTINE INSERT.ITEMS(ID, ...)   READU REC FROM FVAR, ID ELSE NULL   FOR I = 2 TO ARG.COUNT()      VALUE = ARG(I)
\end{verbatim}
LOCATE VALUE IN REC<1> BY 'AL' SETTING POS ELSE
INS VALUE BEFORE REC<POS>
END
NEXT I
WRITE REC TO FVAR, ID
END

This example uses the ... syntax to specify a variable length argument list of the maximum permissible length. It reads a record identified by the ID argument and then inserts all items from the remaining arguments that are not already in the record.

See also:
Object oriented programming, CLASS, DISINHERIT, INHERIT, OBJECT(), PRIVATE, SHARED
PWR() function returns the value of a number raised to a given power.

Format

\[
PWR(expr, pwr.expr)
\]

where

- \( expr \) evaluates to a number or a numeric array.
- \( pwr.expr \) evaluates to a number or a numeric array.

The PWR() function returns the value of \( expr \) raised to the power \( pwr.expr \). It is equivalent to use of the \(^*\) operator.

If either \( expr \) or \( pwr.expr \) is a numeric array (a dynamic array where all elements are numeric), the PWR() function operates on each element in turn and returns another numeric array. The structure of this array will be the same as that of the \( expr \) and \( pwr.expr \) arrays if they are identical. For arrays of differing structure, the structure of the result depends on whether the \( \text{REUSE}() \) function is used.

Example

\[
N = \text{PWR}(T, 3)
\]

This statement finds the value of \( T \) cubed. For small integer values of \( pwr.expr \), use of the multiply operator is faster.
QUOTE()

The QUOTE() function returns a copy of its argument string enclosed in double quotes. The DQUOTE() synonym is identical.

The QUOTES() and DQUOTES() functions are similar but operate on successive elements of a dynamic array, returning a similarly structured dynamic array of results.

Format

```
QUOTE(expr)
```

where

```
expr       evaluates to the source string.
```

The QUOTE() function returns expr enclosed in double quotation marks.

Examples

```
A = QUOTE('ABC123')
```

This statement sets A to the eight character string "ABC123".

```
A = 'ABC' : @VM : 'DEF'
B = QUOTES(A)
```

This program fragment encloses each element of dynamic array A in double quotes, storing the result in B as

"ABC" VM "DEF"

See also:
SQUOTE()
The RAISE() function converts mark characters in a string to the next higher level mark.

**Format**

\[
\text{RAISE(}\text{string}\text{)}
\]

where

\[
\text{string} \quad \text{evaluates to the string in which mark characters are to be converted.}
\]

The RAISE() function replaces mark characters according to the following table:

<table>
<thead>
<tr>
<th>Original</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>Item mark</td>
<td>Item mark (unchanged)</td>
</tr>
<tr>
<td>Field mark</td>
<td>Item mark</td>
</tr>
<tr>
<td>Value mark</td>
<td>Field mark</td>
</tr>
<tr>
<td>Subvalue mark</td>
<td>Value mark</td>
</tr>
<tr>
<td>Text mark</td>
<td>Subvalue mark</td>
</tr>
</tbody>
</table>

**Migration Note**

Note that the behaviour in some other multivalue products may not be identical with regard to the effect of raising character 250 that precedes the text mark in the ASCII character set.

**Example**

```
FORMLIST RAISE(LIST)
```

This statement takes a value mark delimited variable LIST, raises the marks and uses this to create a select list.

**See also:**

LOWER()
**RANDOMIZE**

The `RANDOMIZE` statement initialises the random number generator.

**Format**

```
RANDOMIZE expr
```

where

`expr` evaluates to a number. If omitted or a null string, the time of day is used.

The `RANDOMIZE` statement initialises the seed value of the random number generator function, `RND()`. Supplying the same seed value in successive uses of this statement guarantees that the same pseudo-random number sequence is generated. Note that the sequence returned may vary between QM releases or on different platforms even if the same seed value is set.

If the `expr` value is omitted or given as a null string, the time of day is used thus giving a reasonable chance of a different pseudo-random sequence on successive executions of the program.

**See also:**

`RND()`
RDIV()

The RDIV() function returns the rounded integer result of dividing two values.

**Format**

```
RDIV(dividend, divisor)
```

where

- `dividend` evaluates to the value to be divided.
- `divisor` evaluates to the value by which `dividend` is to be divided.

The RDIV() function divides `dividend` by `divisor` and returns the result as an integer, rounded according to the rule that values with a fractional part of 0.5 or greater are rounded away from zero.

A zero value of `divisor` will cause a run time error.

**Examples**

<table>
<thead>
<tr>
<th>Dividend</th>
<th>Divisor</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>132</td>
<td>10</td>
<td>13</td>
</tr>
<tr>
<td>135</td>
<td>10</td>
<td>14</td>
</tr>
<tr>
<td>-135</td>
<td>10</td>
<td>-14</td>
</tr>
</tbody>
</table>

See also: [DIV()], [IDIV()]
The **READ** statement reads a record from a previously opened file.

**Format**

```
READ var [ENCODING name] FROM file.var, record.id [ON ERROR statement(s)]
{THEN statement(s)}
{ELSE statement(s)}
```

where

**var** is the name of a variable to receive the dynamic array read from the file.

**file.var** is the file variable associated with the file.

**record.id** evaluates to the id of the record to be read.

**statement(s)** are statements to be executed depending on the outcome of the **READ** operation.

At least one of the **THEN** and **ELSE** clauses must be present unless the OPTIONAL.THEN.ELSE setting of the **$MODE** compiler directive is enabled.

The specified record is read into the named variable.

When reading from a directory file, newlines in the data read from the file are replaced by field marks. This action may be suppressed by using the **MARK.MAPPING** statement after opening the file.

The optional **ENCODING** element to this statement sets the character encoding to be used, overriding any encoding set in the **VOC F-type** record or when the file was opened. This is relevant only to directory files and is ignored for other file types. A null string as the encoding name is equivalent to not having the **ENCODING** clause at all. To disable the default encoding, the encoding name should be specified as "NULL".

The **THEN** clause is executed if the **READ** is successful.

The **ELSE** clause is executed if the **READ** fails because no record with the given id is present on the file. If the PICK.READ mode of the **$MODE** directive is used, **var** will be left unchanged, otherwise it will be set to a null string for a normal file or an empty collection for a data collection file. The **STATUS()** function will indicate the cause of the error.

The **ON ERROR** clause is executed for serious fault conditions such as errors in a file's internal control structures. The **STATUS()** function will return an error number. If no **ON ERROR** clause is present, an abort would occur.

**Example**

```
READ ITEM FROM STOCK, ITEM.ID THEN
   ...processing statements...
END ELSE
   DISPLAY "Record " : ITEM.ID : " not found"
END
```
This program fragment reads a record from the a file previously opened to file variable STOCK into variable ITEM. If successful, the processing statements are executed. If the record is not found, a message is displayed.
READ.SOCKET()

The READ.SOCKET() function reads data from a socket.

Format

    READ.SOCKET(skt, max.len, flags, timeout)

where

    skt     is the socket variable returned by ACCEPT.SOCKET.CONNECTION() (stream connections), CREATE.SERVER.SOCKE(T) (datagram connections) or OPEN.SOCKET().
    max.len is the maximum number of bytes to read.
    flags   is a value determining the mode of operation of the socket for this read, formed by adding the values of tokens defined in the SYSCOM KEYS.H record. The flags available in this release are:
               SKT$BLOCKING  Sets the mode of data transfer as blocking.
               SKT$NON.BLOCKING Sets the mode of data transfer as non-blocking.
    timeout is the timeout period in milliseconds. A value of zero implies no timeout.

The READ.SOCKET() function returns data read from the specified socket. The STATUS() function returns zero if the action is successful, or a non-zero error code if an error occurs. A timeout will return an error code of ER$TIMEOUT as defined in the SYSCOM ERR.H record.

Example

    SRVR.SKT = CREATE.SERVER.SOCKE(T("", 0)
    IF STATUS() THEN STOP 'Cannot initialise server socket'
    SKT = ACCEPT.SOCKET.CONNECTION(SRVR.SKT, 0)
    IF STATUS() THEN STOP 'Error accepting connection'
    DATA = READ.SOCKET(SKT, 100, SKT$BLOCKING, 0)
    CLOSE.SOCKE(T SKT
    CLOSE.SOCKE(T SRVR.SKT

This program fragment creates a server socket, waits for an incoming connection, reads a single data packet of up to 100 bytes from this connection and then closes the sockets. The timeout value of 0 in the READ.SOCKET() call specifies that no timeout is to be used.

See also:
Using Socket Connections, ACCEPT.SOCKET.CONNECTION, CLOSE.SOCKE(T, CREATE.SERVER.SOCKE(T), OPEN.SOCKE(T), SELECT.SOCKE(T), SERVER.ADDR(), SET.SOCKE(T.MODE), SOCKET.INFO(), WRITE.SOCKE(T)
READBLK

The READBLK statement reads a given number of bytes from the current file position in a record previously opened using OPENSEQ.

Format

```
READBLK var [ENCODING name] FROM file.var, bytes
{ON ERROR statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

where

- `var` is the name of a variable to receive the data read from the file.
- `file.var` is the file variable associated with the file.
- `bytes` evaluates to the number of bytes to be read.
- `statement(s)` are statements to be executed depending on the outcome of the READBLK operation.

At least one of the THEN and ELSE clauses must be present unless the OPTIONAL.THEN.ELSE setting of the $MODE compiler directive is enabled.

The READBLK statement reads up to `bytes` bytes from the file. If at least one byte is available, the data is returned in `var` and the THEN clause is executed.

The ELSE clause is executed if the READBLK fails. The STATUS() function will indicate the cause of the error. For example, attempting to read when at the end of the file will return ERR$EOF.

The ON ERROR clause is executed for serious fault conditions such as errors in a file's internal control structures. The STATUS() function will return an error number. If no ON ERROR clause is present, an abort would occur.

The optional ENCODING clause sets the character encoding to be used, overriding any encoding set in the VOC F-type record or when the file was opened. A null string as the encoding name is equivalent to not having the ENCODING clause at all. To disable the default encoding, the encoding name should be specified as "NULL".

Note that READBLK is a byte level operation and returns a string in which each character will represent one byte from the file and hence is always in the range char(0) to char(255). When used with the ENCODING clause, the bytes value is the number of bytes to be read, not the character count in the decoded data.

If `file.var` refers to a serial port opened using OPENSEQ, the READBLK statement reads up to `bytes` bytes of data from the port but does not wait if there is less than the requested number of bytes available.

Example

```
READBLK VAR FROM SEQ.F, 100 THEN
...processing statements...
```
END ELSE
   DISPLAY "Data block not read"
END

This program fragment reads 100 bytes from the a file previously opened to file variable SEQ.F into variable VAR.

See also:
CLOSESEQ, CREATE, NOBUF, OPENSEQ, READCSV, READSEQ, SEEK, TIMEOUT, WEOFSEQ, WRITEBLK, WRITECSV, WRITESEQ, WRITESEQF
READCSV

The **READCSV** statement reads a CSV format line of text from a directory file record previously opened for sequential access and parses it into multiple variables.

**Format**

```
READCSV [ENCODING name] FROM file.var [DELIMITER delim] TO var1, var2,...
{THEN statement(s)}
{ELSE statement(s)}
```

where

- **encoding** is the ECS character encoding to be used for the data.
- **file.var** is the file variable associated with the record by a previous **OPENSEQ** statement.
- **delim** is the delimiter separating data elements. This defaults to a comma if omitted. Only the first character of **delim** is significant.
- **var1, var2 ...** are the variables to receive the data read from the file.
- **statement(s)** are statement(s) to be executed depending on the outcome of the **READCSV**.

At least one of the **THEN** and **ELSE** clauses must be present.

The optional **ENCODING** element to this statement sets the character encoding to be used, overriding any encoding set in the **VOC F-type** record or when the file was opened.

A line of text is read from the file. It is then parsed according to the CSV format rules as defined in RFC 4180, placing the elements into the data items identified by **var1**, **var2**, etc. If successful, the **THEN** clause is executed and the **STATUS()** function would return zero.

If there are fewer data items in the line of text than the number of variables supplied, the remaining variables will be set to null strings. If the line of text has more data items than the number of variables supplied, the excess data is ignored. See **CSVDQ()** for a way to parse the CSV string into a dynamic array.

If there are no further fields to be read, the **ELSE** clause is executed and the **STATUS()** function would return ER$RNF (record not found). The target variables will be unchanged.

The CSV rules are described under the **WRITECSV** statement.

**Example**

```
LOOP
   READCSV FROM DELIVERY.F TO PROD.NO, QTY ELSE EXIT
   GOSUB PROCESS.DELIVERY.DETAILS
REPEAT
```

This program fragment reads CSV format lines of text from the record open for sequential access via the DELIVERY.F file variable, placing the elements of the line into PROD.NO and QTY. It then
calls the PROCESS.DELIVERY.DETAILS subroutine to process the new item. The loop terminates when the **ELSE** clause is executed when all fields have been processed.

See also: [CLOSESEQ](#), [CREATE](#), [CSVDQ()](#), [DPARSE.CSV](#), [NOBUF](#), [OPENSEQ](#), [READBLK](#), [READSEQ](#), [SEEK](#), [WEOFSEQ](#), [WRITEBLK](#), [WRITECSV](#), [WRITESEQ](#), [WRITESEQF](#)
READL

The READL statement reads a record from a previously opened file, setting a read lock.

Format

```
READL var [ENCODING name] FROM file.var, record.id {ON ERROR statement(s)}
{LOCKED statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

where

- `var` is the name of a variable to receive the dynamic array read from the file.
- `file.var` is the file variable associated with the file.
- `record.id` evaluates to the id of the record to be read.
- `statement(s)` are statements to be executed depending on the outcome of the READL operation.

At least one of the THEN and ELSE clauses must be present unless the OPTIONAL.THEN.ELSE setting of the $MODE compiler directive is enabled.

The specified record is read into the named variable and a read lock is set. See Locks for full details of QM's locking mechanism.

The optional ENCODING element to this statement sets the character encoding to be used, overriding any encoding set in the VOC F-type record or when the file was opened. This is relevant only to directory files and is ignored for other file types. A null string as the encoding name is equivalent to not having the ENCODING clause at all. To disable the default encoding, the encoding name should be specified as "NULL".

The THEN clause is executed if the READL is successful. The LOCKED clause is executed if the file or record is exclusively locked by another process. The STATUS() function will return the userid of a process holding a lock that is blocking this action or zero if the lock table is full. If the LOCKED clause is omitted and the file or record is locked, the program will wait for the lock to be released.

The ELSE clause is executed if the READL fails because no record with the given id is present on the file. If the PICK.READ mode of the $MODE directive is used var will be left unchanged, otherwise it will be set to a null string for a normal file or an empty collection for a data collection file. The STATUS() function will indicate the cause of the error.

The ON ERROR clause is executed for serious fault conditions such as errors in a file's internal control structures. The STATUS() function will return an error number. If no ON ERROR clause is present, an abort would occur.

Example

```
READL ITEM FROM STOCK, ITEM.ID THEN
   ...processing statements...
   WRITE ITEM TO STOCK, ITEM.ID
```
This program fragment reads a record from the a file previously opened to file variable. STOCK into variable ITEM, setting a read lock on the record. If successful, the processing statements are executed. If the record is not found, a message is displayed and the lock is released.
**READLIST**

The **READLIST** statement reads a select list into a dynamic array.

**Format**

```
READLIST var {FROM list.no}
{THEN statement(s)}
{ELSE statement(s)}
```

where

- **var** is the variable to receive the select list.
- **list.no** is the select list number. If omitted, select list zero is used. A Pick style select list variable may be used instead of a list number.
- **statement(s)** are statement(s) to be performed depending on the outcome of the **READLIST** operation.

The **THEN** and **ELSE** clauses are both optional. If neither is given, the program can recognise failure by **var** being a null string.

The specified select list is read into **var**. If the list had already been partially processed by **READNEXT** statements, only the remaining unprocessed items are stored in **var**.

The select list is empty after the **READLIST** statement is completed.

The **THEN** clause is executed if **var** contains one or more items. Items are separated by field marks. If compatibility with other software is required, it is suggested that programs should be written to accept either field marks or item marks (or a mix) as list separators.

The **ELSE** clause is executed if the select list was not active or if no items remained to be processed. In this case **var** will be set to a null string.

**Exploded Select Lists**

QM supports two styles of select list; a standard list and an exploded list.

A standard select list contains only simple data, usually record ids. An exploded select list is created using the **BY.EXP** or **BY.EXP.DSND** keywords of the query processor to break apart multivalues and subvalues in a field. Each entry in the select list contains four values corresponding to the record id, the value, subvalue and field positions corresponding to the data element associated with the list entry. The **READLIST** statement will return this composite data.

**Example**

```
READLIST S FROM 2 THEN
    WRITE S TO LISTS, "UNPROCESSED"
END
```

This program fragment retrieves the remaining items in select list 2 and, if there are any, writes them to a record UNPROCESSED in file LISTS.
See also:
FORMLIST
READNEXT

The **READNEXT** statement returns the next item from an active select list.

**Format**

```plaintext
READNEXT var {, val.pos {, subval.pos} } {FROM list.no}
{ON ERROR statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

where

- **var** is the variable to receive the select list item.
- **val.pos** is the variable to receive the value position with an exploded select list.
- **subval.pos** is the variable to receive the subvalue position with an exploded select list.
- **list.no** is the select list number. If omitted, the default select list is used. The **READNEXT** statement can also use select list variables returned by the **SELECTV** statement or the RTNLIST option of the **EXECUTE** statement.
- **statement(s)** are statement(s) to be executed depending on the outcome of the **READNEXT** operation.

At least one of the **THEN** and **ELSE** clauses must be present.

The next item in the specified select list is removed from the list and stored in **var**. Although the list may be of any size, a single item extracted by **READNEXT** cannot exceed 32k bytes. Attempting to extract an item larger than this limit will be handled as a fatal error as described below.

The **ON ERROR** clause is executed if a fatal error occurs. The **STATUS()** function will return a value relating to the error. If no **ON ERROR** clause is present, fatal errors result in an abort.

The **THEN** clause is executed if the select list was active and not empty.

The **ELSE** clause is executed if the select list was not active or no items remained to be read. The **var** variable will be set to a null string unless the **PICK.READNEXT** setting of the **$MODE** compiler directive is active, in which case the variable is left unchanged.

The **STATUS()** function will return zero unless the **ON ERROR** clause is executed.

**Migration Note**

Some other multi-value products leave **var** unchanged if the **ELSE** clause is taken. Use of the **PICK.READNEXT** setting of the **$MODE** compiler directive gives this behaviour.

**Exploded Select Lists**

QM supports two styles of select list; a standard list and an exploded list.
A standard select list contains only simple data, usually record ids. The optional val.pos and subval.pos items are always returned as zero with this type of list.

An exploded select list is created using the BY.EXP or BY.EXP.DSND keywords of the query processor to break apart multivalues and subvalues in a field. Each entry contains the record id together with the value and subvalue position corresponding to the data element associated with the list entry.

The optional val.pos and subval.pos components of the READNEXT statement can be used to retrieve this positional data. There are three possible formats:

- If both are present, the value and subvalue positions are returned in these variables. Where the value was not subdivided into subvalues, the subvalue position is returned as zero.
- If only val.pos is present, the value position is returned and any subvalue information is discarded.
- If neither is present, normally only the record id is returned, however, if the program is compiled with the COMPOSITE.READNEXT option of the $MODE compiler directive in force, the data returned in var is made up from the record id, the value position and the subvalue position separated by value marks.

Example

SELECT STOCK.FILE
LOOP
  READNEXT ID ELSE EXIT
  PRINT ID
REPEAT

This program fragment produces a list of the record keys present in STOCK.FILE.
READSEQ

The READSEQ statement reads the next line of text (field) from a directory file record previously opened for sequential access.

Format

```
READSEQ var {ENCODING name} FROM file.var {ON ERROR statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

where

- `var` is the variable to receive the data read from the file.
- `file.var` is the file variable associated with the record by a previous OPENSEQ statement.
- `statement(s)` are statement(s) to be executed depending on the outcome of the READSEQ.

At least one of the THEN and ELSE clauses must be present.

The optional ENCODING clause sets the character encoding to be used, overriding any encoding set in the VOC F-type record or when the file was opened. A null string as the encoding name is equivalent to not having the ENCODING clause at all. To disable the default encoding, the encoding name should be specified as "NULL".

The next line of text is read into var. If successful, the THEN clause is executed and the STATUS() function would return zero.

If there are no further fields to be read, the ELSE clause is executed and the STATUS() function would return ER$RNF (record not found).

If a fatal error occurs, the ON ERROR clause is executed. The STATUS() function can be used to establish the cause of the error. If no ON ERROR clause is present, a fatal error causes an abort.

The FILEINFO() function can be used with key FL$LINE to determine the line number that will be read by the next READSEQ.

Example

```
LOOP
   READSEQ REC FROM STOCK.LIST ELSE EXIT
   GOSUB PROCESS.STOCK.ITEM
REPEAT
```

This program fragment reads fields from the record open for sequential access via the STOCK.LIST file variable and calls the PROCESS.STOCK.ITEM subroutine for each field. The loop terminates when the ELSE clause is executed when all fields have been processed.

See also:
CLOSESEQ, CREATE, NOBUF, OPENSEQ, READBLK, READCSV, SEEK, TIMEOUT, WEOFSEQ, WRITEBLK, WRITECSV, WRITESEQ, WRITESEQF
READU

The READU statement reads a record from a previously opened file, setting an update lock.

Format

```
READU var {ENCODING name} FROM file.var, record.id {ON ERROR statement(s)}
{LOCKED statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

where

- **var** is the name of a variable to receive the dynamic array read from the file.
- **file.var** is the file variable associated with the file.
- **record.id** evaluates to the id of the record to be read.
- **statement(s)** are statements to be executed depending on the outcome of the READU operation.

At least one of the **THEN** and **ELSE** clauses must be present unless the OPTIONAL.THEN.ELSE setting of the $MODE compiler directive is enabled.

The optional **ENCODING** element to this statement sets the character encoding to be used, overriding any encoding set in the VOC E-type record or when the file was opened. This is relevant only to directory files and is ignored for other file types. A null string as the encoding name is equivalent to not having the **ENCODING** clause at all. To disable the default encoding, the encoding name should be specified as "NULL".

The **THEN** clause is executed if the READU is successful. The specified record is read into the named variable and an update lock is set. See **Locks** for full details of QM's locking mechanism.

The **LOCKED** clause is executed if the file or record is locked by another process. The **STATUS** function will return the user id of a process holding a lock that is blocking this action or zero if the lock table is full. If the **LOCKED** clause is omitted and the file or record is locked, the program will wait for the lock to be released.

The **ELSE** clause is executed if the READU fails because no record with the given id is present on the file. If the PICK.READ mode of the $MODE directive is used **var** will be left unchanged, otherwise it will be set to a null string for a normal file or an empty collection for a data collection file. The **STATUS** function will indicate the cause of the error.

The **ON ERROR** clause is executed for serious fault conditions such as errors in a file's internal control structures. The **STATUS** function will return an error number. If no **ON ERROR** clause is present, an abort would occur.

Example

```
READU ITEM FROM STOCK, ITEM.ID LOCKED
   DISPLAY "Record is locked by user " : STATUS()
END THEN
   ...processing statements...
```
WRITE ITEM TO STOCK, ITEM.ID
END ELSE
   DISPLAY "Record " : ITEM.ID : " not found"
   RELEASE STOCK, ITEM.ID
END

This program fragment reads a record from the a file previously opened to file variable. STOCK into variable ITEM, setting an update lock on the record. If successful, the processing statements are executed. If the record is not found, a message is displayed and the record is unlocked. The locked clause displays an error message if the record is locked by another user.
READV

The **READV** statement reads a specific field from a record of a previously opened file.

**Format**

```
READV var [ENCODING name] FROM file.var, record.id, field.expr
  {ON ERROR statement(s) }
  {THEN statement(s) }
  {ELSE statement(s) }
```

where

- `var` is the name of a variable to receive the dynamic array read from the file.
- `file.var` is the file variable associated with the file.
- `record.id` evaluates to the id of the record to be read.
- `field.expr` evaluates to the number of the field to be read.
- `statement(s)` are statements to be executed depending on the outcome of the **READV** operation.

At least one of the **THEN** and **ELSE** clauses must be present unless the **OPTIONAL. THEN. ELSE** setting of the **$MODE** compiler directive is enabled.

The optional **ENCODING** element to this statement sets the character encoding to be used, overriding any encoding set in the **VOC F-type** record or when the file was opened. This is relevant only to directory files and is ignored for other file types. A null string as the encoding name is equivalent to not having the **ENCODING** clause at all. To disable the default encoding, the encoding name should be specified as "NULL".

The specified record is read and the field identified by `field.expr` is extracted into the named variable. If the field does not exist, `var` is set to a null string.

Note that internally, the file system operates at the record level so **READV** reads the entire record, extracts the field and discards the rest. If multiple fields are to be extracted, it is always better to read the whole record and use field extraction operations instead of using multiple **READV** statements.

A `field.expr` value of zero may be used to determine if the record exists. If it does, the **THEN** clause is taken and `var` is set to the record id. If the record does not exist, the **ELSE** clause is taken and `var` will be set to a null string. The record is not transferred into memory, resulting in a significant performance advantage over use of **READ** when the record is very large.

The **THEN** clause is executed if the record is read successfully regardless of whether the specified field is present.

The **ELSE** clause is executed if the **READV** fails because no record with the given id is present on the file. If the **PICK.READ** mode of the **$MODE** directive is used `var` will be left unchanged, otherwise it will be set to a null string. The **STATUS()** function will indicate the cause of the error.
The **ON ERROR** clause is executed for serious fault conditions such as errors in a file's internal control structures. The **STATUS()** function will return an error number. If no **ON ERROR** clause is present, an abort would occur.

When used with a data collection file, the `field.expr` must evaluate to zero as records in this file type are not dynamic arrays.

**Example**

```qmbasic
READV ITEM FROM STOCK, ITEM.ID, 3 THEN
    ...processing statements...
END ELSE
    DISPLAY "Record " : ITEM.ID : " not found"
END
```

This program fragment reads field 3 of a record from the file previously opened to file variable STOCK into variable ITEM. If successful, the processing statements are executed. If the record is not found, a message is displayed.

**See also:**

[WRITEV](#)
READVL, READVU

The READVL statement reads a specific field from a record of a previously opened file, setting a read lock. The READVU statement is similar but sets an update lock.

Format

```
READVL var {ENCODING name} FROM file.var, record.id, field.expr
{ON ERROR statement(s)}
{LOCKED statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

where

- `var` is the name of a variable to receive the dynamic array read from the file.
- `file.var` is the file variable associated with the file.
- `record.id` evaluates to the id of the record to be read.
- `field.expr` evaluates to the number of the field to be read.
- `statement(s)` are statements to be executed depending on the outcome of the operation.

At least one of the THEN and ELSE clauses must be present unless the OPTIONAL.THEN.ELSE setting of the $MODE compiler directive is enabled.

The optional ENCODING element to this statement sets the character encoding to be used, overriding any encoding set in the VOC F-type record or when the file was opened. This is relevant only to directory files and is ignored for other file types. A null string as the encoding name is equivalent to not having the ENCODING clause at all. To disable the default encoding, the encoding name should be specified as "NULL".

The specified record is read and the field identified by `field.expr` is extracted into the named variable. If the field does not exist, `var` is set to a null string.

READVL sets a read lock is set on the record. READVU sets an update lock on the record. See Locks for full details of QM's locking mechanism.

A `field.expr` value of zero may be used to determine if the record exists. `var` will be set to a null string.

The LOCKED clause is executed if the file or record is locked by another process (exclusively in the case of READVL). The STATUS() function will return the user id of a process holding a lock that is blocking this action or zero if the lock table is full. If the LOCKED clause is omitted and the file or record is locked, the program will wait for the lock to be released.

The THEN clause is executed if the record is read successfully regardless of whether the specified field is present.
The **ELSE** clause is executed if the operation fails because no record with the given id is present on the file. If the PICK.READ mode of the `$MODE` directive is used `var` will be left unchanged, otherwise it will be set to a null string. The `STATUS()` function will indicate the cause of the error.

The **ON ERROR** clause is executed for serious fault conditions such as errors in a file's internal control structures. The `STATUS()` function will return an error number. If no **ON ERROR** clause is present, an abort would occur.

**Example**

```qmbasic
READVU ITEM FROM STOCK, ITEM.ID, 3 THEN
  ...processing statements...
  WRITEV ITEM TO STOCK, ITEM.ID, 3
END ELSE
  DISPLAY "Record " : ITEM.ID : " not found"
  RELEASE STOCK, ITEM.ID
END
```

This program fragment reads field 3 of a record from the file previously opened to file variable STOCK into variable ITEM, setting an update lock. If successful, the processing statements are executed and the modified value is written back to the file. If the record is not found, a message is displayed and the record is unlocked.
RECORDLOCKED()

The RECORDLOCKED() function indicates whether a given record is locked.

Format

\[
\text{RECORDLOCKED}(\text{file.var}, \text{record.id})
\]

where

\[
\begin{align*}
\text{file.var} & \quad \text{is the file variable associated with the file.} \\
\text{record.id} & \quad \text{evaluates to the key of the record to be tested.}
\end{align*}
\]

The RECORDLOCKED() function returns a value indicating the state of any locks on record \text{record.id} of the file open as \text{file.var}. The tokens shown in the table below are defined in the KEYS.H record of the SYSCOM file.

<table>
<thead>
<tr>
<th>Value</th>
<th>Token</th>
<th>Lock state</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td>LOCKSOTHER.FILELOCK</td>
<td>Another user holds a file lock</td>
</tr>
<tr>
<td>-2</td>
<td>LOCKSOTHER.READU</td>
<td>Another user holds an update lock</td>
</tr>
<tr>
<td>-1</td>
<td>LOCKSOTHER.READL</td>
<td>Another user holds a read lock</td>
</tr>
<tr>
<td>0</td>
<td>LOCKSNO.LOCK</td>
<td>The record is not locked</td>
</tr>
<tr>
<td>1</td>
<td>LOCKSMY.READL</td>
<td>This user holds a read lock</td>
</tr>
<tr>
<td>2</td>
<td>LOCKSMY.READU</td>
<td>This user holds an update lock</td>
</tr>
<tr>
<td>3</td>
<td>LOCKSMY.FILELOCK</td>
<td>This user holds a file lock</td>
</tr>
</tbody>
</table>

A record may be multiply locked in which case the RECORDLOCKED() function reports only one of the current locks. File locks take precedence over read or update locks. If no file lock is set, read or update locks held by the process in which the RECORDLOCKED() function is performed take precedence over locks held by other processes.

Executing the STATUS() function after a RECORDLOCKED() function indicates that a lock is active will return the user number of the user holding the lock.

Example

\[
\begin{align*}
\text{IF RECORDLOCKED(STOCK, } \text{"ORDER.LIST"}) \text{ THEN} \\
\quad \text{DISPLAY } \text{"Record is locked by user " : STATUS()}
\end{align*}
\]

This program fragment checks if record ORDER.LIST is locked and, if so, reports the user number of the process that holds the lock.
RECORDLOCKL, RECORDLOCKU

The `RECORDLOCKL` statement sets a read lock on a record. The `RECORDLOCKU` statement is similar but sets an update lock.

Format

```
RECORDLOCKL file.var, record.id {ON ERROR statement(s)}
{LOCKED statement(s)}
```

where

- `file.var` is the file variable associated with the file.
- `record.id` evaluates to the key of the record to be locked.
- `statement(s)` are statements to be executed depending on the outcome of the operation.

The `RECORDLOCKL` statement sets a read lock on record `record.id` of the file open as `file.var`. The `RECORDLOCKU` statement sets an update lock.

The `LOCKED` clause is executed if the file or record is locked by another process in a manner than prevents further locking. The `STATUS()` function will return the user id of a process holding a lock that is blocking this action or zero if the lock table is full. If the `LOCKED` clause is omitted and the file or record is locked, the program will wait for the lock to be released.

A process may lock records within files for which it also holds the file lock. These statements may also be used to convert an existing read lock to an update lock or vice versa.

Example

```
RECORDLOCKL STOCK, "ORDER.LIST" LOCKED
   DISPLAY "Waiting. Order list locked by user " : STATUS()
RECORDLOCKL STOCK, "ORDER.LIST"
END
```

This program fragment attempts to lock record `ORDER.LIST` of the file open as `STOCK`. If it is locked, a message is displayed and a second `RECORDLOCKL` statement is executed without a `LOCKED` clause to wait for the lock.
The **REGEX()** function tests whether a string matches a regular expression. This function is not available on Windows systems.

**Format**

```
REGEX(str, regex[, modes])
```

where

- `str` is the string to match against the regular expression.
- `regex` is the regular expression.
- `modes` are additive flags controlling how the regular expression is tested. At this release there is just one option:
  - `REG$NOCASE` Case insensitive.

The **REGEX()** function provides limited support for regular expressions. It tests whether `str` matches regular expression `regex`, returning True if it does and False if it does not. The function also returns False if `regex` is incorrectly formed.

On ECS mode systems, the **REGEX()** function only works correctly for strings formed only from characters with codepoint values U+0000 to U+00FF.

For details of regular expressions, see one of the online tutorials.

**See also:**

- Pattern matching
RELEASE

The RELEASE statement releases read, update or file locks.

Format

    RELEASE {file.var[, record.id]} {ON ERROR statement(s)}

where

    file.var       is the file variable associated with the file.
    record.id     evaluates to the key of the record to be unlocked.
    statement(s)  are statements to be executed depending on the outcome of the operation.

The RELEASE statement operates in three ways according to whether file.var and record.id are specified.

    With no file.var or record.id, all file, read and update locks owned by the process on all files are released.
    With file.var but no record.id, all locks associated with file.var are released.
    With both record.id and file.var, a specific lock is released.

The ON ERROR clause is executed if a fatal error occurs. The STATUS() function can be used to obtain an error code to determine the cause.

The RELEASE statement has no effect inside a transaction.

Examples

    RELEASE STOCK, "ORDER.LIST"

This statement releases any locks on record ORDER.LIST of the file open as STOCK.

    RELEASE

This statement releases all file, read and update locks held by the user.
The \textbf{REM()} function returns the remainder when one value is divided by another.

\textbf{Format}

\textbf{REM}(\textit{dividend}, \textit{divisor})

where

\textit{dividend} evaluates to a number or a numeric array.

\textit{divisor} evaluates to a number or a numeric array.

The \textbf{REM()} function returns the remainder of dividing \textit{dividend} by \textit{divisor}. This is defined as

\[
\text{REM}(x, y) = \text{SIGN}(x) \times \text{MOD}(|x|, |y|)
\]

where the \text{SIGN()} function returns 1 for \(x > 0\), -1 for \(x < 0\) and 0 for \(x = 0\). (\text{SIGN()} is not part of QMBasic. It is used here only to explain the action of the \textbf{REM()} function).

The \textbf{REM()} function differs from the \textbf{MOD()} function when one of its arguments is negative. The following table shows the result of the \textbf{REM()} function.

<table>
<thead>
<tr>
<th>Dividend</th>
<th>Divisor</th>
<th>REM()</th>
</tr>
</thead>
<tbody>
<tr>
<td>530</td>
<td>100</td>
<td>30</td>
</tr>
<tr>
<td>-530</td>
<td>100</td>
<td>-30</td>
</tr>
<tr>
<td>530</td>
<td>-100</td>
<td>30</td>
</tr>
<tr>
<td>-530</td>
<td>-100</td>
<td>-30</td>
</tr>
<tr>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>-100</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>-100</td>
<td>0</td>
<td>-100</td>
</tr>
</tbody>
</table>

If either \textit{dividend} or \textit{divisor} is a numeric array (a dynamic array where all elements are numeric), the \textbf{REM()} function operates on each element in turn and returns another numeric array. The structure of this array will be the same as that of the \textit{dividend} and \textit{divisor} arrays if they are identical. For arrays of differing structure, the structure of the result depends on whether the \textbf{REUSE()} function is used.

\textbf{Example}

\[
N = \text{REM}(T, 30)
\]

This statement finds the remainder of dividing \(T\) by 30 and assigns this to \(N\).

\textbf{See also:} \textbf{MOD()}, \textbf{ROUNDDOWN()}, \textbf{ROUNDUP()}
**REMARK**

The **REMARK** statement, which may be abbreviated to **REM**, enters comment text into a program.

**Format**

```
REMARK text
```

where

`text` is arbitrary comment text.

The **REMARK** statement inserts `text` as a comment in the program which is totally ignored by the compiler. The semicolon delimiter cannot be used to include an executable statement on the same line as a **REMARK** as the entire line after the **REMARK** keyword is ignored.

Comments are more usually inserted using the an asterisk or exclamation mark prefix.

**Examples**

```
REMARK This text is totally ignored
* This text is totally ignored
! This text is totally ignored
A = B + C   ;* This is a trailing comment
```
REMOVE, REMOVE(), REVREMOVE

The **REMOVE** statement and **REMOVE()** function extract characters from a dynamic array up to the next mark character. The **REVREMOVE** statement extracts items in reverse order.

**Format**

```plaintext
REMOVE string FROM dyn.array SETTING var

REMOVE(dyn.array, var)

REVREMOVE string FROM dyn.array SETTING var
```

where

- `string` is the variable to receive the extracted substring.
- `dyn.array` is the dynamic array from which `string` is to be extracted.
- `var` is the variable to be set according to the delimiter that terminates the extracted substring.

The statement

```plaintext
S = REMOVE(X, Y)
```

is equivalent to

```plaintext
REMOVE S FROM X SETTING Y
```

The **REMOVE** operation associates a **remove pointer** with the `dyn.array` from which data is extracted. Whenever a string is assigned to a variable the remove pointer is set to point one character before the start of the string. Subsequent **REMOVE** operations advance the pointer by one character and then extract characters from the position of the remove pointer up to the next mark character or the end of the string. Because the remove pointer gives immediate access to the position at which the **REMOVE** should commence, this operation can be much faster than field, value or subvalue extraction.

The value returned in `var` indicates the delimiter that terminated the **REMOVE**. The delimiter character is not stored as part of the extracted substring. Values of `var` are:

- 0  End of string
- 1  Item mark
- 2  Field mark
- 3  Value mark
- 4  Subvalue mark
- 5  Text mark

The mark character itself can be reconstructed as `CHAR(256 - var)` for a non-zero value of `var`.

Once the end of the `dyn.array` has been reached, the remove pointer remains positioned one character beyond the end of the string and further **REMOVE** operations would return a null string.

The **RMVD()** function is similar to **REMOVE()** but recognises only the field, value and subvalue marks as delimiters.
The remove pointer may be reset to the start of the string by assigning a new value to string. Where it is required to reset the remove pointer without changing the string, the SETREM statement can be used. Alternatively, applications frequently use a statement such as

\[ S = S \]

which will assign S to itself thus resetting the remove pointer.

The REVREMOVE statement is similar to REMOVE but extracts items in reverse order. It will be necessary to use SETREM to position the remove pointer prior to first use of REVREMOVE. The value returned in var will identify the mark character that precedes the extracted text, zero after extracting the item at the start of dyn.array.

Note that the REMOVE operations perform a type conversion on dyn.array if it is not already a string. Thus the program

\[
S = 99 \\
\text{REMOVE X FROM S SETTING DELIM}
\]

would convert S to be a string "99". Although this is unlikely to have any undesirable effects, it is a side effect to be aware of.

Examples

\[
\text{LOOP} \\
\quad \text{REMOVE BOOK.NO FROM BOOK.LIST SETTING DELIM} \\
\quad \text{PRINT "Book number is " : BOOK.NO} \\
\text{WHILE DELIM} \\
\text{REPEAT}
\]

This program fragment extracts entries from the BOOK.LIST dynamic array and prints them. There is an assumption that BOOK.LIST is not a null string (in which case a single null BOOK.NO would be printed).

\[
S = "" \\
\text{LOOP} \\
\quad \text{REMOVE FLD FROM REC SETTING DELIM} \\
\quad S := FLD \\
\quad \text{IF DELIM = 2 OR DELIM = 0 THEN} \\
\quad \quad \text{PRINT S} \\
\quad \quad S = "" \\
\quad \text{END ELSE} \\
\quad \quad S := \text{CHAR}(256 - \text{DELIM}) \\
\quad \text{END} \\
\text{WHILE DELIM} \\
\text{REPEAT}
\]

This program prints fields from REC. Note the use of the ELSE clause to append the delimiter that terminated the substring if it was not a field mark or the end of the string.

This is equivalent to

\[
N = \text{DCOUNT(REC, @FM)} \\
\text{FOR I = 1 TO N} \\
\quad \text{PRINT REC< I>} \\
\text{NEXT I}
\]
but may be much faster where REC is large and has a very large number of fields.

See also:
GETREM(), REMOVEF(), RMVD(), SETREM
REMOVE.BREAK.HANDLER

The REMOVE.BREAK.HANDLER statement allows a program to remove a handler subroutine established using SET.BREAK.HANDLER.

Format

   REMOVE.BREAK.HANDLER

A break handler subroutine can be established using SET.BREAK.HANDLER and will be called if the break key is enabled and the user presses it.

The handler applies to the program that establishes it and to all programs called from it unless they establish their own break handler.

The break handler will be deactivated when the program that established it terminates. The program can also use REMOVE.BREAK.HANDLER to deactivate the handler while the program continues to run. A program cannot remove the break handler of another program further down the call stack. On deactivation of a break handler, any handler belonging to a program further down the call stack becomes active again.

See also:
Interrupting commands, SET.BREAK.HANDLER
The **REMOVEF()** function extracts data from a delimited character string.

**Format**

```plaintext
REMOVEF(string[, delimiter[, count]])
```

where

- **string** is the string from which data is to be extracted.
- **delimiter** is the delimiter character that separates elements of **string**. If omitted, a field mark is used. If **delimiter** is more than one character, only the first character is used. If **delimiter** is a null string, **count** characters are extracted.
- **count** is the number of consecutive delimited elements of **string** to be extracted. If omitted or less than one, this defaults to one.

The **REMOVEF()** function uses the same optimised method as the **REMOVE()** function to extract items from **string** sequentially but uses a specified delimiter character instead of terminating on any mark character.

Whenever a string is assigned to a variable the remove pointer is set to point one character before the start of the string. Subsequent uses of **REMOVEF()** advance the point by one character and then extract characters from the position of the remove pointer up to the next delimiter character or the end of the string. Because the remove pointer gives immediate access to the position at which the **REMOVEF()** should commence, this operation requires no searching and is therefore very fast.

Once the end of the **string** has been reached, the remove pointer remains positioned one character beyond the end of the string and further **REMOVEF()** operations would return a null string.

The **REMOVEF()** function uses the **STATUS()** function to return information about its outcome:

- **0** Successful
- **1** Null **string**
- **2** End of **string**

The remove pointer may be reset to the start of the string by assigning a new value to **string**. Where it is required to reset the remove pointer without changing the string, the **SETREM** statement can be used. Alternatively, applications frequently use a statement such as

```plaintext
S = S
```

which will assign **S** to itself thus resetting the remove pointer.

**Examples**

```plaintext
LOOP
    REF = REMOVEF(REF.LIST, ',')
UNTIL STATUS()
    PRINT "Reference number is " : REF
REPEAT
```

This program fragment prints successive elements from the comma delimited **REF.LIST** variable.
See also:
GETREMO, REMOVE, SETREM
REPLACE()

The `REPLACE()` function replaces a field, value or subvalue of a dynamic array, returning the result.

**Format**

```
REPLACE(dyn.array, field {, value {, subvalue} }, string)
```

where

- `dyn.array` evaluates to a string in which the replacement is to occur.
- `field` evaluates to the field position number. If zero, this argument defaults to one.
- `value` evaluates to the value position number. If omitted or zero, the entire field is replaced.
- `subvalue` evaluates to the subvalue position number. If omitted or zero, the entire value is replaced.
- `string` evaluates to the replacement data.

If `field`, `value` and `subvalue` are not all present, the comma before the `string` argument must be replaced by a semicolon.

The statement

```
S = REPLACE(S, F, V, SV, NEW)
```

is equivalent to

```
S<F, V, SV> = NEW
```

If the specified `field`, `value` or `subvalue` is not present in the `dyn.array`, mark characters are added and the new item is inserted.

A negative value of `field`, `value` or `subvalue` causes a new field, value or subvalue to be appended. The lower ranking items are taken as being one. For example,

```
S = REPLACE(X, -1, 2, 3, Z)
```

appends a new field. The `value` and `subvalue` arguments are treated as though the statement were

```
S = REPLACE(X, -1, 1, 1, Z)
```

See the description of the `S<f,v,sv>` assignment operator for a discussion of how QM appends items, in particular with regard to the COMPATIBLE.APPEND option of the `$MODE` compiler directive.

**Example**

```
S = REPLACE(REC, 3, 1; ITEM)
```

This statement assigns `S` with the result of replacing field 3, value 1 of `REC` by the contents of `ITEM`. The value of `REC` is not changed.
See also:
DEL, DELETE(), EXTRACT(), FIND, FINDSTR, INS, INSERT(), LISTINDEX(), LOCATE, LOCATE()
RESTORE.SCREEN

The `RESTORE.SCREEN` statement restores a rectangular portion of the display screen image previously saved using `SAVE.SCREEN()`.

This statement can only be used with QMConsole and QMTerm sessions and with terminals that support the save and restore screen region functions (e.g. AccuTerm 5.2b upwards with terminal definitions that have the -at suffix).

Format

```
RESTORE.SCREEN image, restore.state
```

where

- `image` is the screen image data to be restored.
- `restore.state` is a Boolean value indicating whether the cursor position, pagination mode and current display attributes are to be restored from the saved data.

The `RESTORE.SCREEN` statement restores the data previously saved in `image` using `SAVE.SCREEN()`. The data cannot be restored to a different screen position from which it was saved. If the `restore.state` expression evaluates to a non-zero value, the pagination mode will also be restored.

Example

```
IMAGE = SAVE.SCREEN(0, 0, 80, 25)
EXECUTE COMMAND.STRING
RESTORE.SCREEN IMAGE, @TRUE
```

The above code fragment saves the screen image, executes the command in variable COMMAND.STRING and then restores the screen image.

See also:

`SAVE.SCREEN`
RETURN

The RETURN statement returns from an internal subroutine entered by GOSUB or an external subroutine entered by CALL.

Format

\[
\text{RETURN} \{expr\} \\
\text{RETURN TO} \ label\\ \\
\text{RETURN FROM PROGRAM}
\]

where

\[
\begin{align*}
\text{label} & \quad \text{is a label in the same program or subroutine as the RETURN statement.} \\
\text{expr} & \quad \text{is the value to be returned from a user written function.}
\end{align*}
\]

The RETURN statement returns from the most recent GOSUB or CALL statement. Thus the same RETURN statement could leave either an internal or catalogued subroutine.

The RETURN expr form of the RETURN statement is only valid in a FUNCTION and returns expr as the result of the function. If the function returns to its caller using a simple RETURN statement with no expr, a null string is returned.

The optional TO label clause causes return from a GOSUB to continue execution at the given label rather than at the statement following the GOSUB. This clause is ignored when returning from a CALL. Excessive use of RETURN TO can lead to programs that are extremely difficult to maintain.

Sometimes a subroutine needs to return to the calling routine but it is not known how many internal subroutines may be active (e.g. in error paths). The standard way to achieve this in all multivalue database products is with a statement of the form

\[
\text{ERROR.LABEL: } \text{RETURN TO ERROR.LABEL}
\]

This slightly strange construct will cause all internal subroutines to return to the RETURN statement and then return to the calling program. This is different from STOP which would also terminate the current sentence.

QMBasic provides a rather more readable (and marginally faster) method by using

\[
\text{RETURN FROM PROGRAM}
\]

Examples

\[
\begin{align*}
\text{SUBROUTINE PRINT.REPORT(ID)} \\
\ldots\text{statements} \ldots \\
\text{RETURN} \\
\text{END}
\end{align*}
\]

This skeleton subroutine performs its task and then returns to its caller.

\[
\begin{align*}
\text{FUNCTION MATMAX(MAT A)} \\
\text{MAX} = \text{A}(1)
\end{align*}
\]
N = INMAT(A)
FOR I = 1 TO N
    IF A(I) > MAX THEN MAX = A(I)
NEXT I

RETURN MAX
END

This function scans a one dimensional matrix and passes back the value of the largest element.

See also:
CALL, GOSUB
REUSE()

The **REUSE()** function determines how arithmetic operators applied to numeric arrays handle unequal numbers of fields, values or subvalues. It can also be used with arguments to the **multivalue functions**.

**Format**

```
REUSE(dyn.array)
```

where

`dyn.array` is a dynamic array.

Arithmetic operators such as addition applied to numeric arrays (dynamic arrays where each element is numeric) operate on each field, value or subvalue in turn. Where the layout of fields, values and subvalues in the two numeric arrays is identical there is no difficulty, each element of one array being added (etc) to its corresponding element from the second array.

If the arrays are of different structure, such as one having more fields than the other or more values in one field than the corresponding field of the other array, the arithmetic operators normally use a default value for the missing item. This value is zero except for the divisor of a division operation which defaults to one.

For dynamic arrays used in the **multivalue functions**, missing items default to a null string.

The **REUSE()** function causes the previous field, value or subvalue to be reused in place of this default value where array structures do not match. The **REUSE()** function applies only to values in expressions; its effect cannot be assigned to a variable but it can be used to qualify an argument in a subroutine or function call.

**Examples**

```
A = "1" : @FM : "2" : @FM : "3"
B = "10" : @FM : "20"
C = A + B
D = A + REUSE(B)
```

In this example, C is set to "11\[FM\]22\[FM\]3" and D to "11\[FM\]22\[FM\]23". The **REUSE()** function causes the final field of B to be reused in the addition with field 3 of A.

```
A = "1" : @FM : "2" : @FM : "3"
C = A + 10
D = A + REUSE(10)
```

This example is similar except that numeric array B has been replaced by a simple numeric constant which can be considered to be a single element numeric array. In this case, C is set to "11\[FM\]2\[FM\]3" and D to "11\[FM\]12\[FM\]13".

```
A = "1":@FM:"2":@VM:"3":@VM:"4":@FM:"5":@VM:"6":@VM:"7":@FM:"8"
B = "10":@FM:"20":@FM:"30":@VM:"40"
```
C = A + B
D = A + REUSE (B)

In this example individual fields and values of A and B are matched into pairs for the addition operations.
C is set to "11FM22VM3VM4FM35VM36VM7FM8".
D is set to "11FM22VM23VM24FM35VM36VM37FM48".

See also:
Multivalue functions, ANDS(), EQS(), GES(), GTS(), IFS(), LES(), LTS(), NES(), NOTS(), ORS()
REUSING()

The REUSING() function returns the state of the REUSE flag on a variable.

Format

    REUSING(var)

where

    var

    is the variable to be examined.

Operators such as addition applied to dynamic arrays operate on each field, value or subvalue in turn. If the arrays are of different structure, such as one having more fields than the other or more values in one field than the corresponding field of the other array, the operators normally use a default value for the missing item.

The REUSE() function causes the previous field, value or subvalue to be reused in place of this default value where array structures do not match. The REUSE() function applies only to values in expressions; its effect cannot be assigned to a variable but it can be used to qualify an argument in a subroutine or function call.

The REUSING() function allows a subroutine to determine whether an argument has the reuse flag set but does not clear the flag. This can easily be achieved by copying the variable, including overwriting itself.

See also:

    REUSE()
RIGHT()

The RIGHT() function returns the trailing part of a string.

Format

```
RIGHT(string, len)
```

where

- `string` is the string from which the leading substring is to be extracted.
- `len` is the length of the substring to be extracted. A value less than zero is replaced by zero.

The RIGHT() function returns the rightmost `length` characters from the given `string`. If `length` is greater than the number of characters in the string, the entire string is returned.

This function is identical in effect to use of the rightmost substring assignment operator. Thus the following two statements are equivalent:

```
SUBSTR = RIGHT(VAR, length)
SUBSTR = VAR[length]
```

Example

```
A = "ABCDEFGHJK"
S = RIGHT(A, 3)
```

This program fragment assigns the string "IJK" to variable S.

See also:

LEFT()
The **RMVD()** function extracts characters from a dynamic array up to the next delimiter character. It is closely related to **REMOVE().**

**Format**

\[ \text{RMVD}(\text{dyn.array}, \text{var}) \]

where

- \textit{dyn.array} is the dynamic array from which \textit{string} is to be extracted.
- \textit{var} is the variable to be set according to the delimiter that terminates the extracted substring.

The **RMVD()** function is similar to the **REMOVE()** function but recognises only the field, value and subvalue marks as delimiters. Text marks and item marks are treated as part of the data. The function associates a remove pointer with the \textit{dyn.array} from which data is extracted. Whenever a string is assigned to a variable the remove pointer is set to point one character before the start of the string. Subsequent **RMVD()** operations advance the pointer by one character and then extract characters from the position of the remove pointer up to the next delimiter character or the end of the string. Because the remove pointer gives immediate access to the position at which the data extraction should commence, this operation can be much faster than field, value or subvalue extraction.

The value returned in \textit{var} indicates the delimiter that terminated the removal. The delimiter character is not stored as part of the extracted substring. Values of \textit{var} are

<table>
<thead>
<tr>
<th>\textit{var}</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>End of string</td>
</tr>
<tr>
<td>2</td>
<td>Field mark</td>
</tr>
<tr>
<td>3</td>
<td>Value mark</td>
</tr>
<tr>
<td>4</td>
<td>Subvalue mark</td>
</tr>
</tbody>
</table>

The mark character itself can be reconstructed as \text{CHAR}(256 - \textit{var}) for a non-zero value of \textit{var}.

Once the end of the \textit{dyn.array} has been reached, the remove pointer remains positioned one character beyond the end of the string and further **RMVD()** operations would return a null string.

The remove pointer may be reset to the start of the string by assigning a new value to \textit{string}. Where it is required to reset the remove pointer without changing the string, the **SETREM** statement can be used. Alternatively, applications frequently use a statement such as

\[ S = S \]

which will assign \textit{S} to itself thus resetting the remove pointer.

Note that the **RMVD()** function performs a type conversion on \textit{dyn.array} if it is not already a string. Thus the program

\[ S = 99 \]
\[ X = \text{RMVD}(S, \text{DELIM}) \]

would convert \textit{S} to be a string "99". Although this is unlikely to have any undesirable effects, it is a side effect to be aware of.
Examples

```qbasic
LOOP
    BOOK.NO = RMVD(BOOK.LIST, DELIM)
    PRINT "Book number is " : BOOK.NO
    WHILE DELIM
        REPEAT
```

This program fragment extracts entries from the BOOK.LIST dynamic array and prints them. There is an assumption that BOOK.LIST is not a null string (in which case a single null BOOK.NO would be printed).

```qbasic
S = ""
LOOP
    FLD = RMVD(REC, DELIM)
    S := FLD
    IF DELIM = 2 OR DELIM = 0 THEN
        PRINT S
        S = ""
    END ELSE
    S := CHAR(256 - DELIM)
    END
    WHILE DELIM
    REPEAT
```

This program prints fields from REC. Note the use of the ELSE clause to append the delimiter that terminated the substring if it was not a field mark or the end of the string.

This is equivalent to

```qbasic
N = DCOUNT(REC, @FM)
FOR I = 1 TO N
    PRINT REC<I>
NEXT I
```

but may be much faster where REC is large and has a very large number of fields.

See also:
GETREM(), REMOVE, REMOVEF(), SETREM
RND()

The RND() function returns a random number.

Format

\[
\text{RND}(\text{expr})
\]

where

\[
\text{expr} \quad \text{evaluates to an integer or a numeric array.}
\]

The RND() function returns a random number. The range of values is determined by the value of \( expr \) rounded towards zero as an integer. If \( expr \) is positive, the number is in the range zero to \( expr \) minus one. If \( expr \) is negative, the number is in the range \( expr \) plus one to zero. If \( expr \) is zero, RND() returns zero.

If \( expr \) is a numeric array (a dynamic array where all elements are numeric), the RND() function operates on each element in turn and returns a numeric array with the same structure as \( expr \).

The seed value of the random number generator may be set using RANDOMIZE. Note that the sequence returned may vary between QM releases even if the same seed value is set.

Example

\[
\text{TWO.DICE} = \text{RND}(6) + \text{RND}(6) + 2
\]

This statement produces a value in TWO.DICE equivalent to throwing a pair of dice. The two calls to the RND() function will each return a value in the range 0 to 5. The values are then brought into the appropriate range by adding two.

See also:

RANDOMIZE
ROUNDDOWN(), ROUNDUP()

The ROUNDDOWN() function returns an integer value rounded towards zero in a given increment. The ROUNDUP() function returns an integer value rounded away from zero in a given increment.

Format

ROUNDDOWN(value, increment)

ROUNDUP(value, increment)

where

value is a numeric value to be rounded.

increment is the integer rounding increment.

The ROUNDDOWN() function returns the integer value rounded towards zero in steps of increment. It is equivalent to use of

IDIV(value, increment) * increment

The ROUNDUP() function returns the integer value rounded away from zero in steps of increment. For a positive value it is equivalent to use of

IDIV(value + increment - 1, increment) * increment

or, for a negative value

IDIV(value - increment + 1, increment) * increment

If increment is less than one, the function returns zero.

Example

FOR I = -10 TO 10
   DISPLAY I, ROUNDDOWN(I, 3), ROUNDUP(I,3)
NEXT I

The above loop displays:

-10 -9 -12
-9 -9 -9
-8 -6 -9
-7 -6 -9
-6 -6 -6
-5 -3 -6
-4 -3 -6
-3 -3 -3
-2 0 -3
-1 0 -3
0 0 0
1 0 3
2 0 3
3 3 3
4 3 6
5 3 6
6 6 6
7 6 9
8 6 9
See also:

CEIL(), FLOOR(), MOD(), REM()
**SADD(), SDIV(), SMUL(), SSUB()**

The `SADD()`, `SDIV()`, `SMUL()` and `SSUB()` functions perform arithmetic on numeric strings of any length.

**Format**

```
SADD(a, b)
SDIV(a, b [, r{, dp}])
SMUL(a, b)
SSUB(a, b)
```

where

- `a`, `b` are numeric strings.
- `r` is the variable to receive the remainder as a numeric string.
- `dp` is the number of decimal places for the calculated result.

These four functions perform addition, division, multiplication and subtraction respectively. The values passed in as arguments `a` and `b` must be strings that contain only digits with an optional leading minus sign and/or decimal point. The behaviour of these functions with a malformed numeric string is undefined.

The returned value will be a numeric string. Trailing zeroes after the decimal point are removed.

The `SDIV()` function has two optional arguments. The third argument, `r`, is used to return the remainder as a numeric string. The fourth argument, `dp`, specifies the precision as the maximum number of decimal places to appear in the result, defaulting to zero if omitted. If `dp` is to be specified but the remainder value is not required, the third argument can specified as a meaningless constant value such as zero or a null string. The `SDIV()` function truncates excess decimal places without rounding.

**Examples**

```
DISPLAY SADD('12345678901234567890', '999988888777766666')
DISPLAY SSUB('12345678901234567890', '987654321')
DISPLAY SMUL('12345678901234567890', '999988888777766666')
DISPLAY SMUL('123.456', '17.81')
```

The above four display statements produce:

```
1123455677890123345561234567890024691356912345665183676390936419777915817559547402198.75136
```

```
DISPLAY SDIV('12345678901234567890', '987654321', R) : ',
Remainder ' : R
DISPLAY SDIV('335', '113', R, 6)
```

The above two statements produce
12499999887, Remainder 339506163
3.141592

See also:
SCMP()
SAVE.SCREEN()

The SAVE.SCREEN() function saves a rectangular portion of the display screen image.

This statement can only be used with QMConsole and QMTerm sessions and with terminals that support the save and restore screen region functions (e.g. AccuTerm 5.2b upwards with terminal definitions that have the -at suffix).

**Format**

$$\text{SAVE.SCREEN}(\text{col}, \text{line}, \text{width}, \text{height})$$

where

- \(\text{col}\) is the screen column (from zero) of the leftmost column to be saved.
- \(\text{line}\) is the screen line (from zero) of the top line to be saved.
- \(\text{width}\) is the width of the screen region to be saved.
- \(\text{height}\) is the height of the screen region to be saved.

The SAVE.SCREEN() function saves the data and display attributes of the screen image within the specified screen area. The value assigned to the variable set by this function can only be used by the RESTORE.SCREEN statement.

**Example**

```plaintext
IMAGE = SAVE.SCREEN(0, 0, 80, 25)
EXECUTE COMMAND.STRING
RESTORE.SCREEN IMAGE, @TRUE
```

The above code fragment saves the screen image, executes the command in variable COMMAND.STRING and then restores the screen image.

See also:

RESTORE.SCREEN
SAVELIST

The SAVELIST statement saves an active select list to the $SAVEDLISTS file.

Format

```
SAVELIST name [FROM list]
{THEN statement(s)}
{ELSE statement(s)}
```

where

- `name` is the name of the $SAVEDLISTS entry to be written.
- `list` is the select list number or select list variable to be saved. If omitted, the default list (list zero) is saved.
- `statement(s)` are statements to be executed depending on the outcome of the operation.

The SAVELIST statement saves an active select in the $SAVEDLISTS file. The original list is destroyed by this operation.

If the list had been partially processed before the SAVELIST statement is performed, only the unprocessed portion of the list is saved.

At least one of the THEN and ELSE clauses must be present. If the list is successfully saved, the THEN clause is executed. If the list cannot be saved for any reason, the ELSE clause is executed.

See also:
GETLIST, WRITELIST
SCAN

The SCAN statement scans the elements of a data collection.

Format

SCAN collection RESET
SCAN collection {FROM start} SETTING name {, value} {THEN statement(s)} {ELSE statement(s)}

where

collection is a data collection variable.
start is the name of the collection element at which the scan is to begin.
name is the variable that will be set to the name of the next element found by the scan.
value is the variable that will be set to the value of the element.

The first form of the SCAN statement resets the scan position to be at the first element of the collection in collating sequence order.

The second form of the SCAN statement returns the next collection element. The optional FROM clause specifies the position of the first item to be returned. If there is no exact match, the scan position commences from where the start element would be in collating sequence order. Scanning a data collection does not descend into lower levels of nested collections.

At least one of the THEN and ELSE clauses must be present. The THEN clause will be executed if the scan is successful and name will be set to the name of the element. The optional value variable will be set to the value of that element. The ELSE clause will be executed if there are no further items to be extracted. The name and value variables will be returned as null strings.

Example

SCAN COL RESET
LOOP
SCAN COL SETTING NAME, VALUE ELSE EXIT
DISPLAY 'Element ', NAME , ' has value ', VALUE
REPEAT

This program fragment prints the name and value of each element of the data collection.

See also:
Data collections
**SCMP()**

The **SCMP()** function performs comparison of two integer numeric strings of any length.

**Format**

\[ \text{SCMP}(a, b) \]

where

\[ a, b \text{ are numeric strings.} \]

The values passed in as arguments \( a \) and \( b \) must be strings that contain only digits. A leading minus sign is allowed.

The returned value will be

\[ -1 \text{ a is less than } b \]
\[ 0 \text{ a is equal to } b \]
\[ 1 \text{ a is greater than } b \]

**See also:**

[SADD0], [SDIV0], [SMUL0], [SSUB0]
SEEK

The **SEEK** statement sets the current read / write position in a directory file record previously opened for sequential access.

**Format**

```
SEEK file.var \{, offset[, relto ]\} \{THEN statement(s)\} \{ELSE statement(s)\}
```

where

- **file.var** is the file variable associated with the record by a previous **OPENSEQ** statement.
- **offset** is the byte position relative to the point given by **relto**. If **offset** and **relto** are both omitted, the file position is set to the start of the file.
- **relto** indicates the point from which **offset** is calculated. Defaults to 0 if omitted.  
  
  - 0  Start of file (offset must be zero or positive).  
  - 1  Current position (offset may be any value)  
  - 2  End of file (offset must be zero or negative)

- **statement(s)** are statement(s) to be executed depending on the outcome of the **SEEK**.

At least one of the **THEN** and **ELSE** clauses must be present. The **THEN** clause is executed if the operation is successful. The **ELSE** clause is executed if the **SEEK** operation fails. The **STATUS()** function can be used to determine the cause of the error.

**Example**

```
SEEK SEQ.F, 0, 2 ELSE ABORT "Seek error"
```

This statement positions to the end of the record ready to append new data.

**See also:**

SELECT, SELECTN, SELECTV

The **SELECT** statement creates a select list containing all record keys from a file.

**Format**

```qmb
SELECT var { TO list.no } { ON ERROR statement(s) }
SELECTN var { TO list.no } { ON ERROR statement(s) }
SELECTV var { TO list.var } { ON ERROR statement(s) }
```

where

- **var** is the file variable associated with an open file or a field mark delimited dynamic array of items to form the list.
- **list.no** is the select list number of the list to be created. If omitted, select list zero is used.
- **list.var** is the select list variable to receive the list. Select list variables can be processed by the `READNEXT` statement as an alternative to using numbered select lists. If the `TO` clause is omitted, a default select list variable is used.
- **statement(s)** are statement(s) to be executed if a fatal error occurs.

The **SELECT** and **SELECTN** statements construct a list of record keys in the file open as **var** and store this as an active select list **list.no** replacing any previously active list. If there are no records in the file, an empty list is created.

The **SELECTV** statement constructs the list in the same way but stores it in a select list variable which can be processed by a subsequent use of `READNEXT`. If the `TO` clause is omitted, the default select list (numbered list 0) is used.

For compatibility with other database products, the action of the **SELECT** statement can be changed to produce a select list variable in the same was as **SELECTV**. This is achieved by setting the **SELECTV** option of the `$MODE` compiler directive.

Also for compatibility with other products, use of the **PICK.SELECT** option of the `$MODE` compiler directive causes **SELECTV** (and **SELECT** when `$MODE SELECTV` is in effect) to behave such that, if the default select list (list 0) is active when the statement is executed, the default list is transferred into **list.var**, ignoring **var** completely. For example

```qmb
EXECUTE "SELECT STOCK WITH DESCR LIKE 'Pen...'
SELECT FVAR TO STOCK.LIST
```

would transfer the list generated by the executed **SELECT** into variable **STOCK.LIST**, ignoring **FVAR**. The **SELECTE** statement provides a neater way to do this.

See [Select Lists](#) for a discussion of how QMBasic optimises select performance.

It is important that a program that does not completely process a select list should clear the remainder of the list because, while the list is active, split and merge operations are suspended on the data file. Thus, leaving a list active may cause the file performance to degrade if updates are made. Note that the file will automatically reconfigure for optimum performance once the select operation has terminated. When using numbered select lists, the partially processed list can be cleared using
CLEARSELECT. For select list variables generated with SELECTV, the program should overwrite the list variable with, for example, a null string.

The @SELECTED variable is set to the number of records selected for a directory file or the number of records in the first non-empty group for a dynamic file.

The optional ON ERROR clause is executed in the event of a fatal error. This covers such situations as disk hardware errors and faults in the internal structure of the file. The STATUS() function will return a value relating to the cause of the error. If no ON ERROR clause is present, a fatal error will result in an abort.

Except where the ON ERROR clause is taken, the STATUS() function will return zero.

Use of a Dynamic Array instead of a File Variable

For compatibility with Pick style environments, QM also supports a variation on these statements where the var is a dynamic array in which each field becomes an entry in the target select list.

Example

SELECT STAFF TO 7

This statement creates a list of the records on the file with file variable STAFF and saves it as active select list 7.
SELECT.SOCKET()

The SELECT.SOCKET() function monitors multiple sockets for events.

Format

SELECT.SOCKET(skt.array, timeout)

where

skt.array is a dimensioned array of socket variables. Any element that is not a reference to an open socket is ignored.

timeout is the time to wait in milliseconds before returning to the program if no events occur. A value of zero causes an immediate return. A negative value waits indefinitely.

The SELECT.SOCKET() enables a developer to write a program that can serve multiple socket connections, pausing execution until an event occurs on any socket.

Use the SKT$INFO.EVENTS key to the SET.SOCKET.MODE() function to set the events to be monitored (read, write, exception) on each socket in the array. Then use SELECT.SOCKET() to wait for an event to occur.

If an event is detected, the function returns a dynamic array with a field to correspond to each element in skt.array. The content of each field is a combination of uppercase letters R for a read event, W for a write event and E for an exception event. Only events that have been selected for monitoring will appear. Fields for which no event has occurred or where the corresponding element of skt.array is not a socket variable will be blank. Alternatively, the SOCKET.INFO() function can be used with key value SKT$INFO.EVENTS against each element of skt.array to check for the socket status.

If the function returns as a result of a timeout or an error, the returned value is a null string and the STATUS() function can be used to determine the cause. A timeout will return ER$TIMEOUT. All other values are errors.

The events that can be monitored are

1 SKT$READ(EVENT Notify availability of a new connection on a server socket Notify availability of data that can be read with READ.SOCKET() Notify closure of connection

2 SKT$WRITE.EVENT Notify when data can be sent with WRITE.SOCKET()

4 SKT$EXCEPTION.EVENT Notify exceptions

Example

The example below shows the general principles of using socket event notification, monitoring multiple incoming connections on port 9000. A real program would need to include additional error handling.

$include keys.h
dim skt(10)

skt(1) = create.server.socket('', 9000, 0)
if status() then stop 'Cannot initialise server socket'

if not(set.socket.mode(skt(1), SKT$INFO.EVENTS, SKT$READ.EVENT)) then
  stop 'Error from set.socket.mode()'
end

loop
  events = select.socket(skt, 5000)
  for i = 1 to 10
    if events<i> then ;* Must be a read event
      if i = 1 then ;* New connection
        * Find an unused table entry
        for j = 2 to 10
          if not(socket.info(skt(j), SKT$INFO.OPEN)) then
            skt(j) = accept.socket.connection(skt(1), 0)
          end if
        end for
        if not(set.socket.mode(skt(j), SKT$INFO.EVENTS, SKT$READ.EVENT)) then
          stop 'Error from set.socket.mode()'
        end if
        exit
      end if
    end if
  end for
  str = read.socket(skt(i), 1000, 0, 0)
  if status() then ;* Socket closed
    skt(i) = 0
  end if
  ...process input...
end

repeat

See also:
Using Socket Connections, ACCEPT.SOCKET.CONNECTION, CLOSE.SOCKET, CREATE.SERVER.SOCKEET(), OPEN.SOCKEET(), READ.SOCKEET(), SERVER.ADDR(), SET.SOCKEET.MODE(), WRITE.SOCKEET()
**SELECTE**

The `SELECTE` statement transfers select list 0 to a select list variable.

**Format**

```
SELECTE TO list.var
```

where

`list.var` is the select list variable to receive the list. Select list variables can be processed by the `READNEXT` statement as an alternative to using numbered select lists.

The `SELECTE` statement transfers the unprocessed portion of the default select list (list 0) to the named variable.
SELECTINDEX, SELECTINDEXV

The SELECTINDEX statement creates a numbered select list from an alternate key index entry. The SELECTINDEXV statement is similar but creates a select list variable.

Format

```
SELECTINDEX index.name [ , value ] FROM file.var [ TO list.no ]

SELECTINDEXV index.name [ , value ] FROM file.var TO list.var
```

where

- `index.name` is the name of the alternate key index to be processed.
- `value` is the value to be located in the index.
- `file.var` is the file variable associated with an open file.
- `list.no` is the select list number of the list to be created. If omitted, select list zero is used.
- `list.var` is the select list variable to be created.

If the `value` is omitted, the SELECTINDEX statement constructs a select list containing all the values of the index identified by `index.name`. If the `value` is included, the SELECTINDEX statement constructs a select list containing keys of records for which the index identified by `index.name` has the given `value`.

Thus, in a file of orders with an index on the customer number field, the first form would return a list of customers referenced by the orders file and the second form would return a list of orders for a specific customer.

The STATUS() function returns zero if the SELECTINDEX is successful, non-zero if it fails because the index does not exist. Selecting records for a `value` that is not present in the index will return an empty list.

The @SELECTED variable is set to the number of entries in the returned list.

The SELECTINDEX operation leaves the internal index pointer used by the SELECTLEFT and SELECTRIGHT statements positioned at the item that has been located or, if not found, at the position where such an item would go.

Use of this statement inside a transaction will not reflect any uncommitted updates to the file.

To understand the order in which items are returned by SELECTINDEX, consider an orders file keyed by order number for which there is an index on a field that contains the customer number. The index will have an entry for each customer who has placed an order and each of these entries will hold a list of the order numbers for that customer. The order of the indexed values (customer numbers, left column in the diagram below) will be sorted left or right aligned according to the format code in the dictionary entry for the customer number field. The order numbers stored for each customer (a single line from the right column in the diagram below) will be sorted according to the format code in the dictionary entry for the @ID item in the customers file. In this example, both items are likely to be defined as right aligned and hence sorted into ascending order.
Examples

SELECTINDEX 'CUST.NO' FROM ORDERS.FILE TO 7
LOOP
    READNEXT CUST.NO FROM 7 ELSE EXIT
    CRT CUST.NO
    SELECTINDEX 'CUST.NO', CUST.NO FROM ORDERS.FILE
    LOOP
        READNEXT ORDER.NO ELSE EXIT
        CRT ORDER.NO
    REPEAT
REPEAT

This program builds a select list of all the customers referenced by the orders file as list 7. The inner loop then constructs a list of the order numbers for each customer in turn.

See also:
SETLEFT, SETRIGHT, SELECTLEFT, SELECTRIGHT
SELECTINFO()

The SELECTINFO() function returns information about a select list.

Format

SELECTINFO(list.no, key)

SELECTINFO(var, key)

where

list.no evaluates to the number of the select list to be examined. If omitted, select list zero is used.

var is a select list variable.

key identifies the action to be performed.

Values for the key to the SELECTINFO() function are defined in the KEYS.H record in the SYSCOM file. These are

1  SL$ACTIVE Returns True if the select list is active, False if it is not active.
2  SL$LEVEL Returns the command processor level at which the list was created.
3  SL$COUNT Returns the number of items remaining to be processed in the select list. If the list is not active, the SELECTINFO() function will return zero.

Use of mode 3 with a list constructed using the QMBasic SELECT statement on a dynamic file requires completion of the selection process and thus may reduce application performance.

Example

SELECT STOCK.FILE
PRINT "Stock file has " : SELECTINFO(0, SL$COUNT) : " records"
CLEARSELECT

This program fragment counts and reports the number of records in the file open with file variable STOCK.FILE.

See also:
Select lists
SELECTLEFT, SELECTRIGHT

The `SELECTLEFT` and `SELECTRIGHT` statements create a numbered select list from the entry in an alternate key index to the left or right of the last entry processed. The `SELECTLEFTV` and `SELECTRIGHTV` statements are similar but create a select list variable.

Format

```
SELECTLEFT index.name FROM file.var {SETTING key} {TO list.no}
SELECTRIGHT index.name FROM file.var {SETTING key} {TO list.no}

SELECTLEFTV index.name FROM file.var {SETTING key} TO list.var
SELECTRIGHTV index.name FROM file.var {SETTING key} TO list.var
```

where

- `index.name` is the name of the alternate key index to be processed.
- `file.var` is the file variable associated with an open file.
- `key` is the variable to be set to the key value associated with the returned list.
- `list.no` is the select list number of the list to be created. If omitted, select list zero is used.
- `list.var` is the select list variable to be created.

The `SELECTLEFT` and `SELECTRIGHT` statements construct a select list from the alternate key index entry to the left or right of the one most recently returned using `SELECTINDEX`, `SELECTLEFT` or `SELECTRIGHT`. The position of the scan can be set to the extreme left using `SETLEFT` or the extreme right using `SETRIGHT`.

These operations allow a program to find a specific value and then walk through successive values in the sorted data structure that makes up an alternate key index.

If `SELECTINDEX` is used to locate a value that does not exist in the index, `SELECTLEFT` will return a list of records for the value immediately before the non-existent one and `SELECTRIGHT` will return a list of records for the value immediately after the non-existent one.

The `STATUS()` function returns zero if the operation is successful, non-zero if it fails. Although other errors may occur, two useful status values are

```
3019 ER$AKNF Specified index does not exist
3030 ER$EOF No further items at end of index
```

The `@SELECTED` variable is set to the number of entries in the returned list, or zero if there are no further index entries to be returned.

Use of these statements inside a transaction will not reflect any uncommitted updates to the file.

See the `SELECTINDEX` statement for an example that describes how an index is sorted.

Examples
KEY = 'M'
SELECTINDEX 'POSTCODE', KEY FROM CLIENTS.FILE
LOOP
    READNEXT CLIENT.NO ELSE EXIT
    CRT CLIENT.NO
    REPEAT
        SELECTRIGHT 'POSTCODE' FROM CLIENTS.FILE SETTING POSTCODE
        WHILE @SELECTED
            WHILE POSTCODE[1, LEN(KEY)] = KEY
                REPEAT
        END IF
    END WHILE
END LOOP

This program displays a list of all clients with postcodes beginning with M.

The SELECTINDEX looks for an index entry for a postcode of "M". This is unlikely to exist and hence the select list will probably be empty. If it did find any records, the inner loop would display these. Having processed this initial list, the SELECTRIGHT moves one step right (i.e. in ascending order) through the index tree and builds a list of these records. The POSTCODE variable is returned as the value of the indexed item located. Processing continues until the SELECTRIGHT finds an item that does not begin with the characters in KEY.

SELECTINDEX 'TIME', TIMESTAMP FROM LOG.F TO 1
IF @SELECTED = 0 THEN
    SELECTLEFT 'TIME' FROM LOG.F TO 1
END IF

The above program fragment finds the record in the file open as LOG.F with the TIME field equal to TIMESTAMP. If there is no such record it finds the record with the nearest time before the requested timestamp. If multiple records have the same timestamp value, select list 1 will contain all of their ids. If TIME was the id of records in the log file, the select list could never contain multiple values.

See Alternate Key Indices for an example of how to perform partial key matching as a user enters characters of an item to be located via the index.

See also:
SELECTINDEX, SETLEFT, SETRIGHT
SEND

The SEND statement sends data to a device previously opened using OPENSEQ.

Format

```
SEND string {;} TO dev
{THEN statement(s)}
{ELSE statement(s)}
```

where

- `string` evaluates to the data to be sent. The data will be followed by the operating system specific newline sequence unless the optional colon is present after the string expression.
- `dev` is the file variable from a previous OPENSEQ.

At least one of the THEN and ELSE clauses must be present. The THEN clause will be executed if the action is successful. The ELSE clause will be executed at an error and the value of the STATUS() function will provide further information regarding the cause of the failure.

See also:
GET, OPENSEQ
SENTENCE()

The SENTENCE() function returns command line that started the current program.

Format

SENTENCE()

The SENTENCE() function is an alternative to use of the @SENTENCE variable.
SEQ()

The SEQ() function returns the character set position value of a character.

Format

\[ \text{SEQ}(\text{char}) \]

where

\[ \text{char} \]

evaluates to the character to be processed.

The SEQ() function returns the character value of char. It is the inverse of the CHAR() or ECHAR() functions. On an ECS mode system, the returned value is in the range 0 to 65535. On a non-ECS mode system, the returned value is in the range 0 to 255.

If char is a null string, SEQ() returns zero. If char is more than one character in length, SEQ() returns the value of the first character.

The SEQS() function takes a dynamic array char and returns a similarly structured dynamic array in which each element contains the ASCII value of the character in the corresponding element of char.

Example

\[ N = \text{SEQ}(\text{KEYIN}()) \]

This statement reads a single character from the keyboard and then uses SEQ() to find its ASCII character set value.

See also:
CHAR(), ECHAR()
SERVER.ADDR()

The SERVER.ADDR() function returns the IP address for a given server name.

Format

    SERVER.ADDR(server.name)

where

    server.name      is the name of the server for which the IP address is required.

The SERVER.ADDR() function can be used to find the IP address of a network server from its name. This function is not usually needed as the QMBasic socket functions work with either IP addresses or server names.

If successful, the STATUS() function will return zero. All error conditions return a null string as the IP address and subsequent use of the STATUS() function will return the error code.

Example

    DISPLAY SERVER.ADDR("openqm.com")

This statement displays the IP address of the openqm.com server.

See also:
Using Socket Connections, ACCEPT.SOCKET.CONNECTION, CLOSE.SOCKET, CREATE.SERVER.SOCKE()T, OPEN.SOCKE()T, READ.SOCKE()T, SELECT.SOCKE()T, SET.SOCKE()T.MODE(), SOCKET.INFO(), WRITE.SOCKE()T
SERVER.WINDOW()

The SERVER.WINDOW() function, available only in Windows QMConsole sessions, opens an asynchronous application window.

Format

```
SERVER.WINDOW(program {, parameters})
```

where

- `program` is the name of the program to be executed or the name of a file with a suffix recognised by Windows.
- `parameters` is a space separated list of parameters to be passed to the `program`. This must be a null string or omitted if `program` is a data file name.

The SERVER.WINDOW() function opens an asynchronous application window to execute the program identified by the function's arguments. This might be, for example, a Word document, an Excel spreadsheet, or a Notepad text file. Note that when using a network connection to QM, this window appears on the Windows server, not the client system. A client side window can be opened using the asynchronous command execution features of a terminal emulator such as AccuTerm. The SERVER.WINDOW() function is of particular use in QMConsole sessions.

The return value from this function is True if it is successful, False if an error occurs. In the case of an error, the STATUS() function and OS.ERROR() functions can be used to find the cause of the error.

The SERVER.WINDOW() function differs from the OS.EXECUTE statement (and the corresponding `SH` and `!` commands) in that it allows execution of a Windows GUI mode program that will have its own window on the server whereas OS.EXECUTE passes a command to the Windows command processor (cmd.exe) and shows the command output as part of the QM session.

Examples

```
OK = SERVER.WINDOW("notepad", "audit.txt")
```

This statement starts the Windows Notepad command to display or edit an item named audit.txt.

```
OK = SERVER.WINDOW("audit.txt")
```

This statement is an alternative way to display or edit the audit.txt file.

See also:

OS.EXECUTE
SET.ARG

The SET.ARG statement updates a subroutine argument value based on its position in the argument list. It is intended for use with subroutines declared with the VAR.ARGS option.

Format

    SET.ARG n, value

where

    n     is the argument list position, numbered from one.
    value is the value to be set.

Subroutines declared with the VAR.ARGS option may have a variable number of arguments. Although each argument must have a name assigned to it in the SUBROUTINE statement, it is often useful to be able to process a series of arguments by indexing this list.

The SET.ARG statement sets the value of argument n. The actual number of arguments passed may be determined using the ARG.COUNT() function. Use of an argument position value less than one or greater than the number of arguments causes the program to abort.

See also:
ARG(), ARG.COUNT, ARG.PRESENT(), SUBROUTINE
**SET.BREAK.HANDLER**

The **SET.BREAK.HANDLER** statement allows a program to establish a handler subroutine that will be called if the user presses the break key.

**Format**

```plaintext
SET.BREAK.HANDLER name
```

where

- `name` evaluates to the name of the break handler. This must be a catalogued subroutine.

The break handler subroutine will be called if the break key is enabled and the user presses it. The subroutine takes a single argument through which it returns an action code telling QM how to handle the break. Possible return values are:

- 0  Display normal break key action prompt
- 1  Continue execution
- 2  Quit (equivalent to use of STOP)
- 3  Abort
- 4  Logout

Any other value will display the normal break key action prompt.

The handler applies to the program that establishes it and to all programs called from it unless they establish their own break handler. The handler may perform any appropriate application processing to determine the action code to be returned but must not execute a **STOP** or **ABORT** statement internally.

The break handler will be deactivated when the program that established it terminates. The program can also use **REMOVE.BREAK.HANDLER** to deactivate the handler while the program continues to run. A program cannot remove the break handler of another program further down the call stack. On deactivation of a break handler, any handler belonging to a program further down the call stack becomes active again.

**See also:**

- [Interrupting commands](#)
- **REMOVE.BREAK.HANDLER**
SET.ECS.MAP()

The SET.ECS.MAP() function selects a given ECS character map.

Format

\[
\text{SET.ECS.MAP}(\text{name})
\]

where

\text{name} \quad \text{evaluates to the name of the ECS character map to be loaded. A null string loads the default map.}

The \text{SET.ECS.MAP}() function allows a program to change the character map on an ECS mode system.

The map name is converted to uppercase internally. Use of a null string as the name loads the default character map as set in the QM configuration parameters.

The function returns a Boolean value indicating whether it was successful. Failure to install a new map leaves the previous map in place. Use of this function on a non-ECS system always returns False.

A program can determine the name of the currently loaded character map by use of the \text{SYSTEM(1021)} function.

Example

\[
\text{OLD.MAP} = \text{SYSTEM(1021)}
\]
\[
\text{IF NOT(SET.ECS.MAP('UNICODE')) THEN}
\]
\[
\qquad \text{STOP 'Cannot load Unicode map'}
\]
\[
\text{END}
\]

The above program fragment saves the current character map name and then attempts to load the Unicode map, terminating the program if unsuccessful.

See also:
Extended Character Set Support, ECS.MAP, SYSTEM()
**SET.EXIT.STATUS**

The **SET.EXIT.STATUS** statement sets the final exit status returned by QM to the operating system.

**Format**

```
SET.EXIT.STATUS value
```

where

- `value` is the exit status value to be set.

By default, QM returns an exit status of zero to the operating system on termination. The **SET.EXIT.STATUS** statement allows an application to return an alternative exit status value to indicate, for example, success or failure. Note that error conditions detected during startup of a QM session return an exit status of 1.

See also the **SET.EXIT.STATUS** command.
SETLEFT, SETRIGHT

The SETLEFT and SETRIGHT statements set the scanning position of an alternate key index at the extreme left or right of the data.

Format

```
SETLEFT index.name  FROM file.var
SETRIGHT index.name  FROM file.var
```

where

- `index.name` is the name of the alternate key index to be processed.
- `file.var` is the file variable associated with an open file.

The SETLEFT and SETRIGHT statements are used with SELECTLEFT and SELECTRIGHT to set the scan position to the first or last entry in an alternate key index.

The STATUS() function returns zero if the operation is successful, non-zero if it fails because the index does not exist.

Examples

```
SETLEFT 'POSTCODE' FROM CLIENTS.FILE
LOOP
  SELECTRIGHT 'POSTCODE' FROM CLIENT.FILE SETTING POSTCODE UNTIL POSTCODE[1,1] >= 'N'
  CRT POSTCODE
REPEAT
```

This program displays a list of all postcodes commencing with a letter in the first half of the alphabet.

See also:

SELECTINDEX, SELECTLEFT, SELECTRIGHT
The **SET.PORT.PARAMS**() function sets the communications parameters for a serial port.

**Format**

```
SET.PORT.PARAMS(fvar, params)
```

where

- `fvar` is the file variable from the **OPENSEQ** statement that was used to open the port.
- `params` is a dynamic array holding the new parameters to be set.

The **SET.PORT.PARAMS**() function returns True if successful, False if an error occurs.

The `params` dynamic array contains the following data:

- Field 1 Port name (ignored)
- Field 2 Baud rate
- Field 3 Parity mode (0 = off, 1 = odd, 2 = even)
- Field 4 Bits per byte (5 to 8)
- Field 5 Stop bits (1 or 2)
- Fields 6 to 9 Ignored (see **GET.PORT.PARAMS**())

To allow for the possibility of additional fields being added in future releases, programs should use the **GET.PORT.PARAMS**() function to retrieve the current settings, modify this as required and then use **SET.PORT.PARAMS**() to set the new parameters.

**Example**

```
PARAMS = GET.PORT.PARAMS(PORT)
PARAMS<2> = 9600
IF NOT(SET.PORT.PARAMS(PORT, PARAMS) THEN STOP 'Error setting parameters'
```

**See also:**

**GET.PORT.PARAMS**(), **SETPORT**
SET.SOCKET.MODE()

The SET.SOCKET.MODE() function sets parameters for an open socket.

Format

SET.SOCKET.MODE(skt, key, value)

where

skt is the socket variable for an open socket.
key identifies the mode to be set:

- SKT$INFO.BLOCKING Default to blocking mode?
- SKT$INFO.NO.DELAY Nagle algorithm disabled?
- SKT$INFO_KEEP.ALIVE Send keep alives?
- SKT$INFO_TRACE Activate socket tracing to the filepathname in value. A null pathname turns off tracing.
- SKT$INFO.EVENTS Specify events to be monitored for use with SELECT.SOCKET().
- SKT$INHERITABLE Make the socket inheritable.

value is the required value of the parameter. Except as explicitly shown above, this is a Boolean value enabling or disabling the mode identified by key.

The SET.SOCKET.MODE() function returns True if the action is successful, False if it fails. The STATUS() function can be used to determine the cause of failure.

The SKT$INFO.EVENTS mode allows a program to specify events that are to be monitored for the socket when using the SELECT.SOCKET() function. The value argument is an additive value chosen from:

1. SKT$READ_EVENT Notify availability of a new connection on a server socket
   Notify availability of data that can be read with READ.SOCKET()
   Notify closure of connection
2. SKT$WRITE_EVENT Notify when data can be sent with WRITE.SOCKET()
3. SKT$EXCEPTION_EVENT Notify exceptions

See also:
Using Socket Connections, ACCEPT.SOCKET CONNECTION, CLOSE.SOCKET, CREATE.SERVER.SOCKET, OPEN.SOCKET, READ.SOCKET, SELECT.SOCKET, SERVER.ADDR, SOCKET.INFO, WRITE.SOCKET
SET.STATUS

The SET.STATUS statement sets a value for return by the STATUS() function.

Format

SET.STATUS expr

where

expr evaluates to the integer value to be returned by subsequent use of STATUS().

The SET.STATUS statement sets the internal process status memory. The value of this item is returned by the STATUS() function.

This statement should rarely be required by application developers but can simplify some error handling. Note that the STATUS() function always returns the absolute (positive) value of the status value.
SET.TIMEZONE

The SET.TIMEZONE statement sets the time zone for use by the epoch conversion code.

Format

    SET.TIMEZONE zone

where

    zone       is the name of the time zone to be set.

The SET.TIMEZONE statement is equivalent to executing the CONFIG command to set the TIMEZONE private configuration parameter. There is no validation performed on the zone name.

A program can retrieve the current time zone using the CONFIG() function with "TIMEZONE" as the argument value.

See also:
Date, Times and Epoch Values, EPOCH(), MVDATE(), MVDATE.TIME(), MVEPOCH(), MVTIME(), SET.TIMEZONE
**SETNLS**

The **SETNLS** statement sets the value of a national language support parameter.

**Format**

```
SETNLS key, value
```

where

- `key` identifies the parameter to be set.
- `value` is the new value for the parameter.

The **SETNLS** statement sets the value of the named national language support parameter. NLS parameter name tokens are defined in the KEYS.H include record.

**Available parameters are:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Key</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NLSSCURRENCY</td>
<td>Default currency symbol. Maximum 8 characters.</td>
</tr>
<tr>
<td>2</td>
<td>NLSSTHOUSANDS</td>
<td>Default thousands separator character.</td>
</tr>
<tr>
<td>3</td>
<td>NLSSDECIMAL</td>
<td>Default decimal separator character.</td>
</tr>
<tr>
<td>4</td>
<td>NLSSIMPLICIT.DEcimal</td>
<td>Default decimal separator character for implicit conversions. See below.</td>
</tr>
<tr>
<td>5</td>
<td>NLSEUROPEAN.DATES</td>
<td>Set/clear European date mode</td>
</tr>
</tbody>
</table>

Setting two or more of the national language support parameters to the same character will have undefined effects.

When using a decimal separator other than the default period, conversion of a numeric character string to a number will recognise either the defined character or a period as being the decimal separator.

Numeric constants in QMBasic programs are not affected by the setting of the decimal separator character and must always be written using periods.

The NLSSDECIMAL setting determines the default decimal separator character to be used when converting numeric values between their internal storage form and their external character form using `FMT()`, `ICONV()` or `OCONV()`. The NLSSIMPLICIT.DEcimal setting determines the decimal separator character that will be used by implicit conversion of a numeric value to its external character representation in other ways such as simply referencing a floating point variable in a `DISPLAY` statement.

In most applications, these two representations of the decimal separator will be the same and hence setting the NLSSDECIMAL value also sets the NLSSIMPLICIT.DEcimal value. Sometimes an
application may want to use, for example, the comma as the decimal separator in reports and most program output but the period as the decimal separator in implicit conversions such as that done by the `INPUT` statement, perhaps to allow validation against a pattern template such as "1-3N.2N". Setting the NLS$SIMPLICIT.DECIMAL value allows this. Note that this must be set after the NLS$DECIMAL value.

**Examples**

```
SETNLS NLS$CURRENCY, 'Eur'
```

The above statement sets the currency prefix as "Eur"

```
SETNLS NLS$DECIMAL, ','
SETNLS NLS$THOUSANDS, '.'
DISPLAY 123.45
DISPLAY "123.45" + 0
DISPLAY "123,45" + 0
```

All three of the DISPLAY statements in the above program fragment will display

```
123,45
```

**See also:**

`GETNLS()`, `NLS`
SETPU

The SETPU statement sets the characteristics of a print unit.

Format

\textbf{SETPU} \textit{key, unit, value}

where

\begin{itemize}
  \item \textit{key} identifies the parameter to set. This may be:
    \begin{itemize}
      \item 1 \texttt{PUSMODE} Print unit mode
      \item 2 \texttt{PUSWIDTH} Characters per line
      \item 3 \texttt{PUSLENGTH} Lines per page
      \item 4 \texttt{PUSTOPMARGIN} Top margin size
      \item 5 \texttt{PUSBOTMARGIN} Bottom margin size
      \item 6 \texttt{PUSLEFTMARGIN} Left margin size
      \item 7 \texttt{PUSSPOOLFLAGS} Various print unit flags
      \item 9 \texttt{PUSFORM} Form name (not used by all spoolers)
      \item 10 \texttt{PUSBANNER} Banner page text
      \item 11 \texttt{PUSLOCATION} Printer / file name
      \item 12 \texttt{PUSCOPIES} Number of copies to print
      \item 15 \texttt{PUSPAGENUMBER} Current page number (see below)
      \item 1006 \texttt{PUSOPTIONS} Options to be passed to the spooler
      \item 1007 \texttt{PUSPREFIX} Pathname of file holding prefix data to be added to the start of the output
      \item 1008 \texttt{PUSSPOOLER} Spooler to be used (ignored on Windows)
      \item 1009 \texttt{PUSOVERLAY} Catalogued overlay subroutine name (see \texttt{SETPTR})
      \item 1010 \texttt{PUSCPI} Characters per inch (may be non-integer value)
      \item 1011 \texttt{PUSPAPER.SIZE} Paper size. See SYSCOM PCL.H
      \item 1012 \texttt{PUSLPI} Lines per inch. Must be 1, 2, 3, 4, 6, 8, 12, 16, 24, 48
      \item 1013 \texttt{PUSWEIGHT} Font stroke weight. See SYSCOM PCL.H
      \item 1014 \texttt{PUSSSYMBOL.SET} Symbol set. See SYSCOM PCL.H
      \item 1015 \texttt{PUSSTYLE} Query processor style. See the Query processor \texttt{STYLE} option for details.
      \item 1016 \texttt{PUSNEWLINE} Newline string for this print unit. Must be \texttt{char(10)}, \texttt{char(13)} or \texttt{char(13):char(10)}
      \item 1017 \texttt{PUSPRINTER.NAME} Name of printer
      \item 1018 \texttt{PUSFILE.NAME} File name for output directed to a file
    \end{itemize}
\end{itemize}
PU$FORM.FEED  Set form feed code for this print unit. Must be char(12) or char(13):char(12)

PU$ENCODING  Set character encoding for this print unit

PU$FONT.SIZE  Set the GDI mode font size. Zero uses the default font.

PU$LINENO  Current line number

The SETPU statement sets the print unit characteristic specified by key to the given value. It is closely related to the SETPU() subroutine.

If successful, STATUS() is set to zero. Otherwise, STATUS() returns an error code.

Mode 15 (PU$PAGENUMBER) can be used to set the current page number before any output to the print unit if a report is to start at a page number other than one. Using this mode after output has commenced may have indeterminate effects.

Example

```
SETPU PU$LOCATION, 3, "LASER"
```

The above statement sets the destination for print unit 3 to be the LASER printer.

See also:

GETPU()
**SETREM**

The **SETREM** statement sets the remove pointer of a string.

**Format**

```
SETREM offset ON string
```

where

- **offset** is the character position (from 1) where the remove pointer is to be set.
- **string** is the string on which the remove pointer position is to be set.

Assigning a character string variable automatically sets the remove pointer to zero, effectively pointing one character before the start of the string. The **SETREM** statement allows an application to set the remove pointer to an arbitrary offset into string. The **STATUS()** function will return zero if the action is successful.

If the offset is less than zero, zero is assumed. If the offset is greater than the length of **string**, the remove pointer is set to point one character beyond the end of the string.

**SETREM** is typically used with **GETREM()** to save and restore the remove pointer position.

**Example**

```
RMV.PTR = GETREM(S)
GOSUB PROCESS.DATA
SETREM RMV.PTR ON S
```

The above code fragment saves the remove pointer associated with string S and restores it after execution of subroutine PROCESS.DATA which might change this remove pointer.

**See also:**

**GETREM, REMOVE**
**SHARED**

The *SHARED* statement declares persistent variables to be shared across all instances of a QMBasic class module.

**Format**

```
SHARED {PRIVATE | PUBLIC} {var1 ,var2...}
```

where

```
var1, etc   are variable names
```

The *SHARED* statement is used to define variables in class modules that are to be visible to all instances of the same class within a QM process. The variables are discarded when the last instance of the class is destroyed.

The variable names may extend over multiple lines by splitting the statement after a comma. For example

```
SHARED PUBLIC VAR1, VAR2,
    VAR3, VAR4
```

The same variables could be defined as

```
SHARED PUBLIC VAR1
SHARED PUBLIC VAR2
SHARED PUBLIC VAR3
SHARED PUBLIC VAR4
```

The compiler assumes that definitions of variables in each *SHARED* statement are a continuation of any previous definitions in the same class module.

Shared variables are initially unassigned and may be defined as either private or public. The *SHARED* statement does not support use of initialisation values. Where shared variables are to be initialised in the CREATE.OBJECT subroutine on creation of just the first instance of the class, this can be achieved by testing whether one shared variable is unassigned using a construct of the form

```
IF UNASSIGNED(variable.name) THEN
    ...do initialisation...
END
```

Shared variable definitions may include matrices. These are defined by including the row and column bounds in the *SHARED* statement, for example

```
SHARED MAT1(5,3)
```

QM supports two styles of matrix with different characteristics. The PICK.MATRIX setting of the $MODE compiler directive can be used to select Pick style matrices.

The default style of matrix includes a zero element and is resizeable. A shared variable list declared as

```
SHARED A,B(3),C
```

has three variables, one of which is a matrix:
A matrix of this type can be re-dimensioned by a later `DIM` statement.

Pick style matrices do not have a zero element and cannot be resized. A matrix of this type is equivalent to a series of simple variables:

```
| A | B(1) | B(2) | B(3) | C |
```

See also:
PRIVATE, PUBLIC
SHIFT()

The **SHIFT()** function performs a logical bit-shift operation on an integer value.

**Format**

\[
\text{SHIFT}(\text{value}, \text{shift.len})
\]

where

\[
\text{value} \quad \text{evaluates to the integer to be shifted.}
\]

\[
\text{shift.len} \quad \text{indicates the number of bit positions by which value is to be shifted.}
\]

The **SHIFT()** function converts **value** to a 32 bit unsigned integer, truncating any fractional part of a non-integer value, and shifts the bit pattern of this value by **shift.len** positions.

A positive value of **shift.len** shifts right (towards the low order end). A negative value of **shift.len** shifts left (towards the high order end).

Values of **shift.len** that are outside the range -32 to +32 have undefined results.

**Example**

```plaintext
FOR I = 30 TO 0 STEP -3
    DISPLAY BITAND(SHIFT(N, I), 7) :
NEXT I
```

This program fragment displays the value of N in octal. The MO conversion mode of the **OCONV()** function would be more appropriate.

**See also:**

**BITAND(), BITNOT(), BITOR(), BITRESET(), BITSET(), BITTEST(), BITXOR()**
SIGNAL()

The SIGNAL() function raises a cross-process application signal.

Format

    SIGNAL(user.no)

where

    user.no    evaluates to the QM number of the process to be signalled.

The SIGNAL() function provides a simple way to notify another QM process of a condition such as a request for a phantom to terminate. It returns True if the process identified by user.no is logged in.

Example

    OK = SIGNAL(SERVER.USER)

This statement raises a cross-process application signal in the QM process with the user number in SERVER.USER.

The signalled process detects this signal using SYSTEM(1062) which will also clear it. For example, a phantom server process might have a main loop as below:

    LOOP
        ...processing...
        UNTIL SYSTEM(1062)
        SLEEP 5
    REPEAT

See also:

    SYSTEM()
SIN()

The SIN() function returns the sine of a value.

Format

\[ \text{SIN}(expr) \]

where

\[ expr \]

evaluates to a number or a numeric array.

The SIN() function returns the cosine of \( expr \). Angles are measured in degrees.

If \( expr \) is a numeric array (a dynamic array where all elements are numeric), the SIN() function operates on each element in turn and returns a numeric array with the same structure as \( expr \).

Example

\[ \text{OPP} = \text{HYP} \times \text{SIN(ANGLE)} \]

This statement finds the length of the opposite side of a right angled triangle from the length of the hypotenuse and the adjacent angle.

See also:
\[ \text{ACOS()}, \text{ASIN()}, \text{ATAN()}, \text{COS()}, \text{TAN()} \]
SLEEP

The SLEEP statement causes the program in which it is executed to pause for a given number of seconds or until a specific time. The synonym RQM may be used in place of SLEEP.

Format

SLEEP {time}

where

time determines the time for which the program is to sleep. If omitted, time defaults to one.

The SLEEP statement operates in one of two ways depending on the format of time.

If time is a number, it is rounded to an integer value and the program sleeps for that number of seconds. If time is negative or zero, the program continues without sleeping. See NAP for a way to pause for less than one second.

If time is not a number, an attempt is made to convert it to a time of day using any of the formats accepted by the ICONV() function MT conversion. If successful, the program sleeps until this time. If the time of day specified by time has already passed, it is assumed to be a reference to that time on the following day. If time cannot be converted to a time of day, the program continues without sleeping.

In all cases, if there is more than one process running, the SLEEP statement causes a process switch to occur. It can therefore be used to relinquish the remainder of the timeslice of the current process if waiting for some event to occur in another process, such as release of a lock.

The actual sleep time may vary from that specified due to process scheduling actions within the operating system.

If the break key is used to interrupt a program which is sleeping, selection of the G option will continue to sleep to the specified time. The Q option will abort the program immediately.

Examples

SLEEP "10:30PM"

This statement causes the program to sleep until half past ten at night.

SLEEP 10

This statement causes the program to pause for 10 seconds

See also:
NAP
SOCKET.INFO()

The `SOCKET.INFO()` function returns information about an open socket.

Format

```
SOCKET.INFO(skt, key)
```

where

- `skt` is the socket variable for an open socket.
- `key` identifies the information to be returned:
  - `SKTSINFO.OPEN` is `skt` a socket variable? Returns True or False.
  - `SKTSINFO.TYPE` Type or socket. Returns one of the following values according to which socket function was used to open the socket:
    - `CREATE.SERVER.SOCKET()`
    - `ACCEPT.SOCKET.CONNECT()`
    - `OPEN.SOCKEET()`
  - `SKTSINFO.PORT` Port number.
  - `SKTSINFO.IP.ADDR` Peer IP address. When used with a server socket, this will be a null string if the `addr` argument to the `CREATE.SERVER.SOCKEET()` was null.
  - `SKTSINFO.BLOCKING` Default blocking mode.
  - `SKTSINFO.NO.DELAY` Nagle algorithm disabled?
  - `SKTSINFO.KEEP.ALIVE` Send keep alives?
  - `SKTSINFO.EVENTS` Report events related to use of `SELECT.SOCKEET()`.
  - `SKTSINFO.PEER.PORT` Port number at other end of connection.
  - `SKTSINFO.NAME` Pathname for Unix local socket.
  - `SKTSINFO.LOCAL.NAME` Local IP address on established TCP connection.
  - `SKTSINFO.PEER.NAME` Reverse DNS lookup for peer name. Returns a null string if unavailable.

The `SOCKET.INFO()` function returns information about an open socket as shown in the parameter descriptions above.
The value returned by SKT$INFO.EVENTS is an additive value formed from:

1 SKT$READ_EVENT A new connection is on a server socket
   Data is available to be read with READ.SOCKET()
   The connection has been closed

2 SKT$WRITE_EVENT Data can be sent with WRITE.SOCKET()

4 SKT$EXCEPTION_EVENT An exception has occurred

See also:
Using Socket Connections, ACCEPT.SOCKET.CONNECTION, CLOSE.SOCKET,
CREATE.SERVER.SOCKET(), OPEN.SOCKET(), READ.SOCKET(), SELECT.SOCKET(),
SERVER.ADDR(), SET.SOCKET.MODE(), WRITE.SOCKET()
SORT()

The SORT() function creates a sort variable.

Format

SORT(key.count, [MAT] flags)

where

- **key.count** specifies the number of keys in the sort. Up to 32 keys may be specified.
- **flags** is the name of a single dimensional array of sort key mode flags prefixed with MAT. For a single key sort, **flags** may be specified as a scalar variable or an expression.

The SORT() function is used at the start of a sort operation to create a sort variable that controls the sort process. The function returns a sort variable that can be used in subsequent sort related operations.

The **flags** value is formed from the items below. The token names shown are defined in the SYSCOM KEYS.H include record.

At most one of the following sort styles:

0  **SRT$LEFT**  A simple left justified comparison, examining corresponding characters from the start of each string until either a difference is found or the end of both strings has been reached.

1  **SRT$RIGHT**  A simple right justified comparison in which the shorter item is effectively padded with leading spaces and the resultant strings are compared character by character from the start until either a difference is found or the end of both strings has been reached. If both strings can be treated as integer values, possibly with a leading sign character, a numeric comparison is performed.

2  **SRT$COMPOUND**  A compound sort in which the two strings are considered to be formed from a series of alternating numeric and non-numeric elements. Numeric elements are sorted into numerical value, non-numeric elements are sorted into collating sequence order. If the first element is numeric, it may have an optional leading sign character. Sign characters appearing later in the strings are treated as being non-numeric characters.

3  **SRT$RIGHT.FLOAT**  The same as mode 1 except that the test for numeric items allows non-integer values. The same as mode 1 except that the test for numeric items allows non-integer values.

Any combination of the following sort options:

16  **SRT$DESCENDING**  The result of the comparison operation is reversed.

32  **SRT$UNIQUE**  Duplicate key values are ignored.
Examples

The following example shows a simple sort with a single key and no data. The first loop enters the data. The second loop displays the sorted results.

```qmbasic
SORTVAR = SORT(1, SRT$LEFT)
LOOP
   INPUT X
   UNTIL X = ""
   SORTADD SORTVAR, X
REPEAT

LOOP
   X = SORTNEXT(SORTVAR)
   UNTIL STATUS()
   DISPLAY X
REPEAT
```

The next example shows a two key sort using a descending order numeric value representing a date as the first key and a case insensitive left aligned sorted client id as the second key. The client name is attached to the keys as a data element.

```qmbasic
DIM KEYS(2)
KEYS(1) = SRT$RIGHT + SRT$DESCENDING + SRT$DATA
KEYS(2) = SRT$CASE
SORTVAR = SORT(2, MAT KEYS)
SELECT ORD.F
LOOP
   READNEXT ORD.ID ELSE EXIT
   READ ORD.REC FROM ORD.F, ORD.ID THEN
      KEYS(1) = ORD.REC<O.DATE>
      KEYS(2) = ORD.REC<O.CLIENT>
      SORTADD SORTVAR, MAT KEYS, ORD.REC<O.NAME>
   END
REPEAT

LOOP
   NAME = SORTNEXT(SORTVAR, MAT KEYS)
   UNTIL STATUS()
   DISPLAY OCONV(KEYS(1), 'D2DMY'), KEYS(2), NAME
REPEAT
```

The above example is broadly equivalent to use of a query

```
LIST ORDERS DATE @ID NAME BY.DSND DATE BY @ID
```
See also:

`Sorting`, `MULTISORT`, `SORTADD`, `SORTCLEAR`, `SORTDATA()`, `SORTNEXT()`
SORTADD

The **SORTADD** statement adds data to the sort system.

**Format**

```
SORTADD sortvar, [MAT] keys [, data]
```

where

- `sort.var` is a sort variable created by a previous use of **SORT()**.
- `keys` is the name of a single dimensional array of sort keys prefixed with MAT. For a single key sort, `keys` may be specified as a scalar variable or an expression. In either case, the data type of this variable may change.
- `data` is the optional data associated with the given keys. If omitted, this defaults to a null string. This value is ignored if the SRTSDATA flag was not set in the first key description passed to **SORT()**.

The **SORTADD** statement adds the given keys and optional data to the sort session related to `sortvar`.

**See also:**
- **Sorting**, **SORT()**, **SORTCLEAR**, **SORTDATA()**, **SORTNEXT()**
SORTCLEAR

The SORTCLEAR statement terminates a sort session, discarding all data.

Format

    SORTCLEAR sortvar

where

    sortvar is a sort variable created by a previous use of SORT().

The SORTCLEAR statement tidies up on completion of a sort operation by clearing the memory based sort tree and deleting any sort work files. A sort session is implicitly cleared by any action that overwrites the sort variable.

See also:
    Sorting, SORT(), SORTADD, SORTDATA(), SORTNEXT()
SORTDATA()

The SORTDATA() function returns the data elements from a sort.

Format

    SORTDATA(sort.var, {max.items})

where

    sort.var is a sort variable created by a previous use of SORT().
    max.items is the maximum number of items to be returned. If omitted or less than one, all data is returned.

The SORTDATA() function is intended as an alternative to a loop containing a SORTNEXT() function where no processing of the individual items is required. It returns the data associated with sorted items as a field mark delimited dynamic array. If the sort does not include a data element, the first key value for each entry is returned.

The max.items argument limits the number of items returned. A further use of SORTDATA() will return subsequent items from the sort.

See also:
Sorting, SORT(), SORTADD, SORTCLEAR, SORTNEXT()
SORTNEXT()

The `SORTNEXT()` function retrieves the next entry from a sort.

**Format**

```
SORTNEXT(sort.var, {{MAT} keys})
```

where

- `sort.var` is a sort variable created by a previous use of `SORT()`.
- `keys` is the optional name of a single dimensional array to receive the key values associated with the retrieved item prefixed with `MAT`. For a single key sort, `keys` may be specified as a simple variable reference.

The two argument form of the `SORTNEXT()` function returns the data associated with the next entry in a sort. The keys are returned via the `keys` array.

The single argument form of the `SORTNEXT()` function returns the data associated with the next entry in a sort or, if the sort has no data element, a null string.

The `STATUS()` function should be used to determine whether the action was successful. This returns non-zero if there is no item to return.

**See also:**
- Sorting, `SORT()`, `SORTADD`, `SORTCLEAR`, `SORTDATA()`
**SORT.COMPARE()**

The **SORT.COMPARE()** function compares two items using a specified set of comparison rules.

**Format**

```
SORT.COMPARE(string1, string2, mode \{, no.case\})
```

where

- `string1` is the first string to be compared.
- `string2` is the second string to be compared.
- `mode` identifies the manner in which the comparison is to be performed.
- `no.case` a Boolean value indicating whether the comparison should be case insensitive. This argument is retained for compatibility with earlier releases but case insensitivity can be specified in the `mode` value.

The **SORT.COMPARE()** function compares `string1` and `string2` according to the sort rules specified by the `mode` value which may be specified as a numeric value or by use of the tokens from the SYSCOM KEYS.H include record shown below.

**Mode 0** SRT$LEFT. A simple left justified comparison, examining corresponding characters from the start of each string until either a difference is found or the end of both strings has been reached.

**Mode 1** SRT$RIGHT. A simple right justified comparison in which the shorter item is effectively padded with leading spaces and the resultant strings are compared character by character from the start until either a difference is found or the end of both strings has been reached. If both strings can be treated as integer values, possibly with a leading sign character, a numeric comparison is performed.

**Mode 2** SRT$COMPOUND. A compound sort in which the two strings are considered to be formed from a series of alternating numeric and non-numeric elements. Numeric elements are sorted into numerical value, non-numeric elements are sorted into collating sequence order. If the first element is numeric, it may have an optional leading sign character. Sign characters appearing later in the strings are treated as being non-numeric characters.

**Mode 3** SRT$RIGHT.FLOAT. The same as mode 1 except that the test for numeric items allows non-integer values.

**Mode 4** SRT$STRICT. Similar to a compound sort but numerically equivalent elements such as "28" and "028" to be treated as being different.

Other additive mode flags are:

- **SRT$DESCENDING** The result of the comparison operation is reversed.
- **SRT$NOCASE** Perform a case insensitive comparison.

In all cases, the `no.case` argument can be used to specify that alphabetic items should be treated as case insensitive, effectively replacing lower case letters with their upper case equivalents.

The **SORT.COMPARE()** function returns

- **-1** `string1` comes before `string2`
0  \textit{string1} is equal to \textit{string2}
1  \textit{string1} comes after \textit{string2}

\textbf{Examples}

Using modes the modes indicated below, the effect of sorting GC5, ND620, GC41 and CD631 would be

- Mode 0: CD631, GC41, GC5, ND620
- Mode 1: GC5, GC41, CD631, ND620
- Mode 2: CD631, GC5, GC41, ND620

\textbf{See also:}\n\texttt{COMPARE()}, \texttt{SORT()}


SOUNDEX()

The SOUNDEX() function returns a four character string determined by the phonetic content of a string. The SOUNDEXS() function is similar to SOUNDEX() but operates on successive elements of a dynamic array, returning a similarly structured dynamic array of results.

Format

SOUNDEX(string)

where

string is the string for which the sound code is to be returned.

The SOUNDEX() function is useful for situations where it is desired to compare or locate items by their spoken sound. For example, names in a telephone directory could be indexed by their SOUNDEX() value to aid location of similar sounding names.

The value returned by SOUNDEX() is made up from the first letter of string in upper case followed by three digits which are found by examination of further characters of string according to the following table.

<table>
<thead>
<tr>
<th>Value</th>
<th>Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>A E H I O U W Y</td>
</tr>
<tr>
<td>1</td>
<td>B F P V</td>
</tr>
<tr>
<td>2</td>
<td>C G J K Q S X Z</td>
</tr>
<tr>
<td>3</td>
<td>D T</td>
</tr>
<tr>
<td>4</td>
<td>L</td>
</tr>
<tr>
<td>5</td>
<td>M N</td>
</tr>
<tr>
<td>6</td>
<td>R</td>
</tr>
</tbody>
</table>

Non-alphabetic characters and letters in group 0 are ignored. Consecutive occurrences of the same letter insert only a single character. If the result is less than four characters long, zeros are added to fill the remaining positions. Thus the word SOUNDEX encodes to S532.

On ECS mode systems, the Soundex algorithm is based only on the letters A to Z, upper and lower case. The character maps do not affect this function.

Example

DISPLAY "Enter name 
INPUT NAME
KEY = SOUNDEX(NAME)
READ OTHER.NAMES FROM PHONONYMS, KEY THEN
   NAME = OTHER.NAMES
END

This program fragment prompts for and reads a name. It then establishes the soundex key for this name and attempts to read a list of similar sounding names from the PHONONYMS file. If found, this list replaces the NAME value.
SPACE()

The SPACE() function returns a string consisting of a given number of spaces. The SPACES() function is similar to SPACE() but operates on successive elements of a dynamic array, returning a similarly structured dynamic array of results.

Format

\[
\text{SPACE}(\text{count})
\]

where

\[
\text{count} \quad \text{evaluates to the desired number of spaces.}
\]

The SPACE() function is a useful way to generate multiple spaces. It can aid readability of programs by removing the need for space filled strings and it can be used to provide variable numbers of spaces where required.

Example

\[
\text{PRINT SPACE (INDENT) : TEXT}
\]

This statement prints the contents of TEXT indented by the number of spaces specified by INDENT.

See also:

STR()
SPLICE()

The SPLICE() function concatenates corresponding elements of a dynamic array, inserting a string between each pair of items.

Format

SPLICE(array1, string, array2)

where

array1 is the first dynamic array.
string is the string to be inserted between each pair of items.
array2 is the second dynamic array.

The SPLICE() function returns the result of concatenating corresponding dynamic array components (fields, values and subvalues) from the supplied arrays, inserting string between each pair.

The REUSE() function can be applied to either or both dynamic arrays. Without this function, any absent trailing values are taken as null strings.

Example

S1 = "ABC":@fm:"DEF"
S2 = "123":@vm:"456":@fm:"789"
X = SPLICE(S1,'-', S2)

The above code fragment concatenates elements of the two strings yielding a result in X of "ABC123VM456FM789"
SQRT()

The SQRT() function returns the square root of a value.

Format

\[
\text{SQRT}(\text{expr})
\]

where

\( \text{expr} \) evaluates to a number or a numeric array.

The SQRT() function returns the square root of \( \text{expr} \).

If \( \text{expr} \) is a numeric array (a dynamic array where all elements are numeric), the SQRT() function operates on each element in turn and returns a numeric array with the same structure as \( \text{expr} \).

A negative value of \( \text{expr} \) will cause a run time error.

Example

\[
N = \text{SQRT}(A)
\]

This statement finds the square root of \( A \) and assigns this to \( N \).
SQUOTE()

The SQUOTE() function returns a copy of its argument string enclosed in single quotes. The SQUOTES() function is similar but operates on successive elements of a dynamic array, returning a similarly structured dynamic array of results.

Format

\[ \text{SQUOTE(expr)} \]

where

\[ expr \]

evaluates to the source string.

The SQUOTE() function returns \( expr \) enclosed in single quotation marks.

Example

\[ A = \text{SQUOTE('ABC123')} \]

This statement sets \( A \) to the eight character string 'ABC123'.

\[ A = 'ABC' : @VM : 'DEF' \]
\[ B = \text{QUOTES}(A) \]

This program fragment encloses each element of dynamic array \( A \) in single quotes, storing the result in \( B \) as

'ABC' VM 'DEF'

See also:

QUOTE()
SSELECT, SSELECTN, SSELECTV

The SSELECT statement creates a select list containing all record keys from a file sorted into order.

Format

\[
\text{SSELECT } file.var \{ \text{TO list.no} \} \{ \text{modes} \} \{ \text{ON ERROR statement(s)} \}
\]

\[
\text{SSELECTN } file.var \{ \text{TO list.no} \} \{ \text{modes} \} \{ \text{ON ERROR statement(s)} \}
\]

\[
\text{SSELECTV } file.var \{ \text{TO list.var} \} \{ \text{modes} \} \{ \text{ON ERROR statement(s)} \}
\]

where

- \( file.var \) is the file variable associated with an open file.
- \( list.no \) is the select list number of the list to be created. If omitted, select list zero is used.
- \( list.var \) is the select list variable to receive the list. Select list variables can be processed by the READNEXT statement as an alternative to using numbered select lists. If the TO clause is omitted, a default select list variable is used.
- \( modes \) is any combination of the following optional qualifiers:
  - DESCENDING: Sorts into descending order
  - NO.CASE: Performs a case insensitive sort
  - RIGHT.ALIGNED: Sorts data in right aligned mode
- \( \text{statement(s)} \) are statement(s) to be executed if a fatal error occurs.

The SSELECT and SSELECTN statements construct a list of record keys in the file open as \( file.var \) and store it as an active select list \( list.no \) replacing any previously active list. If there are no records in the file, an empty list is created. By default, keys will be stored in left justified ascending order. The optional RIGHT.ALIGNED qualifier causes the list to be sorted as right aligned data. The optional DESCENDING qualifier causes the list to be sorted into descending order. The optional NO.CASE qualifier performs a case insensitive sort.

The SSELECTV statement constructs the list in the same way but stores it in a select list variable which can be processed by a subsequent use of READNEXT. If the TO clause is omitted, the default select list (numbered list 0) is used.

The \@SELECTED variable is set to the number of records selected.

For compatibility with other database products, the action of the SSELECT statement can be changed to produce a select list variable in the same way as SSELECTV. This is achieved by setting the SSELECTV option of the $MODE compiler directive.

The optional ON ERROR clause is executed in the event of a fatal error. This covers such situations as disk hardware errors and faults in the internal structure of the file. The STATUS() function will return a value relating to the cause of the error. If no ON ERROR clause is present, a fatal error will result in an abort.

Except where the ON ERROR clause is taken, the STATUS() function will return zero.
Use of a Dynamic Array instead of a File Variable

For compatibility with Pick style environments, QM also supports a variation on \texttt{SSELECT} where the \texttt{file.var} is replaced by a dynamic array in which each field becomes an entry in the target select list.

\textbf{Example}

\begin{verbatim}
SSELECT STAFF TO 7
\end{verbatim}

This statement creates a sorted list of the records on the file with file variable \texttt{STAFF} and saves it as active select list 7.
**STATUS()**

The `STATUS()` function returns information following execution of certain other statements. In many cases, this information gives details of an error condition.

**Format**

```
STATUS()
```

The `STATUS()` function is used to fetch status information set by other statements as documented in their descriptions. Where the value relates to an error condition, the tokens in the ERR.H record of the SYSCOM file can be used. Use of the actual values of error status codes is discouraged.

The `STATUS()` function can be used any number of times to retrieve the current status value but this value may be changed by other statements. In general, the `STATUS()` function should be used as close as possible to the statement that set the value to be retrieved. In particular, use of `CALL` and `EXECUTE` are very likely to result in execution of statements that destroy the previous value of the `STATUS()` function.

There is a standard subroutine, `!ERRTEXT()`, that can be used to translate an error number to an equivalent text message.

See the `OS.ERROR()` function for a way to access operating system level error numbers.

**Example**

```
OPEN "STOCK.FILE" TO FVAR ELSE
    ABORT "Open failed : Error code " : STATUS()
END
```

This program fragment attempts to open a file and, if the `OPEN` fails, reports the error code.
STATUS

The STATUS statement returns a dynamic array containing a variety of information about an open file.

Format

```
STATUS var FROM file.var THEN statement(s) ELSE statement(s)
```

where

- `var` is the variable to receive the dynamic array.
- `file.var` is the file variable associated with an open file.

At least one of the **THEN** and **ELSE** clauses must be present for compatibility with other multivalue products. The implementation of STATUS in QM never executes the **ELSE** clause.

The STATUS statement returns a dynamic array where the fields contain the following information:

1. File position for a sequential file.
2. 1 if at end of file, else 0 (sequential files)
3. Unused on QM
4. Bytes available to read (sequential files)
5. File permission flags in the form used by Linux, etc to define access rights
6. File size
7. Number of hard links (not Windows)
8. User id of owner (not Windows)
9. Group id of owner (not Windows)
10. Inode number (not Windows)
11. Device number (not Windows)
12. Unused on QM
13. Time of last access. Seconds since midnight in user’s local time zone.
14. Date of last access. Pick style day number in user’s local time zone.
15. Time of last modification. Seconds since midnight in user’s local time zone.
16. Date of last modification. Pick style day number in user’s local time zone.
17-19 Unused on QM
20. Operating system file pathname
21. File type (see **FILEINFO()** for a list of values)
22-26 Unused on QM.
27. Operating system file pathname (same as field 20)
28-33 Unused on QM.
34. Time of last access. Epoch value (see **Dates, Times and Epoch Values**).
35. Time of last modification. Epoch value.

See also:
**FILEINFO()**
STOP

The STOP statement terminates the current program. STOPE and STOPM provide compatibility with other multivalue database products.

Format

STOP {message}

where

message evaluates to the message to be displayed.

Control is passed to the calling program, menu or paragraph.

The Pick syntax of STOP can be enabled by including a line

$MODE PICK.ERMSG

in the program before the first STOP statement. This mode is also selected automatically if there is a comma after message.

In this syntax, the STOP statement becomes

STOP {message {, arg...}}

where

message evaluates to the id of a record in the ERMSG file which holds the message to be displayed. If this id is numeric, it will be copied to @SYSTEM.RETURN.CODE.

arg... is an optional comma separated list of arguments to be substituted into the message.

See the ERMSG statement for a description of the ERMSG file message format.

The STOPE statement always uses Pick style message handling and the STOPM statement always uses Information style message handling, regardless of the setting of the PICK.ERMSG option.

Examples

IF NO.OF.ENTRIES = 0 THEN STOP

This statement terminates the program if the value of the variable NO.OF.ENTRIES is zero. No error message is printed. STOP statements without error text messages can result in difficult diagnostic work to locate faults.

OPEN "STOCK.FILE" TO STOCK ELSE
    STOP "Cannot open STOCK.FILE - Error " : STATUS()
END
This program fragment attempts to open a file named STOCK.FILE. If the open fails, the program displays an error message and terminates the program.

See also:
ABORT, ERRMSG
**STR()**

The **STR()** function returns a string made up of a given number of repeated occurrences of another string. The **STRS()** function is similar to **STR()** but operates on successive elements of a dynamic array, returning a similarly structured dynamic array of results.

**Format**

\[
\text{STR(string \{, count\})}
\]

where

- \( \text{string} \) evaluates to the string to be repeated.
- \( \text{count} \) evaluates to the number of repeats of \( \text{string} \) that are required. If omitted, this argument defaults to 1.

The **STR()** function returns \( \text{count} \) occurrences of \( \text{string} \). If \( \text{count} \) is less than one, a null string is returned.

Omitting the \( \text{count} \) argument defaults to 1 and provides an efficient way to force a data item to be stored as a string.

**Examples**

\[
\text{PRINT STR("*", 79)}
\]

This statement prints a line of 79 asterisks.

\[
\text{S = STRS('X':@VM:'Y':@SM:'Z', 2)}
\]

This statement sets \( \text{S} \) to be 
\[
\text{XX@VMYY@SMZZ}
\]

\[
\text{S = STR(N)}
\]

This statement sets \( \text{S} \) to be a string representation of the data in \( \text{N} \).

**See also:**

- **BOOL()**
- **INT()**
- **NUMERIC()**
- **SPACE()**
**SUBR()**

The **SUBR()** function calls a subroutine as a function in an expression. It is normally only used in dictionary I-type items.

**Format**

```
SUBR(name, arg1, arg2, ...)
```

where

- `name` evaluates to the name of the subroutine to be called.
- `arg1`, etc are the arguments to the subroutine.

The **SUBR()** function calls catalogued subroutine `name`, passing `arg1`, `arg2`, etc to it as its arguments. The subroutine must be written to have an additional first argument through which it returns its result which is used as the value of the **SUBR()** function.

A statement such as

```
A = B + SUBR("EVALUATE", C, D)
```

is equivalent to

```
CALL EVALUATE(X, C, D)
A = B + X
```

The `name` argument may be any expression that evaluates to the name of the subroutine. The catalogue look-up process is performed for each execution of the **SUBR()** function unlike a **CALL** statement where the look-up is performed just once for each invocation of the calling program.

When used from a dictionary I-type expression evaluated by the query processor, the @ID and @RECORD variables will contain the id and data of the record currently being processed. If the subroutine called with **SUBR()** modifies these variables, perhaps to use the **ITYPE()** function internally, the original values must be saved and reinstated if the dictionary item is to be usable within the query processor.

When used in a QMBasic program, the **SUBR()** function does not support the MAT keyword to pass a whole matrix as an argument. The **CALL** statement must be used to achieve this.

**Example**

```
SUBROUTINE CUST.ORD(RESULT, CUST.NO)
$INCLUDE FILES.H
   SELECTINDEX 'CUST', CUST.NO FROM ORDERS.F TO 1
   READLIST RESULT FROM 1 ELSE NULL
   RETURN
END
```

The above subroutine takes a customer number as its second argument and uses this to access an alternate key index, returning a list of all orders that were placed by the given customer. This example assumes that the ORDERS.F file variable is in a common block defined in the FILES.H include record and that the file is already open.
The subroutine could alternatively be written as a function:

```fortran
FUNCTION CUST.ORD(CUST.NO)
$INCLUDE FILES.H
    SELECTINDEX 'CUST', CUST.NO FROM ORDERS.F TO 1
    READLIST RESULT FROM 1 ELSE NULL
    RETURN RESULT
END
```

In either case, assuming that the subroutine or function is catalogued as CUST.ORD, it could be used from within a dictionary I-type item by use of a `SUBR()` function such as:

```fortran
SUBR('CUST.ORD', CUST.NO)
```

where CUST.NO is the name of a field within the data records being processed.
SUBROUTINE

The SUBROUTINE statement introduces a subroutine. The abbreviation SUB may be used.

Format

```
SUBROUTINE name{(arg1 {, arg2...}) [VAR.ARGS]}
```

where

- `name` is the name of the subroutine.
- `arg1`, etc are the names of the arguments to the subroutine.

QMBasic programs should commence with a `PROGRAM`, `SUBROUTINE`, `FUNCTION` or `CLASS` statement. If none of these is present, the compiler behaves as though a `PROGRAM` statement had been used with `name` as the name of the source record.

The SUBROUTINE statement must appear before any executable statements. A SUBROUTINE with no arguments is equivalent to a `PROGRAM`. The brackets are optional if there are no arguments. The SUBROUTINE statement may be split over multiple lines by breaking after a comma.

The name used in a SUBROUTINE statement need not be related to the name of the source record though it is recommended that they are the same as this eases program maintenance. The name must comply with the QMBasic name format rules.

A subroutine module is entered by referencing it a `CALL` statement. A subroutine that has no arguments can also be entered by use of the `RUN` command or by executing a command name that corresponds to the name of the program in the system catalogue.

Variable Length Argument Lists and Default Values

The number of arguments in calls to the subroutine must be the same as in the SUBROUTINE statement unless the subroutine is declared with the `VAR.ARGS` option. When `VAR.ARGS` is used, any arguments not passed by the caller will be unassigned. The `ARG.COUNT()` function can be used to determine the actual number of arguments passed. The `ARG.PRESENT()` function can be used to test for the presence of an optional argument by name. If the values of argument variables are changed by the subroutine, these changes are reflected in the variables used in the `CALL` statement that entered the subroutine.

```
SUBROUTINE CREDIT.RATING(CLIENT, CLASS, CODE) VAR.ARGS
```

In this example, if the calling program supplies only one argument, the CLASS and CODE variables will be unassigned. If the calling program provides two arguments, the default value for CLASS is ignored and the CODE variable defaults to "Standard".

When using `VAR.ARGS`, default values may be provided for any arguments by following the argument name with an `=` sign and the required numeric or string value or the @FALSE or @TRUE constants. For example,

```
SUBROUTINE CREDIT.RATING(CLIENT, CLASS = 1, CODE = "Standard") VAR.ARGS
```

In this example, if the calling program supplies only one argument, the CLASS variable will default to 1 and the CODE variable will default to "Standard". It the calling program provides two arguments, the default value for CLASS is ignored and the CODE variable defaults to "Standard".
Default argument values can also be provided when the DEFAULT.UNASS.ARGS option of the $SMODE compiler directive is enabled. In this case, the default value will be applied if the argument variable passed from the calling program is unassigned.

Pass By Reference

Subroutine arguments are normally passed by reference such that changes made to the argument variable inside a subroutine will be visible in the caller’s variable referenced by that argument. The CALL statement allows arguments to be passed by value by enclosing them in brackets. The SUBROUTINE statement also supports this dereferencing syntax. For example

SUBROUTINE INVOICE(P, (Q))

If dereferencing is used with the default argument syntax described above, the default value is placed inside the parenthesis. For example,

SUBROUTINE INVOICE(P, (Q = 7)) VAR.ARGS

Matrix Arguments

An argument may refer to a whole matrix. In this case the argument variable name must be preceded by the keyword MAT and there must be a DIM statement following the subroutine declaration to indicate whether this is a one or two dimensional matrix. Alternatively, the dimensions may be given after the variable name in the SUBROUTINE statement. In either case, the actual dimension values are counted by the compiler to establish whether the matrix has one or two dimensions but are otherwise ignored. Developers frequently use of a dimension value of one to emphasise to readers of the program that the value is meaningless. A default style (not Pick style) matrix passed as an argument can be re-dimensioned in the subroutine by use of the REDIMENSION (or REDIM) statement. Used in any other context, this statement is identical to use of DIM.

For example

SUBROUTINE MATMAX(MAX, MAT A)
  DIM A(1)
  MAX = A(1)
  N = INMAT(A)
  FOR I = 2 TO N
    IF A(I) > MAX THEN MAX = A(I)
  NEXT I
END

This subroutine scans a one dimensional matrix and passes back the value of the largest element via the MAX argument. The first two lines could alternatively be written as

SUBROUTINE MATMAX(MAX, A(1))
SUBSTITUTE()

The SUBSTITUTE() function performs substring replacement on successive elements of a dynamic array, returning a similarly structured dynamic array of results.

Format

\[
\text{SUBSTITUTE}(\text{dyn.array}, \text{old.list}, \text{new.list} \{, \text{delimiter}\})
\]

where

- \text{dyn.array} is the dynamic array to be processed.
- \text{old.list} is list of items to replace.
- \text{new.list} is list of replacement items.
- \text{delimiter} is the single character delimiter separating items in \text{old.list} and \text{new.list}. If omitted, this defaults to a value mark.

The SUBSTITUTE() function processes each element of dynamic array \text{dyn.array} constructing an equivalently structured new dynamic array result. Where an element of \text{dyn.array} contains a value in the \text{old.list}, the result contains the corresponding item from \text{new.list}. Where there is no match with an item in \text{old.list}, the source data is copied to the result dynamic array.

Although this function is defined to operate on dynamic array, it may be equally useful when \text{dyn.array} is a simple single valued string.

Example

A contains \text{DFMDFVMFVMF}

\[
\text{B} = \text{SUBSTITUTE}(\text{A}, \text{'D|P'}, \text{'Done|Pending'}, \text{'|'})
\]

B will contain \text{DoneDFVMDoneFVMPending}
SUBSTRDW(), SUBSTRDWS()

The **SUBSTRDW**() function extracts a substring based on its display width. The **SUBSTRDWS**() function is similar but operates on successive elements of a dynamic array, returning a similarly structured dynamic array of results.

**Format**

\[
\text{SUBSTRDW}(\text{string, start, length}) \\
\text{SUBSTRDWS}(\text{string, start, length})
\]

where

- **string** is the string from which a substring is to be extracted.
- **start** evaluates to the start character position. A start position less than one is treated as one.
- **length** evaluates to the length of the substring to be extracted based on its display width. A negative substring length is treated as zero.

On an ECS mode system, some characters may require two columns for display or printing. The **SUBSTRDW**() function extracts a substring that will require \( len \) columns for display. If the substring extends beyond the end of **string**, the returned data will be shorter than the limit specified by the **length** argument.

On a non-ECS system, this function is equivalent to use of the standard substring operator **string[start, length]**.

**Example**

```
I = 1
LOOP   S = SUBSTRDW(TEXT, I, 30)   UNTIL S = ""    DISPLAY S   I += LEN(S) REPEAT
```

This program fragment displays the data in TEXT as a series of lines that fit a display width of 30 columns. Note how the start position index, \( I \), is incremented by the number of characters in each substring instead of the display width.

**See also:**

**DISPLAY.WIDTH()**
**SUBSTRINGS()**

The **SUBSTRINGS()** function performs substring extraction on successive elements of a dynamic array, returning a similarly structured dynamic array of results.

**Format**

```
SUBSTRINGS(dyn.array, start, length)
```

where

- `dyn.array` is the dynamic array to be processed.
- `start` is the start position for extraction of each substring.
- `length` is the length of each extracted substring.

The **SUBSTRINGS()** function is the multivalued equivalent of the substring extraction operator `[start, length]` and processes each element of `dyn.array` in turn to produce a result dynamic array.

**Example**

A contains  `ABCDEFMFGHIJVMKLMNOVMPQRST`

```
B = SUBSTRINGS(A, 2, 3)
```

B will contain `BCDVFHJVMLMNVMQRS`
SUM()

The \texttt{SUM()} function eliminates the lowest level of a dynamic array by adding the elements to form an item of the next highest level.

**Format**

\[
\texttt{SUM}(\textit{expr})
\]

where 

\[
\textit{expr} \quad \text{evaluates to a numeric array.}
\]

The \texttt{SUM()} function identifies the lowest level elements present in \textit{expr} and forms the sum of each group of elements at this level, replacing the group with an item of the next highest level.

In a numeric array containing subvalues, the subvalues are summed to form values.

If there are no subvalues and the numeric array contains values, the values are summed to form fields.

If there are no subvalues or values, the fields are summed to form a single field.

If only one item remains, the \texttt{SUM()} function returns \textit{expr}.

Non-numeric elements of \textit{expr} are ignored.

**Example**

\[
\text{TOTAL\_PAID} = \text{SUM(PAYMENTS)}
\]

This statement sums a multivalued list of payments to form the total amount paid.

**See also:**

\texttt{SUMMATION()}
SUMMATION()

The SUMMATION() function returns the total value of all elements of a numeric array.

Format

    SUMMATION(expr)

where

    expr    evaluates to a numeric array.

The SUMMATION() function adds together all elements of expr, returning the total value. It is equivalent to repeated use of the SUM() function until just one value remains.

Non-numeric elements of expr are ignored.

Example

    TOTAL.PAID = SUMMATION(PAYMENTS)

This statement sums a multi-valued list of payments to form the total amount paid.

See also:
SUM()
SWAPCASE()

The SWAPCASE() function inverts the case of all alphabetic characters in a string.

Format

\[
\text{SWAPCASE}(\text{string})
\]

where

\[
\text{string} \quad \text{evaluates to the string in which substitution is to occur.}
\]

The SWAPCASE() function returns the value of \text{string} with all uppercase letters converted to lower case and all lowercase letters converted to uppercase. If \text{string} is a variable rather than an expression, the value of the variable is not affected.

Example

\[
\begin{align*}
S &= "\text{ABCdef}"
\text{PRINT} \text{SWAPCASE}(S)
\end{align*}
\]

This program fragment prints the string "abcDEF".

See also:

DOWNCASE(), UPCASE()
**SWAPMARKS()**

The **SWAPMARKS()** function interchanges the mark characters with the Unicode characters that they have displaced in a string.

**Format**

\[
\text{SWAPMARKS}(\text{string})
\]

where

- \( \text{string} \) evaluates to the string in which substitution is to occur.

Multivalue database systems define that characters 251 to 255 (U+00FB to U+00FF) are reserved as the mark characters (text mark, subvalue mark, value mark, field mark, item mark). The Unicode character set defines these character positions as accented characters typically found in European languages. To resolve this conflict, QM moves the accented characters internally to an alternative position in the Unicode Private Use Area to become code points U+F8FB to U+F8FF, exchanging them with their Unicode positions where necessary on external interfaces.

The **SWAPMARKS()** function allows an application to interchange these two sets of five characters where internal processing may be simplified by conforming to the Unicode character positions.

Execution of this function on a non-ECS mode system returns the original string.

**Example**

\[
\begin{align*}
S &= \text{VMPRINT}
\quad \text{SEQ}(S), \text{SEQ}(\text{SWAPMARKS}(S))
\end{align*}
\]

This program fragment prints

\[
253 \quad 63741
\]
SYSMSG()

The SYSMSG() function returns a message text from the MESSAGES file.

Format

\[
\text{SYSMSG}(\text{key \{, args...\}})
\]

where

- \text{key} identifies the message to be returned.
- \text{args} is a list of up to four argument values to be substituted into the message.

The SYSMSG() function is part of the QM multi-language message handling subsystem, returning a message in the currently selected language.

The \text{key} argument is a message number corresponding to a message stored in the MESSAGES file. The function first attempts to read this message with the prefix code for the currently selected language. If this record does not exist, it tries again with no prefix to retrieve the English language message.

The message may contain placeholders %1, %2, %3 and %4 which will be replaced with the corresponding optional \text{args} values from the SYSMSG() function. Note that the message library also contains messages that use the C programming language style %d, %s, etc placeholders. These messages are used by the QM kernel and should not be referenced in QMBasic programs.

Message numbers 10000 - 19999 are reserved for application developers.

See also:
Multi-language applications, LOAD.LANGUAGE, SET.LANGUAGE
SYSTEM()

The `SYSTEM()` function returns information regarding the status of various aspects of the system.

**Format**

```
SYSTEM(key)
```

where

- `key` identifies the information to be returned.

The `SYSTEM()` function is provided for compatibility with other data management products. Many of the `key` values correspond to those found in other multivalue database products though some values are implemented inconsistently across products. Values 1000 and above are usually specific to QM.

The following `key` values are implemented. All other `key` values return a zero value. There are tokens defined for these key values in the SYSCOM KEYS.H include record.

<table>
<thead>
<tr>
<th>Key</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Returns 1 if a PRINTER ON statement is in effect</td>
</tr>
<tr>
<td>2</td>
<td>Current page width of the default print unit (printer 0 if PRINTER ON has been done, otherwise the screen)</td>
</tr>
<tr>
<td>3</td>
<td>Current page length of the default print unit (printer 0 if PRINTER ON has been done, otherwise the screen)</td>
</tr>
<tr>
<td>4</td>
<td>Lines remaining on current page of the default print unit (printer 0 if PRINTER ON has been done, otherwise the screen)</td>
</tr>
<tr>
<td>5</td>
<td>Current page number of the default print unit (printer 0 if PRINTER ON has been done, otherwise the screen)</td>
</tr>
<tr>
<td>6</td>
<td>Current line number of the default print unit (printer 0 if PRINTER ON has been done, otherwise the screen)</td>
</tr>
<tr>
<td>7</td>
<td>Terminal type (same as <code>@TERM.TYPE</code>)</td>
</tr>
<tr>
<td>9</td>
<td>Cumulative processor time used (mS) by this QM session</td>
</tr>
<tr>
<td>10</td>
<td>Input waiting in the DATA queue? (1 if so, 0 if not)</td>
</tr>
<tr>
<td>11</td>
<td>Select list 0 active? (1 if so, 0 if not)</td>
</tr>
<tr>
<td>12</td>
<td>Time in seconds since midnight (same as <code>TIME()</code>). The default behaviour of the <code>SYSTEM(12)</code> function is to return an integer number of seconds. Enabling the TIME.MS setting of the <code>SMODE</code> compiler directive causes the return value of the <code>SYSTEM(12)</code> function to have a fractional part to millisecond precision.</td>
</tr>
<tr>
<td>13</td>
<td>Relinquish timeslice. Always returns 1.</td>
</tr>
<tr>
<td>14</td>
<td>Returns the number of characters in the type-ahead buffer. This is unreliable for piped input.</td>
</tr>
<tr>
<td>16</td>
<td>Returns the command processor level (same as <code>@LEVEL</code>)</td>
</tr>
<tr>
<td>19</td>
<td>Returns a unique value formed from the internal form date and time in the user’s time zone as two five digit numbers. If the function is used more than once system wide in</td>
</tr>
</tbody>
</table>
the same second, an alphabetic suffix is added to create a unique value. This function is unreliable in applications where users span multiple time zones.

20 Returns the system wide reference number for last print job passed to the operating system spooler (Linux only). See also the PUSSEQNO key to GETPU().

21 Returns a unique value formed from the internal form date and a sequential number that resets each day.

22 User number (same as @USERNO)

23 Break key enabled? (1 if so, 0 if not)

24 Input echo enabled? (1 if so, 0 if not)

25 Is this a phantom process?

26 Returns the current input prompt character

27 Returns the operating system uid for the user's process. (Not Windows)

28 Returns the operating system effective uid for the user's process. (Not Windows)

29 Returns the operating system gid for the user's process. (Not Windows)

30 Returns the operating system effective gid for the user's process. (Not Windows)

31 Licence number

32 Returns the system (QMSYS) directory pathname

33 Returns the name of the program that called this subroutine

35 Returns current number of interactive users

38 Returns temporary directory pathname

42 Returns telnet connection IP address, null for a console user

45 Returns the program catalogue name or pathname if not catalogued.

91 Returns 1 on Windows, 0 on other platforms

1000 Returns 1 if EXECUTE CAPTURING is in effect, 0 otherwise

1001 Returns 1 if case inversion is enabled, 0 otherwise

1002 Returns the program call history. This is a dynamic array in which each program is represented by a field, the current program being in field 1. The first value in each field contains the program name. Subsequent values are divided into two subvalues containing the program address and line number (where available) for each internal subroutine call (GOSUB) in the program. The first value in each field has an additional third subvalue containing the compile time of the program as an epoch value.

1003 Returns a dynamic array containing a list of open files. Each field has two values; the first holds the internal file number, the second holds the file's pathname.

1004 Returns the peak number of files that have been open at one time since QM was started.

1005 Returns the combined date and time value as DATE() * 86400 + TIME() based on the user's time zone.

1006 Returns 1 if running on a Windows NT style system (XP and later). This will always be true on Windows as earlier releases are no longer supported.
1007 Returns the current transaction number, zero if not in a transaction.
1008 Returns the current transaction nesting level, zero if not in a transaction.
1009 Returns the system byte ordering, 1 for high byte first, 0 for low byte first.
1010 Returns the platform name; Windows, Linux, FreeBSD, AIX, Mac, PDA, Solaris, Rpi.
1011 Returns the pathname of the QM configuration file
1012 Returns the QM version number.
1013 Returns user limit, excluding users reserved for phantom processes.
1014 Returns user limit, including users reserved for phantom processes.
1015 Returns the name of the host computer system.
1016 Returns the remaining number of licensed non-phantom users.
1017 Returns the tcp/ip port number for a socket connection.
1018 Returns the device licensing connection limit.
1019 Returns the offset in seconds of the user's local time zone from UTC. This will be negative for zones to the west of longitude zero.
1020 Returns the time of day in milliseconds since midnight UTC.
1021 Returns the currently selected ECS base map name. On a non-ECS system, this returns the current map name, null if using the default map.
1022 Returns the unique qmid value for a user mode installation of QM, zero for a standard installation.
1023 Returns True on 64 bit systems, False on 32 bit systems.
1024 Returns the current working directory pathname when QM was entered.
1025 Returns a dynamic array where field 1 is a multivalued list of environment variable names and field 2 is a corresponding list of their values.
1026 Returns xxx when QM is entered using "qm xxx".
1027 Returns the name of the serial port when logged in on a serial connection.
1028 Returns the system id of the active QM licence, zero if the licence is not system specific.
1029 Returns the current internal subroutine depth.
1030 Returns login time as date * 86400 + time in the user's local time zone using Pick date numbering.
1031 Returns operating system process id.
1032 Returns and clears the break pending flag, set if the break key is pressed with breaks disabled.
1033 Returns True if a COMO file is active.
1034 Returns True if COMO output is active but suspended.
1035 Returns the time as seconds since 00:00:00 on January 1 1970, Universal Coordinated Time (UTC) independent of the user's time zone. Same as the EPOCH() function.
1036 Returns the period in tenths of a second for which login was delayed waiting for a spare user. See the **LGNWAIT** configuration parameter.

1037 Returns a dynamic array with one field for each open QM file. In each field, value 1 holds the internal file number, value 2 holds the file pathname and value 3 is a subvalue mark delimited list of QM user numbers for which the file is open.

1038 Returns login time as an epoch value.

1040 Returns True if operating system filenames are case insensitive.

1041 Used on a replication publisher system, returns a field mark delimited list of replication subscribers actively transferring data.

1042 Returns the system id of the server. This will differ from SYSTEM(1028) for a USB installation or a licence that is not system specific.

1043 Returns the current CSV mode that determines quoting rules as set by **CSV.MODE**.

1044 Returns True if the system is running in ECS mode.

1045 Returns time at which QM was started as an epoch value.

1046 Returns the current CSV delimiter character set by **CSV.MODE**.

1047 Returns a globally unique identifier (GUID) string. Platforms that do not support this operation return a null string.

1048 Returns a rolling sequence number that is unique across all sessions until QM is restarted when it is reset to 1.

1049 Returns True if **HUSH** is in effect, otherwise returns False.

1050 Returns True if the current user is a QM system administrator, otherwise returns False. This will be True if any of...
   a) The user is an administrator at the operating system level.
   b) The user logs in via a Windows QMConsole session.
   c) The user is defined as an administrator using the **ADMIN.USER** command.
   d) QM security is disabled (see the **SECURITY** command).

1051 Returns True if QM is running as a USB installation.

1052 Returns count of direct descendant child (phantom) processes.

1053 Returns the highest user number that can be assigned on this system.

1054 Returns field mark delimited list of active packages.

1055 Returns a field mark delimited dynamic array containing statistical information related to the **LGNWAIT** configuration parameter:
   1 The number of times login has been delayed.
   2 The number of successful logins.
   3 The number of processes currently delayed waiting for login.
   4 The peak number of processes waiting for login.
   5 The number of login attempts that failed after waiting.

1056 Returns a field mark delimited dynamic array containing user count information:
   1 The number of interactive users.
   2 The number of QMClient users.
   3 The number of interactive phantom users.
   4 The number of non-interactive phantom users.
   5 The peak number of concurrent licensed users.

1057 Returns the parent process user number, zero if the current process is not a phantom or the parent has terminated.
QM allows users to add definitions for their own `SYSTEM()` function `key` values by writing a QMBasic subroutine that performs whatever processing is required. This subroutine takes two arguments. The first is used to return the result and the second is the `key` value passed in. This subroutine must be catalogued as `$SYSTEM`. 

```text
1058 Returns a field mark delimited list of child phantom process user numbers.
1059 Returns the system licence site text.
1060 Returns the pool name for a QMClient connection pooling session.
1061 Returns QMSYS drive letter on Windows, null elsewhere. Same as `@DRIVE`
1062 Returns and clears cross-process application signal state.
1065 Returns diagnostic data from the transaction cache as a dynamic array. Each field represents a cached action and is formed from four value mark separated items; the transaction number, file number, id and mode (`D` = delete, `W` = write). Use `SYSTEM(1003)` to translate file numbers to pathnames.
1066 Returns last form queue number assigned with `SP.ASSIGN`.
1067 Returns default group size in bytes.
1068 Returns Proc output buffer status. True if secondary buffer is active.
1069 Returns Proc input buffer status. True if secondary buffer is active.
1070 Returns the current `PRECISION` setting.
1071 Returns a dynamic array of file information after `OSREAD`.
1072 Returns the IP address on which QM listens for incoming telnet, QMClient, QMNet and replication connections. Null if listening on all configured IP addresses. See the `IPADDR` configuration parameter.
1073 Returns a field mark delimited list of grouped child phantom process user numbers.
1075 Returns a dynamic array of connection pooling configuration information, one field per pool with values for name, connection limit, timeout and current idle process count.
1076 Returns the current `CALL` depth.
1077 Returns a field mark delimited list of group names to which the user belongs (not Windows).
1078 Returns a field mark delimited list of group numbers to which the user belongs (not Windows).
1079 Is this a QMClient process?
1080 Returns True if program was started directly or indirectly from a Proc, else returns False.
1081 Returns epoch time as milliseconds since 1 Jan 1970 UTC.
1082 Returns time of day as milliseconds since midnight in user's time zone.
1083 Returns information about the last deadlock as a dynamic array with a field for each process in the deadlock, divided into three values as the user number, file pathname and record id. The record id is omitted for a file lock.
1086 Returns True if the program has case sensitivity for string comparisons enabled, otherwise returns False.
```
**TAN()**

The **TAN()** function returns the tangent of a value.

**Format**

```
TAN(expr)
```

where

```
expr evaluates to a number or a numeric array.
```

The **TAN()** function returns the tangent of *expr*. Angles are measured in degrees.

If *expr* is a numeric array (a dynamic array where all elements are numeric), the **TAN()** function operates on each element in turn and returns a numeric array with the same structure as *expr*.

**Example**

```
OPP = ADJ * TAN(ANGLE)
```

This statement finds the length of the opposite side of a right angled triangle from the length of the adjacent side and the angle between it and the hypotenuse.

**See also:**

**ACOS()**, **ASIN()**, **ATAN()**, **COS()**, **SIN()**
TCLREAD

The TCLREAD statement retrieves the sentence that started the current program.

Format

    TCLREAD var

where

    var    is the variable to receive the sentence.

The TCLREAD statement is an alternative to use of the @SENTENCE variable.
TERMINFO(), TERMINFO.EX()

The TERMINFO() function returns information from the terminfo database. The TERMINFO.EX() function returns extended data.

Format

TERMINFO()
TERMINFO(cap.name)

TERMINFO.EX()
TERMINFO.EX(cap.name)

where

cap.name evaluates to the name of a terminfo capability.

The TERMINFO() function enables programs to examine the terminfo database to establish capabilities of the currently selected terminal type.

In the first form, TERMINFO() returns a dynamic array containing a wide range of capability information about the terminal. The structure of this dynamic array is defined in the TERMINFO.H include record in the SYSCOM file. Additional entries may be added in future releases but existing entries will not be moved. The terminfo database capability name to which each item corresponds is shown in the comment text in this include record.

The second form of the TERMINFO() function returns the value of the named capability. The cap.name argument should evaluate to a capability name as used in terminfo source files. This name is case sensitive. Unrecognised capabilities and those for which the terminfo database has no entry will be returned as null strings.

QM extends the terminfo system to allow a string capability to have multiple definitions as alternative codes that might be received from the terminal device. For compatibility with earlier releases, the TERMINFO() function returns only the first string associated with a capability name. The TERMINFO.EX() function returns all strings defined for the capability, separating them by value marks.

As an example, the vt100-at terminfo entry has two alternative strings defined for the kmous capability, corresponding to a single click and a double click of the left mouse button. A program using the KEYCODEV() function to decode incoming data would see both of these as KV$MOUSE.

See also:
TERM, The Terminfo database, @(), KEYCODEV()
TESTLOCK()

The TESTLOCK() function returns the state of a task lock.

Format

\[
\text{TESTLOCK}(\text{lock.num})
\]

where

\[
\text{lock.num}
\]
evaluates to the lock number in the range 0 to 63.

The TESTLOCK() function enables programs to test which user, if any, owns a specific task lock. If the lock is currently owned by a QM user, the TESTLOCK() function returns their user number. If the lock is available, zero is returned.

See also:
CLEAR.LOCKS, LIST.LOCKS, LOCK (Command), LOCK (QMBasic), UNLOCK
THROW

The **THROW** statement throws an exception that can be caught in a **TRY / CATCH** construct.

**Format**

```plaintext
THROW exception {, data}
```

where

- `exception` is the case insensitive name of the exception to be thrown. This may be a quoted constant value or an expression that evaluates to the exception name.

An **exception** is a named event, typically an error, raised within an application by use of the **THROW** statement and caught by a **TRY/CATCH** construct elsewhere in the application. The actual name has no significance to QM but should ideally relate to the situation that the exception handles. See **Exception Handling** for further information on exception names.

The **@EXCEPTION** variable is set to contain the name of the exception being thrown and **@EXCEPTION.ORIGIN** is set to a dynamic array in which field 1 holds the program name and field 2 holds the line number at which the exception was thrown. If the program has no cross-reference tables, the line number will be -1.

The optional `data` element of this statement is an expression that will be evaluated and stored in **@EXCEPTION.DATA**. This may be a simple data item such as a number or a string, or it can be a complex data item such as a file variable, a socket or an instance of a class. In the case of complex items, it may be necessary for the exception handler to clear **@EXCEPTION.DATA** after any processing so that, for example, the file associated with a file variable can be closed.

If the data element is omitted, **@EXCEPTION.DATA** is set to a null string.

If an exception is thrown for which there is no corresponding **TRY/CATCH** exception handler, an abort occurs.

**Example**

```plaintext
THROW 'ACCOUNT.INVALID'
```

The above statement throws an exception named ACCOUNT.INVALID.

**See also:**

**TRY/CATCH**
**TIME()**

The `TIME()` function returns the current time as the number of seconds since midnight.

**Format**

```
TIME()
```

The `TIME()` function returns the number of seconds since midnight within the user’s time zone as an integer value. The `OCONV()` function can be used to format this in a number of ways for display.

The default behaviour of the `TIME()` function is to return an integer number of seconds. Enabling the `TIME.MS` setting of the `$MODE` compiler directive causes the return value of the `TIME()` function to have a fractional part to millisecond precision.

See the `SYSTEM(1082)` function for a way to return the time of day as an integer number of milliseconds since midnight.

**Example**

```
DISPLAY OCONV(TIME(), "MTS")
```

This statement displays the time in the form `hh:mm:ss` using the 24 hour clock.

**See also:**

`@TIME, DATE(), EPOCH()`
TIMEDATE()

The **TIMEDATE()** function returns the current time and date as a string.

**Format**

```
TIMEDATE()
```

The **TIMEDATE()** function returns the current time and date as a 20 character string in the form

```
hh:mm:ss dd mmm yyyy
```

where

- **hh**: hours in 24 hour format, zero filled
- **mm**: minutes, zero filled
- **ss**: seconds, zero filled
- **dd**: day of month, zero filled
- **mmm**: first three letters of the month name, first letter uppercase
- **yyyy**: year

**Example**

```
DISPLAY @(60, 0) : TIMEDATE()
```

This statement displays the time and date at the top right of the display.
TIMEOUT

The **TIMEOUT** statement sets a timeout for **READBLK** and **READSEQ**.

**Format**

```
TIMEOUT file.var, interval
```

where

- **file.var** is the file variable associated with a file opened using **OPENSEQ**.
- **interval** is the timeout period in seconds. A negative value disables the timeout.

The **TIMEOUT** statement can be used when **OPENSEQ** is used to open a FIFO (named pipe) or a serial port. If no input is received by **READBLK** or **READSEQ** in the given **interval**, the read terminates.

The **TIMEOUT** statement is ignored on Windows systems and for files that are not FIFOs or serial ports.
TOTAL()

The TOTAL() function accumulates totals for use with the CALC query processor keyword. It is only available in dictionary I-type items.

Format

TOTAL(expr)

where

expr is an expression.

The TOTAL() function can be used in dictionary I-type expressions. While processing the detail lines of a report, the TOTAL() function returns the value of the expression but also accumulates a running total internally. When the query includes fields prefixed by the CALC keyword, the expression is re-evaluated on the total lines of the report using the accumulated value in place of the TOTAL() function.
TRANS(), RTRANS(), XLATE()

The TRANS() function returns a field or the entire record from a named data file. It is normally only used in dictionary I-type items. The synonym XLATE() may be used.

The RTRANS() function is similar but has a slight difference described below for closer compatibility with some other environments.

Format

\[
\text{TRANS}([\text{DICT}] \text{ file.name}, \text{ record.id, field, action})
\]

\[
\text{RTRANS}([\text{DICT}] \text{ file.name, record.id, field, action})
\]

where

- **file.name** is the name of the file from which data is to be retrieved. The allowable forms of the file.name differ between use in a dictionary I-type or use in a QMBasic program:
  - In a QMBasic program, file.name is evaluated in the same way as any other expression.
  - In a dictionary I-type record, the file.name may be specified with or without quotes. If file.name is an expression that evaluates to the file name, it must be enclosed in parentheses.
  - In all usage, if the evaluated file name commences with DICT (case insensitive) separated from the actual name by a single space, the dictionary of the named file is used.

- **record.id** evaluates to the id of the record to be retrieved. When used in a QMBasic program, this must be the actual record id. When used in an I-type dictionary expression, this may be
  - the name of a D or I-type item defined in the same dictionary which contains the id of the record to be retrieved.
  - a literal record id enclosed in quotes.

- **field** an expression that evaluates to the field number of the field to be returned. A field value of zero returns the record id and can be used to check the existence of a record. A field value of -1 indicates that the entire record is to be returned. When used in a dictionary I-type expression this can also be
  - A data item (A/C/D/E/I/S-type) defined in the target file's dictionary. If the field name does not follow the rules for construction of QMBasic variable names, it must be quoted. This form is not valid when the file.name is an expression.
  - A field number
  - @RECORD or -1 to return the entire record.

If field is given as expression that evaluates to the field position, this must be enclosed in brackets to avoid potential syntactic ambiguity.
**action** determines the action taken if the file cannot be opened, the record does not exist, or the required field is null. This may evaluate to:

- **C**: Return the record id.
- **V**: Print a warning message and return a null value.
- **X**: Return a null value (default).

In addition,

- **B**: Causes the file to be opened in binary mode with mark mapping suppressed (directory files only).

The **TRANS()** function returns the specified data with any mark characters lowered by one level (e.g. value marks become subvalue marks).

If **record.id** is multivalued, the **TRANS()** function extracts each requested record and returns a multivalued result with the data from each record separated by a value mark.

The **RTRANS()** function is identical to **TRANS()** except that it does not lower the mark characters. This makes it impossible to distinguish between the results of retrieving a multivalued field from a single record and retrieving a single valued field from multiple records.

**Examples**

```
TOTAL.VALUE = QTY * TRANS('STOCK', PART.NO, 'PRICE', 'X')
```

The above statement used in a dictionary I-type record reads from the STOCK file a record (or list of records) whose id(s) can be found in the PART.NO variable. The X error code causes the **TRANS()** function to return a null value for any record that cannot be found. When using **TRANS()** in a QMBasic program, the **field** argument must be an expression that evaluates to the field number. For example:

```
TOTAL.VALUE = QTY * TRANS('STOCK', PART.NO, 6, 'X')
```

```
X = TRANS(DICT 'ORDERS', 'DISCOUNT', -1, 'X')
X = TRANS('DICT ORDERS', 'DISCOUNT', -1, 'X')
```

Both of the above statements perform the same action. Either might be used, for example, to retrieve a record named DISCOUNT from the dictionary of the ORDERS file.
TRANSACTION ABORT, TRANSACTION COMMIT, TRANSACTION START

The TRANSACTION START/COMMIT/ABORT statements provide an alternative to use of the BEGIN TRANSACTION, COMMIT, ROLLBACK and END TRANSACTION statements.

Format

TRANSACTION START
THEN {statements}
ELSE {statements}

TRANSACTION COMMIT
THEN {statements}
ELSE {statements}

TRANSACTION ABORT

A transaction is a group of updates that must either be performed in their entirety or not at all. The TRANSACTION START statement starts a new transaction. All updates within the transaction are cached and only applied to the database when the TRANSACTION COMMIT statement is executed. Execution of the program then continues at the statement following the TRANSACTION COMMIT.

The TRANSACTION ABORT statement terminates the transaction, discarding any cached updates. Execution continues at the statement following the TRANSACTION ABORT.

The THEN and ELSE clauses are optional and are provided for compatibility with other products. Within QM any errors occurring in a TRANSACTION START or TRANSACTION COMMIT will result in run time errors.

Deletes and writes inside a transaction will fail unless the program holds an update lock on the record or the file. All locks obtained inside the transaction are retained until the transaction terminates and are then released. Locks already owned when the transaction begins will still be present after the transaction terminates, even if the record is updated or deleted within the transaction.

Closing a file inside a transaction appears to work in that the file variable is destroyed though the actual close is deferred until the transaction terminates and any updates have been applied to the file. Rolling back the transaction will not reinstate the file variable.

Access to indices using SELECTINDEX, SELECTLEFT or SELECTRIGHT inside a transaction will not reflect any updates within the transaction as these have not been committed.

Updates to sequential records opened using OPENSEQ are not affected by transactions.

Transactions may be nested. If the TRANSACTION START statement is executed inside an active transaction, the active transaction is stacked and a new transaction commences. Termination of the new transaction reverts to the stacked transaction. The default behaviour of QM is that transactions are durable such that updates in a child transaction are applied to the data files on use of TRANSACTION COMMIT. The NON.DURABLE.TXN setting of the QMBasic $MODE compiler directive, makes transactions non-durable such that updates in a child transaction are inherited by the parent transaction on commit.

The following operations are banned inside transactions:
  CLEARFILE
  PHANTOM
Example

TRANSACTION START
READU CUST1.REC FROM CUST.F, CUST1.ID ELSE
   TRANSACTION ABORT
RETURN
END
CUST1.REC<C.BALANCE> -= TRANSFER.VALUE
WRITE CUST1.REC TO CUST.F, CUST1.ID

READU CUST2.REC FROM CUST.F, CUST2.ID ELSE
   TRANSACTION ABORT
RETURN
END
CUST2.REC<C.BALANCE> += TRANSFER.VALUE
WRITE CUST2.REC TO CUST.F, CUST2.ID
TRANSACTION COMMIT

The above program fragment transfers money between two customer accounts. The updates are only committed if the entire transaction is successful.

See also:
Transactions, BEGIN TRANSACTION
TRANSLITERATE()

The `TRANSLITERATE()` function returns a transliterated version of a character string containing only 8 bit characters.

**Format**

```
TRANSLITERATE(string)
```

where

*string* is the string to be transliterated.

The `TRANSLITERATE()` function takes a string that may contain ECS characters and returns a version containing only 8 bit characters. The replacements used for this transformation are determined by the transliteration values defined in the active ECS character map. Characters for which no transliteration has been provided are replaced by a period.

In a non-ECS mode system, the `TRANSLITERATE()` function returns the string that is passed as its argument.

**Example**

```
S = 'Müller'
DISPLAY TRANSLITERATE(S)
```

In this program fragment, variable S contains a string that includes the German u-umlaut character. If the character maps are set up such that this character transliterates to “ue”, the output would be

Mueller

Note that in an ECS mode system the u-umlaut, which is normally defined as being character 252 in the 8 bit character set, has been moved to become ECS codepoint U+F8FC to allow the five mark characters reserved by the multivalue data model to remain in their traditional positions as characters 251 to 255.
TRIM()

The TRIM() function removes excess characters from a string.

Format

\[ \text{TRIM}(\text{string}) \]
\[ \text{TRIM}(\text{string}, \text{character}, \text{mode}) \]

where

\text{string} evaluates to the string to be trimmed.
\text{character} is the character to be removed
\text{mode} evaluates to a single character which determines the mode of trimming:

- A: Remove all occurrences of \text{character}.
- B: Remove all leading and trailing occurrences of \text{character}.
- C: Replace multiple instances of \text{character} with a single character.
- D: Remove all leading and trailing spaces, replacing multiple embedded spaces with a single space. The value of \text{character} is ignored.
- E: Remove all trailing spaces. The value of \text{character} is ignored.
- F: Remove all leading spaces. The value of \text{character} is ignored.
- L: Remove all leading occurrences of \text{character}.
- R: Remove all leading and trailing occurrences of \text{character}, replacing multiple embedded instances of \text{character} with a single character.
- T: Remove all trailing occurrences of \text{character}.

The first format of the TRIM() function removes all leading and trailing spaces from \text{string} and replaces multiple embedded spaces by a single space.

The second form is more generalised and allows other characters to be removed.

Examples

\[ X = \text{" 1 2 3 "} \]
\[ Y = \text{TRIM}(X) \]

This program fragment removes excess spaces from string \(X\) setting \(Y\) to "1 2 3"

\[ X = \text{"ABRACADABRA"} \]
\[ Y = \text{TRIM}(X, \text{\'A\'}, \text{\'A\'}) \]

This program fragment removes all occurrence of the letter A from string \(X\) setting \(Y\) to "BRCDBR"

\[ X = \text{"ABRACADABRA"} \]
\[ Y = \text{TRIM}(X, \text{\'A\'}, \text{\'B\'}) \]

This program fragment removes all occurrence of the letter A from string \(X\) setting \(Y\) to "BRCDBR"
This program fragment removes leading and trailing occurrences of the letter A from string X setting Y to "BARCADABR"

See also: TRIMB(), TRIMF(), TRIMS(), TRIMW()
**TRIMB()**

The TRIMB() function removes excess spaces from the back of a string. The TRIMBS() function is similar to TRIMB() but operates on each element of a dynamic array and returns an equivalently structured dynamic array of trimmed strings.

**Format**

\[ \text{TRIMB}(\text{string}) \]

where

\[ \text{string} \] evaluates to the string to be trimmed.

The TRIMB() function removes all trailing spaces from \[ \text{string} \].

**Examples**

\[
\begin{align*}
A &= " \ 1 \ 2 \ 3 \ " \\
B &= \text{TRIMB}(A)
\end{align*}
\]

This program fragment removes excess spaces from string A setting B to " 1 2 3"

\[
\begin{align*}
A &= " \ 1 \ 2 \ 3 \ : \ @FM \ : \ " \ 4 \ 5 \ 6" \\
B &= \text{TRIMBS}(A)
\end{align*}
\]

This program fragment is similar to the previous example but it shows the way in which TRIMBS() operates on the two fields separately. B becomes " 1 2 3FM 4 5 6"

**See also:**

TRIM(), TRIMF(), TRIMS(), TRIMW()
**TRIMF()**

The TRIMF() function removes excess spaces from the front of a string. The TRIMFS() function is similar to TRIMF() but operates on each element of a dynamic array and returns an equivalently structured dynamic array of trimmed strings.

**Format**

```
TRIMF(string)
```

where

```
string
```
evaluates to the string to be trimmed.

The TRIMF() function removes all leading spaces from `string`.

Where `string` is delimited by mark characters, the TRIMF() function works on each delimited substring as a separate item.

**Examples**

```
A = "1  2  3  
B = TRIMF(A)
```

This program fragment removes excess spaces from string A setting B to "1  2  3  

```
A = "1  2  3  : @FM : " 4  5  6"
B = TRIMFS(A)
```

This program fragment is similar to the previous example but it shows the way in which TRIMFS() operates on the two fields separately.

B becomes "1  2  3  " 4  5  6  

**See also:**

TRIM(), TRIMB(), TRIMS(), TRIMW()
TRIMS()

The TRIMS() function removes excess characters from strings in a dynamic array, operating on each element in turn and returning an equivalently structured dynamic array of trimmed strings.

Format

TRIMS(string)

TRIMS(string, character{, mode})

where

string evaluates to the string to be trimmed.

calendar is the character to be removed

mode evaluates to a single character which determines the mode of trimming:

A Remove all occurrences of calendar.

B Remove all leading and trailing occurrences of calendar.

C Replace multiple instances of calendar with a single character.

D Remove all leading and trailing spaces, replacing multiple embedded spaces with a single space. The value of calendar is ignored.

E Remove all trailing spaces. The value of calendar is ignored.

F Remove all leading spaces. The value of calendar is ignored.

L Remove all leading occurrences of calendar.

R Remove all leading and trailing occurrences of calendar, replacing multiple embedded instances of calendar with a single character.

T Remove all trailing occurrences of calendar.

The first format of the TRIMS() function removes all leading and trailing spaces from each dynamic array element of string and replaces multiple embedded spaces by a single space.

The second form is more generalised and allows other characters to be removed.

Example

A = " 1 2 3 " : @FM : " 4 5 6"

B = TRIMS(A)

B becomes "1 2 3FM4 5 6"

See also:

TRIM(), TRIMB(), TRIMF(), TRIMW()
**TRIMW(), TRIMWS()**

The **TRIMW()** function removes excess whitespace characters from a string. The **TRIMWS()** function is similar to **TRIMW()** but operates on each element of a dynamic array and returns an equivalently structured dynamic array of trimmed strings.

**Format**

```
TRIMW(string)
```

where

```
string       evaluates to the string to be trimmed.
```

The **TRIMW()** function removes all leading and trailing whitespace from *string* and replaces multiple embedded whitespace characters with a single space (character 32).

In the non-ECS version of QM, whitespace corresponds to the space and tab characters. In the ECS version, the whitespace attribute is defined by the currently loaded character map.

**Example**

```
X = "  1" : CHAR(9) : "2  3  ":
Y = TRIMW(X)
```

This program fragment removes excess spaces from string X setting Y to "1 2 3"

**See also:**

TRIM(), TRIMB(), TRIMF(), TRIMS()
TRY / CATCH

The TRY/CATCH construct allows creation of exception handlers in a manner broadly similar to that of some other programming languages.

Format

```
TRY {statement}
   {statements}
CATCH exception {qualifiers}[, exception {qualifiers}...}
   {statements}
CATCH exception {qualifiers}[, exception {qualifiers}...}
   {statements}
END
```

where

- `statement(s)` are any executable QMBasic statements.
- `exception` is an exception name. The name may optionally be enclosed in quotes. If unquoted, the name must comply with QMBasic name format rules.
- `qualifiers` are chosen from:
  - **DUMPING**: Generates a process dump file when the exception is thrown.
  - **SAVING.STACK**: Sets @EXCEPTION.STACK to contain the stack history in the same form as use of SYSTEM(1002). This can also be set by default by use of the EXCEPTION.SAVE.STACK option of the $MODE compiler directive.

A single TRY construct may have multiple CATCH clauses.

An exception is a named event, typically an error, raised within an application by use of the THROW statement. The actual name has no significance to QM but should ideally relate to the situation that the exception handles. Exception names are case insensitive and may be up to 63 characters.

A TRY/CATCH construct executes the statement(s) in the TRY clause, monitoring for named exceptions being thrown.

On throwing an exception, the application exits from all lower level programs, subroutines, etc and continues execution in the CATCH clause that handles this exception. Where an object oriented programming object is discarded by an exception, the DESTROY.OBJECT subroutine will be executed in the normal way.

Whilst execution of subroutines that return is acceptable within the TRY clause, it is essential that programs do not jump into or out of this clause. Failure to follow this rule may have undesirable results.

A single CATCH clause may catch multiple exception names. On arrival in the CATCH clause, the @EXCEPTION variable will contain the name of the exception that has been caught and the @EXCEPTION.DATA variable will contain any data associated with this exception. The
@EXCEPTION.ORIGIN variable is set to a dynamic array in which field 1 holds the program name from which the exception was thrown and field 2 holds the line number in that program. If the program has no cross-reference tables, the line number will be -1.

If the exception name in the CATCH clause is followed by the DUMPING or SAVING.STACK options, the additional diagnostic data is generated when the exception is thrown, before unwinding the call stack to the exception handler.

Examples

LOOP
  READNEXT ACC.ID ELSE EXIT
  TRY
    CALL PROCESS.ACCOUNT
    CATCH ACCOUNT.INVALID
      DISPLAY 'Account ' : ACC.ID : ' is invalid'
    END
  END
REPEAT

The above rather simple example shows a program fragment that processes successive items from a select list. If the PROCESS.ACCOUNT subroutine, or any lower level action called from it, throws an ACCOUNT.INVALID exception, the program will continue execution at the DISPLAY statement.

TRY
  CALL MYSUB
  CATCH TERMINATE DUMPING
    DISPLAY 'Process status dumped'
  END

The above example establishes an exception handler around a call to MYSUB. If the user defined TERMINATE exception occurs, a process dump file will be created prior to arrival in the CATCH clause.

A useful report of the call stack can be produced with the code fragment below executed from the CATCH clause. This is also available as a standard catalogued QMBasic subroutine named !EXC.STACK.

IF @EXCEPTION.STACK # ' ' THEN
  DISPLAY ' Program name............... Addr.. Line'
  LEVEL = 1
  FOR EACH PROG IN @EXCEPTION.STACK DELIMITER @FM
    NAME = FMT(PROG<1,1>, '30L')
    DISPLAY FMT(LEVEL, 'R%3 ') :
    SUBDATA = FIELD(PROG<1>, @VM , 2, *)
    SUBADDR = VSLICE(SUBDATA, 0, 1)
    SUBLINE = VSLICE(SUBDATA, 0, 2)
    LOOP
      DISPLAY FMT(REMOVE(NAME, MORE.NAME), '30L') : ' ' :
      DISPLAY DTX(REMOVE(SUBADDR, MORE.ADDR), 6) : ' ' :
      DISPLAY REMOVE(SUBLINE, MORE.LINE)
      WHILE MORE.NAME OR MORE.ADDR OR MORE.LINE
      DISPLAY ' ' :
    REPEAT
LEVEL += 1
NEXT PROG
END

See also:
Exception handling, CAUGHT(), EXC_STACK, THROW
TTYGET()

The TTYGET() function returns a dynamic array containing the current terminal settings.

Format

TTYGET()

The TTYGET() function allows an application that alters terminal settings to read and save the original terminal settings for restore on exit.

The dynamic array currently contains the fields listed below. Further fields may be added in future.

<table>
<thead>
<tr>
<th>Field</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ctrl-C treated as the break key? (PTERM BREAK mode)</td>
</tr>
<tr>
<td>2</td>
<td>Case inversion on? (PTERM CASE mode)</td>
</tr>
<tr>
<td>3</td>
<td>Break character value (PTERM BREAK n)</td>
</tr>
<tr>
<td>4</td>
<td>Output newline sequence (PTERM NEWLINE)</td>
</tr>
<tr>
<td>5</td>
<td>Input return key code (PTERM RETURN)</td>
</tr>
</tbody>
</table>

See also:
PTERM, TTYSET
TTYSET

The **TTYSET** statement sets the terminal modes.

**Format**

```
TTYSET var
```

where

```
var
```

is a dynamic array of terminal mode settings.

The **TTYSET** statement allows an application that alters terminal settings to restore previously saved settings on exit.

The format of the dynamic array *var* is described under the **TTYGET()** function. Because this dynamic array may be extended in future releases, programs must ensure that any additional fields returned by **TTYGET()** are restored on use of **TTYSET()**.

See also:

**PTERM, TTYGET()**
UNASSIGNED()  

The UNASSIGNED() function tests whether a variable is unassigned.

**Format**

```
UNASSIGNED(var)
```

where

- `var` is the variable to be tested.

All QMBasic variables except those in common blocks are initially unassigned. Any attempt to use the contents of the variable in an expression would cause a run time error until such time as a value has been stored in it. The UNASSIGNED() function allows a program to test whether a variable is unassigned, returning True if it is unassigned or False if it is assigned.

**Example**

```
SUBROUTINE VALIDATE(ACCOUNT.CODE, ERROR)
BEGIN CASE
   CASE UNASSIGNED(ACCOUNT.CODE)
      ERROR = 1
   CASE ACCOUNT.CODE MATCHES '3N-5N'
      ERROR = 2
      ...etc...
   CASE 1
      ERROR = 0
END CASE
RETURN
END
```

This program fragment validates an account code. The use of the UNASSIGNED() function prevents an abort if the variable has not been assigned.

See also: ASSIGNED()
UNLOCK

The **UNLOCK** statement releases one of 64 system wide task locks.

**Format**

```
UNLOCK \{ lock.num \} \{ THEN statement(s) \} \{ ELSE statement(s) \}
```

where

- `lock.num` evaluates to the lock number in the range 0 to 63. If omitted or specified as a negative value, all task locks owned by the process will be released.
- `statement(s)` are statements to be executed depending on the outcome of the **UNLOCK** operation.

The **THEN** and **ELSE** clauses are both optional.

The **UNLOCK** statement releases the specified task lock if it has previously been acquired using the **LOCK** statement. There is no means for a program to determine which task locks are held by the user except by attempting to lock each in turn and checking the **STATUS()** value. Beware that unlike read, update and file locks, task locks are only automatically released on leaving QM, not on return to the command prompt.

When `lock.num` is omitted or evaluates to a negative number, all task locks owned by the process are released. This action is always successful, even if no locks are owned.

The **THEN** clause is executed if the lock is held by this process. The value of the **STATUS()** function will be zero.

The **ELSE** clause is executed if the lock is not owned by this process. The value of the **STATUS()** function will be ER$LCK if the lock is owned by another process or ER$NLK if it is not owned by any process.

**Example**

```
LOCK 7 THEN
...processing statements...
UNLOCK 7
END
ELSE ABORT "Cannot obtain task lock"
```

This program fragment obtains task lock 7, performs some critical processing and then releases the lock.

**See also:**  
**CLEAR.LOCKS, LIST.LOCKS, LOCK** (Command), **LOCK** (QMBasic), **TESTLOCK()**
UNTIL

The UNTIL statement is used in conjunction with the FOR / NEXT or LOOP / REPEAT constructs to determine whether execution of the loop should continue.

Format

UNTIL expr

where

expr evaluates to a numeric value

The UNTIL statement causes execution of the innermost FOR/NEXT or LOOP/REPEAT construct to terminate if the value of expr is non-zero. It is equivalent to a statement such as

IF expr # 0 THEN EXIT

If the CONDITIONAL.STATEMENTS option of the $MODE compiler directive is active, expr may be one of the following statements that in its normal form has a THEN/ELSE clause:

FILELOCK, FILEUNLOCK, FIND, FINDSTR, FLUSH, GETLIST, LOCATE, MATREAD, MATREADCSV, MATREADL, MATREADU, OPEN, OPENPATH, OPENSEQ, OSREAD, READ, READBLK, READCSV, READL, READLIST, READNEXT, READEQ, REAUD, READY, READVL, READVU, SEEK, WRITEBLK, WRITECSV, WRITESEQ

When used in this way, the LOCKED or ON ERROR clauses normally associated with some of these statements are not available.

Examples

FOR I = 1 TO 20
UNTIL A(I) < 0
    DISPLAY A(I)
NEXT I

This program fragment displays elements of matrix A. The loop terminates if an element is found with a negative value.

LOOP
UNTIL OSREAD 'C:\FAX.DATA'
    SLEEP 300
REPEAT

The above program fragment, which requires the CONDITIONAL.STATEMENTS option of the $MODE compiler directive to be active, repeatedly attempts to read the C:\FAX.DATA file at five minute intervals. When successful, the loop exits.

See also:
EXIT, WHILE
UPCASE()

The **UPCASE()** function returns a string with all letters converted to upper case.

**Format**

```plaintext
UPCASE(string)
```

where

```plaintext
string  evaluates to the string in which substitution is to occur.
```

The **UPCASE()** function returns the value of `string` with all letters converted to upper case. If `string` is a variable rather than an expression, the value of the variable is not affected.

On an ECS mode system, the uppercase equivalent of each character is defined by the currently selected character map. On other systems, the standard letter pairings of the ASCII character set are used.

**Example**

```plaintext
NAME = "Thomas Smith"
PRINT UPCASE(NAME)
```

This program fragment prints the string "THOMAS SMITH".

**See also:**

**DOWNCASE()**
VALIDATE()

The VALIDATE() function validates a record against data integrity constraints.

Format

\[ \text{VALIDATE}(fvar, id, data) \]

where

- \( fvar \) is the file variable associated with an open file.
- \( id \) is the id of the record being processed.
- \( data \) is the data record.

The \text{VALIDATE}() function allows an application to test whether the record in \textit{data} satisfies the validation rules for the file open to \( fvar \). If successful, the function returns True. Using \text{VALIDATE}() against a hashed file that has no constraints defined will also return True.

If the validation fails, \text{VALIDATE}() returns False and the \text{STATUS}() function can be used to determine why it failed. A status of zero indicates that the constraints rules were not met. The failing field number will be in \@FNO and the corresponding field name will be in \@FNAME. A non-zero status value indicates that some other error occurred such as the file being of an inappropriate type.

Example

The processing performed by the VERIFYCONSTRAINTS command is essentially as below:

```qbasic
LOOP 
  READNEXT ID FROM ID.LIST ELSE EXIT 
  READ REC FROM DATA.F, ID THEN 
    IF NOT(VALIDATE(DATA.F, ID, REC)) THEN 
      DISPLAY ID : ' (FIELD ' : @FNO ; ')'
    END 
  END 
END
```

See also:

- Data integrity constraints
- VERIFYCONSTRAINTS
**VARTYPE()**

The **VARTYPE()** function returns a value indicating the data type associated with a variable.

**Format**

\[ \text{VARTYPE}(\text{variable}) \]

where

\[ \text{variable} \] is the variable to be tested.

QM stores data internally in a type variant manner. The actual data type associated with a variable may change as new data is stored. The **VARTYPE()** function allows a program to determine the internal data type being used. It returns one of the following values. The symbolic names shown are defined in the SYSCOM KEYS.H include record.

<table>
<thead>
<tr>
<th>Value</th>
<th>Symbolic Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>VT$UNASSIGNED</td>
<td>Unassigned</td>
</tr>
<tr>
<td>1</td>
<td>VT$BOOLEAN</td>
<td>Boolean value</td>
</tr>
<tr>
<td>2</td>
<td>VT$INTEGER</td>
<td>Integer value</td>
</tr>
<tr>
<td>3</td>
<td>VT$FLOATNUM</td>
<td>Floating point value</td>
</tr>
<tr>
<td>4</td>
<td>VT$SUBR</td>
<td>Subroutine reference</td>
</tr>
<tr>
<td>5</td>
<td>VT$STRING</td>
<td>String variable</td>
</tr>
<tr>
<td>6</td>
<td>VT$FILE.REF</td>
<td>File variable</td>
</tr>
<tr>
<td>7</td>
<td>VT$NULLVALUE</td>
<td>SQL style null value</td>
</tr>
<tr>
<td>8</td>
<td>VT$COLLECTION</td>
<td>Data collection</td>
</tr>
<tr>
<td>9</td>
<td>VT$IMAGE</td>
<td>Screen image from <strong>SAVE.SCREEN()</strong></td>
</tr>
<tr>
<td>10</td>
<td>VT$ARRAY</td>
<td>Array in data collection</td>
</tr>
<tr>
<td>11</td>
<td>VT$SELLIST</td>
<td>Select list</td>
</tr>
<tr>
<td>13</td>
<td>VT$SOCK</td>
<td>Socket variable</td>
</tr>
<tr>
<td>15</td>
<td>VT$OBJ</td>
<td>Object instance of class module</td>
</tr>
<tr>
<td>18</td>
<td>VT$FILEWAIT</td>
<td>Monitor variable from <strong>FILE.EVENT()</strong></td>
</tr>
<tr>
<td>20</td>
<td>VT$SORT</td>
<td>Sort data from <strong>SORT()</strong></td>
</tr>
</tbody>
</table>

**Example**

```plaintext
IF VARTYPE(FVAR) = VT$FILE.REF THEN
   CLOSE(FVAR)
END
```

This program fragment tests whether FVAR references an open file and, if so, closes the file. The same effect could have been achieved using the FL$OPEN mode of the **FILEINFO()** function.
**VOCPATH()**

The **VOCPATH()** function returns the pathname for a file referenced via the VOC.

**Format**

$$\text{VOCPATH}(\text{filename} \{, \text{dict.flag}\})$$

where

- **filename** is the file name for which the pathname is to be returned. The extended file name syntaxes are allowed.
- **dict.flag** evaluates to True to return the dictionary pathname, False to return the data pathname. If omitted, the data pathname is returned.

The **VOCPATH()** function processes the VOC file to return the pathname for the supplied **filename**. If **filename** corresponds to a Q-pointer, the function resolves the remote reference.

The optional **dict.flag** argument allows a program to specify whether the returned pathname should be that of the data or dictionary components of the file. The function also recognises use of a prefix of **DICT** and a single space on the front of the **filename**, returning the dictionary pathname regardless of the value of **dict.flag**.

If the function is successful, the return value is the file pathname and the **STATUS()** function will return zero. If the **filename** cannot be resolved to a pathname, the return value will be a null string and the **STATUS()** function will return an explanatory error number.

**Examples**

```vq
DISPLAY VOCPATH('SALES')
```

This statement would display the pathname of the data part of the SALES file.

```vq
DISPLAY VOCPATH('SALES', @true)
```

Adding the **dict.flag** as True would display the pathname of the dictionary part of the SALES file.

```vq
DISPLAY VOCPATH('DICT SALES')
```

Use of the **DICT** prefix would also display the pathname of the dictionary part of the SALES file.

```vq
DISPLAY VOCPATH('CUSTOMERS,NORTH')
```

Used with a multi-file reference, the pathname returned will be that of the NORTH subfile of the CUSTOMERS file.

```vq
DISPLAY VOCPATH('HR:PAYROLL')
```
This example uses an extended file name syntax to resolve the pathname of the PAYROLL file in the HR account.
VOID

The VOID statement discards the result of an associated expression.

Format

VOID expr

where

expr is an expression.

The VOID statement evaluates the supplied expression and discards the result. It is intended for use when calling functions for which the returned value is not used by the program. Use of VOID removes the need for a dummy variable and possible compiler warning messages regarding a variable that is set but never used.

Example

VOID KEYIN()

The above statement waits for the user to press a key but discards the input data.
The **VSLICE()** function returns a string formed by extracting a given value or subvalue position from a dynamic array.

**Format**

\[
\text{VSLICE}(\text{string, vpos}) \\
\text{VSLICE}(\text{string, vpos, svpos})
\]

where

- \(\text{string}\) is the string from which the data is to be extracted.
- \(\text{vpos}\) evaluates to the value position to be extracted.
- \(\text{svpos}\) evaluates to the subvalue position to be extracted.

The first form of the **VSLICE()** function processes \(\text{string}\) to build a new dynamic array containing only the specified value position from each field. Subvalues are returned as part of each value in the result string.

The second form processes \(\text{string}\) to build a new dynamic array containing only the specified subvalue position from each field. If \(\text{vpos}\) is less than one, the subvalue is extracted from all value positions and value marks are included in the returned string to match the structure of the original data. If \(\text{vpos}\) is one or greater, the subvalue is extracted only from the specified value position and no value marks are included in the result.

If \(\text{vpos}\) and \(\text{svpos}\) are both less than one, the **VSLICE()** function returns the source \(\text{string}\).

**Examples**

The following examples are all based on a \(\text{string}\) that contains

\[1\text{FM}2\text{VM}3\text{SM}4\text{VM}5\text{FM}6\text{VM}7\text{SM}8\text{VM}9\]

\[
\text{VSLICE}(\text{S, 0}) \quad 1\text{FM}2\text{VM}3\text{SM}4\text{VM}5\text{FM}6\text{VM}7\text{SM}8\text{VM}9 \\
\text{VSLICE}(\text{S, 1}) \quad 1\text{FM}2\text{FM}6 \\
\text{VSLICE}(\text{S, 2}) \quad \text{FM}3\text{SM}4\text{FM}7\text{SM}8 \\
\text{VSLICE}(\text{S, 3}) \quad \text{FM}5\text{FM}9 \\
\text{VSLICE}(\text{S, 0, 1}) \quad 1\text{FM}2\text{VM}3\text{VM}5\text{FM}6\text{VM}7\text{VM}9 \\
\text{VSLICE}(\text{S, 0, 2}) \quad \text{FM}3\text{SM}4\text{FM}VM8 \\
\text{VSLICE}(\text{S, 0, 3}) \quad \text{FM}5\text{FM}9 \\
\text{VSLICE}(\text{S, 1, 1}) \quad 1\text{FM}2\text{FM}6 \\
\text{VSLICE}(\text{S, 1, 2}) \quad \text{FM}3\text{FM}7 \\
\text{VSLICE}(\text{S, 2, 1}) \quad \text{FM}3\text{FM}7 \\
\text{VSLICE}(\text{S, 2, 2}) \quad \text{FM}4\text{FM}8 \\
\text{VSLICE}(\text{S, 2, 3}) \quad \text{FM}5\text{FM}9 \\
\text{VSLICE}(\text{S, 3, 1}) \quad \text{FM}5\text{FM}9 \\
\text{VSLICE}(\text{S, 3, 2})
\]
WAIT.FILE.EVENT

The WAIT.FILE.EVENT() function waits for a file monitoring event (Windows only).

Format

WAIT.FILE.EVENT(event, timeout)

where

- **event** is either an event monitoring variable created using FILE.EVENT() or a dimensioned matrix containing these variables.
- **timeout** is the maximum period in seconds to wait. A zero or negative value causes an infinite wait.

The WAIT.FILE.EVENT() function allows a program to wait for a change within the Windows file system. It can be used, for example, to wait until a new file is created within a directory without needing to write a loop that periodically compares the directory content with a previously recorded list of items.

The event(s) to be monitored are established using the FILE.EVENT() function. Where only a single event is to be monitored, the value returned by this function can be stored in a simple scalar variable. If multiple events are to be monitored, the returned values should be stored in a dimensioned array. The WAIT.FILE.EVENT() function will ignore any elements of the array that are not file monitoring event values. See FILE.EVENT() for a list of the events that can be trapped.

The WAIT.FILE.EVENT() function uses an efficient event waiting mechanism within the Windows operating system. When a monitored event occurs, the function returns the index of the associated element of the event array or, if this was a scalar variable, 1. If the function returns because the timeout has expired, it returns zero.

When using an array of events, an event can be removed from the list by overwriting the relevant element of the event array with any other type of data. Because use of a file event monitoring variable in any other way returns a non-negative number, setting the element to a negative value or a null string provides a reliable way to recognise unused entries. If the array contains no file event monitoring variables, the WAIT.FILE.EVENT() function simply sleeps for the given timeout period.

Beware that when the SAFEDIR configuration parameter is set to 1, if this function is used to monitor a directory into which items are written using QM, two file notification events will occur for each write.

Examples

```qmb
EVT = FILE.EVENT("C:\IMPORT", FE$FILE.NAME)
LOOP
   IF WAIT.FILE.EVENT(EVT, 10) > 0 THEN
       ...Processing...
   ELSE
       ...Periodic actions...
   END
END
REPEAT
```
The above program fragment creates a file monitoring event to trap changes to the C:\IMPORT directory as a result of creating, deleting or renaming a file. The `WAIT.FILE.EVENT()` function waits for an event to occur and then executes some processing code before waiting for the next event. In this example, the wait includes a ten second timeout to allow the program to perform some other periodic tasks.

```
DIM EVT(3)
EVT(1) = FILE.EVENT("C:\IMPORT", FE$FILE.NAME + FE$SUBTREE)
EVT(2) = FILE.EVENT("C:\EXPORT", FE$FILE.NAME + FE$SUBTREE)
LOOP
  IF WAIT.FILE.EVENT(EVT, 0) > 0 THEN
    ...Processing...
  END
REPEAT
```

In this example, two directories and their sub-directories are monitored. Note that the EVT array has been dimensioned as three elements but only two are used. The `WAIT.FILE.EVENT()` function will ignore the unused element. Monitoring of either directory could be temporarily disabled in the processing code by setting the EVT array element to -1. Similarly, the processing code could add a new monitored event in EVT(3).
WAKE

The WAKE statement awakens another process that has executed a PAUSE.

Format

WAKE user.no

where

user.no is the QM user number of the process to be awoken.

The WAKE statement resumes execution of another process that has executed a PAUSE statement.

If the WAKE is executed before the other process attempts to pause, the program is not suspended. Multiple wake events occurring in this way will only awaken the target process once.

The WAKE statement attempts to use an inter-process signalling mechanism to resume execution of the other process. Due to operating system limitations, this is usually only possible if both processes are running with the same user id. If this is not the case, the target process may take up to about a second to restart.
WEOFSEQ

The **WEOFSEQ** statement truncates a record open for sequential access at the current position.

**Format**

```
WEOFSEQ file.var {ON ERROR statement(s)}
```

where

- `file.var` is the file variable associated with the record by a previous **OPENSEQ** statement.
- `statement(s)` are statement(s) to be executed if the action fails.

The **WEOFSEQ** statement truncates the record at the current position. Performed immediately after the **OPENSEQ**, this will remove all data from the record. Performed after one or more **READSEQ** operations have been performed, all subsequent data is cleared from the record.

The **ON ERROR** clause is executed if a fatal error occurs. The **STATUS()** function can be used to determine the cause of the error. If no **ON ERROR** clause is present, a fatal error causes an abort.

**Example**

```
OPENSEQ "STOCKS", "STOCK.LIST" TO STOCK.LIST THEN
    WEOFSEQ STOCK.LIST
ELSE
    IF STATUS() THEN ABORT "Cannot open stocks list"
END
```

This program fragment opens the record STOCKS for sequential access. If it already exists, the **THEN** clause of the **OPENSEQ** is taken and the existing data is removed using **WEOFSEQ**.

**See also:**

CLOSESEQ, CREATE, NOBUF, OPENSEQ, READBLK, READCSV, READSEQ, WRITEBLK, WRITECSV, WRITESEQ, WRITESEQF
WHILE

The WHILE statement is used in conjunction with the FOR / NEXT or LOOP / REPEAT constructs to determine whether execution of the loop should continue.

Format

WHILE expr

where

expr evaluates to a numeric value

The WHILE statement causes execution of the innermost FOR/NEXT or LOOP/REPEAT construct to terminate if the value of expr is zero. It is equivalent to a statement such as

IF expr = 0 THEN EXIT

If the CONDITIONAL STATEMENTS option of the $MODE compiler directive is active, expr may be one of the following statements that in its normal form has a THEN/ELSE clause:

FILELOCK, FILEUNLOCK, FIND, FINDSTR, FLUSH, GETLIST, LOCATE, MATREAD, MATREADCSV, MATREADL, MATREADU, OPEN, OPENPATH, OPENSEQ, OSREAD, READ, READBLK, READCSV, READL, READLIST, READNEXT, READSEQ, READU, READY, READVL, READVU, SEEK, WRITEBLK, WRITECSV, WRITESEQ

When used in this way, the LOCKED or ON ERROR clauses normally associated with some of these statements are not available.

Examples

LOOP
   REMOVE ITEM FROM LIST SETTING DELIMITER
   DISPLAY ITEM
WHILE DELIMITER
   DISPLAY ITEM
REPEAT

This program fragment displays items removed from dynamic array LIST. The loop is terminated when the value of DELIMITER becomes zero.

SELECT FV
LOOP
   WHILE READNEXT ID
      ...processing...
   REPEAT

The above program fragment, which requires the CONDITIONAL STATEMENTS option of the $MODE compiler directive to be active, uses the READNEXT statement as a conditional element in the loop.

See also:
EXIT, UNTIL
WRITE

The WRITE statement writes a record to a previously opened file. The WRITEU statement is identical but preserves any lock on the record.

Format

WRITE var {ENCODING name} TO file.var, record.id {ON ERROR statement(s)}
WRITE var {ENCODING name} TO file.var CREATING.SEQKEY id.var {ON ERROR statement(s)}

where

var is the name of a variable containing the data to be written.
file.var is the file variable associated with the file.
record.id evaluates to the id of the record to be written.
id.var is the variable to receive the record id.
statement(s) are statements to be executed if the write fails.

The keyword ON may be used in place of TO.

In the first form of the WRITE statement, the contents of var are written to the file using the supplied record id. Any existing record of the same id is replaced by this action.

When writing to a directory file, field marks in the data to be written are replaced by newlines. This action may be suppressed by using the MARK.MAPPING statement after opening the file.

The optional ENCODING element to this statement sets the character encoding to be used, overriding any encoding set in the VOC F-type record or when the file was opened. This is relevant only to directory files and is ignored for other file types. A null string as the encoding name is equivalent to not having the ENCODING clause at all. To disable the default encoding, the encoding name should be specified as "NULL".

The WRITE statement releases any read or update lock on this record. The WRITEU statement preserves the lock. Within a transaction, the lock is retained until the transaction terminates and then released regardless of which statement is used. Attempting to write a record in a transaction will fail if the process does not hold an update lock on the record or the file.

The ON ERROR clause is executed for serious fault conditions such as errors in a file's internal control structures. It is also executed if the write operation is disallowed by a pre-write trigger function associated with the file. The STATUS() function will return an error number. If no ON ERROR clause is present, an abort would occur.

Although it is generally acceptable to assume that a WRITE that does not execute the ON ERROR clause or abort was successful, the WRITE operation normally returns zero from an immediately following use of the STATUS() function. On ECS mode systems, status value ERSECS.DATA will be returned if a record containing ECS characters is written to a non-ECS mode hashed file or a directory file with no encoding. The write will have completed but only the low order byte of each character will have been written.
When writing a new record to a directory file on a Linux or Unix system, the operating system level file that represents the QM record is created using the current umask value. For sessions that connect directly to QM rather than from the operating system shell (direct telnet and QMClient), the umask value is specified by the \texttt{UMASK} configuration parameter or, if this parameter is not present, a default of 002. This behaviour can be modified by executing a \texttt{UMASK} command within the session.

### Automatic Generation of Sequential Numeric Keys

The second form uses the \texttt{CREATING.SEQKEY} clause to create a sequentially numbered record id, returning this id in \texttt{id.var}. This form of write is only valid with hashed files, including those opened over a \texttt{QMNet} connection. Use with data replication is likely to fail as the internal sequence number will not be updated on the subscriber system.

Used inside a transaction, the record id is generated when the \texttt{WRITE} statement is executed, not at transaction commit. This could lead to gaps in sequential record numbering if a transaction does not commit. Gaps can also occur as a result of errors detected after the point when locking must be enforced such as rejection of the write by a trigger function. Because the record id is generated automatically, a program cannot lock it prior to the \texttt{WRITE}. Instead, an update lock is automatically applied for the duration of the internal processing of the write operation. Except when using \texttt{WRITEU}, this lock will be released before continuing with the next program statement.

By default, the sequential record numbering begins at 1 for a newly created file. The \texttt{FCONTROL()} function provides a way for an application to access and update the internal sequence number. The \texttt{FILEINFO()} function allows retrieval of the last used sequential record key. The sequence number can be updated using the \texttt{CONFIGURE.FILE} command.

Application developers should take great care when using automatic key generation. It is essential that all records are written using this method or by reserving a key value with the \texttt{FC$GET.SEQ.KEY} mode of the \texttt{FCONTROL()} function. Failure to comply with this rule could result in records being overwritten as there is no check whether a record with the automatically generated record id already exists. In addition, if the system suffers a power failure or similar condition in which a record may have been written but the file header (where the sequential counter is maintained) has not been flushed to disk. This situation is no different from that which would occur with other methods of allocating sequential numeric keys, however, the \texttt{qmfix} tool will detect and correct any inconsistency. It is good operational practice to run \texttt{qmfix} after a system failure regardless of whether automatic key generation is in use. Note that this issue applies only to system failures, not application errors or forced logout of a QM process.

### Use of a THEN/ELSE Clause

For compatibility with the way in which triggers operate in some other multivalue products, the \texttt{WRITE} statement has an optional \texttt{THEN/ELSE} clause. Because this would otherwise lead to a syntactic ambiguity, compilation of programs that use this clause requires the \texttt{WRITE.DELETE.THEN.ELSE} option of the \texttt{$MODE} compiler directive to be enabled. Once this is done, the optional \texttt{THEN/ELSE} clauses can be included in their usual position, after the \texttt{ON ERROR} clause.

When a \texttt{WRITE} statement has a \texttt{THEN/ELSE} clause, a failure returned from a trigger function, typically as a result of a pre-write data validation error, will cause the \texttt{ELSE} clause to be executed instead of the \texttt{ON ERROR} clause.

### Examples
WRITE ITEM TO STOCK, ITEM.ID

This statement writes the content of ITEM to a record with the id in ITEM.ID on the file previously opened to file variable STOCK.

WRITE LOG.REC TO LOG.F CREATING.SEQKEY LOG.ID

This statement writes the content of LOG.REC to the file opened to file variable LOG.F, using an automatically generated sequential record id that is returned in the LOG.ID variable.
WRITE.SOCKET()

The WRITE.SOCKET() function writes data to a socket.

Format

WRITE.SOCKET(skt, data, flags, timeout)

where

\( \text{skt} \) is the socket variable returned by ACCEPT.SOCKET.CONNECTION() (stream connection), CREATE.SERVER.SOCKE() (datagram connection) or OPEN.SOCKE().

\( \text{data} \) is the data to be written.

\( \text{flags} \) is a value determining the mode of operation of the socket for this write, formed by adding the values of tokens defined in the SYSCOM KEYS.H record. The flags available in this release are:

- SKT$BLOCKING: Sets the mode of data transfer as blocking.
- SKT$NON.BLOCKING: Sets the mode of data transfer as non-blocking.

If neither blocking flag is given, the blocking mode set when the socket was opened is used.

\( \text{timeout} \) is the timeout period in milliseconds. A value of zero implies no timeout.

The WRITE.SOCKET() function writes the given data and returns the number of bytes written. If non-blocking mode is used or a timeout occurs, this byte count may be less than the length of the data. The remaining data can be written with a subsequent call to WRITE.SOCKET() when buffer space becomes available.

The STATUS() function returns zero if the action is successful, or a non-zero error code if an error occurs. A timeout will return an error code of ER$TIMEOUT as defined in the SYSCOM ERR.H record.

Example

```
SKT = OPEN.SOCKE("193.118.13.14", 3000, SKT$BLOCKING)
IF STATUS() THEN STOP 'Cannot open socket'
N = WRITE.SOCKE(SKT, DATA, 0, 0)
CLOSE.SOCKE SKT
```

This program fragment opens a connection to port 3000 of IP address 193.118.13.14, sends the data in DATA and then closes the socket.

See also:
Using Socket Connections, ACCEPT.SOCKE.CONNECTION, CLOSE.SOCKE, CREATE.SERVER.SOCKE(), OPEN.SOCKE(), READ.SOCKE(), SELECT.SOCKE(), SERVER.ADDR(), SET.SOCKE.MODE(), SOCKET.INFO()
WRITEBLK

The WRITEBLK statement writes data at the current file position in a record previously opened using OPENSEQ.

Format

```
WRITEBLK var {ENCODING name} TO file.var
{ON ERROR statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

where

- `var` is the name of a variable holding the data to be written.
- `file.var` is the file variable associated with the file.
- `statement(s)` are statements to be executed depending on the outcome of the WRITEBLK operation.

At least one of the THEN and ELSE clauses must be present.

The THEN clause is executed if the WRITEBLK is successful.

The ELSE clause is executed if the WRITEBLK fails. The STATUS() function will indicate the cause of the error.

The ON ERROR clause is executed for serious fault conditions such as errors in a file's internal control structures. The STATUS() function will return an error number. If no ON ERROR clause is present, an abort would occur.

The optional ENCODING clause sets the character encoding to be used, overriding any encoding set in the VOC F-type record or when the file was opened. A null string as the encoding name is equivalent to not having the ENCODING clause at all. To disable the default encoding, the encoding name should be specified as "NULL".

If the record was opened with the SHARED option of OPENSEQ, the data will be appended to any existing record content.

Example

```
WRITEBLK VAR TO SEQ.F ELSE STOP 'Write error'
```

This program fragment writes data to a file previously opened to file variable SEQ.F.

See also:
CLOSESEQ, CREATE, NOBUF, OPENSEQ, READBLK, READCSV, READSEQ, WRITECSV, WEOFSEQ, WRITESEQ, WRITESEQF
WRITECSV

The WRITECSV statement writes data at the current file position in a record previously opened using OPENSEQ. The data to be written is assembled from one or more variables and written in CSV format.

Format

```
WRITECSV var1, var2, ... {ENCODING name} TO file.var
{ON ERROR statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

where

- `var1, var2, ...` are the items to be written to the file. If any of the variables contains field marks, each field is treated as a separate item in the resultant CSV data.
- `file.var` is the file variable associated with the file.
- `statement(s)` are statements to be executed depending on the outcome of the WRITECSV operation.

At least one of the THEN and ELSE clauses must be present.

The optional ENCODING element to this statement sets the character encoding to be used, overriding any encoding set in the VOC F-type record or when the file was opened.

The data in the named variables is assembled as a CSV format text string which is then written to the file with a newline appended. The THEN clause is executed if the WRITECSV is successful.

If the record was opened with the SHARED option of OPENSEQ, the data will be appended to any existing record content.

The ELSE clause is executed if the WRITECSV fails. The STATUS() function will indicate the cause of the error.

The ON ERROR clause is executed for serious fault conditions such as errors in a file's internal control structures. The STATUS() function will return an error number. If no ON ERROR clause is present, an abort would occur.

CSV Format

CSV format is used by many applications. QM adheres to the CSV standard (RFC 4180).

Items are enclosed in double quotes if they contain commas or double quotes. Embedded double quotes are replaced by a pair of double quotes.

The CSV.MODE statement can be used to select variations on the quoting rules applied or to change the delimiter character.

Examples
WRITECSV PROD.NO, QTY TO SEQ.F ELSE STOP 'Write error'

This program fragment writes the contents of the PROD.NO and QTY variables as a CSV item to a
file previously opened to file variable SEQ.F.

WRITECSV S<1>, S<2>, S<3> TO SEQ.F ELSE STOP 'Write error'

This program fragment writes the contents of fields 1 to 3 of S as CSV data to a file previously
opened to file variable SEQ.F.

WRITECSV S TO SEQ.F ELSE STOP 'Write error'

If dynamic array S in the previous example had only three fields, this program fragment writes
exactly the same data, treating each field as a separate CSV item.

See also:
CLOSESEQ, CREATE, CSVDQ(), CSV.MODE, DPARSE.CSV, NOBUF, OPENSEQ,
PRINTCSV, READBLK, READCSV, READSEQ, WEOFSEQ, WRITESEQ, WRITEBLK,
WRITESEQF
WRITELIST

The WRITELIST statement writes a dynamic array to the $SAVEDLISTS file.

Format

```
WRITELIST ids TO list.name
```

where

- **ids** evaluates to the data to be written. This should be a field mark delimited dynamic array.
- **list.name** is the name of the item to be written in the $SAVEDLISTS file.

The keyword **ON** may be used in place of **TO**.

The WRITELIST statement allows a program to form a saved select list from a field mark delimited dynamic array. It is equivalent to opening the $SAVEDLISTS file and writing the ids variable to a record named list.name. The file is opened in non-transactional mode such that the write occurs immediately, not on transaction commit.

It is important that the ids list does not contain any mark characters other than the field marks between each item.

Example

```
READ ITEMS FROM STOCK.FILE, STORE.CODE THEN
    WRITELIST ITEMS TO 'STORE.LIST'
END
```

The above program fragment reads a record from the file open as STOCK.FILE and then writes this to $SAVEDLISTS as "STORE.LIST".

See also:

GETLIST, SAVELIST
WRITESEQ

The WRITESEQ statement writes a string to a directory file record previously opened for sequential access. WRITESEQF is identical except that it force writes the data to disk.

Format

```
WRITESEQ var { ENCODING name } TO file.var { ON ERROR statement(s)}
{ THEN statement(s) }
{ ELSE statement(s) }
```

where

`var` is the variable containing the data to be written.

`file.var` is the file variable associated with the record by a previous OPENSEQ statement.

`statement(s)` are statement(s) to be executed depending on the outcome of the WRITESEQ.

The keyword TO may be replaced by ON. At least one of the THEN and ELSE clauses must be present.

The optional ENCODING clause sets the character encoding to be used, overriding any encoding set in the VOC F-type record or when the file was opened. A null string as the encoding name is equivalent to not having the ENCODING clause at all. To disable the default encoding, the encoding name should be specified as "NULL".

The data in `var` and a newline is written to the record at the current file position, overwriting any data already present. The THEN clause is executed if the write is successful.

The default behaviour of WRITESEQ is to use the operating system dependent newline (CR/LF pair on Windows, LF on other platforms). This can be modified by use of the mode codes in field 6 of the F-type VOC entry or by use of the FCONTROL() function.

The ELSE clause is executed if the WRITESEQ operation fails. On ECS mode systems, STATUS() value ER$ECS.DATA will be returned if data containing ECS characters is written to a directory file with no encoding. The write will have completed but only the low order byte of each character will have been written.

If a fatal error occurs, the ON ERROR clause is executed. The STATUS() function can be used to establish the cause of the error. If no ON ERROR clause is present, a fatal error causes an abort.

The FILEINFO() function can be used with key FL$LINE to determine the line number that will be written by the next WRITESEQ. This information is not valid if the SEEK, READBLK or WRITEBLK statements have been used.

If the record was opened with the SHARED option of OPENSEQ, the data will be appended to any existing record content.

The WRITESEQF statement is identical to WRITESEQ except that execution of the next QMBasic statement does not occur until the data has been written to disk. With WRITESEQ, the data may still be in internal buffers.
Example

`WRITESEQ REC TO STOCK.LIST ELSE ABORT "Write error"

This statement writes the data in REC to the record open for sequential access via the STOCK.LIST file variable.

See also:
`CLOSESEQ, CREATE, NOBUF, OPENSEQ, READBLK, READCSV, READSEQ, WEOFSEQ, WRITEBLK, WRITECSV`
WRITEV

The WRITEV statement writes a specific field to a record of a previously opened file. The WRITEVU statement is identical but preserves any lock on the record.

Format

WRITEV var [ENCODING name] TO file.var, record.id, field.expr
{ON ERROR statement(s)}

where

var is the name of a variable containing the data to be written.
file.var is the file variable associated with the file. This may not be a data collection file.
record.id evaluates to the id of the record to be written.
field.expr evaluates to the number of the field to be written.
statement(s) are statements to be executed if the write fails.

The keyword ON may be used in place of TO.

The optional ENCODING element to this statement sets the character encoding to be used, overriding any encoding set in the VOC F-type record or when the file was opened. This is relevant only to directory files and is ignored for other file types. A null string as the encoding name is equivalent to not having the ENCODING clause at all. To disable the default encoding, the encoding name should be specified as "NULL".

The contents of var are written to field field.expr of record record.id of the file. If the record does not already exist, it will be created by this operation. The WRITEV statement releases any read or update lock on this record. The WRITEVU statement preserves the lock. Within a transaction, the lock is retained until the transaction terminates and then released regardless of which statement is used. Attempting to write a record in a transaction will fail if the process does not hold an update lock on the record or the file.

A field.expr value of zero is treated as a reference to field one. A negative field number causes the var string to be appended as a new field at the end of the record.

The ON ERROR clause is executed for serious fault conditions such as errors in a file's internal control structures. The STATUS() function will return an error number. If no ON ERROR clause is present, an abort would occur.

Although it is generally acceptable to assume that a WRITEV that does not execute the ON ERROR clause or abort was successful, the WRITEV operation normally returns zero from an immediately following use of the STATUS() function. On ECS mode systems, status value ERSECS.DATA will be returned if a record containing ECS characters is written to a non-ECS mode hashed file or a directory file with no encoding. The write will have completed but only the low order byte of each character will have been written.

Example
WRITEV ITEM TO STOCK, ITEM.ID, 3

This program fragment writes the value of ITEM to field 3 of record ITEM.ID in a file previously opened to file variable STOCK.

See also:
READV
**XTD()**

The **XTD()** function converts a string of hexadecimal characters to a number.

**Format**

\[ \text{XTD}(expr) \]

where

\[ expr \]

evaluates to the hexadecimal string to be converted.

The **XTD()** function converts the supplied \( expr \) hexadecimal string to a number. If \( expr \) contains any characters other than 0-9 or A-F (upper or lower case) or is a null string, the function returns the original value of \( expr \).

**See also:**

[DTX()]
6.6 Character Values for Terminal Input

KEYIN() and KEYCODE()

The table below shows the keys that produce each decimal character value from the `KEYIN()` function on Windows systems using QMConsole, on all systems using QMTerm, or when using `KEYCODE()` to decode key sequences.

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Ctrl-A</td>
<td>Ctrl-B</td>
<td>Ctrl-C</td>
<td>Ctrl-D</td>
<td>Ctrl-E</td>
<td>Ctrl-F</td>
<td>Ctrl-G</td>
<td>Ctrl-H</td>
</tr>
<tr>
<td>1</td>
<td>Bsp</td>
<td>Tab</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Ctrl-J</td>
<td>Ctrl-K</td>
<td>Ctrl-L</td>
<td>Ctrl-M</td>
<td>Ctrl-N</td>
<td>Ctrl-O</td>
<td>Ctrl-P</td>
<td>Ctrl-Q</td>
</tr>
<tr>
<td>3</td>
<td>Ctrl-rtn</td>
<td>Return</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Ctrl-T</td>
<td>Ctrl-U</td>
<td>Ctrl-V</td>
<td>Ctrl-W</td>
<td>Ctrl-X</td>
<td>Ctrl-Y</td>
<td>Ctrl-Z</td>
<td>Esc</td>
</tr>
<tr>
<td>5</td>
<td>Ctrl-}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Ctrl-^</td>
<td>Ctrl-</td>
<td>Space</td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
</tr>
<tr>
<td>7</td>
<td>Ctrl-</td>
<td>Space</td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
</tr>
<tr>
<td>8</td>
<td>Ctrl-</td>
<td>Space</td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
</tr>
<tr>
<td>9</td>
<td>Ctrl-</td>
<td>Space</td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
</tr>
</tbody>
</table>

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00x</td>
<td>Ctrl-A</td>
<td>Ctrl-B</td>
<td>Ctrl-C</td>
<td>Ctrl-D</td>
<td>Ctrl-E</td>
<td>Ctrl-F</td>
<td>Ctrl-G</td>
<td>Ctrl-H</td>
</tr>
<tr>
<td>01x</td>
<td>Bsp</td>
<td>Tab</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>02x</td>
<td>Ctrl-J</td>
<td>Ctrl-K</td>
<td>Ctrl-L</td>
<td>Ctrl-M</td>
<td>Ctrl-N</td>
<td>Ctrl-O</td>
<td>Ctrl-P</td>
<td>Ctrl-Q</td>
</tr>
<tr>
<td>03x</td>
<td>Ctrl-rtn</td>
<td>Return</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>04x</td>
<td>Ctrl-T</td>
<td>Ctrl-U</td>
<td>Ctrl-V</td>
<td>Ctrl-W</td>
<td>Ctrl-X</td>
<td>Ctrl-Y</td>
<td>Ctrl-Z</td>
<td>Esc</td>
</tr>
<tr>
<td>05x</td>
<td>Ctrl-}</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>06x</td>
<td>Ctrl-^</td>
<td>Ctrl-</td>
<td>Space</td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
</tr>
<tr>
<td>07x</td>
<td>Ctrl-</td>
<td>Space</td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
</tr>
<tr>
<td>08x</td>
<td>Ctrl-</td>
<td>Space</td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
</tr>
<tr>
<td>09x</td>
<td>Ctrl-</td>
<td>Space</td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
</tr>
<tr>
<td>10x</td>
<td>Ctrl-</td>
<td>Space</td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
</tr>
<tr>
<td>11x</td>
<td>Ctrl-</td>
<td>Space</td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
</tr>
<tr>
<td>12x</td>
<td>Ctrl-</td>
<td>Space</td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
</tr>
<tr>
<td>13x</td>
<td>Ctrl-</td>
<td>Space</td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
</tr>
<tr>
<td>14x</td>
<td>Ctrl-</td>
<td>Space</td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
</tr>
<tr>
<td>15x</td>
<td>Ctrl-</td>
<td>Space</td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
</tr>
<tr>
<td>16x</td>
<td>Ctrl-</td>
<td>Space</td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
</tr>
<tr>
<td>17x</td>
<td>Ctrl-</td>
<td>Space</td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
</tr>
<tr>
<td>18x</td>
<td>Ctrl-</td>
<td>Space</td>
<td>!</td>
<td>&quot;</td>
<td>#</td>
<td>$</td>
<td>%</td>
<td>&amp;</td>
</tr>
</tbody>
</table>

4.0-0
Character value tokens are defined in the KEYIN.H record of the SYSCOM file. Codes User0 to User9 are only returned by the KEYCODE() function.
KEYINV() and KEYCODEV()

The table below shows the code point values returned by the `KEYINV()` for special characters on Windows and by use of `KEYCODEV()` on all systems.

<table>
<thead>
<tr>
<th>U+F88x</th>
<th>U+F89x</th>
<th>U+F8Ax</th>
<th>U+F8Bx</th>
<th>U+F8Cx</th>
<th>U+F8Dx</th>
<th>U+F8Ex</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>F1</td>
<td>Ctrl-F5</td>
<td>Alt-F9</td>
<td>Page Down</td>
<td>User 6</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F2</td>
<td>Ctrl-F6</td>
<td>Alt-F10</td>
<td>Home</td>
<td>User 7</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>F3</td>
<td>Ctrl-F7</td>
<td>Alt-F11</td>
<td>End</td>
<td>User 8</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>F4</td>
<td>Ctrl-F8</td>
<td>Alt-F12</td>
<td>Insert</td>
<td>User 9</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>F5</td>
<td>Ctrl-F9</td>
<td>Shift-F1</td>
<td>Delete</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>F6</td>
<td>Ctrl-F10</td>
<td>Shift-F2</td>
<td>Ctrl-Tab</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>F7</td>
<td>Ctrl-F11</td>
<td>Shift-F3</td>
<td>Ctrl-Pg Up</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>F8</td>
<td>Ctrl-F12</td>
<td>Shift-F4</td>
<td>Ctrl-Pg Down</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>F9</td>
<td>Alt-F1</td>
<td>Shift-F5</td>
<td>Mouse</td>
<td>Ctrl-Home</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>F10</td>
<td>Alt-F2</td>
<td>Shift-F6</td>
<td>Ctrl-End</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>F11</td>
<td>Alt-F3</td>
<td>Shift-F7</td>
<td>User 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>F12</td>
<td>Alt-F4</td>
<td>Shift-F8</td>
<td>Csr Left</td>
<td>User 1</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Ctrl-F1</td>
<td>Alt-F5</td>
<td>Shift-F9</td>
<td>Csr Right</td>
<td>User 2</td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>Ctrl-F2</td>
<td>Alt-F6</td>
<td>Shift-F10</td>
<td>Csr Up</td>
<td>User 3</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>Ctrl-F3</td>
<td>Alt-F7</td>
<td>Shift-F11</td>
<td>Csr Down</td>
<td>User 4</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td>Ctrl-F4</td>
<td>Alt-8</td>
<td>Shift-F12</td>
<td>Page Up</td>
<td>User 5</td>
<td></td>
</tr>
</tbody>
</table>
6.7 \textbf{@-Variables}

QMBasic provides a number of special variables and constants with names prefixed by the @ character. Some @-variables can be updated by QMBasic programs though most are read-only.

Many of the @-variables are also available for use in I-type definitions or within paragraphs. A complete list of @-variables appears below.

\textbf{Compile-time Constants}

These constants are available in QMBasic programs and I-type definitions to improve readability.

- \@AM Attribute mark (synonym for \@FM)
- \@FM Field mark
- \@IM Item mark
- \@SM Subvalue mark
- \@SVM Subvalue mark (synonym for \@SM)
- \@TM Text mark
- \@VM Value mark

- \@FALSE Boolean value. Used as a number, this evaluates to zero.
- \@TRUE Boolean value. Used as a number, this evaluates to one.

- \@BUILD.TIME Date/time at which the program was compiled as an epoch value.

\textbf{Variables}

Except where indicated in the descriptions, these items are read-only

- \@ABORT.CODE A value indicating the cause of execution of the last abort. This variable is particularly useful within \texttt{ON.ABORT} paragraphs or programs invoked from them. Values are:
  
  0 No abort has occurred
  1 A QMBasic ABORT statement or the ABORT command has been used
  2 The Abort option has been selected after the break key was pressed
  3 An internal error has been detected

  The value of \@ABORT.CODE is initially zero and is reset to zero only by the EXECUTE statement

- \@ABORT.MESSAGE Contains the text of any message associated with the most recent abort event.

- \@ACALL.FNO The field number associated with a CALL correlative in an A or S-type dictionary item.

- \@ACCOUNT Synonym for \@WHO.
@ANS  Contains the result of the last virtual attribute expression evaluated. This variable can be updated, usually only in C-type dictionary items.

@BPV  Previous value of breakpoint item when using X breakpoint control option.

@COMMAND  The last command entered at the command prompt or initiated using the QMBasic EXECUTE statement. This variable is stacked across an EXECUTE and reverts to its previous value on completion of the executed command. If multiple commands are executed in a single EXECUTE statement, the @COMMAND variable will be updated at each command.

@COMMAND.STACK  This variable holds the history of commands executed at the command prompt as a field mark delimited dynamic array. The most recent command is field 1.

@CONV  Extended information for user defined conversion codes.

@CRTHIGH  Contains the number of lines per page of the display.

@CRTWIDE  Contains the width of the display.

@DATA  Data from REFORMAT command.

@DATA.PENDING  Contains the data on the DATA queue, if any. Each item, including the last, is followed by an item mark character.

@DATE  The internal format date value (days since 31 December 1967) at which the last command started execution. Any changes made to this variable will also be reflected in the values of the @DAY, @MONTH, @YEAR and @YEAR4 variables described below.

@DAY  The day of the month at which the last command started execution as a two digit value. Changing @DATE will also change this value.

@DICTRECS  The query processor sets this variable to an item mark delimited copy of the dictionary records that were used to construct the display clause elements of the query (including any item with the BREAK.SUP prefix). For dictionary records that contain object code (C/I types and A/S types with correlatives), the object code is omitted. Field 1, value 2 of each element contains the item name. This variable can be of use in post-processing query processor output.

@DRIVE  The drive letter of the QMSYS directory (Windows only, null on other platforms).

@DS  Contains the operating system specific directory delimiter character, \ on Windows, / on other platforms.

@EXCEPTION  Contains the name of the last exception thrown by the QMBasic THROW statement. This variable can be modified by applications.

@EXCEPTION.DATA  Contains any qualifying data associated with the last exception thrown by the QMBasic THROW statement. This variable can be modified by applications.

@EXCEPTION.ORIGIN  Contains the program name and line number at which the last exception was thrown separated by a field mark. If the program has no cross-reference tables, the line number will be -1. This variable can be modified by applications.
@EXCEPTION.STACK
Dynamic array containing a full stack dump in the form created by SYSTEM(1002) at the point of throwing an exception for which the SAVE.STACK option of the CATCH clause was used. This variable can be modified by applications.

@FILE.NAME
The name of the file referenced in the most recent query processor command. This variable may be updated by a QMBasic program.

@FILENAME
Synonym for @FILE.NAME

@FMT
The query processor sets this variable to a field mark delimited list of the width and justification codes for each item in the display clause of the query (including any item with the BREAK.SUP prefix). This variable can be of use in post-processing query processor output.

@FNAME
Contains the field name for a data integrity constraints validation error.

@FNO
Contains the field number for a data integrity constraints validation error.

@GID
User’s group id number for all platforms except Windows. Same as SYSTEM(29).

@HOSTNAME
The name of the server computer system. Same as SYSTEM(1015).

@ID
The record id of the record being processed by a query processor command or an I-type function. This variable may be updated by a QMBasic program.

@IP.ADDR
The IP address associated with a network user. Same as SYSTEM(42).

@ITYPE.MODE
This variable can be used to determine the mode of execution of an I-type. It has three possible values:
0 Normal
1 Evaluation of the old index value when updating or deleting a record from a file with an alternate key index.
2 Evaluation of the new index value when updating or adding a record to a file with an alternate key index.

@LEVEL
The current command processor depth (EXECUTE level). The initial command processor is level one, each EXECUTE level increments this by one, decrementing on return from that level.

@LINE
The current source line number in the module being compiled, possibly useful in error messages. See also @WHERE.

@LOGNAME
User’s login name. On Windows, this is converted to uppercase.

@LPTRHIGH
Contains the number of lines per page of print unit zero. Depending on the current setting of the PRINTER flag, this may refer to the display or to the printer.

@LPTRWIDE
Contains the width of print unit zero. Depending on the current setting of the PRINTER flag, this may refer to the display or to the printer.

@MBUTTON
The button number related to the most recent mouse click trapped by KEYCODE() or KEYCODEV().

@MCOL
The screen column number related to the most recent mouse click trapped by KEYCODE() or KEYCODEV().

@MONTH
The month in which the last command started execution as a two digit value. Changing @DATE will also change this value.
@MROW  The screen row number related to the most recent mouse click trapped by KEYCODE() or KEYCODEV().

@NB  Break number level. Set to zero on detail lines and one upwards on break lines. A value of 255 represents the grand total line.

@NI  Item counter. Used in I-types, this holds the number of records retrieved by the query processor command. This variable can be updated, primarily for interaction with I-type dictionary items.

@NULL  SQL style null data constant for compatibility with data collections that may be converted to JSON string format.

@OPTION  Contains a copy of field 4 of the V-type VOC entry when a verb starts execution. Use of this variable enables related commands to be handled by a single program.

@PARASENTENCE  The sentence that invoked the most recent paragraph or sentence. On entering a command at the keyboard, this variable will be set to the same value as @COMMAND. If the command is a paragraph or sentence which invokes a further paragraph or sentence, the value will be updated to be the command which started this new paragraph or sentence.

@PATH  The pathname of the current account.

@PIB  The PROC primary input buffer. This variable can be modified by applications.

@POB  The PROC primary output buffer. This variable can be modified by applications.

@PSTAT  May be set by an application to contain diagnostic text that will be included in the output of the PSTAT command.

@QM.GROUP  The QM user group to which the user running this session belongs, a null string if none.

@QMSYS  The pathname of the system account.

@RECORD  The data of the record being processed by an I-type function. This variable may be updated by a QMBasic program.

@SELECTED  Contains the total record count for the most recent SELECT or SSELECT operation. Note that a QMBasic SELECT operation against a dynamic file processes the file one group at a time and this variable will show the record count for the group being processed. Note also that processing items from the select list does not decrement this value. To find the current state of a select list, use the SELECTINFO() function.

@SENTENCE  The currently active sentence. This is different from @COMMAND if the command runs a paragraph, sentence or menu.

@SEQNO  Hold file sequence number for most recent print job using uniquely sequenced file numbers.

@SIB  The PROC secondary input buffer. This variable can be modified by applications.

@SOB  The PROC secondary output buffer. This variable can be modified by applications.

@SOCKET  Socket file variable in a phantom process started with socket inheritability enabled. This variable can be modified by applications.
@STDFIL  Default file variable.

@SYSTEM.RETURN.CODE  A status value returned from most commands.

@SYS.BELL  This variable is available to QMBasic programs and initially contains the ASCII BEL character (character 7) which, when sent to the display, causes the audible warning to sound. The BELL OFF command changes @SYS.BELL to a null string and BELL ON reverts to the default character. Thus use of @SYS.BELL in QMBasic programs results in an audible alarm which can be disabled by the user.

@TERM.TYPE  Terminal type.

@TIME  The internal format time value (seconds since midnight) at which the last command started execution. This value may be updated by a QMBasic program.

@TRANSACTION.ID  The unique id number for the currently active transaction. Zero if no transaction is active. Same as SYSTEM(1007).

@TRANSACTION.LEVEL  The transaction depth. Zero when no transaction is active, incremented for each active transaction, decremented when a transaction terminates. Same as SYSTEM(1008).

@TRIGGER.RETURN.CODE  A status value returned set by trigger functions that return a STATUS() value of ER$TRIGGER.

@TTY  Terminal device name. This variable is provided for compatibility with other systems. It contains one of the following values:
- console  QMConsole interactive session on Windows
- /dev/...  QMConsole interactive session on other platforms
- telnet  Telnet session
- phantom  Phantom process
- port  Serial port connection
- startup  Process started from STARTUP configuration parameter
- vbsrvr  QMClient process
- Other process types may be added in future.

@UID  User's user id number for all platforms except Windows. Same as SYSTEM(27).

@USER  Synonym for @LOGNAME

@USER0 to @USER4  These variables are initially set to zero and may be updated by QMBasic programs to provide status information, etc. QM places no rules on the use of these variables and does not update them at any time.

@USERNO  User number.

@USER.NO  Synonym for @USERNO.

@USER.RETURN.CODE  This variable is initially set to zero and may be updated by QMBasic programs to provide status information, etc. QM places no rules on the use of this variable and does not update it at any time.

@VOC  This @VOC variable can be used as the file variable for the VOC in place of opening it explicitly within user written application code.

@WHERE  The current source line number in the module being compiled together with line numbers in nested include records in the form "21.4.3". Possibly useful in error messages. See also @LINE.
@WHO
User's account name.

@YEAR
The last two digits of the year in which the last command started execution. Changing @DATE will also change this value.

@YEAR4
The four digit year number in which the last command started execution. Changing @DATE will also change this value.
6.8 Standard Subroutines and Classes

QMBasic includes a set of standard subroutines and object oriented programming class modules that may be called from user programs. These all have an exclamation mark prefix to the subroutine or class name.

- `!ABSPATH()` Form an absolute pathname from a directory and file path
- `!ACCOUNT.RULES` Determine list of accounts that a user may or may not enter
- `!ATVAR()` Return value of an @-variable
- `!CALLHTTP` Class module to send HTTP request
- `!CED()` Subroutine interface to the data collection editor
- `!CHMOD()` Set file permissions
- `!CLOG()` Command logger class module
- `!DUMP()` Show string in hexadecimal and character format.
- `!ERRLOG.FILTER()` User supplied error log message filter. See error logging
- `!ERRTEXT()` Return text description of an error number
- `!EXC.STACK()` Display the call stack at an exception.
- `!FINDPROG()` Return information about catalogued program
- `!GETPU()` Get print unit characteristics
- `!LISTU()` Get raw data similar to LISTU command
- `!OSCOPY()` Copy an operating system file or directory
- `!PACKAGE()` Add, update or remove a registered package licence
- `!PATHTKN()` Process special tokens in a VOC or ACCOUNTS file pathname
- `!PARSER()` Command line parser
- `!PCL()` PCL control code functions
- `!PHLOG()` Return the log record name for a running phantom process
- `!PICK()` Display a pick list of options
- `!PICKLIST()` Display a pick list of options
- `!PSTAT()` Get raw data similar to PSTAT command
- `!QMCLIENT()` QMClient interface from QMBasic (class module)
- `!SCREEN()` Screen driver
- `!SETPU()` Set print unit characteristics
- `!SETVAR()` Set the value of an @-variable
- `!SORT()` Sort a delimited list
- `!USERNAME()` Return user name for a given user number
- `!USERNO()` Return a list of user numbers for a given user name
- `!VIEW()` Display a field mark delimited dynamic array in scrollable form
- `!VOCREC()` Read a VOC record, following remote pointers

**UniVerse / Prime Information Compatibility Subroutines**

Both UniVerse and Prime Information include subroutines that aid portability but in most cases are better written using other features of Basic. QM provides a tool that will add many of these compatibility subroutines to the global catalogue. To use this tool, type

```
*UVSUBS
```

at the QM command prompt. This will catalogue the subroutines with an exclamation mark prefix character. A hyphen prefix may be used instead by typing

```
*UVSUBS PREFIX -
```

The subroutines can be removed from the catalogue using the same commands but with the DELETING keyword added.

The subroutines installed by this tool and their QMBasic equivalents are
CALL !ADDS(result, a, b)  \[\text{result} = \text{ADDS}(a, b)\]
CALL !ANDS(result, a, b)  \[\text{result} = \text{ANDS}(a, b)\]
CALL !BPIOCP()  \[\text{PRINTER RESET}\]
CALL !CATS(result, a, b)  \[\text{result} = \text{CATS}(a, b)\]
CALL !CHARS(result, a)  \[\text{result} = \text{CHARS}(a)\]
CALL !COUNTS(result, a, b)  \[\text{result} = \text{COUNTS}(a, b)\]
CALL !DIVS(result, a, b)  \[\text{result} = \text{DIVS}(a, b)\]
CALL !EQS(result, a, b)  \[\text{result} = \text{EQS}(a, b)\]
CALL !EXIST(a, b)  \[b = \text{CATALOGUED}(a) \neq 0\]
CALL !FIELDS(result, a, b, c, d)  \[\text{result} = \text{FIELDS}(a, b, c, d)\]
CALL !FMTS(result, a, b)  \[\text{result} = \text{FMTS}(a, b)\]
CALL !FOLDS(result, a, b)  \[\text{result} = \text{FOLDS}(a, b)\]
CALL !GES(result, a, b)  \[\text{result} = \text{GES}(a, b)\]
CALL !GET.USERS(a, b, c, d, e)  \[\text{No simple equivalent}\]
CALL !GTS(result, a, b)  \[\text{result} = \text{GTS}(a, b)\]
CALL !HUSHIT(a)  \[\text{Use of HUSH statement}\]
CALL !ICONVS(result, a, b)  \[\text{result} = \text{ICONVS}(a, b)\]
CALL !IDENT(result)  \[\text{result} = \@who : \@fm : \@logname : \@fm : \@userno\]
CALL !IFS(result, a, b, c)  \[\text{result} = \text{IFS}(a, b)\]
CALL !INDEXS(result, a, b, c)  \[\text{result} = \text{INDEXS}(a, b, c)\]
CALL !LENS(result, a)  \[\text{result} = \text{LENS}(a)\]
CALL !LES(result, a, b)  \[\text{result} = \text{LES}(a, b)\]
CALL !LTS(result, a, b)  \[\text{result} = \text{LTS}(a, b)\]
CALL !MULS(result, a, b)  \[\text{result} = \text{MULS}(a, b)\]
CALL !NEGS(result, a)  \[\text{result} = \text{NEGS}(a)\]
CALL !NES(result, a, b)  \[\text{result} = \text{NES}(a, b)\]
CALL !NOTS(result, a)  \[\text{result} = \text{NOTS}(a)\]
CALL !NUMS(result, a)  \[\text{result} = \text{NUMS}(a)\]
CALL !OCONVS(result, a, b)  \[\text{result} = \text{OCONVS}(a, b)\]
CALL !ORS(result, a, b)  \[\text{result} = \text{ORS}(a, b)\]
CALL !PTERM(a)  \[\text{EXECUTE 'PTERM' : a}\]
CALL !SEQS(result, a)  \[\text{result} = \text{SEQS}(a)\]
CALL !SLEEP(a)          NAP a
CALL !SPACES(result, a) result = SPACES(a)
CALL !SPLICE(result, a, b, c) result = SPLICE(a, b, c)
CALL !STRS(result, a, b) result = STRS(a, b)
CALL !SUBS(result, a, b) result = SUBS(a, b)
CALL !SUBSTRINGS(result, a, b, c) result = SUBSTRINGS(a, b, c)
CALL !SUMMATION(result, a) result = SUMMATION(a)
CALL !TRIMSBS(result, a) result = TRIMBS(a)
CALL !TRIMFS(result, a) result = TRIMFS(a)
CALL !TRIMS(result, a) result = TRIMS(a)
CALL !USER.TYPE(a, b) a = if @tty='phantom' then 65 else 1; b = system(1050)
!ABSPATH() subroutine forms an absolute pathname from a directory and file path.

**Format**

```call !ABSPATH(path, dir, file)```

where

- `path` is the returned absolute pathname.
- `dir` is the directory path to be used when prefixing the pathname.
- `file` is the file path to be processed.

The `!ABSPATH()` subroutine uses the supplied directory and file path to construct an absolute pathname.

- If `file` commences with `@QMSYS`, `path` is returned as the `file` value with the `@QMSYS` token replaced by the QMSYS account pathname.
- If `file` commences with a directory separator character, `path` is returned as `file`.
- If `file` commences with a Windows drive specification, `path` is returned as `file`.
- Otherwise, `path` is formed by concatenating `dir` and `file`, inserting a directory separator character if required.

**Examples**

<table>
<thead>
<tr>
<th>Dir</th>
<th>File</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any</td>
<td>@QMSYS\ACCOUNTS</td>
<td>C:\QMSYS\ACCOUNTS</td>
</tr>
<tr>
<td>Any</td>
<td>\SALES\CUSTOMERS</td>
<td>\SALES\CUSTOMERS</td>
</tr>
<tr>
<td>Any</td>
<td>C:\SALES\CUSTOMERS</td>
<td>C:\SALES\CUSTOMERS</td>
</tr>
<tr>
<td>C:\SALES</td>
<td>CUSTOMERS</td>
<td>C:\SALES\CUSTOMERS</td>
</tr>
<tr>
<td>C:\</td>
<td>CUSTOMERS</td>
<td>C:\CUSTOMERS</td>
</tr>
</tbody>
</table>
!ACCOUNT.RULES

The !ACCOUNT.RULES() subroutine returns a list of QM accounts that a specified user may or may not enter.

Format

CALL !ACCOUNT.RULES(username, accounts, barred)

where

username is the user name to be looked up.
accounts is a multivalued list of account names.
barred is True if the list of accounts is those from which the user is barred, False if it is a list of allowed accounts.

The !ACCOUNT.RULES() subroutine is part of the QM application level security system. Within this system, a user name can be registered as having access only to a restricted set of accounts. Alternatively, a user can be allowed access to all except specific accounts.

The subroutine looks at the QM account register and returns the list of accounts and a flag indicating whether this is a list of allowed accounts or barred accounts. The username is case insensitive on Windows but case sensitive on all other environments.

A null accounts list is returned if the user name is not in the user register or if no account rules are applied to the user.

See also:
Application level security, ADMIN.USER
!ATVAR()

The !ATVAR() subroutine retrieves the value of an @-variable.

Format

CALL !ATVAR(value, name)

where

value is the returned value.

name is the name of the @-variable to be retrieved. The leading @ character may optionally be omitted. Variable names are case insensitive.

The !ATVAR() subroutine returns the value of the named @-variable. Although intended for accessing user defined variables, it can also return the standard variables.

The !ATVAR() function sets the value returned by the STATUS() function. This will be zero if the specified variable is found, non-zero if it is not recognised.

Example

CALL !ATVAR(VALUE, "@MYVAR")

or

DEFFUN ATVAR(NAME) CALLING "!ATVAR"
VALUE = ATVAR("@MYVAR")

Both of these examples retrieve the value of the @MYVAR variable.

See the !SETVAR() subroutine for a way to set the value of an updateable @-variable.
!CALLHTTP

The !CALLHTTP class module allows an application to initiate HTTP style requests to a remote server.

The object is instantiated with a statement of the form

```
SESSION = OBJECT('!CALLHTTP')
```

Five HTTP request message types are supported; GET, POST, HEAD, PUT and DELETE. Each has a corresponding public function in the class that takes the request message as its only argument and returns a Boolean value indicating whether the action was successful.

The connection details are set using the following properties prior to calling the request function:

- **HOST** The server name or IP address.
- **PORT** The server port number. Defaults to 80 if not specified.
- **PARAMS** For a GET or HEAD action, if the request message does not contain a "?" character and the PARAMS property is not a null string, a "?" and the PARAMS property value are appended to the request message. For a POST message, if the PARAMS property is not a null string, it is sent as the message body. The PARAMS property is reset to a null string after all request messages.
- **EXT.HDR** Additional message header items for GET or HEAD as a dynamic array with a field for each item containing a name/value pair in the form "name: value". The header items will be transmitted in the order in which they appear in the dynamic array. Alternatively, the additional items may be set as a single level data collection in which case the order of transmission is the collating sequence of the names in the collection.
- **AUTH** Authentication details in the form "user:password" for a request requiring authentication.

The server and port can alternatively be specified as part of instantiating the object though actual connection does not occur until a request is submitted.

```
SESSION = OBJECT('!CALLHTTP', host, port)
```

The parameters to a GET, HEAD, POST or PUT message can be provided in three mutually exclusive manners using public functions that return a true/false status:

- As part of the request message argument
  ```
  SESSION->GET('/?t0=h&t1=openqm')
  ```
- As the second argument to the GET message
  ```
  SESSION->GET('/', '?t0=h&t1=openqm')
  ```
- By use of the PARAMS property
  ```
  SESSION->PARAMS = 't0=h&t1=openqm'
  SESSION->GET('/')
  ```

On return from a GET, HEAD or POST request, the following properties contain information returned from the server:

- **HEADER** The HTTP response header as a data collection with each item as a name/value pair.
- **HTTP.STATUS** The HTTP status code (e.g. 404 for item not found), zero if no HTTP error.
- **BODY** For a GET or POST request, this is the response body.
If the request returns False, the error message is available via the ERROR property.

Examples

```qbasic
SESSION = OBJECT('!CALLHTTP')
SESSION->HOST = 'www.openqm.com'
IF SESSION->GET('/downloads.htm') THEN
   DISPLAY SESSION->BODY
END ELSE
   DISPLAY SESSION->ERROR
END
```

The above program fragment uses the !CALLHTTP class to retrieve an item from the OpenQM web site (actually just a redirection link). If successful, the page body is displayed. If an error occurs, the error message is displayed.

```qbasic
SESSION = OBJECT('!CALLHTTP', 'www.openqm.com')
IF SESSION->GET('/?t0=h&t1=openqm') THEN
   DISPLAY SESSION->BODY
END ELSE
   DISPLAY SESSION->ERROR
END
```

The above program fragment uses a GET message that includes parameters. It also shows how the host name can be passed in the object instantiation rather than by explicitly setting the HOST property.

```qbasic
SESSION = OBJECT('!CALLHTTP', 'www.openqm.com')
SESSION->PARAMS = 't0=h&t1=openqm'
IF SESSION->GET('/') THEN
   DISPLAY SESSION->BODY
END ELSE
   DISPLAY SESSION->ERROR
END
```

The above program fragment is a variation on the preceding example, using the PARAMS property to define the parameters to be appended to the request message. Substituting POST for GET as the session method on the third line of this example would send the request as a POST message.

See also:
Data collections, Simple Web Services
!CED()

The !CED() subroutine invokes the data collection editor for a collection passed as an argument.

Format

CALL !CED(collection)

where

collection is the data collection to be processed. This may be updated on return.

The !CED() subroutine allows a data collection variable to be viewed or modified. For details of use, see the CED command.

Example

VAR = JPARSE(JSON.STRING)
CALL !CED(VAR)

This example uses JPARSE() to parse a JSON (JavaScript Object Notation) string into a data collection and then allows the user to view or edit this data.

See also:
Data collections, CED
!CHMOD

The !CHMOD() subroutine sets the mode value (permissions) for a given pathname.

Format

CALL !CHMOD(status, pathname, modes)

where

status is the returned status value.
pathname is the pathname of the file system item to be amended.
modes is the required mode value (See below).

The !CHMOD() subroutine is equivalent to use of the Linux chmod command to set the mode value (permissions) for a given file. If successful, the status argument is set to True. If unsuccessful, status is returned as False. The subroutine is not applicable to Windows systems and will always return False on such systems.

The subroutine argument structure allows it to be defined and used as a function as shown in one of the examples below.

File permission values in Linux are usually represented as a three digit octal value with one digit for the permissions applicable to the file owner, one for other members of the group to which the file is assigned, and one for the permissions applicable to all other users. Each digit is formed from adding three elements; 4 for read permission, 2 for write permission, and 1 for execute permission. The !CHMOD() subroutine modes argument is a character string containing this three digit octal value. Because of the type variant nature of QMBasic, the modes can also be written as a numeric value which will be treated as a character string internally by the subroutine.

Examples

CALL !CHMOD(OK, PATH, "744")

or

DEFFUN CHMOD(PATH, MODES) CALLING "!CHMOD"
OK = CHMOD(PATH, 744)

Both of these examples set the permissions associated with the file identified by PATH to 744 (rwxr--r--).
!DUMP()

The !DUMP() subroutine shows string data in hexadecimal and character form.

Format

CALL !DUMP(text [, lptr])

where

- **text** is the string to be printed.
- **lptr** is the optional print unit number for the output. If omitted, the output is sent to the screen.

The !DUMP() subroutine is intended to be used as a diagnostic aid in programs that work with binary data. The **text** string is reported in the same form as use of the DUMP command to show the contents of a database record.

```
00000000: 43 61 72 74 77 72 69 67 68 74 2C 20 44 FE 37 20 | Cartwright, D^7
00000010: 53 70 72 69 6E 67 20 47 72 6F 76 65 FD 4E 6F 74 | Spring Grove]Not
00000020: 74 69 6E 67 68 61 6D FE 31 2D 31 FD 33 2D 31 | tingham"1-1|3-1
```

In an ECS mode system, if the data contains any characters outside the 8-bit set, each character is shown as four hexadecimal digits. The above example becomes

```
00000000: 0043 0061 0072 0074 0077 0072 0069 0067 0068 0074 002C 0020 0044 00FE 0037 20 | Cartwright, D^7
00000008: 0068 0074 002C 0020 0044 00FE 0037 20 | ht, D^7
00000009: 0053 0070 0072 0069 0067 0068 0061 006D 00FE 0037 20 | Spring Grove]Not
00000008: 0074 006E 006F 006E 0067 0068 0061 006D 00FE 0037 20 | tingham"1-1|3-1
```
!ERRTEXT()

The !ERRTEXT() subroutine returns a text description of an error number.

Format

CALL !ERRTEXT(text, qm.error[, os.error])

where

- **text** is the returned descriptive text.
- **qm.error** is the QM error number, typically from the STATUS() function.
- **os.error** is the operating system error number, typically from the OS.ERROR() function. If this argument is omitted, it defaults to the current value of the OS.ERROR() function.

The !ERRTEXT() subroutine can be used to retrieve a text description of a QM error number for display to a user or entry into a log file.

Where relevant, the associated operating system error number will be inserted into the text. For this to be correct, the !ERRTEXT() subroutine must either be called before any actions are performed that might lose this value (e.g. file operations) or the error number must be passed as the optional os.error argument.

If qm.error is not recognised, the subroutine returns qm.error as the text description.

Examples

CALL !ERRTEXT(TEXT, STATUS())
DISPLAY 'Error ' : STATUS() : ' ' : TEXT

or

DEFFUN ERRTEXT(ERRNO) CALLING "/!ERRTEXT"
DISPLAY 'Error ' : STATUS() : ' ' : ERRTEXT(STATUS())
!EXC.STACK()

The !EXC.STACK() subroutine displays a report based on the content of the @EXCEPTION.STACK variable after an exception is thrown.

Format

CALL !EXC.STACK

This call executes a system supplied subroutine that contains

```plaintext
IF @EXCEPTION.STACK # '' THEN
  DISPLAY ' Program name.................. Addr.. Line'
  LEVEL = 1
  FOR EACH PROG IN @EXCEPTION.STACK DELIMITER @FM
    NAME = FMT(PROG<1,1>, '30L')
    DISPLAY FMT(LEVEL, 'R%3 ') :
    SUBDATA = FIELD(PROG<1>, @VM , 2, *)
    SUBADDR = VSLICE(SUBDATA, 0, 1)
    SUBLINE = VSLICE(SUBDATA, 0, 2)
    LOOP
      DISPLAY FMT(REMOVE(NAME, MORE.NAME), '30L') : ' ' :
      DISPLAY DTX(REMOVE(SUBADDR, MORE.ADDR), 6) : ' ' :
      DISPLAY REMOVE(SUBLINE, MORE.LINE)
      WHILE MORE.NAME OR MORE.ADDR OR MORE.LINE
        DISPLAY ' ' :
        REPEAT
        LEVEL += 1
      NEXT PROG
  END

Note that @EXCEPTION.STACK is only set if the CATCH clause that catches the exception uses the SAVING.STACK qualifier. This allows applications to use exceptions as part of the normal flow of execution rather than only for error handling without paying the performance cost of gathering the stack information.

See also:

Exception handling, THROW, TRY/CATCH
!FINDPROG()

The !FINDPROG() subroutine searches the catalogue for a named program, returning information about it.

Format

CALL !FINDPROG(result, name)

where

result is the returned data.

name is the case insensitive name of the program to find.

The !FINDPROG() subroutine examines the three catalogue areas in the same order as used when searching for a program within an application (local, private, global). For each instance of the program that is found, information about that program is returned in dynamic array result. This dynamic array has five multivalued fields with a value for each program found:

1. Catalogue mode: L (local), P (private), G (global).
2. Source pathname, if available.
3. Compile time as an epoch value.
4. Comments (from $* and $COPYRIGHT), possibly multi-subvalued.
5. The full pathname of the object code.

Applications using this subroutine should allow for further fields to be added to the returned data in future releases.

If the program is not found, result will be returned as a null string.

Example

CALL !FINDPROG(result, 'INVOICE')
NUM.INSTANCES = DCOUNT(result<1>, @VM)
FOR I = 1 TO NUM.INSTANCES
  DISPLAY result<1,I> : ' ' : result<2,I>
NEXT I

This program fragment searches for a catalogued program named INVOICE and, for each instance found, displays the catalogue mode and the source pathname.

See also:
FIND.PROGRAM, CATALOGUED()
!GETPU()

The !GETPU() subroutine gets the characteristics of a print unit.

Format

CALL !GETPU(key, unit, value, status)

where

key identifies the parameter to retrieved. This is as for the GETPU() function.

unit evaluates to the print unit number.

value is the variable to receive the given parameter.

status is the return status value. Zero if the action is successful, a non-zero error code if the action fails.

The !GETPU() subroutine retrieves the print unit characteristic specified by key, storing it in value. It is closely related to the GETPU() function.

Example

CALL !GETPU(PU$MODE, 3, MODE, STATUS)

The above statement gets the mode of print unit 3, storing it in MODE.
!LISTU()

The !LISTU() subroutine returns the raw data used by the LISTU command.

Format

\[
\text{CALL !LISTU}(\text{dyn.array})
\]

where

\[
\text{dyn.array} \quad \text{is the dynamic array returned by the subroutine.}
\]

The !LISTU() subroutine allows a developer to construct their own variant of LISTU, perhaps as a screen within an application.

The data returned in \text{dyn.array} is a dynamic array where each field is multivalued with a value for each active QM process in ascending user number order. The fields are:

1. User number.
2. Operating system level process id. This can be negative on some Windows platforms.
4. IP address for a direct network connection to QM. Entry from the operating system command prompt will not set this.
5. Parent user id for a phantom process. This is zero if the process is not a phantom or if the parent process has logged out.
6. User name
7. Terminal device name (where relevant)
8. Login time as date \* 86400 + time in the local time zone of the user executing the subroutine.
9. Account name
10. Login time as an epoch value
11. Transaction data. Three subvalues (id, start time as epoch value, command processor level). Where a process has nested transactions, only the top level is reported.
12. Process sub-type. Currently only applies to phantoms where this value will be IP for an interactive phantom, null for all other process types.

Further fields or process types may be added in future releases.

User type counts can be retrieved in a QMBasic program using SYSTEM(1056).

Example

\[
\begin{align*}
\text{CALL !LISTU(USERS)} \\
\text{NUM.USERS = DCOUNT(USERS<1>, @VM)} \\
\text{FOR I = 1 TO NUM.USERS} \\
\quad \text{IF USERS<3,I> = 'I' THEN} \\
\quad \quad \text{DISPLAY FMT(USERS<1,I>, '3R') : ' ' : USERS<6,I>} \\
\quad \text{END} \\
\text{NEXT I}
\end{align*}
\]
The above program fragment displays a list of the user numbers and user login names of all interactive processes.

See also:
LISTU
The !MULTISORT() subroutine sorts multiple related lists simultaneously.

**Format**

```
CALL !MULTISORT(mode, keys, data1 [, data2...])
```

where

- `mode` is the sort mode to be used.
- `keys` is a field mark delimited list to be sorted.
- `data1` is a parallel list to be sorted, maintaining pairing of items with `keys`. A minimum of 1 and a maximum of 10 `data` lists may be specified.

The !MULTISORT() subroutine sorts multiple field mark delimited lists in parallel, maintaining the relationship between items in the `keys` list and the `data` lists. The `keys` and `data` lists are updated on return.

The subroutine uses char(0) internally so the lists to be sorted must not contain this character.

The mode argument determines how the data will be sorted

At most one of the following sort styles:

- **0 SRT$LEFT** A simple left justified comparison, examining corresponding characters from the start of each string until either a difference is found or the end of both strings has been reached.
- **1 SRT$RIGHT** A simple right justified comparison in which the shorter item is effectively padded with leading spaces and the resultant strings are compared character by character from the start until either a difference is found or the end of both strings has been reached. If both strings can be treated as integer values, possibly with a leading sign character, a numeric comparison is performed.
- **2 SRT$COMPOUND** A compound sort in which the two strings are considered to be formed from a series of alternating numeric and non-numeric elements. Numeric elements are sorted into numerical value, non-numeric elements are sorted into collating sequence order. If the first element is numeric, it may have an optional leading sign character. Sign characters appearing later in the strings are treated as being non-numeric characters.
- **3 SRT$RIGHT.FLOAT** The same as mode 1 except that the test for numeric items allows non-integer values. The same as mode 1 except that the test for numeric items allows non-integer values.

Any combination of the following sort options:

- **16 SRT$DESCENDING** The result of the comparison operation is reversed.
- **32 SRT$UNIQUE** Duplicate key values are ignored.
64 SRT$NOCASE      Case insensitive sort.

Example

KEYS = CHANGE('17|4|38|25|66', '|', @FM)
DATA = CHANGE('A|B|C|D|E', '|', @FM)
CALL !MULTISORT(SRT$RIGHT, KEYS, DATA)
FOR I = 1 TO 5
    DISPLAY KEYS<I>, DATA<I>
NEXT I

The above program fragment sets up KEYS and DATA as two parallel lists. The CHANGE() function is used here for clarity as a literal string cannot contain a field mark. After using the !MULTISORT() subroutine the displayed data would be:

4   B
17  A
25  D
38  C
66  E

See also:
SORT()
The `!OSCOPY()` subroutine copies an operating system level file or directory to a specified location.

**Format**

```plaintext
CALL !OSCOPY(src.path, tgt.path [, no.query [, flush [, append]]])
```

where

- `src.path` is the pathname of the file or directory to be copied.
- `tgt.path` is the pathname of destination.
- `no.query` is a Boolean value that determines whether a query prompt is displayed to confirm that the target directory is to be created. This argument is optional and defaults to True.
- `flush` is a Boolean value that determines whether the file cache should be flushed prior to the copy. This argument is optional and defaults to True. See `FLUSH.DH.CACHE` for more information.
- `append` is a Boolean value that, if True, causes `!OSCOPY()` to append to `tgt.path` if `src.path` is a file that already exists.

The `!OSCOPY()` subroutine provides an easy way to copy an operating system file or directory.

In all cases, `tgt.path` is the pathname of the new file or directory, not the parent directory into which items will be copied.

The `!OSCOPY()` function locks each item as it progresses. When copying a QM hashed file, it is essential that the file is not open in any QM process as this may result in incorrect data being copied.

On return from `!OSCOPY()`, the `STATUS()` function with return zero if the copy was successful or an error code if the copy failed. Note that failure may occur at any stage of copying a directory structure and updates performed before the error will not be undone.

**Example**

```plaintext
CALL !OSCOPY("C:\SALES", "C:\BACKUPS\SALES", @TRUE)
```

The above program fragment copies `C:\SALES` (perhaps an operating system directory representing a QM hashed file) to a subdirectory of the same name under `C:\BACKUPS`. The third argument suppresses the prompt that would otherwise occur if the `C:\BACKUPS` directory does not exist.

**See also:**

`OSCOPY`
!PACKAGE()

The !PACKAGE() subroutine allows an application installer to add or remove a package licence.

Format

CALL !PACKAGE(result, name, auth.string)

where

result is the variable to receive status information.
name identifies the package.
auth.string is the licence authorisation code.

The !PACKAGE() subroutine maintains the registered package licence data.

The subroutine returns result as zero if successful or an error code if it fails.

The subroutine can also be declared as a function:

DEFFUN PACKAGE(NAME, AUTH.STRING) CALLING "!PACKAGE"

and then used as

ERR = PACKAGE(NAME, AUTH.STRING)

Example

CALL !PACKAGE(ERR, "MYPKG", "WHDY-BJGD-LNDD-PPQH")
IF ERR THEN DISPLAY 'Error ' : ERR : ' licensing package'

See also:
Package licensing, ENTER.PACKAGE(), EXIT.PACKAGE(),
PACKAGE.LICENSE, !PACKAGE()
The `!PARSER()` subroutine parses a command line.

### Format

```
CALL !PARSER(key, type, string, keyword {, voc.rec, quote.char})
```

where

- **key** identifies the operation to be performed:
  - 0 `PARSER$RESET` Prepares to parse the data in `string`.
  - 1 `PARSER$GET.TOKEN` Returns the next token from the data.
  - 2 `PARSER$GET.REST` Returns all remaining tokens as a single string.
  - 3 `PARSER$EXPAND` Inserts `string` before the remaining tokens.
  - 4 `PARSER$LOOK.AHEAD` Previews the next token.
  - 5 `PARSER$MFILE` Like `PARSER$GET.TOKEN` but allows multifile syntax.

- **type** is the returned token type:
  - 0 `PARSER$END` End of data reached.
  - 1 `PARSER$TOKEN` A token has been returned in `string`.
  - 2 `PARSER$STRING` A quoted string. The quotes are removed in `string`.
  - 3 `PARSER$COMMA` A comma has been found.
  - 4 `PARSER$LBR` A left bracket has been found.
  - 5 `PARSER$RBR` A right bracket has been found.

- **string** is the returned token string. For `key` values 0 and 3, this is the string passed into the parser.

- **keyword** is the returned token keyword number as defined in the VOC and in the SYSCOM `PARSER.H` record. This is negative if the token is not a VOC keyword. This argument is ignored for `key` values 0 and 3.

- **voc.rec** is an optional argument, returned as the VOC record when `string` corresponds to a VOC key.

- **quote.char** is an optional argument, returned as the quoting character for `PARSER$STRING` type items.

The `!PARSER()` subroutine can be used to parse the elements of a command line.

### Example

```
CALL !PARSER(PARSER$RESET, 0, @SENTENCE, 0)
CALL !PARSER(PARSER$GET.TOKEN, TOKEN.TYPE, STRING, KEYWORD) ;* Verb
```
LOOP
    CALL !PARSER(PARSER$GET.TOKEN, TOKEN.TYPE, STRING, KEYWORD)
UNTIL TOKEN.TYPE = PARSER$END
    ...process token...
REPEAT
!PATHTKN()

The !PATHTKN() subroutine processes special tokens in a VOC or ACCOUNTS file pathname.

Format

CALL !PATHTKN(outpath, inpath)

where

outpath     is the returned processed pathname. This may be the same variable as inpath.
inpath      is the pathname to be processed. This may be the same variable as outpath.

Pathnames recorded in the VOC or the QMSYS ACCOUNTS file may include special tokens that represent variable components. The !PATHTKN() subroutine processes a pathname, substituting the expansions for these tokens.

The special tokens are:

@DRIVE     The drive letter for the QMSYS directory (Windows only).
@HOME       The user's home directory as defined by the corresponding operating system environment variable (HOMEPATH on Windows, HOME elsewhere).
@QMSYS      The full pathname of the QMSYS directory.
@TMP        The pathname of the system temporary directory as defined by the TEMPDIR configuration parameter.

The token must be the leading part of the pathname.

The !PATHTKN() subroutine is also defined as a function in the SYSCOM KEYS.H include record:

outpath = PARSE.PATHNAME.TOKENS(inpath)

Examples

The entry for the QMSYS account in the ACCOUNTS register is simply @QMSYS

This ensures that the entire system can be moved without needing to update the QMSYS account location.

When using QM installed on a USB flash drive on Windows, creating an account on the USB device sets the ACCOUNTS register entry as @DRIVE:pathname

The account is therefore accessible even if the flash drive takes on a different drive letter in later use.
The !PCL() subroutine constructs PCL control strings for various useful operations. It is intended for use as a series of functions defined in the SYSCOM PCL.H include record.

**Format**

```call !pcl(string, key, arg1, ...)```

where

- `string` is the returned control string.
- `key` identifies the operation to be performed. See the PCL.H include record for the relationship between the `key` values and the functions described below.
- `arg1,...` are arguments defining the exact action. The !PCL() subroutine takes a variable length argument list.

The !PCL() subroutine should be called using the function interfaces defined below. The returned string can be sent to a PCL compatible printer to perform the requested action.

All page positioning values are measured using the PCL coordinate grid where (0,0) is at the top left of the page and the grid scale is 300 per inch. There is a useful grid template printing program, PCL.GRID, in the BP file of the QMSYS account.

**Note:** The quality of PCL implementations varies widely and these functions may not give the expected results on some printers. In particular, setting some font metrics may cause inconsistent character placement. It is the application developer’s responsibility to ensure that the printed results are acceptable.

The following functions are defined in the SYSCOM PCL.H include record. Each returns the relevant control string to perform its action.

- **PCL.BOX(left, top, width, height, pen.width, radius)**
  Draws a rectangular box using the given position (left, top) and size (width, height) values. The `pen.width` determines the width of the lines used to draw the box. The `radius` value determines the radius of rounded corners. A value of zero results in square corners.

- **PCL.Copies(copies)**
  Sets the number of copies to be printed.

- **PCL.Cursor(x, y)**
  Sets the current cursor position to the given coordinates. Subsequent text output will occur at this point.

- **PCL.Duplex(mode)**
  Sets duplex mode. 0 = off, 1 = long edge binding, 2 = short edge binding.

- **PCL.Font(font)**
  Sets the font details for text output. The `font` argument consists of comma separated list of case insensitive items chosen from the following list. Features that are not specified retain their previous values. Not all printers support all options.

  Font names: ARIAL, COURIER, CG-TIMES, LETTER-GOTH, LINEPRINTER, UNIVERS

  Character sets: ASCII, LATIN-1, PC-8, ROMAN-8
Type style: UPRIGHT, COMPRSSED, CONDENSED, CONDENSEDITALIC, EXPANDED, INLINE, ITALIC, OUTLINE, OUTLINESHADOW, SHADOW
Weight: ULTRA-THIN, EXTRA-THIN, THIN, EXTRA-LIGHT, LIGHT, DEMI-LIGHT, SEMI-LIGHT, MEDIUM, SEMI-BOLD, DEMI-BOLD, BOLD, EXTRA-BOLD, BLACK, EXTRA-BLACK, ULTRA-BLACK
Spacing: FIXED, PROPORTIONAL
Size: nPT (point size), PITCH n (characters per inch)
Composite: REGULAR (equivalent to UPRIGHT, MEDIUM)

PCL.HLINE(x, y, length, pen.width)
Draws a horizontal line starting at the given position and extending to the right, using the specified length and pen.width values.

PCL.LEFT.MARGIN(col)
Sets the left margin at column col.

PCL.ORIENTATION(layout)
Specifies the page format. The layout argument is a string and may be PORTRAIT or LANDSCAPE.

PCL.PAPER.SIZE(size)
Specifies the page size. The size argument is a string chosen from A3, A4, B5, C5, COM10, DL, EXECUTIVE, LEDGER, LEGAL, LETTER and MONARCH.

PCL.RESET()
Resets the printer.

PCL.RESTORE.CURSOR()
Restores a previously saved cursor position.

PCL.SAVE.CURSOR()
Saves the current cursor position. Note that there is a limit to the number of nested cursor save operations.

PCL.VLINE(x, y, length, pen.width)
Draws a vertical line starting at the given position and extending downwards, using the specified length and pen.width values.

The source version of the PCL() subroutine is in the BP file of the QMSYS account so that users can add further options. A copy of any changes should be retained as this item will be replaced when an upgrade is installed.

Example

$INCLUDE PCL.H
PRINTER ON
PRINT PCL.RESET() :
PRINT PCL.FONT('Courier, Pitch 10, Regular') :
PRINT PCL.BOX(300,300,300,100,2,15) :
PRINT PCL.CURSOR(350,380) : 'Hello' :

The above program prints the word Hello in a box with rounded corners.
!PHLOG()

The !PHLOG() subroutine returns the name of the log record for a running phantom process.

Format

CALL !PHLOG(name, userno)

where

name is the returned log record name.

userno is the user number of the phantom process.

A phantom process normally creates a log record in the $COMO file containing any output that would have gone to the screen if the command had been run in an interactive session. The name of this log record is PHn_date_time where n is the QM user number of the phantom process and date_time is the date/time at which the phantom started in the form ddmmyy_hhmms.

The !PHLOG() subroutine allows a process to determine the name of the log record within the $COMO file for a phantom so long as the phantom is still active. If the phantom is started using the NO.LOG option of the PHANTOM command or it has terminated, name is returned as a null string.

The PHANTOM command provides a mechanism with which the logging data can be directed to an alternative location. When using this, the name returned by the !PHLOG() subroutine is formed from the logging file name and record name separated by a single space.

Because the !PHLOG() subroutine returns the log name via its first argument, it can be declared as a function as in the example below.

Example

DEFFUN PHLOG(U) CALLING "!PHLOG"
EXECUTE "PHANTOM OVERNIGHT.REPORT" SETTING UNO
DISPLAY "Log is " : PHLOG(UNO)

Note how the SETTING clause of the EXECUTE statement is used to capture the @SYSTEM.RETURN.CODE value from the PHANTOM command. Simply using @SYSTEM.RETURN.CODE as the argument to PHLOG() would not work as this would be the return status of the EXECUTE operation, not the executed command.

See also: CHILD(), LIST.PHANTOMS, PHANTOM, STATUS
!PICK()

The !PICK() subroutine displays a list of entries from which a user may select one.

Format

CALL !PICK(item, top.line, item.list, title, pos)

where

item is the returned item. This will be returned as a null string if no item is selected

top.line is the line number of the topmost display line to be used. The pick list display uses from this line to the bottom of the screen.

item.list is a field mark delimited list of items to display. Long items will be truncated to fit a single line.

title is a short text description of the items being processed. This is displayed alongside the item count at the bottom of the screen. This may be a null string to omit the title.

pos enables programs to return to a pick list at the position of a previously displayed item. On initial entry, this should be zero or a null string. If a variable name is used for pos, this variable will be updated to contain position information related to the list. A subsequent call to !PICK() using this updated value will display the screen as it was when the previous item was selected. Programs should not make any assumption about the format of this variable as it may change between QM releases.

The !PICK() subroutine displays a list of items as specified in list. The user can move through this list using the following keys:

- Down one line: Cursor down D Ctrl-N
- Up one line: Cursor up U Ctrl-P Ctrl-Z
- Down page: Page down N Ctrl-V
- Up page: Page up P Esc-V
- Top: Home T Esc-<
- Bottom: End B Esc->

Example

OPEN "ACCOUNTS" TO ACC.F ELSE STOP "Cannot open ACCOUNTS file"
SSELECT ACC.F
READLIST LIST ELSE NULL
CALL !PICK(ITEM, 0, LIST, "Accounts", POS)
IF ITEM # "" THEN
   READ ACC.REC FROM ACC.F, ITEM THEN
      ...processing...
END
END
The above example shows a list of records in the ACCOUNTS file and processes the selected record.

See also:
'PICKLIST()'
!PICKLIST()

The !PICKLIST() subroutine displays a list of entries from which a user may select one.

Format

```
CALL !PICKLIST(value, list, return.col, index.col)
```

where

- **value** is the returned item. This will be returned as a null string if no option is selected.
- **list** is a field mark delimited list of items to process. The display can show multiple items for each entry (e.g. a code and an expanded text) in which case each field is divided into values corresponding to the columns to be shown. The number of displayed columns is determined by the number of values in the first field.
- **return.col** identifies which column (from 1) of the selected item is to be returned as value.
- **index.col** is the column number (from 1) for shortcut entry. The list must be sorted into ascending order of this column. A value of zero implies that no shortcut is to be allowed.

The !PICKLIST() subroutine displays a box containing the items to from list. The user can use the up and down cursor keys to move through this list. If the list is longer than can be displayed, the subroutine will scroll the displayed items. The page up and down key can be used to move rapidly through the entries. The return key selects the current item, returning it in the value argument.

If index.col is non-zero, the user may enter the initial characters of an entry in the chosen column to position directly to the first entry starting with the entered prefix. The characters entered are displayed in the lower border of the box.

When used on a QMTerm or Windows QMConsole session or on a terminal that supports screen region save and restore, the area of the screen overwritten by the pick list box is automatically saved and restored. Programs using other terminal systems will need to arrange their own system to recover the screen.

Example

```
LIST = 'Blue':@FM:'Green':@FM:'Red'
CALL !PICKLIST(ITEM, LIST, 1, 1)
DISPLAY 'Selected item was ' : ITEM
```

See also:

- !PICK()
!PSTAT()

The !PSTAT() subroutine returns information about a QM process.

**Format**

```
CALL !PSTAT(data, userno)
```

where

- `data` is the variable to receive the result.
- `userno` is the user number of the QM process to be analysed.

The !PSTAT() subroutine returns a dynamic array containing process status data similar to that used by the PSTAT command.

<table>
<thead>
<tr>
<th>Field 1</th>
<th>Process information:</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>User number</td>
</tr>
<tr>
<td>V2</td>
<td>Process id</td>
</tr>
<tr>
<td>V3</td>
<td>Process type</td>
</tr>
<tr>
<td>V4</td>
<td>User name</td>
</tr>
<tr>
<td>V5</td>
<td>Client IP address for network connection</td>
</tr>
<tr>
<td>V6</td>
<td>Opcode</td>
</tr>
<tr>
<td>V7</td>
<td>Process wait cause</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field 2</th>
<th>Last command executed</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Field 3</th>
<th>Program stack. Each value corresponds to a program/subroutine in the call history. Each value is further divided into subvalues. The first subvalue holds the program name. Remaining subvalues contain the program offset and line number separated by a text mark, repeated for each internal subroutine layer (GOSUB). The line number is -1 if not known.</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Field 4</th>
<th>If waiting for a lock:</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>User number of process owning lock</td>
</tr>
<tr>
<td>V2</td>
<td>File pathname</td>
</tr>
<tr>
<td>V3</td>
<td>id (null if waiting for a file lock)</td>
</tr>
<tr>
<td>V4</td>
<td>Time started waiting as an epoch value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field 5</th>
<th>Account name</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Field 6</th>
<th>QMNet client id data (if this is a QMNet server process)</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>User name</td>
</tr>
<tr>
<td>V2</td>
<td>User number</td>
</tr>
<tr>
<td>V3</td>
<td>Process id</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Field 7</th>
<th>QMNet server data for open connections. One value per connection, each with five subvalues:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SV1</td>
<td>Server name</td>
</tr>
<tr>
<td>SV2</td>
<td>Number of open files</td>
</tr>
<tr>
<td>Field 8</td>
<td>Transaction data. Value per active transaction, each with three subvalues:</td>
</tr>
<tr>
<td>--------</td>
<td>---------------------------------------------------------------------------</td>
</tr>
<tr>
<td>SV1</td>
<td>Transaction id</td>
</tr>
<tr>
<td>SV2</td>
<td>Start time as epoch value</td>
</tr>
<tr>
<td>SV3</td>
<td>Command processor level</td>
</tr>
</tbody>
</table>

**Field 9**  
@PSTAT
## !QMCLIENT

The !QMCLIENT class module provides an object oriented interface to the QMClient API for use within QMBasic programs.

An QMClient object is instantiated using a QMBasic statement of the form

```qmbasic
session = object('!qmclient')
```

The table below lists the QMClient API calls and their actions available with this object. All names are case insensitive.

<table>
<thead>
<tr>
<th>Call Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QMConnect</td>
<td>bool = session-&gt;Connect(host, port, username, password, account)</td>
</tr>
<tr>
<td>QMConnectPool</td>
<td>bool = session-&gt;Connect(host, port, username, password, account, pool)</td>
</tr>
<tr>
<td>QMCall</td>
<td>session-&gt;Call(subr{, args})</td>
</tr>
<tr>
<td>QMClearFile</td>
<td>session-&gt;ClearFile(fno)</td>
</tr>
<tr>
<td>QMClearSelect</td>
<td>session-&gt;ClearSelect(listno)</td>
</tr>
<tr>
<td>QMClose</td>
<td>session-&gt;Close(fno)</td>
</tr>
<tr>
<td>QMConnected</td>
<td>bool = session-&gt;Connected</td>
</tr>
<tr>
<td>QMCreateObject</td>
<td>objno = session-&gt;CreateObject(class {, args})</td>
</tr>
<tr>
<td>QMDelete</td>
<td>session-&gt;Delete(fno, id)</td>
</tr>
<tr>
<td>QMDeleteu</td>
<td>session-&gt;Deleteu(fno, id)</td>
</tr>
<tr>
<td>QMDestroyObject</td>
<td>session-&gt;DestroyObject(objno)</td>
</tr>
<tr>
<td>QMDisconnect</td>
<td>session-&gt;Disconnect</td>
</tr>
<tr>
<td>QMEndCommand</td>
<td>session-&gt;EndCommand</td>
</tr>
<tr>
<td>QMEnterPackage</td>
<td>n = session-&gt;EnterPackage(name)</td>
</tr>
<tr>
<td>QMEvaluate</td>
<td>str = session-&gt;Evaluate(dict.fno, name, data, id)</td>
</tr>
<tr>
<td>QMEvalConv</td>
<td>str = session-&gt;EvalConv(dict.fno, name, data, id)</td>
</tr>
<tr>
<td>QMExecute</td>
<td>str = session-&gt;Execute(cmd, err)</td>
</tr>
<tr>
<td>QMExitPackage</td>
<td>n = session-&gt;ExitPackage(name)</td>
</tr>
<tr>
<td>QMGet</td>
<td>str = session-&gt;Get(objno, name {, args})</td>
</tr>
<tr>
<td>QMGetVar</td>
<td>str = session-&gt;GetVar(name)</td>
</tr>
<tr>
<td>QMIConv</td>
<td>str = session-&gt;Iconv(data, code)</td>
</tr>
<tr>
<td>QMIConvs</td>
<td>str = session-&gt;IConvs(data, code)</td>
</tr>
<tr>
<td>QMIndices</td>
<td>str = session-&gt;Indices(fno, name)</td>
</tr>
<tr>
<td>QMIsECS</td>
<td>bool = session-&gt;IsECS</td>
</tr>
<tr>
<td>QMLogto</td>
<td>bool = session-&gt;Logto(acc)</td>
</tr>
<tr>
<td>QMMarkMapping</td>
<td>session-&gt;MarkMapping(fno, state)</td>
</tr>
<tr>
<td>QMNextPartial</td>
<td>str = session-&gt;NextPartial(fno, listno)</td>
</tr>
<tr>
<td>QMConv</td>
<td>str = session-&gt;Oconv(data, code)</td>
</tr>
<tr>
<td>QMConvs</td>
<td>str = session-&gt;OConvs(data, code)</td>
</tr>
<tr>
<td>QMOpen</td>
<td>fno = session-&gt;Open(name)</td>
</tr>
<tr>
<td>QMOpenSeq</td>
<td>fno = session-&gt;OpenSeq(filename, id, modes)</td>
</tr>
</tbody>
</table>
QMPoolIdle
QMRead  str = session->Read(fno, id, err)
QMReadBlk str = session->ReadBlk(fno, bytes)
QMReadl  str = session->Readl(fno id, wait, err)
QMReadList str = session->ReadList(listno, err)
QMReadNext str = session->ReadNext(listno, err)
QMReadSeq str = session->ReadSeq(fno, err)
QMReadu str = session->Readu(fno, id, wait, err)
QMRecordLock session->RecordLock(fno, id, update, wait)
QMRecordLocked n = session->RecordLocked(fno, id)
QMRelease  session->Release(fno, id)
QMRespond str = session->Respond(response, err)
QMRevision  str = session->Revision
QMSend  session->Seek(fno, offset, relto)
QMSend  session->Select(fno, listno)
QMSendIndex str = session->SelectIndex(fno, indexname, indexvalue, listno)
QMSendLeft str = session->SelectLeft(fno, indexname, listno)
QMSendPartial str = session->SelectPartial(fno, listno)
QMSendRight str = session->SelectRight(fno, indexname, listno)
QMSend  session->Set(objno, name {, args})
QMSendLeft session->SetLeft(fno, indexname)
QMSendRight session->SetRight(fno, indexname)
QMSend  session->ServerStatus
QMTrapCallAbort session->TrapCallAbort(mode)
QMSend  session->Write(fno, id, data)
QMSendBlk session->WriteBlk(fno, data)
QMSendSeq session->WriteSeq(fno, data)
QMSendSeqKey session->WriteSeqKey(fno, data)
QMSend  session->Writeu(fno, id, data)
QMSendSeqKey session->WriteuSeqKey(fno, data)

For a more detailed description, see QMClient.
The !SCREEN() subroutine performs screen based input using a screen definition created using the SCRB command.

**Format**

```
CALL !SCREEN(scrn, data, step, status)
```

where

- `scrn` is a dynamic array holding the screen definition.
- `data` is the data record to be processed.
- `step` holds the step number at which screen execution is to commence. If this is a variable, it will be updated on exit to contain the step at which execution ended.
- `status` identifies the termination cause on returning to the calling program.

The !SCREEN() subroutine executes the screen starting at `step` except for the special `step` values described below.

<table>
<thead>
<tr>
<th><code>step</code></th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Clear the screen and paint text and data from all steps except those items with X in their mode value.</td>
</tr>
<tr>
<td>-1</td>
<td>Paint text and data from all steps except those items with X in their mode value without clearing the screen.</td>
</tr>
<tr>
<td>-2</td>
<td>Clear the screen without painting any data.</td>
</tr>
<tr>
<td>-3</td>
<td>Return a single keystroke value.</td>
</tr>
</tbody>
</table>

On returning to the calling program, `status` contains

-3 Illegal exit key code found in screen definition.
-2 Illegal validation code found in screen definition.
-1 Step number error.
0 Normal exit (X action code)
1 Exit key (escape) used with action code X
2 Backstep key used with no step history.

`n` Function key used. `n` is the key value as in KEYIN.H.

**Example**

```
READ SCRN FROM SCR.F, 'MY.SCREEN' ELSE ABORT 'Cannot read screen'
DATA = ''
CALL !SCREEN(SCRN, DATA, STEP, SCR.STATUS)
```

The above code fragment reads a screen definition and executes the screen driver to process the data record using this definition.
!SETPU()

The !SETPU() subroutine sets the characteristics of a print unit.

Format

    CALL !SETPU(key, unit, value, status)

where

- **key**: identifies the parameter to set. This is as for the SETPU statement.
- **unit**: evaluates to the print unit number.
- **value**: is the value to set for the given parameter.
- **status**: is the return status value. Zero if the action is successful, a non-zero error code if the action fails.

The !SETPU() subroutine sets the print unit characteristic specified by **key** to the given **value**. It is closely related to the SETPU statement.

Example

    CALL !SETPU(PU$LOCATION, 3, "LASER", STATUS)

The above statement sets the destination for print unit 3 to be the LASER printer.
!SETVAR()

The !SETVAR() subroutine sets the value of a user defined @-variable. It can also update some standard @variables.

Format

CALL !SETVAR(name, value)

where

name is the name of the @-variable to be set. The leading @ character may optionally be omitted. The name may be up to 32 characters and is case insensitive.

value is the value to be set. This may not include the mark characters.

The !SETVAR() subroutine sets the value of the named user defined @-variable. It can also set other standard @variables that are not read-only (e.g. @USER0) though these can be set using simple assignment statements.

The !SETVAR() function sets a status value that can be retrieved using the STATUS() function. This will be zero if the action is successful, or a non-zero error code if the name is invalid.

Example

CALL !SETVAR("@MYVAR", 71)

This example sets the user defined @MYVAR to 71.

See the !ATVAR() subroutine for a way to retrieve the value of a user defined @-variable.
The !SORT() subroutine sorts the elements of a dynamic array according to a specified sorting rule.

**Format**

```
CALL !SORT(in.list, out.list, sort.rule)
```

where

- `in.list` is the dynamic array containing the items to be sorted. Any mark character (or a mix of different mark characters) may be used to separate the items.
- `out.list` is the variable to receive the sorted dynamic array. The items will be separated by field marks.
- `sort.rule` defines the manner of sorting. This is a string containing characters from the following:
  - **Sort style:**
    - `F` Sort as right aligned values, comparing numeric values as floating point values
    - `L` Sort as left aligned values (default)
    - `R` Sort as right aligned values, comparing numeric values as integers
    - `S` Similar to a compound sort but elements that are numerically equivalent values such as "28" and "028" as treated as being different.
    - `X` Compound sort for mixed numeric and non-numeric data (see Sorting)
  - **Sort order:**
    - `A` Sort in ascending order (default)
    - `D` Sort in descending order
  - **Sort options:**
    - `C` Case insensitive sort
    - `N` Ignore null elements
    - `U` Return unique items. Multiple occurrences of an item are replaced by just one item.

Invalid or conflicting `sort.rule` elements are ignored.

The !SORT() subroutine sorts elements of `in.list` into the order defined by `sort.rule`, returning the sorted list in `out.list`. The value of `in.list` is not changed unless it refers to the same variable as `out.list`.

Right aligned sorts should normally be used when sorting numeric data.

**Example**

```
CUSTOMER.LIST = ""
SELECT INVOICES
LOOP
   READNEXT ID ELSE EXIT
   READ INVOICE.REC FROM INVOICES, ID THEN
      CUSTOMER.LIST<-1> = INVOICE.REC<CUSTOMER.NAME>
```

4.0-0
END
REPEAT
CALL !SORT(CUSTOMER.LIST, CUSTOMER.LIST, "AU")

The above program fragment reads all the records from the INVOICE file and builds a list of customer names. This is then sorted, removing duplicates.

This approach will be faster than using LOCATE and INS to build a sorted list unless there are a very large number of duplicates.

See also:
Sorting, SORT.COMpare()}
!USERNAME()

The !USERNAME() subroutine returns the user login name for a given QM user number.

Format

CALL !USERNAME(name, userno)

or

DEFFUN USERNAME(userno) CALLING "!USERNAME"

name = USERNAME(userno)

where

name is the returned user login name.

userno is the user number to locate.

The !USERNAME() subroutine returns name as the login name associated with a given user number. If there is no user logged in with that userno, a null string is returned.

Example

READU INV.REC FROM INV.F, INV.NO
LOCKED
    CALL !USERNAME(UNAME, STATUS())
    PRINTERR "Invoice is locked by user " : UNAME
END THEN
    GOSUB PROCESS.INVOICE
END

The above program fragment displays the login name of the user holding the lock if the READU is blocked by another user.
!USERNO()

The !USERNO() subroutine returns a list of QM user numbers for a given user name.

Format

CALL !USERNO(userno, username)

or

DEFFUN USERNO(username) CALLING "!USERNO"

userno = USERNO(username)

where

username is the user name to locate.

userno is a field mark delimited list of QM user numbers for the given user name.

The !USERNO() subroutine returns a field mark delimited list of the QM user numbers of processes running with the given user name. The user name is case insensitive. If there is no user logged in with that username, a null string is returned.

Example

INPUT USERNAME
CALL !USERNO(UNO, USERNAME)
IF UNO # "" THEN
   CRT "User numbers are: " : CHANGE(UNO, @FM, ", ", )
END ELSE
   CRT "There are no users logged in with this user name"
END

The above program fragment displays a comma separated list of QM users logged in under a given user name.
!VIEW()

The !VIEW() subroutine displays a record or other field mark delimited dynamic array as a scrollable text window.

Format

CALL !VIEW(data, user.keys, exit.key {, top {, lines}})

where

- **data** is the dynamic array to be displayed.
- **user.keys** is a sequence of characters that will cause exit from the subroutine in addition to use of Q as described below. This may be a null string.
- **exit.key** is returned as the character that caused the subroutine to exit.
- **top** is the topmost screen line to be used. If omitted, this defaults to zero.
- **lines** is the number of screen lines to be used. If omitted, this defaults to the entire portion of the screen below the top line.

The !VIEW() subroutine displays a field mark delimited dynamic array such as a program source text in a window that can be scrolled or panned. For compatibility with several other QM tools, the !VIEW() subroutine recognises multiple keys for most functions:

- Move to previous page: U, P, Cursor up, Page up, Ctrl-P, Ctrl-Z
- Move to next page: D, N, Cursor down, Page down, Ctrl-N
- Pan right: R, Cursor right, Ctrl-F
- Pan left: L, Cursor left, Ctrl-B
- Move to top of record: T, Home
- Move to bottom of record: B, End
- Exit: Q

Example

CALL !VIEW(REC, 'X', KEY)

The above statement displays the content of REC in scrollable form. The X key can be used as an alternative to Q to exit from the subroutine.
!VOCREC()

The !VOCREC() subroutine reads a VOC record, following links to remote records.

Format

CALL !VOCREC(rec, id)

or

DEFFUN VOCREC(id) CALLING "!VOCREC"

rec = VOCREC(id)

where

rec is the variable to receive the result.

id is the record id of the record to be read.

The !VOCREC() subroutine attempts to read record id from the VOC file. If not found, it tries again using an uppercase version of id.

If the record read from the VOC is an R-type item, the subroutine follows the link, again translating to uppercase if the record is not found exactly as specified in the R-type link.

If the original VOC record or the target of the R-type link is not found, the rec variable is set to a null string, otherwise rec contains the retrieved data.

The STATUS() function returns zero if a record was found or an error code if not.

Example

CALL !VOCREC(STYLE.REC, STYLE.NAME)
IF STATUS() THEN STOP 'Style record not found'

The above program fragment reads the VOC record identified by STYLE.NAME, following any remote link. If no record can be found, the program terminates with an error message.
6.9  QMBasic Debugger

The QM interactive debugger enables the developer to step through an application program in a convenient manner, stopping at desired points and examining data items.

Programs to be debugged must be compiled with the DEBUGGING option to the BASIC command or by including the $DEBUG compiler directive in the program source. At run time, the debugger will stop at selected places in the execution of these programs but will run normally through programs not compiled in this mode. Catalogued programs, subroutines and class modules may be debugged in exactly the same way as other programs.

The debugger is activated either by use of the DEBUG command in place of RUN or by a DEBUG statement encountered during execution of a program. The latter method enables debug mode to be entered part way through execution of the program or on meeting some specific condition. The debugger can also be entered from the break key action prompt if any program currently being executed has been compiled in debug mode.

During application development it is often worth compiling the entire application in debug mode. Execution of the program with the RUN command will not invoke the debugger unless a DEBUG statement is encountered. There is a small performance impact of running a debug mode program in this way but it is usually not significant.

Phantom processes and those acting as the server side of a QMClient connection can be debugged using the PDEBUG command.

The debugger will identify the program from which it was entered and locate the source program record. If this is not available, a warning is displayed and execution of the program continues in non-debug mode though other programs and subroutines called by it will still be subject to debugging if their source records are available.

When used with QMConsole on a Windows system, via the QMTerm terminal emulator or AccuTerm using a terminal type with the -at suffix, the debugger operates in full screen mode. When used on other terminals, the debugger output is mixed with the application output.

Full Screen Mode

The display is divided into two areas. The upper portion of the screen shows the source program with the line about to be executed highlighted. The lower portion of the screen is used to echo commands and to display their responses. The top line of the screen displays the program name and current line and element number. The display may be toggled between the debugger and the application by use of the F4 key. Full screen mode also supports a command stack similar to that found at the command prompt.

Where the terminfo definition for the terminal device in use includes mouse support, the full screen debugger additionally displays a menu bar at the top of the screen and a pop-up menu when the mouse is clicked on a source line. All mouse actions in the debugger use the left mouse button.

The top menu is divided into three sub-menus; Run, Debug and View. The Run menu allows the user to step through a program in various ways or to terminate debugging. The Debug menu allows a breakpoint to be set or cleared on the current line. The View menu has options to view the application screen, to show all variables, and to show the call stack.

Clicking on the line number shown to the left of a source line toggles a breakpoint on that line.
Clicking on the source text shows a context sensitive pop-up menu that allows the user to set or clear a breakpoint, show the content of the variable named at the mouse click position or set watch mode on a variable. Debugger actions related to variables in the pop-up menu cannot be used for private variables in local subroutines or functions and cannot include qualifiers for matrix indices or dynamic array references. These advanced options must be accessed via the debugger command prompt.

**Debugger Configuration**

On first entry to the debugger in a QM session, it checks for the presence of an X-type VOC record named $DEBUG.OPTIONS. If this is present, fields 2 onwards may contain configuration data for the debugger. The parameters are case insensitive and are as shown below.

- **NO.SOURCE.WARNING** Suppresses display of the warning message if the source code for a program module cannot be found.
- **WINDOW n** Sets the number of lines in the command area of the screen when using the debugger in full screen mode. The value must be in the range 5 to 15.

Unrecognised parameters are ignored.

**Using the Debugger**

The current position in a program is referenced by a line number and an element number. Most QBASIC source lines hold only a single element (element 0) but lines with multiple statements separated by semicolons or clauses of **IF/THEN/ELSE** constructs, etc, are considered to represent separate execution elements. The debugger can step line by line or element by element through a program.

The debugger cannot step through statements inserted into a program from an include record. In such cases, it will step over the included statements as though they were part of the immediately preceding statement.

Debugger commands fall into two groups; function keys and word based commands. In many cases both forms are available. Not all terminals support function keys.

Entering a blank line as a debugger command repeats the last command.

**Function Key Commands**

(Some function keys may not be available on all terminal emulations)

- F1 Display help screen
- F2 Abort program
- F3 Stop program
- F4 Display user screen (normal program output)
- F5 Free run
- F6 Step over subroutine call (internal or external)
- F7 Step program element
- F8 Step line
- Ctrl-F7 Run to parent program / subroutine (internal or external)
- Ctrl-F8 Exit program, returning to parent program or external subroutine

If an application dynamically rebinds the codes sent by keys used by the debugger, setting the DEBUG.REBIND.KEYS mode of the **OPTION** command will cause the debugger to reset these to the bindings specified in the terminfo entry for the current terminal type on each entry to the debug
screen. Note that the debugger cannot revert to the user bindings on exit as it has no way to determine what these were. This feature is available only with AccuTerm.

Word Based Commands

Where a short form is available, this is the upper case portion of the command as shown. Commands may be entered in any mix of upper and lower case.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABORT</td>
<td>Quit the program, generating an abort.</td>
</tr>
<tr>
<td>BRK ( n )</td>
<td>Set a breakpoint on line ( n ). Shows breakpoints if ( n ) omitted.</td>
</tr>
<tr>
<td>CLR</td>
<td>Clear all breakpoints.</td>
</tr>
<tr>
<td>CLR ( n )</td>
<td>Clear breakpoint on line ( n ).</td>
</tr>
<tr>
<td>DUMP ( var ) ( path )</td>
<td>Dumps a variable to an operating system level file.</td>
</tr>
<tr>
<td>EP</td>
<td>Exit program, returning to parent program or external subroutine.</td>
</tr>
<tr>
<td>Exit</td>
<td>Exit subroutine, returning to parent program, internal or external subroutine.</td>
</tr>
<tr>
<td>Goto ( n )</td>
<td>Continue execution at line ( n ).</td>
</tr>
<tr>
<td>HELP</td>
<td>Display help page.</td>
</tr>
<tr>
<td>INFO</td>
<td>Displays information about the program being debugged.</td>
</tr>
<tr>
<td>PDUMP</td>
<td>Generate a process dump file.</td>
</tr>
<tr>
<td>Quit</td>
<td>Quit the program, generating an abort.</td>
</tr>
<tr>
<td>Run</td>
<td>Free run.</td>
</tr>
<tr>
<td>Run ( n )</td>
<td>Run to line ( n ).</td>
</tr>
<tr>
<td>SET ( var=value )</td>
<td>Change content of a program variable (see below)</td>
</tr>
<tr>
<td>STACK</td>
<td>Display the call stack. The current program is shown first.</td>
</tr>
<tr>
<td>Step ( n )</td>
<td>Execute ( n ) lines.</td>
</tr>
<tr>
<td>Step ( .n )</td>
<td>Execute ( n ) elements.</td>
</tr>
<tr>
<td>StepOver</td>
<td>Step over subroutine call (internal or external)</td>
</tr>
<tr>
<td>STOP</td>
<td>Quit the program, generating a stop.</td>
</tr>
<tr>
<td>UnWatch</td>
<td>Cancels an active watch action.</td>
</tr>
<tr>
<td>View</td>
<td>Display user screen (normal program output)</td>
</tr>
<tr>
<td>Watch ( { var } )</td>
<td>Watches the named variable (see below). Shows watched variable if ( var ) omitted.</td>
</tr>
<tr>
<td>XEQ ( command )</td>
<td>Execute ( command ). Note that some commands may interfere with correct operation of the debugger.</td>
</tr>
<tr>
<td>^</td>
<td>Toggle expansion of non-printing characters.</td>
</tr>
</tbody>
</table>

The following commands apply only to full screen mode debugging:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SRC</td>
<td>Revert to default program source display</td>
</tr>
</tbody>
</table>
The following commands apply only to non-full screen mode debugging:

- **SRC**
  - **SRC name**
    - Show source of program name.
  - **SRC n**
    - Display around line n of currently displayed program.
  - **SRC +n**
    - Move display forward n lines in program.
  - **SRC -n**
    - Move display backward n lines in program.
  - **WINdow n**
    - Sets the number of lines (range 5 to 15) in the command area of the screen.

Displaying Program Variables

Entering a variable name preceded or followed by a slash (/) or a question mark (?) displays the type and content of the given variable (var/, /var, var?, ?var). This name may be local variable or a variable in a common block defined in the current program. If the common block has not been linked at the time the command is entered, the variable will appear as unassigned. For programs compiled with case insensitive names, the debugger is also case insensitive.

Private local variables in a subroutine declared using the **LOCAL** statement can be referenced using a name formed by concatenating the subroutine name and variable name with a colon between them. For programs compiled with case insensitive names, the local subroutine, a name of the form :var assumes the same local subroutine name as the previous local variable reference.

The debugger will not recognise names defined using **EQUATE** or **$DEFINE**.

The debugger recognises variable names **STATUS()**, **INMAT()**, **COL1()**, **COL2()** and **OS.ERROR()** to display the corresponding system variable. All @-variables may also be displayed except for @VOC (which is a file variable) and those representing constants such as @FM and @TRUE.

When debugging an object instance, the names of arguments to a public subroutine or function are not known to the debugger. Instead, a special positional reference of the form "*Arg1" is needed when displaying argument variables. The *Arg prefix is case insensitive and the argument number may include leading zeroes. For example, display of the second argument could use a debugger command such as

/ *arg2

All of the qualifiers to variable references such as array indices and dynamic array element positions may be used.

Display of long strings is broken into short sections to fit the available display space. Entering Q at the continuation prompt will terminate display.

When displaying strings with an active remove pointer, the position of this pointer is also shown.
If the variable is a matrix, the name may be followed by the index value(s) for the element to be displayed. Entry of the name without an index will display the dimensions of the matrix. Subsequent presses of the return key display successive elements of the matrix until either all elements have been displayed or another command is entered.

CLI.REC/
Array: DIM (20)
<RETURN>
CLI.REC(0) = Unassigned
<RETURN>
CLI.REC(1) = String (8 bytes): "J Watson"
<RETURN>
CLI.REC(2) = 13756
CLI.REC(8)/
Integer: 86

The variable name may be followed by a field, value or subvalue reference which will be used to restrict the display if the data is a string. Note that this qualifier has no effect on other data types.

REC/
String (11 bytes, R=4): "487m912m338"
REC<1>/
String (3 bytes): "487"
REC<2,1>/
String (3 bytes): "912"

Entering a slash alone will repeat the most recent display command.

Analysis of very large character strings is sometime easier from outside the debugger. The DUMP command can be used to dump the contents of a variable to an operating system level file that can then be processed with other tools.

The /* command, available in full screen mode only, displays all variables in the program, one per line. The page up, page down, cursor up and cursor down keys can be used to move through the data. When the current line marker in the leftmost column is positioned on an array or a data collection, pressing the return key shows the elements of the item. When positioned on a character string, pressing the return key shows the string data in hexadecimal and character form. In all cases, the Q key returns to the previous screen.

Changing Program Variables

The SET command can be used to alter the value of a variable.

SET var = value to set a numeric value
SET var = "string" to set a string value. Double quotes, single quotes or backslashes may be used to enclose the string.
SET var = true to set a Boolean True value
SET var = false to set a Boolean False value
SET var = null to set an SQL style null value
SET var(row,col) = value to set a matrix element

Watching Variables

The WATCH command causes the debugger to monitor the named variable.
WATCH var
Whenever a value is assigned to this variable (even if the value is the same as currently stored), the debugger will stop program execution and display the new value. Only one variable can be watched at a time.

There is an extended form of WATCH that will only report the new value and stop if the value meets a specific condition. This form of WATCH is

WATCH var op value
where
var is the variable to watch.

op is the operator used to test the condition. This may be any of the standard relational operators (=, #, <, >, <=, >=, EQ, NE, LT, GT, LE, GE) or the LIKE or MATCHES operator for pattern matching. The debugger supports UNLIKE in this context.

value is the value to be used for comparison. This may be a number or a string enclosed in single quotes, double quotes or backslashes.

The UNWATCH command cancels the watch action. The watch action is automatically cancelled when the watched variable ceases to exist. This might be return from the program in which the variable exists, re-dimensioning a common block, etc.

Referencing Equate Tokens

If a program is compiled in debug mode and with the $MODE DEBUG,EQUATES compiler mode enabled, equate tokens from EQUATE or DEFINE used in the program source code can also be used in the debugger for displaying, changing and watching variables.
6.10 Process Dump Files

QM includes the option to generate a process dump file containing a detailed report of the state of the process. There are five ways to generate a process dump:

A process dump will be created automatically if the DUMP.ON.ERROR mode of the OPTION command is active and the process aborts either due to an error detected by QM or from use of the ABORT statement in a QMBasic program.

Use of the DUMPING option in the CATCH clause of a TRY/CATCH block.

Selection of the P option following use of the break key.

Use of the QMBasic PDUMP statement.

Use of the PDUMP command. This can be used to generate a dump of a different process such as a phantom or a QMClient process.

By default, the process dump is directed to an operating system level file named qmdump.n in the QMSYS account directory where n is the QM user number. The directory to receive the dump file can be changed using the DUMPDIR configuration parameter. Because use of PDUMP potentially allows a user to view data from another user’s session, reducing system security, the PDUMP configuration parameter can be used to restrict the command to dumping other processes running under the same user name.

The file consists of a number of sections detailing the current state of the user process at the time of the error.

1. Environment data
   QM version number
   Licence number and site name
   User number
   Process id
   Parent user number (zero except in phantom processes)
   User name
   Values of STATUS() and OS.ERROR()
   Transaction id, command processor level and start time (only if in a transaction)

2. @-variables
   @-variables that are likely to be useful in determining the cause of an error.

3. Active numbered select lists

4. Locks
   The report shows all task locks, file locks and record locks owned by the process.

5. Program stack
   This contains an entry for each program, working backwards from the program in which the error occurred.
   For each program, the dump shows
   Program number (used in some cross-references within the dump)
   Program name, instruction address and line number. Line numbers cannot be shown if the program was compiled with no cross reference tables or these were removed when the program was catalogued.
Program status flags showing various special program states.
GOSUB return stack, if not empty.
Variables. Local variables are sorted alphabetically. Elements of a common block are shown in memory order and are only dumped on the first program that references the block. Non-printing characters in strings are replaced by \mn where \mn is the hexadecimal character value. Backslash characters are shown as \\ Character string data is not line wrapped to simplify exploration of the data using tools such as the SED editor.
6.11 Error Numbers

Error numbers are defined in the ERR.H record of the SYSCOM file.

1  ER$ARGS    Command arguments invalid or incomplete
2  ER$NCOMO   Como file not active
3  ER$ICOMP   I-type compilation error
4  ER$ACC.EXISTS  Account name already in register
5  ER$NO.DIR    Directory not found
6  ER$NOT.CREATED  Unable to create directory
7  ER$STOPPED  Processing terminated by user in response to a "continue" prompt
8  ER$INVA.PATH  Invalid pathname
9  ER$NOT.CAT   Item not in catalogue
10  ER$PROCESS  Unable to start new process
11  ER$USER.EXISTS  User name already in register
12  ER$UNSupported    This operation is not supported on this platform
13  ER$TERMINFNO  No terminfo definition for this function
14  ER$NO.ACC    Account name not in register
15  ER$TERMINATED  Query command terminated by user
16  ER$NO.LIST   No select list when one is required
1000  ER$PARAMS  Invalid parameters
1001  ER$MEM     Cannot allocate memory
1002  ER$LENGTH  Invalid length
1003  ER$BAD.NAME  Bad name
1004  ER$NOT.FOUND  Item not found
1005  ER$IN.USE  Item is in use
1006  ER$BAD.KEY  Bad action key
1007  ER$PRT.UNIT  Bad print unit
1008  ER$FAILED  Action failed
1009  ER$MODE    Bad mode setting
1010  ER$TXN    Operation not allowed in a transaction
1011  ER$TIMEOUT  Timeout
1012  ER$LIMIT  User limit reached
1013  ER$EXPIRED  Package licence has expired
1014  ER$NO.CONFIG  Cannot find configuration file
1015  ER$RDONLY.VAR  Variable is read-only
1016  ER$NOT.PHANTOM  Not a phantom process
1017  ER$CONNECTED  Device already connected
1018  ER$INVA.ITYPE  Invalid I-type
1019 ER$RANGE Value is out of range
1020 ER$BARRED Action is barred
1021 ER$CONSOLE Only valid in a Windows QMConsole session
1022 ER$SUSPENDED QM is suspended
1023 ER$PARSE.ERROR Parsing error
1024 ER$BUILD.ERROR Data build error
2000 ER$INVA.OBJ Invalid object code
2001 ER$CFNF Catalogued function not found
2100 ER$TI.NAME Invalid terminal type name
2101 ER$TI.NOENT No terminfo entry for given name
2102 ER$TI.MAGIC Termino magic number check failed
2103 ER$TI.INVHDR Invalid terminfo header data
2104 ER$TI.STRSZ Invalid terminfo string length
2105 ER$TI.STRMEM Error allocating terminfo string memory
2106 ER$TI.NAMEMEM Error allocating terminfo name memory
2107 ER$TI.BOOLMEM Error allocating terminfo Boolean memory
2108 ER$TI.BOOLRD Error reading terminfo Booleans
2109 ER$TI.NUMMEM Error allocating terminfo numbers memory
2110 ER$TI.NUMRD Error reading terminfo numbers
2111 ER$TI.STROMEM Error allocating terminfo string offsets memory
2112 ER$TI.STRORD Error reading terminfo string offsets
2113 ER$TI.STRTBL Error reading terminfo string table
2114 ER$TI.NAMERD Error reading terminal type name
3000 ER$IID Illegal record id
3001 ER$SFNF Subfile not found
3002 ER$NAM Bad file name
3003 ER$FNF File not found
3004 ER$NDIR Not a directory file
3005 ER$NDYN Not a dynamic file
3006 ER$RNF Record not found
3007 ER$NVR No VOC record
3008 ER$NPN No pathname in VOC record
3009 ER$VNF VOC file record not F type
3010 ER$IOE I/O error
3011 ER$LCK Lock is held by another process
3012 ER$NLK Lock is not held by this process
3013 ER$SEQ Not a sequential file
3014 ER$NEOF Not at end of file
<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3015</td>
<td>ER$SQRD  Sequential file record read before creation</td>
</tr>
<tr>
<td>3016</td>
<td>ER$SQNC  Sequential record not created due to error</td>
</tr>
<tr>
<td>3017</td>
<td>ER$SQEX  Sequential record already exists (CREATE)</td>
</tr>
<tr>
<td>3018</td>
<td>ER$RDONLY  Update to read only file</td>
</tr>
<tr>
<td>3019</td>
<td>ER$AKNF  AK index not found</td>
</tr>
<tr>
<td>3020</td>
<td>ER$INVAPATH  Invalid pathname</td>
</tr>
<tr>
<td>3021</td>
<td>ER$EXCLUSIVE  Cannot gain exclusive access to file</td>
</tr>
<tr>
<td>3022</td>
<td>ER$STRIGGER  Trigger function error</td>
</tr>
<tr>
<td>3023</td>
<td>ER$NOLOCK  Attempt to write/delete record with no lock</td>
</tr>
<tr>
<td>3024</td>
<td>ER$REMOTE  Open of remote file not allowed</td>
</tr>
<tr>
<td>3025</td>
<td>ER$NOTNOW  Action cannot be performed now</td>
</tr>
<tr>
<td>3026</td>
<td>ER$SPORT  File is a port</td>
</tr>
<tr>
<td>3027</td>
<td>ER$NPORT  File is not a port</td>
</tr>
<tr>
<td>3028</td>
<td>ER$SQSEEK  Seek to invalid offset in sequential file</td>
</tr>
<tr>
<td>3029</td>
<td>ER$SQREL  Invalid SEEK relto in sequential file</td>
</tr>
<tr>
<td>3030</td>
<td>ER$EOF  End of file</td>
</tr>
<tr>
<td>3031</td>
<td>ER$CNF  Multifile component not found</td>
</tr>
<tr>
<td>3032</td>
<td>ER$MFC  Multifile reference with no component name</td>
</tr>
<tr>
<td>3033</td>
<td>ER$PNF  Port not found</td>
</tr>
<tr>
<td>3034</td>
<td>ER$BAD.DICT  Bad dictionary entry</td>
</tr>
<tr>
<td>3035</td>
<td>ER$PERM  Permissions error</td>
</tr>
<tr>
<td>3036</td>
<td>ER$SEEK.ERROR  Seek error</td>
</tr>
<tr>
<td>3037</td>
<td>ER$WRITE.ERROR  Write error</td>
</tr>
<tr>
<td>3038</td>
<td>ER$VFS.NAME  Bad class name in VFS entry</td>
</tr>
<tr>
<td>3039</td>
<td>ER$VFS.CLASS  VFS class routine not found</td>
</tr>
<tr>
<td>3040</td>
<td>ER$VFS.NGLBL  VFS class routine is not globally catalogued</td>
</tr>
<tr>
<td>3041</td>
<td>ER$ENCRYPTED  Access denied to encrypted file</td>
</tr>
<tr>
<td>3042</td>
<td>ER$NOT.A.FILE  Pathname or variable is not a file</td>
</tr>
<tr>
<td>3043</td>
<td>ER$COMMIT.FAIL  Transaction commit failed</td>
</tr>
<tr>
<td>3044</td>
<td>ER$ROLLBACK  Transaction rolled back</td>
</tr>
<tr>
<td>3045</td>
<td>ER$SIZE  File or record is too large for this operation</td>
</tr>
<tr>
<td>3046</td>
<td>ER$FILETYPE  Invalid file type for this operation</td>
</tr>
<tr>
<td>3047</td>
<td>ER$PART  Invalid part number</td>
</tr>
<tr>
<td>3048</td>
<td>ER$AK.ERR  Execution error in AK expression</td>
</tr>
<tr>
<td>3049</td>
<td>ER$DNF  Device not found</td>
</tr>
<tr>
<td>3050</td>
<td>ER$IDX.VERIFY  Index verification error</td>
</tr>
<tr>
<td>3051</td>
<td>ER$QPTR.PATH  Q-pointer with account name barred</td>
</tr>
<tr>
<td>3052</td>
<td>ER$ACC.BARRED  Access to this account is barred</td>
</tr>
</tbody>
</table>
3053  ER$TRG.NOCAT  Trigger function not catalogued
3054  ER$ECS  File requires ECS
3055  ER$ECS.MAP  ECS map not available
3056  ER$ECS.DATA  Data requires ECS
3057  ER$NOT.ECS  QMNet server not ECS
3058  ER$VFS.NO.OPEN  No VSOPEN function in VFS handler
3059  ER$TEMP.FILE  Cannot open temporary file
3060  ER$MAXRLOCK  MAXRLOCK record lock limit reached
3061  ER$LOCKFULL  Lock table is full
3062  ER$CONSTRAINTS  Data integrity constraints validation error
3063  ER$NHASHED  Not a hashed file
4000  ER$SRVRMEM  Insufficient memory for packet buffer
4001  ER$CONTEXT  Context error
4002  ER$FILE.BAR  QMClient file access is disabled
4003  ER$EXEC.BAR  QMClient command execution is disabled
4004  ER$SUBR.BAR  QMClient subroutine restrictions active
4200  ER$FUNCNAME  Function name not recognised
4204  ER$VFS.KEY  Unrecognised VFS key
4205  ER$VFS.PIPE  Failed to create VFS pipe
4206  ER$VFS.CHILD  Failed to start VFS child
4207  ER$VFS.CONNECT  Failed to connect to VFS child
5000  ER$NO.DLL  DLL not found
5001  ER$NO.API  API not found
5002  ER$NO.TEMP  Cannot open temporary file
5003  ER$UID.NAME  Cannot convert user name to UID
5004  ER$GID.NAME  Cannot convert group name to GID
6031  ER$NO.EXIST  Item does not exist
6032  ER$EXISTS  Item already exists
6033  ER$NO.SPACE  No space for entry
6034  ER$INVALID  Validation error
7000  ER$NETWORK  Networked file not allowed for this operation
7001  ER$SERVER  Unknown server name
7002  ER$WSA.ERR  Failed to start Window socket library
7003  ER$HOSTNAME  Invalid host name
7004  ER$NOSOCKET  Cannot open socket
7005  ER$CONNECT  Cannot connect socket
7006  ER$RECV.ERR  Error receiving socket data
7007  ER$RESOLVE  Cannot resolve server name
7008 ER$LOGIN  Login rejected
7009 ER$XREMOTE  Remote server disallowed access
7010 ER$ACCOUNT  Cannot connect to account
7011 ER$HOST.TABLE  Host table is full
7012 ER$BIND  Error binding socket
7013 ER$SKT.CLOSED  Socket has been closed
7014 ER$PROTOCOL  Unrecognised protocol
7015 ER$SKT.TYPE  Unrecognised socket type
7016 ER$NET.READ  QMClient data read error
7017 ER$NET.WRITE  QMClient data write error
7100 ER$RPL.PROT  Replication protocol error
7101 ER$SEND.ERR  Error sending socket data
8001 DHESFILE.NOT.OPEN  DH.FILE pointer is NULL
8002 DHESNOT.A.FILE  DH.FILE does not point to a file descriptor
8003 DHE$SID.LEN.ERR  Invalid record id length
8004 DHE$SEEK.ERROR  Error seeking in DH file
8005 DHE$READ.ERROR  Error reading DH file
8006 DHE$WRITE.ERROR  Error writing DH file
8007 DHE$NAME.TOO.LONG  File name is too long
8008 DHE$SIZE  File exceeds maximum allowable size
8009 DHE$STAT.ERR  Error from stat()
8100 DHESOPEN.NO.MEMORY  No memory for DH.FILE structure
8101 DHESFILE.NOT.FOUND  Cannot open primary subfile
8102 DHESOPEN1.ERR  Cannot open overflow subfile
8103 DHE$PSFH.FAULT  Primary subfile header format error
8104 DHE$OSFH.FAULT  Overflow subfile header format error
8105 DHE$BUFFERS  Unable to allocate file buffers
8106 DHESINVA.FILE.NAME  Invalid file name
8107 DHE$TOO.MANY.FILES  The limit on the number of open files has been reached. See the NUMFILES configuration parameter.
8108 DHE$NO.MEM  No memory to allocate group buffer
8109 DHESAK.NOT.FOUN D  Cannot open AK subfile
8110 DHESAK.HDR.READ. ERROR  Error reading AK header
8111 DHESAK.HDR.FAULT  AK subfile header format error
<table>
<thead>
<tr>
<th>Error Code</th>
<th>Error Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>8112</td>
<td>DHESAK.ITYPE.ERR  Format error in AK I-type code</td>
</tr>
<tr>
<td>8113</td>
<td>DHESAK.NODE.ERR  Unrecognised node type</td>
</tr>
<tr>
<td>8114</td>
<td>DHESNO.SUCH.AK  Reference to non-existent AK</td>
</tr>
<tr>
<td>8115</td>
<td>DHESAK.DELETE.ERR Error deleting AK subfile</td>
</tr>
<tr>
<td>8116</td>
<td>DHESEXCLUSIVE  File is open for exclusive access</td>
</tr>
<tr>
<td>8117</td>
<td>DHESTRUSTED     Requires trusted program to open</td>
</tr>
<tr>
<td>8118</td>
<td>DHESVERSION    Incompatible file version</td>
</tr>
<tr>
<td>8119</td>
<td>DHESID.LEN      File may contain id longer than MAXIDLEN</td>
</tr>
<tr>
<td>8120</td>
<td>DHESAK.CROSS.CHECK Relocated index pathname cross-check failure</td>
</tr>
<tr>
<td>8121</td>
<td>DHESHASH.TYPE   Unsupported hash type</td>
</tr>
<tr>
<td>8122</td>
<td>DHESAK.PERM     Permissions error on AK subfile</td>
</tr>
<tr>
<td>8123</td>
<td>DHESMAX.OPENS   File open maximum number of times in one process</td>
</tr>
<tr>
<td>8201</td>
<td>DHESILLEGAL.GROUP P.SIZE Group size out of range</td>
</tr>
<tr>
<td>8202</td>
<td>DHESILLEGAL.MIN. MODULUS Minimum modulus &lt; 1</td>
</tr>
<tr>
<td>8203</td>
<td>DHESILLEGAL.BIG.REC.SIZE Big record size invalid</td>
</tr>
<tr>
<td>8204</td>
<td>DHESILLEGAL.MERGE ELSEAD Merge load invalid</td>
</tr>
<tr>
<td>8205</td>
<td>DHESILLEGAL.SPLIT LOAD Split load invalid</td>
</tr>
<tr>
<td>8206</td>
<td>DHESFILE.EXISTS File exists on create</td>
</tr>
<tr>
<td>8207</td>
<td>DHESCREATE.DIR.ERR Cannot create directory</td>
</tr>
<tr>
<td>8208</td>
<td>DHESCREATE0.ERR Cannot create primary subfile</td>
</tr>
<tr>
<td>8209</td>
<td>DHESCREATE1.ERR Cannot create overflow subfile</td>
</tr>
<tr>
<td>8210</td>
<td>DHESPFSH.WRITE.ERR Failure writing primary subfile header ERROR</td>
</tr>
<tr>
<td>8211</td>
<td>DHESINIT.DATA.ERR Failure initialising data bucket</td>
</tr>
<tr>
<td>8212</td>
<td>DHESILLEGAL.HASH Invalid hashing algorithm</td>
</tr>
<tr>
<td>8213</td>
<td>DHESOSFH.WRITE.ERR Failure writing overflow subfile header ERROR</td>
</tr>
<tr>
<td>8301</td>
<td>DHESRECORD.NOT.FOUND Record not in file</td>
</tr>
<tr>
<td>8302</td>
<td>DHESBIG.CHAIN.END Found end of big record chain early</td>
</tr>
<tr>
<td>8303</td>
<td>DHESNOT.BIG.REC  Big record pointer does not point to big record block</td>
</tr>
<tr>
<td>8401</td>
<td>DHESNO.SELECT   No select is active</td>
</tr>
<tr>
<td>Code</td>
<td>Description</td>
</tr>
<tr>
<td>-------</td>
<td>----------------------------------</td>
</tr>
<tr>
<td>8402</td>
<td>DHESOPEN2.ERR</td>
</tr>
<tr>
<td>8403</td>
<td>DHESGLS.WRITE.ERR</td>
</tr>
<tr>
<td>8404</td>
<td>DHESGLS.TRUNCATE.ERR</td>
</tr>
<tr>
<td>8501</td>
<td>DHESAK.NAME.LEN</td>
</tr>
<tr>
<td>8502</td>
<td>DHESAK.EXISTS</td>
</tr>
<tr>
<td>8503</td>
<td>DHESAK.TOO.MANY</td>
</tr>
<tr>
<td>8504</td>
<td>DHESAK.CREATE.ERR</td>
</tr>
<tr>
<td>8505</td>
<td>DHESAK.HDR.WRITE.ERROR</td>
</tr>
<tr>
<td>8506</td>
<td>DHESAK.WRITE.ERR</td>
</tr>
<tr>
<td>8601</td>
<td>DHESPSF.CHSIZE.ERR</td>
</tr>
<tr>
<td>8701</td>
<td>DHESALL.LOCKED</td>
</tr>
<tr>
<td>8702</td>
<td>DHESSPLIT.HASH.ERROR</td>
</tr>
<tr>
<td>8703</td>
<td>DHESWRONG.BIG.REC</td>
</tr>
<tr>
<td>8704</td>
<td>DHESFREE.COUNT.ZERO</td>
</tr>
<tr>
<td>8705</td>
<td>DHESFDS.OPEN.ERR</td>
</tr>
<tr>
<td>8706</td>
<td>DHESPOINTER.ERROR</td>
</tr>
<tr>
<td>8707</td>
<td>DHESNO.INDICES</td>
</tr>
<tr>
<td>8708</td>
<td>DHESRECLLEN.ERROR</td>
</tr>
<tr>
<td>8900</td>
<td>DHESECB.TYPE</td>
</tr>
<tr>
<td>8901</td>
<td>DHESVAULT.CLOSED</td>
</tr>
<tr>
<td>9000</td>
<td>DHERPL.LIMIT</td>
</tr>
<tr>
<td>9001</td>
<td>DHERPL.MKDIR</td>
</tr>
<tr>
<td>9002</td>
<td>DHERRCB.TYPE</td>
</tr>
</tbody>
</table>

Cannot open select list
Error from write()
Error from chsize()
Index name too long
AK already exists
Too many AKs to create a new one
Unable to create AK subfile
Error writing AK subfile header
Error writing AK node
Error compacting primary subfile
All buffers are locked
Record does not hash to either group in split
Big record chain error
Overflow free count zero
Cannot reopen subfile
Internal file pointer fault
File has no AKs
Invalid record length
ECB has incorrect type
Key vault not open
Too many replication targets
Error creating replication buffer area
RCB type error
6.12 Package Licensing

Packages provide a way in which a software developer can ensure that their application is used only on systems for which it is licensed and with a defined user limit. QM supports two distinct types of package: **registered packages** that can be licensed by QM resellers via the openqm.com web site and **private packages** that manage the licensing internally.

**Registered Packages**

Registered packages are managed by the QM licensing system. A reseller can register a package name with Zumasys and then use the web site to generate licences to use that package. Registered packages are identified by names of up to 15 characters that commence with an uppercase letter and may have an associated user limit and expiry date. They are tied to a specific QM licence by QM's own licence authentication system. QM resellers who own registered packages will have access to the relevant licensing screens on the openqm.com web site.

A package licence is enabled by use of the `PACKAGE.LICENCE` command in the QMSYS account. The package name and authorization code are entered by the user and then validated by QM before being applied to the internal registered package list. This command can also be used to remove a package licence or to amend the authorisation code.

**Private Packages**

Private packages are applications that are released directly by their developer. They are identified by names of up to 15 characters that commence with a lowercase letter. All aspects of licence authentication is the responsibility of the application. The only involvement of QM in this process is tracking the number of users currently active in the package.

**Entering a Package**

An application uses the QMBasic `ENTER.PACKAGE()` function to indicate that it is entering the package.

```
ENTER.PACKAGE(name)
```

where

```
name
```

is the name of the package.

For a registered package this function returns True if it is successful, False if it fails. In the event of failure, the `STATUS()` function can be used to determine the cause.

For a private package, the function returns the number of users in the package, including the one that executed this function. The application can determine from its own licensing rules whether the user limit has been reached. If so, it should exit the package as described below.

Where a package is to be tied to a specific QM licence, the licence number can be obtained using `SYSTEM(31)`.

**Exiting from a Package**

An application uses the QMBasic `EXIT.PACKAGE()` function to indicate that it is leaving the package.
EXIT.PACKAGE(name)

where

name is the name of the package.

This function returns True if it is successful, False if it fails. In the event of failure, the STATUS() function can be used to determine the cause. Note that a registered package may be configured not to allow exit.

QM will automatically update the internal data managing package access if a process terminates without exiting from a package that it had previously entered.

Identifying Active Packages

An application can obtain a field mark delimited list of currently active packages by use of the SYSTEM(1054) function. Although this should not normally be required as the process should be able to track package entry and exit, it might be useful, for example, to test if a malicious user has attempted to defeat the user limit mechanism by executing an EXIT.PACKAGE() function. A test such as

IF LISTINDEX(SYSTEM(1054), @FM, package.name) THEN

could be used periodically to confirm that the process is still active in that package.

Application Installation

An application installer can add a licence for a registered package by using the !PACKAGE() subroutine.

See also:
ENTER.PACKAGE(), EXIT.PACKAGE(), PACKAGE.LICENCE, !PACKAGE()
6.13 Building a Self-Installing Application

If you are developing an application to be provided as a complete user-installable package, you probably want to automate as much as possible of this. Ideally, you would like the user to need only to execute a single program to install both QM and the application software. This section describes two ways to do this.

Using a Commercial Installer Package

On Windows systems, we recommend use of the Astrum InstallWizard from Thraex Software (as used by QM itself) but the following process should map onto other self-installer packages.

Whatever installer package you use, it needs to install both QM and the application. The complication is that this process needs to run QM to create the account that will hold the application. The steps to achieve this are:

1. Unpack all the application files to wherever they need to go. The directory that will become the application account can be created during this process but the only QM specific subdirectory that should be created is the private catalogue (cat). You can place your own application install program into the cat subdirectory for later use. If you need to pre-load dictionary items or data file records, these should be unpacked into a temporary location.

2. The self-extracting file must also include the relevant version of QM as its own self-extracting file. This should be unpacked into a temporary directory.

3. Once everything has been unpacked, the process now needs to install QM by executing the QM self-extracting program. On Windows systems, use of the /silent command line option will suppress most user interaction.

4. Now that QM is installed (or upgraded), you need to use it to create the application account. The process should check whether the account already exists by looking for the VOC file and, if not, execute QM with a single command line option of "CREATE.ACCOUNT account.name account.path NO.QUERY". The quotes are required in this command and the working directory should be the QMSYS account. The CREATE.ACCOUNT command will not fail if the cat subdirectory already exists.

5. Next, you need to execute your own application installer program that should have been included in the contents of the unpacked private catalogue directory. This is done by executing QM with a command line option that is the catalogued item name and with a working directory of the newly created account.

6. Finally, you need to remove any temporary files.

So, what does the catalogued install program need to do?

- We recommend that it should start by executing a COMO command to create a log file of its progress.
- Create any application files that do not already exist.
- Copy dictionary items from a temporary set of dictionaries unpacked from the install file. By doing this rather than simply overwriting the dictionaries, anything that had been added will not be lost when updating an existing installation.
• Build any indices that are required.
• Create any application specific VOC entries such as paragraphs and sentences.

Using an Installation Script

The QM installer includes a mechanism that can be used to link it to installation of your own software. To use this, you distribute your software on a CD or as a downloadable zip file. This should contain all of your application software and the QM installation package together with a text file named qmcfg.dat and the qmsetup program described below. This program can be found in the bin subdirectory of the QMSYS account of your QM development installation and may be freely distributed.

To run the installation, the user executes the qmsetup program. This reads the qmcfg.dat file that contains data to control the installation process. There are two sections to this file which may appear in either order. Blank lines and lines commencing with a # character are ignored.

The [qm] section contains QM configuration parameters that will be updated in the qmconfig file by the installer. All QM configuration parameters except for the licence details may be updated in this way. Registered QM resellers can contact Zumasys for details of how to generate and apply QM licences automatically as part of the installation process.

```
[qm]
NUMFILES=200
NUMLOCKS=400
ERRLOG=10
```

The [application] section contains instructions to the qmsetup program. These are executed in the order in which they appear.

```
[application]
EULA=licence.txt
RUN=qm_4-0-0.exe /silent
COPY=appdir;c:\program files\myapp
CD=c:\program files\myapp
RUN=qm setup
```

The EULA command identifies a simple text file that will be displayed to the user as the end user licence agreement. This must be approved by the user before the installer moves on.

The RUN command uses the operating system shell to execute the given command. In the example above, the first RUN command is used to execute the QM installer.

The COPY command copies a file or directory. The source and destination pathnames should be separated by a semicolon. Any spaces in the names are taken as being part of the name.

The CD command changes the current working directory. If no pathname is given, the directory holding the install script is assumed.

Any command may contain <mypath> which will be replaced with the pathname of the directory from which the installation was run. Thus

```
CD=
```

is equivalent to

```
CD=<mypath>
```
6.14 Building a CGI Web Server Application

There are advanced third party web based packages available for QM such as Pavuk or DesignBais but for many applications use of built-in QM features may be sufficient. See Simple Web Services for more information.

Alternatively, a simple CGI program gives an easy way to achieve web connectivity with no additional software as described below. This uses the QMClient API which can also be called from PHP, ASP, etc for alternative development styles.

See "Using the C API" in the QMClient API section for details of libraries that must be included when this application is compiled and linked. The executable program file should be placed in the cgi-bin subdirectory of the relevant web account.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <\qmsys\syscom\qmclilib.h>       [ Windows ]
#include </usr/qmsys/SYSCOM/qmclilib.h>   [ Unix/Linux based systems ]

// Set the following six lines as appropriate for your system
#define SERVER_ADDRESS "localhost" /* Server network address or name... */
#define SERVER_PORT 4243             /* ...and port on which to connect */
#define SERVER_USER "xxx"            /* Server user name... */
#define SERVER_PASSWORD "xxxxx"      /* ...and password */
#define SERVER_ACCOUNT "xxxxx"       /* Web server QM account name */
#define SERVER_PROGRAM "xxxxx"       /* Catalogued program to run on server */

char NullString[] = "";
char InputData[32767] = "";        /* Incoming data stream */
char Response[99999] = "";        /* Response to send */
char * ClientIP;                  /* Client IP address */
char * ClientUser;                /* User name if web authentication used */

/*
 ==---------------------------------------------------------------
 == */

int main()
{
  char * RequestMethod;
  char * p;

  RequestMethod = getenv("REQUEST_METHOD");
  if (RequestMethod == NULL)
  {
    printf("Program must be executed by a Web browser\n");
    return 1;
  }

```
ClientIP = ((p = getenv("REMOTE_ADDR")) != NULL)?p:NullString;
ClientUser = ((p = getenv("REMOTE_USER")) != NULL)?p:NullString;

if (!strcmp(RequestMethod,"GET"))
{
    if ((p = getenv("QUERY_STRING")) != NULL) strcpy(InputData, p);
}
else if (!strcmp(RequestMethod,"POST"))
{
    if ((p = getenv("CONTENT_LENGTH")) != NULL)
    {
        fread(InputData, atoi(p), 1, stdin);
    }
}

/* Check for locally processed screens */

ParseInputData();

if (!QMConnect(SERVER_ADDRESS, SERVER_PORT, SERVER_USER,
SERVER_PASSWORD, SERVER_ACCOUNT))
{
    strcpy(Response, "Failed to connect. The server may be offline.");
}
else
{
    QMCall(SERVER_PROGRAM, 4, InputData, ClientIP, ClientUser,
Response);
    QMDisconnect();
}

printf("Content-type: text/html\n\n");
printf("<meta http-equiv="Pragma" content="no-cache">\n");
printf("%s\n", Response);

return 0;
}

Web requests received by this program will be passed to the catalogued QMBasic subroutine identified by SERVER_PROGRAM. The declaration of this is

    SUBROUTINE SERVER(INPUT.DATA, CLIENT.IP, CLIENT.USER, RESPONSE)

When used with HTML forms with the method attribute set to "post" or "get", any data sent with the form will be passed to the QM server program via the first argument (INPUT.DATA). Typically, the form would include an item in this data that can be used to determine the screen being processed.

The CLIENT.IP is the network address of the client user and can be used for simple security checking.

If the user has been authenticated using the conventional web user authentication process, the user name appears in CLIENT.USER. If authentication has not been performed, this will be a null string.

The server subroutine must return valid HTML data to be returned to the web client via the RESPONSE argument. For applications that return very large HTML strings it may be better for the
QMBasic component to write the data to a temporary file and pass this name back to the C program. This avoids the need for the Response variable in the above example to be sized to fit the largest string that could ever be returned.
Part 7

QMClient API
7 QMClient API

The QMClient API allows an external application to connect to a QM server. There are API equivalents to the major file handling statements of QMBasic as well as a range of string functions for dynamic array data manipulations, functions to execute commands and call catalogued subroutines on the server, etc. Where both ends of the connection support it, the network traffic is encrypted for security.

QMClient allows multiple simultaneous connections from a single client process. This allows development of applications that draw on data from multiple sources or transfer data between accounts or servers.

QMClient is usable from many languages and environments. The table below shows the library name, function definition record in the SYSCOM file. Except as noted below, the library files appropriate to the version of QM in use are normally automatically installed in the bin subdirectory of the QMSYS account. Because the client software might be for a different system type, these files are also on the Downloads page of the openqm.com web site. Some QMClient variants are separately downloadable items on the Resources and Solutions page of the openqm.com web site as noted below.

<table>
<thead>
<tr>
<th>Language</th>
<th>Library</th>
<th>Definitions</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Visual Basic .Net</td>
<td>qmclivb.dll + qmclilib.dll</td>
<td></td>
<td>Add reference to qmclivb.dll and use &quot;Imports qm.qmclient&quot; to access library functions</td>
</tr>
<tr>
<td>Visual Basic 2005</td>
<td>qmclient.dll + qmclient.lib</td>
<td>qmclient2005.vb</td>
<td></td>
</tr>
<tr>
<td>Power Basic</td>
<td>qmclient.dll + qmclient.lib</td>
<td>qmclient.inc</td>
<td></td>
</tr>
<tr>
<td>PureBasic (Windows)</td>
<td>qmclilib.dll + qmclilib.lib</td>
<td>qmclient.pb</td>
<td></td>
</tr>
<tr>
<td>PureBasic (Linux)</td>
<td>qmclilib.so + qmclilib.lib</td>
<td>qmclient.pb</td>
<td></td>
</tr>
<tr>
<td>C, C++ (Windows, Microsoft)</td>
<td>qmclilib.dll + qmclilib.lib</td>
<td>qmclilib.h</td>
<td></td>
</tr>
<tr>
<td>C, C++ (Linux)</td>
<td>libqmcclient.a (static)</td>
<td>qmclilib.h</td>
<td></td>
</tr>
<tr>
<td>C, C++ (Mac)</td>
<td>libqmcclient.a (static)</td>
<td>qmclilib.h</td>
<td></td>
</tr>
<tr>
<td>C#</td>
<td>qmclilib.dll</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Java (Windows, JNI)</td>
<td>JNI_QMClient.dll</td>
<td></td>
<td>These two versions are the original implementation of the QMClient Java API using platform specific Java Native</td>
</tr>
</tbody>
</table>
javaxnijniinterface libraries, downloadable from the Resources and Solutions area. They are no longer updated or supported.

This version is a pure Java implementation of the QMClient Java API and is portable across all platforms.

This item is downloadable from the Resources and Solutions area.

In global catalogue

It may be necessary to add the location of the qmclient.py module to the PYTHONPATH environment variable.

The secret of writing efficient client server applications is to perform the bulk data processing on the server and only handle user interface issues on the client. This minimises the data transferred between the systems and hence optimises performance.

QMClient has some security issues that need special consideration.

A QMClient session is an interactive use of the system and hence is counted in determining the licensed user limit on the server.

The QMClient libraries listed above may be freely distributed with application software. The QM licence redistribution clauses apply to the server software, not the QMClient client-side components. Although every attempt is made to ensure compatibility with older versions of the client libraries, there may be times when changes require the distributed client components to be updated.

There are some aspects of the QMClient API that differ between the various client languages. The following descriptions cover the most commonly used languages. For the less widely used languages such as PowerBasic, see the comments at the head of the relevant item in the SYSCOM file.

Using the C API

Use of the C programmers’ API is different depending on the compiler in use. On Unix/Linux based systems, programs need to include the libqmclient.a library when linking the application. The Linux version of QM also includes a shared object version of QMClient API named qmclilib.so. On Windows, the qmclilib.dll dynamic link library is used and the qmclilib.lib import library should be included when linking the application. All of these components can be found in the bin subdirectory of the QMSYS account. The function definitions can be found in the qmclilib.h include record in the SYSCOM file.

Functions that return Boolean values, return 0 for False, 1 for True in the C API library.

With the exception of QMError(), all API calls that return strings dynamically allocate memory to hold the returned data. It is the calling program’s responsibility to release this memory using the QMFree() function when it is no longer required. On Windows systems, this function must be used in place of the standard free() C runtime library routine to ensure compatibility with the memory allocator used by the QMClient library. On other platforms, use of QMFree() is strongly recommended but this should be compatible with the C run time library free() function.
The QMClient C API has three modes of operation for handling character string data:

1. The functions documented in the following sections use 8-bit characters. Using these functions to return data that is outside the 8-bit character set (e.g., reading a record containing ECS characters) will return only the low order 8 bits of each character. No error status will be reported.

2. The QMConnectionType() function can be used to select a mode in which the functions that pass character strings in 8-bit form do so using UTF-8 encoding. This allows an application using the 8-bit character functions to use characters from the entire Unicode Basic Multilingual Plane. Note that use of this mode requires that characters in the upper half of the 8-bit character set, including the multivalue mark characters, are correctly encoded as their UTF-8 equivalents. The five mark characters encode as:

   - Item mark: "\xC3\xBF"
   - Field mark: "\xC3\xBE"
   - Value mark: "\xC3\xBD"
   - Subvalue mark: "\xC3\xBC"
   - Text mark: "\xC3\xBB"

3. All functions in the QMClient C API that return a character string or have at least one character string argument have wide character variants formed by appending a W to the function name. For example, the wide character version of QMDel() is named QMDelW(). In the wide character versions, return values or arguments shown in the function definitions as char * become wchar_t *. Note that, for compatibility with the default behaviour of the C run time library, the definition of this data type is a 16 bit unsigned integer on Windows but a 32 bit unsigned integer on Linux and Unix systems.

   The wide character API functions can be used regardless of whether the server to which the connection is open is in ECS mode or not, however, passing characters with values over 255 in API calls in this situation may have unexpected effects. The QMIsECS() function can be used to determine whether the server supports ECS or not.

QMBasic subroutines and functions can update their argument values. For the C API this would require that the memory space allocated to a subroutine argument that may be updated by the server must be large enough to hold the longest updated value that could occur. Failure to observe this rule may result in memory corruption on the client side of the connection. To avoid this issue, the C API supports an alternative mechanism for handling returned argument values where the original argument variable is not updated but instead the new value can be retrieved with the QMGetArg() function. To make use of this system, the client software should use QMCallx() instead of QMCall(). The QMCreateObject(), QMGet() and QMSet() functions also allow updated argument values to be retrieved with QMGetArg().

Using the Visual Basic .Net API

Use of QMClient from Visual Basic .Net requires two dynamic link libraries; qmclivb.dll and qmclilib.dll. Inclusion of the first of these libraries as a reference from the Visual Basic project and adding

   Imports qm.qmclient

to the Visual Basic classes makes the QMClient functions available. The descriptions for each QMClient function in this documentation show the VB.Net data types.

The dynamic link libraries must be installed on the client system. They are automatically placed in the Windows directory (not necessarily c:\windows) when QM is installed. These components may be freely copied and distributed as necessary. They can also be downloaded separately from the openqm.com web site.

All strings in the Visual Basic .Net API are Unicode.
Where a client application calls a QMBasic subroutine or function on the server that updates its argument values, the modified value will be visible in the client process too when using `QMCall()`. The alternative method of calling subroutines with `QMCallx()` and handling returned argument values via the `QMGetArg()` function is also available in the Visual Basic .Net API.

**Using the Visual Basic 2005 API**

QM currently retains support for Visual Basic 2005 using the `qmclient.dll` dynamic link library and a Visual Basic module named `QMClient 2005.vb` in the SYSCOM file that contains the API function definitions. Many of the function argument values passed by the VB.Net API as integers are passed as short integers in the old Visual Basic API and hence differ from the VB.Net definitions shown in the function descriptions in this documentation. See the `QMClient2005.vb` file for the correct definitions.

The `QMClient.dll` library must be installed on the client system. This library is placed in the Windows directory (not necessarily c:\windows) when QM is installed. These components may be freely copied and distributed as necessary.

All strings in the Visual Basic 2005 API are 8-bit.

Where a client application uses `QMCall()` to call a QMBasic subroutine or function on the server that updates its argument values, the modified values will be visible in the client process.

Visual Basic versions 5 and 6 are no longer supported though the function definition items for these in the SYSCOM file will not be deleted by the installer.

**Using the QMBasic Class Module API**

The `QMClient` class module is supplied as a globally catalogued item named `!QMCLIENT`. To create a QMClient session, the object is instantiated with a statement of the form

```vbnet
session = object("!qmclient")
```

The session is then connected to a server using the `CONNECT()` method:

```vbnet
ok = session->connect(hostname, port, user, password, account)
```

There is no equivalent of the `QMConnectLocal()` function in the QMBasic class module API.

Functions that return Boolean values, return 0 for False, 1 for True in the QMBasic class module API. Status codes referenced with a prefix of `SV_` in the function descriptions are defined in the SYSCOM KEYS.H include record with a prefix of `SV$`. For example, `SV_OK` becomes `SV$OK` for the QMBasic variant of the QMClient API.

All examples in the detailed function descriptions assume that the object has been instantiated as a variable named "session".

**Using the Java QMClient API**

Two distinct versions of the Java API are available. The original implementation is platform specific because it uses the Java Native Interface (JNI). This version can be downloaded from the OpenQM web site `Resources and Solutions` area. The Java QMClient API installation package includes separate HTML documentation covering its usage.
A pure Java version of the API was introduced at QM release 3.4-7 and can be found on the main downloads page of the web site. This version is portable across all platforms that support Java. Documentation for the functions in this API appears in the sections that follow. To use this API, the client program should use

```java
import qmclient.*;
```

and then reference the functions in the form qm.Name() such that, for example, the QMExtract() function becomes

```java
var = qm.Extract(dyn_array, field, value, subvalue);
```

All references to the Java API in the following sections relate to the portable Java API. All examples in the function descriptions assume that the object has been instantiated as a variable named "qm".

### Using QMClient from Python

The qmclient.py item in the SYSCOM file has definitions and descriptions for Python versions of the API functions and also has comment text at the top of the module that describes how to use it. Essentially, the Python program should use

```python
import qmclient.py as qm
```

and then reference the functions in the form qm.Name() such that, for example, the QMExtract() function becomes

```python
var = qm.Extract(dyn_array, field, value, subvalue)
```

If the qmclient.py file is not in the current directory, the PYTHONPATH environment variable must be set as a list of directories to search for imported modules in the same form as used for the PATH environment variable. Also note that on Windows, which has a case insensitive file system, the casing of Python module filenames must match that used in the import statement.

Because Python does not support passing arguments by reference, QMClient functions that use this capability in other languages are modified to return two items as a list such that, for example, QMRead() becomes

```python
rec, err = qm.Read(fno, id)
```

or QM Locate() becomes

```python
found, pos = qm.Locate(item, dyn_array, field, value, subvalue, order)
```

All examples in the function descriptions assume that the object has been instantiated as a variable named "qm".

### Error Handling in QMClient Applications

QMClient was originally developed for use in applications that had a directly associated user screen on which error messages could be displayed. It has subsequently found use in applications that do not have a directly associated screen such as the server side of a web application. In such situations when using some languages, it will be necessary to use the QMConnectionType function to disable display of error messages. To ensure that older applications continue to run unchanged, the default behaviour of QMClient is to display errors either using a pop-up message box (Windows) or by sending the message to the stderr error channel (Linux).

From release 2.11-3, nearly all QMClient functions start by setting the QMError() error string to null. Applications can test if QMError() is a null string in order to determine whether an error has
Errors are returned in this way even if the older style displayed messages are still enabled. The functions that do not change the `QMError` value are `QMFree`, `QMStatus` and `QMError` itself. Alternatively, applications can check the server status code as described later in this section.

### Multi-threading

The QMClient API supports multi-threading so long as two threads do not attempt an operation on the same session simultaneously. If a session could never be accessed by more than one thread simultaneously, no special coding is needed. In languages such as Java or Python where each QM session is instantiated as a separate object, each function call references the specific object to which it applies. For other languages, sessions are identified by a numeric value and a thread may switch between multiple sessions using `QMSetSession()` in which case QMClient will correctly track the currently active session in each thread. If a single session could be accessed by multiple threads simultaneously, it will be necessary to use semaphores or similar synchronisation techniques to ensure that actions on a single session do not overlap.

### API Function Summary

For the C API, "W" indicates that a wide string version of the function is available.

<table>
<thead>
<tr>
<th>Function</th>
<th>VB.Net</th>
<th>VB 2005</th>
<th>C</th>
<th>QMBasic</th>
</tr>
</thead>
<tbody>
<tr>
<td>QMCall</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Y</td>
</tr>
<tr>
<td>QMCallx</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>QMChange</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>QMChecksum</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>QMCreateObject</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Y</td>
</tr>
<tr>
<td>QMClearFile</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>QMClearSelect</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>QMClose</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>QMConnect</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Y</td>
</tr>
<tr>
<td>QMConnected</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>QMConnectionType</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>QMConnectLocal</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>QMConnectPool</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Y</td>
</tr>
<tr>
<td>QMCreateObject</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Y</td>
</tr>
<tr>
<td>QMDcount</td>
<td>Y</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QMDel</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td></td>
</tr>
<tr>
<td>QMDeluxe</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Y</td>
</tr>
<tr>
<td>QMDestroyObject</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>QMDecrypt</td>
<td>Y</td>
<td>W</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QMEncrypt</td>
<td>W</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Call a subroutine on the server
Call a subroutine on the server, returning updated arguments with QMGetArg()
Replace substrings in a string
Calculate a checksum value
Clear a file
Clear a select list
Close a file
Establish a session via a network connection
Verify whether connection is open
Set options that affect QMClient behaviour
Establish a session on the local system
Establish a connection pooling session via a network connection
Instantiate a QMBasic class instance object
Count delimited items in a string
Decrypt data
Delete a field, value or subvalue from a dynamic array
Delete a record from an open file
Delete a record from an open file, retaining the lock
Discard a QMBasic class instance object
Terminate a QMClient session
Terminate all QMClient sessions from this client
Encrypt data
<table>
<thead>
<tr>
<th>Function</th>
<th>VB.Net</th>
<th>VB 2005</th>
<th>C</th>
<th>QMBasic</th>
</tr>
</thead>
<tbody>
<tr>
<td>QMEncryptx</td>
<td>W</td>
<td></td>
<td></td>
<td>Encrypt data with random initialisation vector</td>
</tr>
<tr>
<td>QMEndCommand</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y Terminate a command started with QMExecute()</td>
</tr>
<tr>
<td>QMEnterPackage</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Y Enter a licensed package</td>
</tr>
<tr>
<td>QMError</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Return the text associated with the most recent error</td>
</tr>
<tr>
<td>QMEvalConv</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Evaluate a dictionary data definition item, applying conversion codes</td>
</tr>
<tr>
<td>QMEvaluate</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Y Evaluate a dictionary data definition item</td>
</tr>
<tr>
<td>QMExecute</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Y Execute a command</td>
</tr>
<tr>
<td>QMExitPackage</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Y Exit from a licensed package</td>
</tr>
<tr>
<td>QMExtract</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Extract a field, value or subvalue from a dynamic array</td>
</tr>
<tr>
<td>QMField</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Extract a substring from a delimited string</td>
</tr>
<tr>
<td>QMGet</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Y Return the value of a public variable or function in a QMBasic object</td>
</tr>
<tr>
<td>QMGetArg</td>
<td>Y</td>
<td></td>
<td>W</td>
<td>Get a returned argument value from QMCallx(), QMCreateObject(), QMGet() or QMSet()</td>
</tr>
<tr>
<td>QMGetSession</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Return currently selected session number</td>
</tr>
<tr>
<td>QMGetVar</td>
<td>Y</td>
<td></td>
<td>W</td>
<td>Y Fetch the value of an @-variable on the server</td>
</tr>
<tr>
<td>QMConv</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Y Apply input conversion codes</td>
</tr>
<tr>
<td>QMIConvs</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Y Apply input conversion codes to multivalued data</td>
</tr>
<tr>
<td>QMIndices</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Y Retrieve information about indices</td>
</tr>
<tr>
<td>QMIIns</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Y Insert a field, value or subvalue in a dynamic array</td>
</tr>
<tr>
<td>QMLsECS</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y Test if server is running in ECS mode</td>
</tr>
<tr>
<td>QMLocate</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Y Locate a field, value or subvalue in a dynamic array</td>
</tr>
<tr>
<td>QMLogto</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Y Switch to a different account</td>
</tr>
<tr>
<td>QMMarkMapping</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y Enable or disable mark mapping for a directory file</td>
</tr>
<tr>
<td>QMMatch</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Y Match a string against a pattern template</td>
</tr>
<tr>
<td>QMMatchfield</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Y Return the portion of a string that matches a pattern template element</td>
</tr>
<tr>
<td>QMNextPartial</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Y Return next part of a select list</td>
</tr>
<tr>
<td>QMOCConv</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Y Apply output conversion codes</td>
</tr>
<tr>
<td>QMOCConvs</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Y Apply output conversion codes to multivalued data</td>
</tr>
<tr>
<td>QMOpen</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Y Open a file</td>
</tr>
<tr>
<td>QMOpenSeq</td>
<td>Y</td>
<td></td>
<td>W</td>
<td>Y Open a sequential file</td>
</tr>
<tr>
<td>QMPoolIdle</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y Move a connection pooling session into an idle wait state</td>
</tr>
<tr>
<td>QMRead</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Y Read a record without locking</td>
</tr>
<tr>
<td>Function</td>
<td>VB.Net</td>
<td>VB 2005</td>
<td>C</td>
<td>QMBasic</td>
</tr>
<tr>
<td>----------------</td>
<td>--------</td>
<td>---------</td>
<td>-----</td>
<td>----------------------------------------------</td>
</tr>
<tr>
<td>QMReadBlk</td>
<td>Y</td>
<td>W</td>
<td>Y</td>
<td>Read a given number of bytes from a sequential file</td>
</tr>
<tr>
<td>QMReadl</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Read a record with a shareable read lock</td>
</tr>
<tr>
<td>QMReadList</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Read a select list</td>
</tr>
<tr>
<td>QMReadNext</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Read the next item from a select list</td>
</tr>
<tr>
<td>QMReadSeq</td>
<td>Y</td>
<td>W</td>
<td>Y</td>
<td>Read a line of text from a sequential file</td>
</tr>
<tr>
<td>QMReadu</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Read a record with an update lock</td>
</tr>
<tr>
<td>QMRecordlock</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Lock a record</td>
</tr>
<tr>
<td>QMRecordlocked</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Query the lock state of a record</td>
</tr>
<tr>
<td>QMRelease</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Release a record lock</td>
</tr>
<tr>
<td>QMReplace</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Replace a field, value or subvalue in a dynamic array</td>
</tr>
<tr>
<td>QMRespond</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Respond to a request for input from a command started with QMExecute()</td>
</tr>
<tr>
<td>QMRevision</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Return the client and server QM version numbers</td>
</tr>
<tr>
<td>QMRTrans</td>
<td>Y</td>
<td>W</td>
<td>Y</td>
<td>Perform an RTRANS() function to fetch data</td>
</tr>
<tr>
<td>QMSeek</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Position in a sequential file</td>
</tr>
<tr>
<td>QMSelect</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Generate a select list</td>
</tr>
<tr>
<td>QMSelectIndex</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Generate a select list from an alternate key index</td>
</tr>
<tr>
<td>QMSelectLeft</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Move left in an alternate key index</td>
</tr>
<tr>
<td>QMSelectPartial</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Optimised select operation, returning multiple keys</td>
</tr>
<tr>
<td>QMSelectRight</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Move right in an alternate key index</td>
</tr>
<tr>
<td>QMSet</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Set the value of a public variable or execute a public subroutine in a QMBasic object</td>
</tr>
<tr>
<td>QMSetLeft</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Position at the left in an alternate key index</td>
</tr>
<tr>
<td>QMSetRight</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Position at the right in an alternate key index</td>
</tr>
<tr>
<td>QMSetSession</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Select session to which subsequent actions apply</td>
</tr>
<tr>
<td>QMStatus</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Return the STATUS() value from the server</td>
</tr>
<tr>
<td>QMSysystem</td>
<td>Y</td>
<td>W</td>
<td>Y</td>
<td>Execute a QMBasic SYSTEM() function on the server</td>
</tr>
<tr>
<td>QMTrans</td>
<td>Y</td>
<td>W</td>
<td>Y</td>
<td>Perform a TRANS() function to fetch data</td>
</tr>
<tr>
<td>QMTrapCallAbort</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Enable client trapping of aborts in QMCall()</td>
</tr>
<tr>
<td>QMtxn</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Start, commit or abort a transaction on the server</td>
</tr>
<tr>
<td>QMWriteSeq</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Truncate a sequential file item</td>
</tr>
<tr>
<td>QMWrite</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Write a record</td>
</tr>
<tr>
<td>QMWriteBlk</td>
<td>Y</td>
<td>W</td>
<td>Y</td>
<td>Write a given number of bytes to a sequential file</td>
</tr>
<tr>
<td>QMWriteSeq</td>
<td>Y</td>
<td>W</td>
<td>Y</td>
<td>Write a line of text to a sequential file</td>
</tr>
<tr>
<td></td>
<td>VB.Net</td>
<td>VB 2005</td>
<td>C</td>
<td>QMBasic</td>
</tr>
<tr>
<td>--------------------------</td>
<td>--------</td>
<td>---------</td>
<td>-----</td>
<td>------------------------------</td>
</tr>
<tr>
<td>QMWriteSeqKey</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Y</td>
</tr>
<tr>
<td>QMWriteu</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Y</td>
</tr>
<tr>
<td>QMWriteuSeqKey</td>
<td>Y</td>
<td>Y</td>
<td>W</td>
<td>Y</td>
</tr>
</tbody>
</table>

**The Server Status Code**

Many functions return a server error status referenced as Err in the detailed function descriptions. In variants of the QMClient API that allow function argument variables to be updated by the function (C, VB.Net, QMBasic class module), this variable is an integer argument, passed by reference. In the Python variant, the error status is returned as a second return value. For the Java variant, applications should test the ServerError property.

In all cases, the value returned will be one of the following that broadly correspond to the various clauses applicable to the equivalent QMBasic statements.

- **0 SV_OK** Action successful
- **1 SV_ON_ERROR** Action took the ON ERROR clause to recover from a situation that would otherwise cause the server process to abort.
- **2 SV_ELSE** Action took the ELSE clause. In most cases the QMStatus() function can be used to determine the error number.
- **3 SV_ERROR** An error occurred for which extended error text can be retrieved using the QMError() function.
- **4 SV_LOCKED** The action was blocked by a lock held by another user. The QMStatus() function can be used to determine the blocking user.
- **5 SV_PROMPT** A command executed on the server is waiting for input. The only valid client functions when this status is returned are QMRespond(), QMEndCommand and QMDisconnect.

The tokens shown above are defined in the SYSCOM include record or module relevant to the language in use. For the Java and QMBasic variants of the API, the names are modified to be SV$xxx instead of SV_xxx.
7.1 Select lists in QMClient sessions

Because a QMClient session operates in two parts, one on the client and one on the server, it is important to understand how this affects use of select lists.

A select list may be constructed on the server by use of the QMSelect() or QMSelectPartial() functions or by execution of a program (QMCall) or a command (QMExecute).

Using the QMSelect() function followed by repeated use of the QMReadNext() function builds the select list on the server and then returns items one by one. This will use the same group by group optimisation that is described with the QMBasic SELECT statement but, because the select list is maintained on the server, retrieval of each entry requires passing of a message pair between the client and the server which may be slow. However, because the select list actually exists at the server side of the QMClient session, it is available for use by commands and programs executed on the server within the session.

A QMClient application can read the entire select list in a single operation using the QMReadList() function and then extract entries using the QMExtract() function. This requires the list to be fully constructed before it can be returned to the client and hence loses the performance advantage gained from the group by group selection process.

Use of QMSelectPartial() provides a way to gain much of the performance advantage of the group by group select but without needing to retrieve each record id separately. This function performs an optimised select operation on the server and then returns a number of ids as a field mark delimited string. The QMNextPartial() function then retrieves a further set of record ids until the list is exhausted. Because QMSelectPartial() returns the first part of the list immediately, the full list is not available on the server for use by commands or programs executed within the QMClient session.

QMNextPartial() is effectively the same as a series of QMReadNext() operations that merge the results into a single string before returning it to the server. It can therefore also be used to retrieve data from a select list constructed by QMSelect(), a program or a command.

There is a further consideration for processes that use QMReadNext() or QMNextPartial() and also update the file by adding new records. Because the optimised selection process is finding records one by one as processing progresses, any records written during the processing may appear later in the returned list. Using QMReadList() to construct and retrieve the entire list before processing commences ensures that records added during processing will not be included in the list.

If a QMClient application uses QMSelectPartial() but the server pre-dates introduction of this function, the entire list is returned similar to use of QMSelect() followed by QMReadList(). Similarly, use of QMNextPartial() with a server that does not support it is equivalent to use of QMReadList().
7.2 Security Issues of the QMClient API

In most systems, a normal terminal user is taken directly into the application on logging in and the application itself controls what the user can do. The ON.ABORT paragraph provides a mechanism to ensure that, even if the application fails, the user cannot fall back to a command prompt.

With QMClient, the client session is effectively at a command prompt from which it can open, read and write files, execute commands, or call subroutines. It becomes the responsibility of the client software to control what the user can do. A knowledgeable user with a valid user name and password could, however, develop a client session that connects in the same way as the application and then goes on to do almost anything. Setting appropriate access rights on files may help but is unlikely to be a perfect solution to this potential security threat.

The QMCLIENT configuration parameter can be used to control the level of access that a QMClient session has. It starts with the value defined in the QM configuration parameters and can be modified to a higher level using the CONFIG command but cannot be taken to a lower level in this way. Because QMClient sessions execute the LOGIN paragraph on connection, the CONFIG command is easily executed from this paragraph.

It may also be useful to validate the client network address (See @IP.ADDR) in the LOGIN paragraph.
7.3 QMClient Example

The program samples below show how the QMClient API can be used to build a simple application that displays details from a sales order system. They are based on the QM demonstration files that can be created using the `SETUP_DEMO` command. For simplicity, some error handling that would be appropriate to a real application has been omitted.

### C

Example to be supplied.

### VB.Net

Example to be supplied.

### QMBasic Class Module

```vba
program example
    prompt ''

    session = object('!qmclient')

    if not(session->Connect('127.0.0.1', -1, 'username', 'password', 'demo')) then
        stop 'Failed to connect to server: ' : session->Error
    end

    * Open files

    fSales = session->Open('SALES')
    if fSales = 0 then stop 'Cannot open SALES file'

    fStock = session->Open('STOCK')
    if fStock = 0 then stop 'Cannot open STOCK file'

    fCustomers = session->Open('CUSTOMERS')
    if fCustomers = 0 then stop 'Cannot open CUSTOMERS file'

    * Prompt for order number repeatedly until a blank is entered

    loop
        display 'Enter order number, blank to terminate:' :
        input SaleId
        until SaleId = ''

        SaleRec = session->Read(fSales, SaleId, Err)

        SaleDate = SaleRec<1>
        SaleCust = SaleRec<2>
        SaleItems = SaleRec<3>
        SaleNumItems = dcount(SaleItems, @vm)

        CustRec = session->Read(fCustomers, SaleCust, Err)
        CustName = CustRec<1>

        display
        display
        display 'Order Number ' : SaleId : ' (' : CustName : ')
        display
```
display 'Cust Date.... Item Description Qty Price Total'
Prefix = SaleCust : ' ' : oconv(SaleDate, 'D2DMDL[,A3]')
SaleTotal = 0
for line = 1 to SaleNumItems
  ItemId = SaleRec<3,line>
  ItemQty = SaleRec<4,line>
  ItemPrice = SaleRec<5,line>
  ItemPriceExt = oconv(ItemPrice, 'MD2')
  StockRec = session->Read(fStock, ItemId, Err)
  StockDescr = StockRec<1>
  ItemTotal = ItemPrice * ItemQty
  SaleTotal += ItemTotal;
  ItemTotalExt = oconv(ItemTotal, 'MD2')
  * 0  1  2  3  4  5  6
  7
  1234567890123456789012345678901234567890123456789012345678901234567890
  * Cust Date.... Item Description Qty Price Total
  * xxxxxxxxxxxxxxx xxx xx xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx xxx
  xxxxxxx xxxxxxx
  s = space(76)
s[1,15] = Prefix
s[18,4] = ItemId
s[24,30] = StockDescr
s[56,3] = fmt(ItemQty, '3R')
s[61,7] = fmt(ItemPriceExt, '7R')
s[61,7] = fmt(ItemTotalExt, '7R')
display s
  Prefix = ''
next line
SaleTotalExt = oconv(SaleTotal, 'MD2')
display fmt('=======', '76R')
display fmt(SaleTotalExt, '76R')
repeat
  session->Disconnect()
end

Java
import qmclient.*;
import java.io.*;

class ex
{
  private static qmclient qm;

  public static void main(String[] args) throws IOException
  {
    int fSales;    // SALES file
int fStock;            // STOCK file
int fCustomers;        // CUSTOMERS file
String SaleId;        // SALES record id
String SaleRec;       // SALES record
String SaleDate;      // Date
String SaleCust;      // Customer id
String SaleItems;     // Items in order
int SaleNumItems;     // Order line count
int SaleTotal;        // Total value of order, internal
format
String SaleTotalExt;  // Total value of order, external
format
String CustRec;       // Customer record
String CustName;      // Customer name
String ItemId;        // Item id
String ItemQty;       // Quantity in order line
String ItemPrice;     // Price, internal format
String ItemPriceExt;  // Price, external format
int ItemTotal;        // Item total, internal format
String ItemTotalExt;  // Item total, external format
String StockRec;      // Stock record
String StockDescr;    // Item description
String Prefix;        // Single valued items as line prefix
int line;             // Order line counter

qm = new qmclient();

if (!qm.ConnectLocal("demo"))
{
    System.out.println("Failed to connect to server: "+
qm.Error);
    return;
}

// Open files

fSales = qm.Open("SALES");
if (fSales == 0)
{
    System.out.println("Cannot open SALES file");
    return;
}

fStock = qm.Open("STOCK");
if (fStock == 0)
{
    System.out.println("Cannot open STOCK file");
    return;
}

fCustomers = qm.Open("CUSTOMERS");
if (fCustomers == 0)
{
    System.out.println("Cannot open CUSTOMERS file");
    return;
}

// Prompt for order number repeatedly until a blank is entered
while(true)
{
    System.out.print("Enter order number, blank to terminate:");
    SaleId = System.console().readLine();
    if (SaleId.length() == 0) break;

    SaleRec = qm.Read(fSales, SaleId);
    SaleDate = qm.Extract(SaleRec, 1, 0, 0);
    SaleCust = qm.Extract(SaleRec, 2, 0, 0);
    SaleNumItems = qm.Dcount(SaleItems, "\u00FD");
    CustRec = qm.Read(fCustomers, SaleCust);
    CustName = qm.Extract(CustRec, 1, 0, 0);

    System.out.println("";  
    System.out.println("";  
    System.out.println("" + Order Number " + SaleId +  
    " " + qm.OConv(SaleDate,  
    "D2DMYL,A3")")};  
    SaleTotal = 0;
    for(line = 1; line <= SaleNumItems; line++)
    {
        ItemId = qm.Extract(SaleRec, 3, line, 0);
        ItemQty = qm.Extract(SaleRec, 4, line, 0);
        ItemPrice = qm.Extract(SaleRec, 5, line, 0);
        ItemTotal = Integer.valueOf(ItemPrice) * 
        Integer.valueOf(ItemQty);
        SaleTotal += ItemTotal;
        ItemTotalExt = qm.OConv(String.valueOf(ItemTotal), "MD2");

        System.out.printf("%15s  %4s  %-30s  %3s  %7s  %7s%n",  
        Prefix, ItemId, StockDescr, ItemQty, ItemPriceExt,  
        ItemTotalExt);
    
    Prefix = "";
}

SaleTotalExt = qm.OConv(String.valueOf(SaleTotal), "MD2");
System.out.printf("%76s%n", "======");
System.out.printf("%76s%n", SaleTotalExt);
}
qm.Disconnect();
}
import qmclient as qm
import sys

VM = chr(253)

if not(qm.ConnectLocal("DEMO")):
    print("Failed to connect to server: ", qm.Error())
    sys.exit()

# Open files

fSales = qm.Open("SALES")
if fSales == 0:
    print("Cannot open SALES file")
    sys.exit()

fStock = qm.Open("STOCK")
if fStock == 0:
    print("Cannot open STOCK file")
    sys.exit()

fCustomers = qm.Open("CUSTOMERS")
if fCustomers == 0:
    print("Cannot open CUSTOMERS file")
    sys.exit()

# Prompt for order number repeatedly until a blank is entered

while True:
    SaleId = input("Enter order number, blank to terminate:")
    if SaleId == ": break

    SaleRec, Err = qm.Read(fSales, SaleId)
    SaleDate = qm.Extract(SaleRec, 1, 0, 0)
    SaleCust = qm.Extract(SaleRec, 2, 0, 0)
    SaleItems = qm.Extract(SaleRec, 3, 0, 0)
    SaleNumItems = qm.Dcount(SaleItems, VM)

    CustRec, Err = qm.Read(fCustomers, SaleCust)
    CustName = qm.Extract(CustRec, 1, 0, 0)

    print("Cust   Date.....  Item  Description                    Qty    Price    Total")
    Prefix = SaleCust + "  " + qm.OConv(SaleDate, "D2DMYL[,A3"]
    SaleTotal = 0
for line in range(1, SaleNumItems + 1):
    ItemId = qm.Extract(SaleRec, 3, line, 0)
    ItemQty = qm.Extract(SaleRec, 4, line, 0)
    ItemPrice = qm.Extract(SaleRec, 5, line, 0)
    ItemPriceExt = qm.OConv(ItemPrice, "MD2")

    StockRec, Err = qm.Read(fStock, ItemId)
    StockDescr = qm.Extract(StockRec, 1, 0, 0)
    ItemTotal = int(ItemPrice) * int(ItemQty)
    SaleTotal += ItemTotal
    ItemTotalExt = qm.OConv(str(ItemTotal), "MD2")
    print('{:<15} {:>4} {:<30} {:>3} {:>7}'.format(Prefix, ItemId, StockDescr, ItemQty, ItemPriceExt, ItemTotalExt))
    Prefix = ""

    SaleTotalExt = qm.OConv(str(SaleTotal), "MD2")
    print('{:>76}'.format("======="))
    print('{:>76}'.format(SaleTotalExt))

qm.Disconnect()
7.4 QMCall()

The QMCall() function calls a catalogued subroutine on the server. It is analogous to the QMBasic CALL statement.

The subroutine may take up to 20 arguments that must be passed as strings. QMClient does not provide a method to call subroutines with a greater number of arguments or those that are defined to pass a QMBasic dimensioned matrix as an argument.

The called subroutine may make use of any of the standard QMBasic programming statements and functions, however, it may not perform terminal input that is not handled via the data queue as there is no terminal associated with the server process. Any terminal output will be discarded unless redirected to a COMO file.

The QMCLIENT configuration parameter can be used to restrict QMCall() to calling only subroutines that have been compiled with the $QMCALL compiler directive. Use of this feature prevents a malicious application that uses the QMClient API calling other subroutines that might not perform adequate security validation to protect the system.

Handling values returned via the subroutine arguments differs depending on the variant of the QMClient API in use. Some update the argument variables directly, others require use of the QMGetArg() function. The language specific descriptions below give more details.

The example program fragments below call a subroutine on the server that sets the credit limit for a specific client. The subroutine takes three arguments, the third of which is used to return an error message if the action fails. Note how the ErrMsg variable is set to a null string before calling the subroutine so that any old content of this variable is not unnecessarily sent across the network to the server. To optimise network traffic, the server only returns arguments values if they have been modified by the called subroutine.

C

```c
void QMCall(char * SubrName, short int ArgCount, ArgList...) {
    ErrMsg[0] = '\0';
    QMCall("SET.CREDIT.LIMIT", 3, Client, NewLimit, ErrMsg);
    if (ErrMsg[0] != '\0') printf("Failed - %s\n", ErrMsg);
}
```

Use of this function in the C API is not recommended as the size of any argument variable that may be overwritten by the subroutine must be large enough to receive the updated value. Failure to observe this rule will result in memory corruption. See the QMCallx() function for an alternative way to handle returned argument values.

VB.Net

```vbnet
QMCall(ByVal SubrName as String, ByVal ArgCount as Short, Optional ByRef Arg1 as String, ...)

ErrMsg = ""
QMCall("SET.CREDIT.LIMIT", 3, Client, NewLimit, ErrMsg)
If ErrMsg <> "" Then MsgBox(ErrMsg, MsgBoxStyle.Exclamation, "Failed")
```

If the subroutine modifies the values of any of its arguments, this will be reflected in the variables specified in ArgList on return from the call.
QMBasic Class Module

Call(SubrName, ArgList...)

ErrMsg = ""
session->Call("SET.CREDIT.LIMIT", Client, NewLimit, ErrMsg)
if ErrMsg # "" then display "Failed - " : ErrMsg

If the subroutine modifies the values of any of its arguments, this will be reflected in the variables
specified in ArgList on return from the call.

Java

Call(String SubrName, String Arg1, String Arg2...)

ErrMsg = "";
qmc.Call("SET.CREDIT.LIMIT", Client, NewLimit, ErrMsg);
ErrMsg = qmc.GetArg(3);
if (ErrMsg != "") System.out.printf("Failed - %s\n", ErrMsg);

Note use of the GetArg() method to retrieve the value of the modified ErrMsg argument. The
ErrMsg variable in the call could have been replaced with a literal null string as it is an output only
argument.

Python

Call(SubrName, ArgCt, Arg1, Arg2...)

ErrMsg = ""
qmc.Call("SET.CREDIT.LIMIT", 3, Client, NewLimit, ErrMsg)
ErrMsg = qmc.GetArg(3);
if ErrMsg == "": print("Failed", ErrMsg)

Note use of QMGetArg() to retrieve the value of the modified ErrMsg argument. The ErrMsg
variable in the call could have been replaced with a literal null string as it is an output only
argument.

See also:
QMCallx, QMTrapCallAbort
7.5 \textbf{QMCallx()}

The \texttt{QMCallx()} function calls a catalogued subroutine on the server. It is analogous to the QMBasic \texttt{CALL} statement. This function is only present in the C and VB.Net versions of the QMClient API. For other versions, use \texttt{QMCall()}.

The subroutine may take up to 20 arguments that must be passed as strings. QMClient does not provide a method to call subroutines with a greater number of arguments or those that are defined to pass a QMBasic dimensioned matrix as an argument.

The return value from \texttt{QMCallx()} is zero if successful or non-zero in the case of an error. The actual return value is one of the SV_	extit{xxx} codes as defined in the qmclilib.h include record.

Unlike \texttt{QMCall()}, the updated values of any arguments modified by the subroutine do not overwrite the original argument variables. Instead, they can be retrieved using the \texttt{QMGetArg()} function, removing the need to know the size of the returned data prior to the call.

The called subroutine may make use of any of the standard QMBasic programming statements and functions, however, it may not perform terminal input that is not handled via the data queue as there is no terminal associated with the server process. Any terminal output will be discarded unless redirected to a COMO file.

The \texttt{QMCLIENT} configuration parameter can be used to restrict \texttt{QMCallx()} to calling only subroutines that have been compiled with the \texttt{QMCALL} compiler directive. Use of this feature prevents a malicious application that uses the QMClient API calling other subroutines that might not perform adequate security validation to protect the system.

The example below calls a subroutine on the server that sets the credit limit for a specific client. The subroutine takes three arguments, the third of which is used to return an error message if the action fails. Note how the ErrMsg variable is set to a null string before calling the subroutine so that any old content of this variable is not unnecessarily sent across the network to the server. To optimise network traffic, the server only returns arguments values if they have been modified by the called subroutine.

**C**

```c
int QMCallx(char * SubrName, short int ArgCount, ArgList...)

ErrMsg[0] = '\0';
QMCallx("SET.CREDIT.LIMIT", 3, Client, NewLimit, ErrMsg);
ErrMsg = QMGetArg(3);
if (ErrMsg[0] != '\0') printf("Failed - %s\n", ErrMsg);
```

The return value from this function is the ServerError status and will be SV_OK if the call is successful. The memory allocated by the \texttt{QMGetArg()} function must subsequently be released using \texttt{QMFree()}.

**VB.Net**

```vbnet
QMCallx(ByVal SubrName as String, ByVal ArgCount as Short, Optional ByVal Arg1 as String, ...) as Integer

ErrMsg = ""
If QMCallx("SET.CREDIT.LIMIT", 3, Client, NewLimit, ErrMsg) Then
ErrMsg = QMGetArg(3)
```

4.0-0
If ErrMsg <> "" Then MsgBox(ErrMsg, MsgBoxStyle.Exclamation, "Failed")
End If

The return value from this function is the ServerError status and will be SV_OK if the call is successful. Note the use of QMGetArg() to retrieve the updated argument value.

See also:
QMCall, QMTrapCallAbort
7.6 QMChange()

The QMChange() function replaces occurrences of one substring with another in a string. It is analogous to the QMBasic CHANGE() function.

The function arguments are:

- **Src** is the string to be processed.
- **OldStr** is the substring to be replaced.
- **NewStr** is the replacement substring.
- **Occurrences** is the number of occurrences of **OldStr** to be replaced. If omitted or specified as less than one, all occurrences are replaced. Note that the Visual Basic 2005 API passes this argument by reference.
- **Start** is the occurrence number from one of the first occurrence of **OldStr** to be replaced. If omitted or less than one, replacement commences at the first occurrence of **OldStr**. Note that the Visual Basic 2005 API passes this argument by reference.

The QMChange() function returns a new string with the specified substrings replaced. This function is evaluated on the client system and does not require a server connection to be open.

One typical use of QMChange() is to replace mark characters with carriage return / line feed pairs when transferring data from a dynamic array to a multi-line text box.

The examples below change three uppercase letter "A" to lowercase "a" in the supplied string starting at the second occurrence, returning the result.

**C**

```c
char * QMChange(char * Src, char * OldStr, char * NewStr, int Occurrences, int Start)
X = QMChange("ABRACADABRA", "A", "a", 3, 2);
```

The memory allocated to receive the result of the QMChange() function must be released using QMFree() when no longer required.

**VB.Net**

```vbnet
QMChange(ByVal Src as String, ByVal OldStr as String, ByVal NewStr as String, Optional ByVal Occurrences as Integer, Optional ByVal Start as Integer) as String
X = QMChange("ABRACADABRA", "A", "a", 3, 2)
```

The **Occurrences** and **Start** arguments are optional, defaulting to zero and one respectively.

**QMBasic Class Module**

This function is not supported by the QMCClient class module as it is a QMBasic function.

**Java**

```java
String Change(String Src, String OldStr, String NewStr, int Occurrences, int Start)
X = qm.Change("ABRACADABRA", "A", "a", 3, 2);
```
Python

\textbf{Change}(\textit{Src, OldStr, NewStr, Occurrences, Start})

\begin{verbatim}
X = qm.Change("ABRACADABRA", "A", "a", 3, 2)
\end{verbatim}

The \textit{Occurrences} and \textit{Start} arguments are optional, defaulting to zero and one respectively.
7.7 QMChecksum()

The QMChecksum() function returns a checksum value for supplied data. It is analogous to the QMBasic CHECKSUM() function with the default algorithm.

The function takes a single argument, Data, as the string for which the checksum is to be evaluated. Note that the algorithm used by QM may be different from that of other multivalue products and hence may yield a different result for the same data.

When using a QMClient session that is operating in UTF-8 mode, the checksum is based on the decoded character string.

This function is evaluated on the client system and does not require a server connection to be open.

C

    int QMChecksum(char * Data)
    X = QMChecksum("ABCDE");

VB.Net

    QMChecksum(ByVal Data as String) As Integer
    X = QMChecksum("ABCDE")

QMBasic Class Module

    This function is not supported by the QMClient class module as it is a QMBasic function.

Java

    This function is not currently implemented in the QMClient Java API.

Python

    This function is not currently implemented in the QMClient Python API.
7.8 QMClearFile()

The QMClearFile() function deletes all content from a file. It is analogous to the QMBasic CLEARFILE statement.

The function takes a single argument, FileNo, as the file number returned from a previous use of the QMOpen() function.

The code fragment in the examples below opens the CLIENTS file and clears it.

**C**

```c
void QMClearFile(int FileNo)
    fClients = QMOpen("CLIENTS");
    QMClearFile(fClients);
```

**VB.Net**

```vbnet
QMClearFile(ByVal FileNo as Integer)
    fClients = QMOpen("CLIENTS")
    QMClearFile(fClients)
```

**QMBasic Class Module**

```qbasic
ClearFile(FileNo)
    fClients = session->Open("CLIENTS")
    session->ClearFile(fClients)
```

**Java**

```java
ClearFile(int FileNo)
    fClients = qm.Open("CLIENTS");
    qm.ClearFile(fClients);
```

**Python**

```python
ClearFile(FileNo)
    fClients = qm.Open("CLIENTS")
    qm.ClearFile(fClients)
```
7.9 QMClearSelect()

The **QMClearSelect()** function clears a select list. It is analogous to the QMBasic `CLEARSELECT` statement.

The function takes a single argument, *ListNo*, which is the select list number to be cleared. This must be in the range 0 to 10. No error occurs if the list was not active.

Applications that could leave unprocessed items in the list should always clear when it is no longer required. This is particularly important with the default select list (list 0) as items left in this list may affect commands executed on the server later in the session.

The program fragment in the examples below builds select list 1 and uses it to process records from the file open as fClients. It exits from the loop when it finds the first record where field 1 is empty. Because this could leave a partially processed select list, **QMClearSelect()** is used to clear the list.

### C

```c
void QMClearSelect(int ListNo)
{
    QMSelect(fClients, 1);
    do {
        Id = QMReadNext(1);
        if (Id == NULL) break;
        Rec = QMRead(fClients, Id, Err);
        QMFree(Id);
        Fld = QMExtract(Rec, 1, 0, 0);
        QMFree(Rec);
        if (Fld[0] == '\0') break;
        QMFree(Fld);
    } while(1);
    QMClearSelect(1);
}
```

Note that this example leaves variables *Fld* pointing to a dynamically allocated memory area on exit from the loop. This must be released using **QMFree()** when no longer needed.

### VB.Net

```vbnet
QMClearSelect ByVal ListNo as Integer
{
    QMSelect fClients, 1
    Do
        Id = QMReadNext(1)
        If ID is Nothing Then Exit Do
        Rec = QMRead(fClients, Id, Err)
        If QMExtract(Rec, 1, 0, 0) = "" Then Exit Do
    Loop
    QMClearSelect 1
}
```

### QMBasic Class Module

```vbnet
ClearSelect(ListNo)
{
    session->Select(fClients, 1)
    loop
        Id = session->ReadNext(1)
        until Id = ""
        Rec = session->Read(fClients, Id, Err)
}
```
until Rec<1> = ""
repeat
    session->ClearSelect(1)

Java

ClearSelect(int ListNo)

    qm.Select(fClients, 1);
    do {
        Id = qm.ReadNext(1);
        if (qm.ServerError != SV$OK) break;
        Rec = qm.Read(fClients, Id);
        Fld = qm.Extract(Rec, 1, 0, 0);
    } while(Fld.length() != 0);
    qm.ClearSelect(1);

Python

ClearSelect(ListNo)

    QMSelect(fClients, 1)
    while True:
        Id = qm.ReadNext(1)
        if Id = "": break
        Rec, Err = QMRead(fClients, Id)
        If Err != SV_OK: break
        Fld = QMExtract(Rec, 1, 0, 0)
        if Fld == "": break
    qm.ClearSelect(1)
7.10 QMClose()

The QMClose() function closes a file. It is analogous to the QMBasic CLOSE statement.

The function takes a single argument, FileNo, which is the file number returned by a previous QMOpen() call.

The server maintains a list of files open for processing by the client application. The QMClose() function causes the server to close the specified file. It is not normally necessary to close files as there is no practical limit to the number of files that the server can hold open at once, however, for best performance applications should close files if they are unlikely to be referenced for a considerable time.

The program fragment in the examples below opens the CLIENTS file, reads the record identified by ClientNo, and then closes the file. A real program should test the error status from the read to determine if the action was successful.

**C**

```c
void QMClose(int FileNo)
{
    fClients = QMOpen("CLIENTS");
    Rec = QMRead(fClients, ClientNo, Err);
    QMClose(fClients);
}
```

Note that this example leaves variables Rec pointing to a dynamically allocated memory area that must be released using QMFree() when no longer needed.

**VB.Net**

```vbnet
QMClose ByVal FileNo as Integer
{
    fClients = QMOpen("CLIENTS")
    Rec = QMRead(fClients, ClientNo, Err)
    QMClose(fClients)
}
```

**QMBasic Class Module**

```qmb
Close(FileNo)
{
    fClients = session->Open("CLIENTS")
    Rec = session->Read(fClients, ClientNo, Err)
    session->Close(fClients)
}
```

**Java**

```java
Close(int FileNo)
{
    fClients = qm.Open("CLIENTS");
    Rec = qm.Read(fClients, ClientNo);
    qm.Close(fClients);
}
```

**Python**

```python
Close(FileNo)
{
    fClients = qm.Open("CLIENTS")
    Rec = qm.Read(fClients, ClientNo)
    qm.Close(fClients)
}
7.11 QMConnect()

The QMConnect() function establishes a QMClient session.

The function arguments are:

- **Host** is the IP address or name of the server system. Use of IPV6 addresses in the C and VB.Net variants of the QMClient API requires that the QMConnectionType() function is used first to enabled IPV6 support.

- **Port** is the port number to which connection is to be made. Set this to -1 to use the default QMClient port.

- **UserName** is the user name under which the server process is to run.

- **Password** is the password for the given UserName.

- **Account** is the name of the QM account to be accessed.

The QMConnect() function attempts to establish a QMClient process on the system identified by the Host argument. If successful, the function returns True. If unsuccessful, the function returns False and the QMError() function can be used to retrieve a text error message identifying the cause of the failure.

Host can reference the local machine. For an alternative method of starting a local QM session, see the QMConnectLocal() function.

A single client may open multiple connections simultaneously. For the C and VB.Net variants of QMClient, the internal session number associated with the session opened by QMConnect() can be retrieved using QMGetSession(). All subsequent QMClient function calls relate to the most recently opened session unless QMSetSession() is used to select an alternative session. For the QMBasic class module, Java and Python variants of QMClient, each session is instantiated as a separate object.

QMClient sessions run the MASTER.LOGIN and/or LOGIN paragraph (if present). A QMClient session can be recognised within these paragraphs by testing the value of @TTY which will be "vbsrvr" for QMClient. Note that a prompt for input within the actions performed by the paragraph cannot be handled via QMClient.

The program fragments in the examples below attempt to connect to the server identified by Host, using login credentials in UserName, Password and Account. If unsuccessful, an error message is displayed including the text returned by QMError().

**C**

```c
int QMConnect(char * Host, int Port, char * UserName, char * Password, char * Account)
if (!QMConnect(Host, -1, UserName, Password, Account))
{
    printf("Failed to connect - %s\n", QMError());
}
```

**VB.Net**

```vbnet
QMConnect(ByVal Host as String, ByVal Port as Integer, ByVal UserName as String,
ByVal Password as String, ByVal Account as String) as Boolean
```
If Not QMConnect(Host, -1, UserName, Password, Account) Then
    MsgBox(QMError(), MsgBoxStyle.Exclamation, "Failed to connect")
End If

**QMBasic Class Module**

Connect(Host, Port, UserName, Password, Account)

    session = object("!qmclient")
    if not(session->Connect(Host, -1, UserName, Password, Account))
        stop "Failed to connect - " : session->Error
    end

**Java**

    boolean Connect(String Host, int Port, String UserName, String Password, String Account)
    if (!qm.Connect(Host, -1, UserName, Password, Account)) {
        System.out.println("Failed to connect - " + qm.Error());
    }

**Python**

    Connect(Host, Port, UserName, Password, Account)
    if not qm.Connect(Host, -1, UserName, Password, Account):
        print("Failed to connect - ", qm.Error())

See also:
QMConnectLocal(), QMDisconnect
7.12 QMConnected()

The `QMConnected()` function confirms whether a QMClient session is connected to a server.

The examples below test whether a QMClient connection is open and, if not, display an error message.

**C**

```c
int QMConnected()
if (!QMConnected())
{
    printf("Not connected");
}
```

**VB.Net**

```vbnet
QMConnected() as Boolean
If Not QMConnected() Then
    MsgBox("Not connected", MsgBoxStyle.Exclamation)
End If
```

**QMBasic Class Module**

```qbasic
Connected
if not(session->Connected) then
    display "Not connected"
end
```

**Java**

```java
boolean Connected(void)
if (!qm.Connected())
{
    System.out.println("Not connected");
}
```

**Python**

```python
Connected()
if not qm.Connected():
    print("Not connected")
```
7.13 QMConnectionType()

The QMConnectionType() function sets parameters that affect the behaviour of QMClient.

The function takes a single argument, Type, which is a bit significant value that controls the session parameters.

Type can be formed from any of the following additive values:

1  The next session to be opened will not take part in the device licensing system.
2  The next session to be opened will not use encrypted data traffic. This may result in increased transfer speeds but at the expense of weakened security. Note that the login authentication data (user name and password) is always encrypted when both the client and the server support this, regardless of the setting of this parameter.
4  The next session to be opened must use encrypted data traffic. If the server does not support encryption, the session will not be established.
8  Suppress display of error messages from within the QMClient library. Once set, this mode persists even if a later call to QMConnectionType does not include this value. Applications must test the value returned by QMError() to determine if an error has occurred.
32  All functions passing strings as 8-bit data use UTF-8 representation (C API only)
64  Enable IPV6 support.

Additional bit values may be used in future releases or for internal undocumented purposes.

The examples below use the QMConnectionType() function to disable network traffic encryption before attempting to connect to the server.

**C**

```c
void QMConnectionType(int Type)

QMConnectionType(2);
if (!QMConnect(Host, -1, UserName, Password, Account))
{
    printf("Failed to connect - %s\n", QMError());
}
```

**VB.Net**

```vbnet
QMConnectionType ByVal Type as Integer

QMConnectionType(2)
If Not QMConnect(Host, -1, UserName, Password, Account) Then
    MsgBox(QMError(), MsgBoxStyle.Exclamation, "Failed to connect")
End If
```

**QMBasic Class Module**

```vbnet
session->ConnectionType = Type

session = object("!qmclient")
session->ConnectionType = 2
if not(session->Connect(Host, -1, UserName, Password, Account))
then
    stop "Failed to connect - " : session->error()
```
The QMConnectionType function is implemented as a public property variable in this version of the QMClient API.

**Java**

```java
int ConnectionType
q.m.ConnectionType = 2;
if (!q.m.Connect(Host, -1, UserName, Password, Account))
    { 
        System.out.println("Failed to connect - " + q.m.Error()); 
    }
```

The QMConnectionType function is implemented as a public property variable in this version of the QMClient API.

**Python**

```python
ConnectionType(Type)
q.m.ConnectionType(2)
if not q.m.Connect(Host, -1, UserName, Password, Account):
    print("Failed to connect - ", q.m.Error())
```
7.14 QMConnectLocal()

The QMConnectLocal() function attempts to establish a QMClient process on the local system. The process runs as the user executing the function. If successful, the function returns True. If unsuccessful, the function returns False and the QMError() function can be used to retrieve a text error message identifying the cause of the failure.

A single client may open multiple connections simultaneously. For the C and VB.Net variants of QMClient, the internal session number associated with the session opened by QMConnectLocal() can be retrieved using QMGetSession(). All subsequent QMClient function calls relate to the most recently opened session unless QMSetSession() is used to select an alternative session. For the QMBasic class module, Java and Python variants of QMClient, each session is instantiated as a separate object.

QMClient sessions run the LOGIN paragraph (if present) but not the MASTER.LOGIN paragraph. A QMClient session can be recognised within this paragraph by testing the value of @TTY which will be "vbsrvr" for QMClient. Note that a prompt for input within the actions performed by the LOGIN paragraph cannot be handled via QMClient.

Use of QMConnectLocal() needs to know the location of the QMSYS directory. If this is not in its default location (C:\QMSYS on Windows, /usr/qmsys on other platforms), the QMSYS environment variable must be set to specify the location.

C

```c
int QMConnectLocal(char * Account)
if (!QMConnectLocal(Account)) {
     printf("Failed to connect - %s\n", QMError());
}
```

VB.Net

```vbnet
QMConnectLocal(ByVal Account as String) as Boolean
If Not QMConnectLocal(Account) Then
    MsgBox(QMError(), MsgBoxStyle.Exclamation, "Failed to connect")
End If
```

QMBasic Class Module

ConnectLocal() is not supported by the QMBasic class module API.

Java

```java
boolean ConnectLocal(String Account)
if (!qm.ConnectLocal(Account)) {
    System.out.println("Failed to connect - " + qm>Error());
}
```

The server must be running QM version 3.4-7 or later.

Python

```python
ConnectLocal(Account)
```
if not qm.ConnectLocal(Account):
    print("Failed to connect - ", qm.Error())

See also:
QMConnect(), QMDisconnect
7.15 QMConnectPool()

The QMConnectPool() function establishes a QMClient session using connection pooling.

The function arguments are:

- **Host** is the IP address or name of the server system. Use of IPV6 addresses requires that the QMConnectionType() function is used first to enabled IPV6 support.

- **Port** is the port number to which connection is to be made. Set this to -1 to use the default QMClient port.

- **UserName** is the user name under which the server process is to run.

- **Password** is the password for the given **UserName**.

- **Account** is the name of the QM account to be accessed.

- **Pool** is the connection pool name.

The QMConnectPool() function attempts to establish a QMClient process on the system identified by the **Host** argument. If successful, the function returns True. If unsuccessful, the function returns False and the QMError() function can be used to retrieve a text error message identifying the cause of the failure. If there is an idle QMClient process that matches the user, account and pool names, the function connects to this session rather than creating a new one.

A single client may open multiple connections simultaneously. For the C and VB.Net variants of QMClient, the internal session number associated with the session opened by QMConnectPool() can be retrieved using QMGetSession(). All subsequent QMClient function calls relate to the most recently opened session unless QMSetSession() is used to select an alternative session. For the QMBasic class module, Java and Python variants of QMClient, each session is instantiated as a separate object.

QMClient sessions run the MASTER.LOGIN and/or LOGIN paragraph (if present). A QMClient session can be recognised within these paragraphs by testing the value of @TTY which will be "vbsrvr" for QMClient. Note that a prompt for input within the actions performed by the paragraph cannot be handled via QMClient.

```c
int QMConnectPool(char * Host, int Port, char * UserName, char * Password, char * Account, char * Pool)

if (!QMConnectPool(Host, -1, UserName, Password, Account, Pool)) {
    printf("Failed to connect - %s\n", QMError());
}
```

```vb.net
QMConnectPool(ByVal Host as String, ByVal Port as Integer, ByVal UserName as String,
ByVal Password as String, ByVal Account as String, ByVal Pool as String) as Boolean

If Not QMConnectPool(Host, -1, UserName, Password, Account, Pool) Then
```
MsgBox(QMError(), MsgBoxStyle.Exclamation, "Failed to connect")
End If

**QMBasic Class Module**

*Bool sessionゐConnectPool(Host, Port, UserName, Password, Account, Pool)*

*session = object("!qmclient")*
*if not(session->ConnectPool(Host, -1, UserName, Password, Account, Pool)) then*
*stop "Failed to connect - " : session->error() end*

**Java**

*boolean ConnectPool(String Host, int Port, String UserName, String Password, String Account, String Pool)*

*if (!qm.ConnectPool(Host, -1, UserName, Password, Account, Pool)) {*
*    System.out.println("Failed to connect - " + qm.Error());*
*}*

**Python**

*ConnectPool(Host, Port, UserName, Password, Account, Pool)*

*if not qm.ConnectPool(Host, -1, UserName, Password, Account, Pool):*
*    print("Failed to connect - ", qm.Error())*

**See also:**
Connection pooling, QMDisconnect, QMPoolIdle()
7.16 QMCreateObject()

The **QMCreateObject()** function creates an instance of the named class on the server, returning a numeric value that can be used in other QMClient functions that reference this object. This value is returned as zero if instantiation fails.

The function arguments are:

- **Class** is the name of the QM class from which the object is to be instantiated.
- **ArgCt** is the number of argument values following. This argument is not present in all variants of the QMClient API.
- **Arg1...** is a series of argument values as strings. A maximum of 20 arguments may be given.

The argument values are passed into the CREATE.OBJECT public subroutine, if this exists.

When using the C and VB.Net versions of the QMClient API, arguments must be passed as strings. Any argument values updated by the CREATE.OBJECT subroutine can be retrieved using the **QMGetArg()** function.

For the old Visual Basic version of the QMClient API, the original variables are updated with any modifications from the CREATE.OBJECT subroutine.

The object remains instantiated until either the connection is closed or the **QMDestroyObject()** function is used.

The **QMCALL** configuration parameter can be used to restrict **QMCreateObject()** to instantiating only classes that have been compiled with the **QMCALL** compiler directive. Use of this feature prevents a malicious application that uses the QMClient API calling other subroutines that might not perform adequate security validation to protect the system.

The example program fragments below instantiate an object based on the CONNECTION class, passing in two argument strings.

**C**

```c
int QMCreateObject(char * Class, short int ArgCount, char * Arg1...)

Obj = QMCreateObject("CONNECTION", 2, IpAddr, Port);
```

Arguments must be passed as strings. Any argument values updated by the CREATE.OBJECT subroutine can be retrieved using the **QMGetArg()** function.

**VB.Net**

```vbnet
QMCreateObject(ByVal Class as String, ByVal ArgCount as Short, Optional ByVal Arg1 as String, ...) as Integer

Obj = QMCreateObject("CONNECTION", 2, IpAddr, Port)
```

Arguments must be passed as strings. Any argument values updated by the CREATE.OBJECT subroutine can be retrieved using the **QMGetArg()** function.

**QMBasic Class Module**

```vbnet
CreateObject(Obj, Class, ArgList...)
```
Objno = obj->CreateObject(Obj, "CONNECTION", IpAddr, Port)

**Java**

```java
int CreateObject(char * Class, int ArgCount, char * Arg1...)
Obj = qm.CreateObject("CONNECTION", 2, IpAddr, Port);
```

Arguments must be passed as strings. Any argument values updated by the CREATE.OBJECT subroutine can be retrieved using the `QMGetArg()` function.

**Python**

```python
CreateObject(Class, ArgCount, Arg1...)
Obj = qm.CreateObject("CONNECTION", 2, IpAddr, Port)
```

Arguments must be passed as strings. Any argument values updated by the CREATE.OBJECT subroutine can be retrieved using the `QMGetArg()` function.

**See also:**
- `QMDestroyObject`
- `QMGet()`
- `QMSet`
7.17 **QMDcount()**

The `QMDcount()` function counts delimited items in a string. It is analogous to the QMBasic `DCOUNT()` function.

The function arguments are:

- `Src` is the string to be processed
- `Delim` is the delimiter separating substrings. This may be more than one character.

The `QMDcount()` function is usually used to count fields, values or subvalues in a dynamic array but can be used to count elements in any string that is separated by any substring.

This function is evaluated on the client system and does not require a server connection to be open.

The examples below return the number of comma separated items in the Address variable.

**C**

```c
int QMDcount(char * Src, char * Delim)
X = QMDcount(Address, "," );
```

**VB.Net**

```vbnet
QMDcount(ByVal Src as String, ByVal Delim as String) as Integer
X = QMDcount(Address, "," )
```

**QMBasic Class Module**

This function is not supported by the QMClient class module as it is a QMBasic function.

**Java**

```java
int Dcount(char * Src, String Delim)
X = qm.Dcount(Address, "," );
```

**Python**

```python
Dcount(Src, Delim)
X = qm.Dcount(Address, "," )
```
7.18 QMDecrypt()

The QMDecrypt() function decrypts data that has been encrypted for secure storage or transmission. It is analogous to the QMBasic DECRYPT() function.

The function arguments are:

- **Data** is the encrypted data to be decrypted.
- **KeyString** is the encryption key to be used.

The QMDecrypt() function applies the AES 128 bit encryption algorithm to the supplied data and returns the decrypted text. The key string may be up to 64 characters in length and may contain any character. It is automatically transformed into a form that is usable by the AES algorithm. For optimum data security, the key should be a minimum of 16 characters though shorter keys can be used.

The encrypted data is structured so that it can never contain characters from the C0 control group (characters 0 to 31) or the mark characters. As a result of this operation, the encrypted data is slightly longer than the decrypted data.

The QMDecrypt() function decrypts data that was encrypted with either QMEncrypt() or QMEncryptx().

This function is evaluated on the client system and does not require a server connection to be open.

The examples below decrypt the data in variable Var using an encryption key of "MyKey".

**C**

```c
char * QMDecrypt(char * Data, char * KeyString)
X = QMDecrypt(Var, "MyKey");
```

Note that the function returns a pointer to a newly allocated memory area that must be released using QMFree() when no longer needed.

**VB.Net**

```vbnet
QMDecrypt(ByVal Data as String, ByVal KeyString as String) as String
X = QMDecrypt(Var, "MyKey")
```

**QMBasic Class Module**

This function is not supported by the QMClient class module as it is a QMBasic function.

**Java**

This function is currently not supported by the QMClient Java API.

**Python**

This function is currently not supported by the QMClient Python API.

See also: QMEncrypt()
7.19 QMDel()

The QMDel() function deletes a field, value or subvalue from a dynamic array. It is analogous to the QMBasic DELETE() function.

The function arguments are:

- **Src** is the dynamic array to be processed.
- **Fno** is the number of the field to be deleted. If less than 1, 1 is assumed.
- **Vno** is the number of the value to be deleted. If less than 1, the entire field is deleted.
- **Svno** is the number of the subvalue to be deleted. If less than 1, the entire value is deleted.

The QMDel() function returns a new dynamic array with the given field, value or subvalue deleted. If the required item is not found, the original string is returned unchanged.

The examples below return a copy of the dynamic array Rec with field 1, value Pos removed.

**C**

```c
char * QMDel(char * Src, int Fno, int Vno, int Svno)
NewRec = QMDel(Rec, 1, Pos, 0);
```

Note that the function returns a pointer to a newly allocated memory area which must be released using QMFree() when no longer needed. A statement such as

```
Rec = QMDel(Rec, 1, Pos, 0);
```

will overwrite the Rec pointer without releasing the previously allocated memory.

**VB.Net**

```vbnet
QMDel(ByVal Src as String, ByVal Fno as Integer, ByVal Vno as Integer, ByVal Svno as Integer) as String
NewRec = QMDel(Rec, 1, Pos, 0)
```

**QMBasic Class Module**

This function is not supported by the QMClient class module as it is a QMBasic function.

**Java**

```java
String Del(String Src, int Fno, int Vno, int Svno)
NewRec = qm.Del(Rec, 1, Pos, 0);
```

**Python**

```python
Del(Src, Fno, Vno, Svno)
NewRec = qm.Del(Rec, 1, Pos, 0)
```
7.20 QMDelete()

The QMDelete() function deletes a record from a file. It is analogous to the QMBasic DELETE statement.

The function arguments are:

- `FileNo` is the file number returned by a previous QMOpen() call.
- `Id` is the id of the record to be deleted.

The QMDelete function deletes the named record from the file open as `FileNo`. No error occurs if the record does not exist.

Applications should always obtain an update lock for a record before deleting it. The lock is released by this function.

The program fragment in the examples below opens the CLIENTS file, deletes the record identified by `ClientNo`, and then closes the file. Note the use of a record update lock to comply with recommended programming rules.

**C**

```c
void QMDelete(int FileNo, char * Id)

fClients = QMOpen("CLIENTS");
QMRecordlock(fClients, ClientNo, TRUE, TRUE);
QMDelete(fClients, ClientNo);
QMClose(fClients);
```

**VB.Net**

```vbnet
QMDelete ByVal FileNo as Integer, ByVal Id as String

fClients = QMOpen("CLIENTS")
QMRecordlock(fClients, ClientNo, True, True)
QMDelete(fClients, ClientNo)
QMClose(fClients)
```

**QMBasic Class Module**

```qbasic
Delete(FileNo, Id)

fClients = session->Open("CLIENTS")
session->Recordlock(fClients, ClientNo, @true, @true)
session->Delete(fClients, ClientNo)
session->Close(fClients)
```

**Java**

```java
Delete(int FileNo, String Id)

fClients = qm.Open("CLIENTS");
qm.Recordlock(fClients, ClientNo, true, true);
qm.Delete(fClients, ClientNo);
qm.Close(fClients);
```

**Python**

```python
```

---
Delete(FileName, Id)

fClients = qm.Open("CLIENTS")
qm.Recordlock(fClients, ClientNo, True, True);
qm.Delete(fClients, ClientNo);
qm.Close(fClients);
7.21 QMDeleteu()

The QMDeleteu() function deletes a record from a file, retaining the record lock. It is analogous to the QMBasic DELETEU statement.

The function arguments are:

- FileNo: the file number returned by a previous QMOpen() call.
- Id: the id of the record to be deleted.

The QMDeleteu() function deletes the named record from the file open as FileNo. No error occurs if the record does not exist.

Applications should always obtain an update lock for a record before deleting it. The lock is not released by this function.

The program fragment in the examples below opens the CLIENTS file and deletes the record identified by ClientNo. Note the use of a record update lock to comply with recommended programming rules.

**C**

```c
void QMDeleteu(int FileNo, char * Id)
{
    fClients = QMOpen("CLIENTS");
    QMRecordlock(fClients, ClientNo, TRUE, TRUE);
    QMDeleteu(fClients, ClientNo);
}
```

**VB.Net**

```vbnet
QMDeleteu ByVal FileNo as Integer, ByVal Id as String
{
    fClients = QMOpen("CLIENTS")
    QMRecordlock(fClients, ClientNo, True, True)
    QMDeleteu(fClients, ClientNo)
}
```

**QMBasic Class Module**

```qmbasic
Deleteu(FileNo, Id)
{
    fClients = session->Open("CLIENTS")
    session->Recordlock(fClients, ClientNo, @true, @true)
    session->Deleteu(fClients, ClientNo)
}
```

**Java**

```java
Deleteu(int FileNo, String Id)
{
    fClients = qm.Open("CLIENTS");
    qm.Recordlock(fClients, ClientNo, true, true);
    qm.Deleteu(fClients, ClientNo);
}
```

**Python**

```python
Deleteu(FileNo, Id)
{
    fClients = qm.Open("CLIENTS");
    qm.Recordlock(fClients, ClientNo, True, True);
    qm.Deleteu(fClients, ClientNo);
}
```
7.22 QMDestroyObject()

The QMDestroyObject() function discards an instantiated object.

The function takes one argument, Objno as the numeric identifier returned by a previous use of QMCreateObject().

The DESTROY.OBJECT subroutine will be executed if it exists.

The example program fragments below destroy an object previously instantiated with the reference value in ObjNo.

**C**

```c
void QMDestroyObject(int Objno)
QMDestroyObject(ObjNo);
```

**VB.Net**

```vbnet
QMDestroyObject(ByVal Objno as Integer)
QMDestroyObject(ObjNo)
```

**QMBasic Class Module**

```qmbasic
DestroyObject(objno)
session->DestroyObject(ObjNo)
```

**Java**

```java
DestroyObject(int Objno)
qm.DestroyObject(ObjNo);
```

**Python**

```python
DestroyObject(Objno)
qm.DestroyObject(ObjNo)
```

See also: QMCreateObject(), QMGet(), QMSet
7.23 QMDisconnect()

The `QMDisconnect()` function terminates a QMClient session previously established using `QMConnect()`, `QMConnectPool()` or `QMConnectLocal()`.

```c
void QMDisconnect(void)
QMDisconnect();
```

```vbnet
QMDisconnect()
QMDisconnect
```

**QMBasic Class Module**

```java
Disconnect
session->Disconnect
```

```python
Disconnect()
qm.Disconnect();
```

See also: `QMConnect()`, `QMConnectLocal()`, `QMConnectPool()`, `QMDisconnectAll`
7.24 QMDisconnectAll()

The QMDisconnectAll() function terminates all QMClient sessions from a client process.

C

    void QMDisconnectAll(void)
    QMDisconnectAll();

VB.Net

    QMDisconnectAll()
    QMDisconnectAll

QMBasic Class Module

    This function is not supported by the QMClient class module.

Java

    This function is not supported by the QMClient Java API.

Python

    This function is not supported by the QMClient Python API.

See also:
QMConnect(), QMConnectLocal(), QMConnectPool(), QMDisconnect()
7.25 QMEncrypt(), QMEncryptx()

The QMEncrypt() function decrypts data for secure storage or transmission. It is analogous to the QMBasic ENCRYPT() function.

The QMEncrypt() function applies the AES 128 bit encryption algorithm to the supplied data and returns the encrypted text. The key string may be up to 64 characters in length and may contain any character. It is automatically transformed into a form that is usable by the AES algorithm. For optimum data security, the key should be a minimum of 16 characters though shorter keys can be used.

The encrypted data is structured so that it can never contain characters from the C0 control group (characters 0 to 31) or the mark characters. As a result of this operation, the encrypted data is slightly longer than the resultant decrypted data.

The QMEncryptx() function is similar but uses a random initialisation vector for improved security. This results in a small increase in the size of the encrypted data.

These functions are evaluated on the client system and do not require a server connection to be open.

The examples below encrypt the data in variable Var using an encryption key of "MyKey".

C

```c
char * QMEncrypt(char * Data, char * KeyString)
X = QMEncrypt(Var, "MyKey");
```

Note that the function returns a pointer to a newly allocated memory area that must be released using QMFree() when no longer needed.

VB.Net

```vbnet
QMEncrypt(ByVal Data as String, ByVal KeyString as String) as String
X = QMEncrypt(Var, "MyKey")
```

QMBasic Class Module

This function is not supported by the QMClient class module as it is a QMBasic function.

Java

This function is currently not supported by the QMClient Java API.

Python

This function is currently not supported by the QMClient Python API.

See also: QMDecrypt()
7.26 QMEndCommand()

The QMEndCommand() function aborts a command executed on the server that is requesting input.

This function may only be used when an immediately preceding QMExecute() or QMRespond() function has returned a status of SV_PROMPT.

The example program fragments below execute a command on the server. If this command prompts for input, it is terminated.

C

```c
void QMEndCommand(void, int * Err);
Response = QMExecute(Command, &Err);
if (Err == SV_PROMPT) QMEndCommand();
```

The Response variable points to dynamically allocated memory that must be released with QMFree() when no longer needed.

VB.Net

```vbnet
QMEndCommand
Response = QMExecute(Command, Err)
if Err = SV_PROMPT Then QMEndCommand
```

QMBasic Class Module

```qbasic
EndCommand
Response = session->Execute(Command, Err)
if Err = SV_PROMPT Then session->EndCommand()
```

Java

```java
String EndCommand(String Command)
Response = qm.Execute(Command);
if (qm.ServerError == SV_PROMPT) qm.EndCommand();
```

Python

```python
EndCommand()
Response, Err = qm.Execute(Command)
if Err == SV_PROMPT: qm.EndCommand()
```

See also: QMExecute(), QMRespond()
7.27 QMEnterPackage()

The QMEnterPackage() function attempts to enter a licensed software package. It is analogous to
the QMBasic ENTER PACKAGE() function.

The function takes one argument, the package name and behaves differently according to the
package type:

- For a **registered package**, it confirms that the named package is installed, has not expired, and
  that the user limit has not been reached. If all is ok, the user is recorded as being active in the
  package and the function returns 1. If an error occurs, the function returns 0 and the error
  code can be found using the QMStatus() function.

- For a **private package**, the function returns the number of users in the package, including the
  one attempting to enter. An error will return zero and the error code can be found using the
  QMStatus() function. It is the application's responsibility to apply any concurrent user limits
  or other constraints.

The examples below attempt to enter a registered package named SALESINFO, reporting an error if
this fails.

**C**

```c
int QMEnterPackage(name)
if (QMEnterPackage("SALESINFO") == 0) {
    printf("Software is not licensed on this server");
    exit(0);
}
```

**VB.Net**

```vbnet
QMEnterPackage(name As String) As Integer
If QMEnterPackage("SALESINFO") = 0 Then
    MsgBox "Software is not licensed on this server",
    MsgBoxStyle.Exclamation
End If
```

**QMBasic Class Module**

```vbnet
EnterPackage(name)
if session->EnterPackage("SALESINFO") = 0 then
    display "Software is not licensed on this server"
    stop
end
```

**Java**

```java
int EnterPackage(String name)
if (qm.EnterPackage("SALESINFO") == 0) {
    System.out.println("Software is not licensed on this server");
}
```

**Python**

```python
```
EnterPackage(name)

if qm.EnterPackage("SALESINFO") == 0:
    print("Software is not licensed on this server")

See also:
Package licensing, QMExitPackage()
7.28 QMError()

The **QMError()** function returns extended error message text.

```c
char * QMError(void)
{
    if (!QMConnect(Host, -1, UserName, Password, Account))
    {
        printf("Failed to connect - %s\n", QMError());
    }
}
```

Note that in the C API, this function returns a pointer to a statically allocated error message buffer. The calling program must not attempt to free this memory. The standard **QMError()** and wide character **QMErrorW()** functions share a single buffer. The pointer returned by one of these functions is invalid after calling the other. There is a variant on this function, **QMErrorAllocW()** that returns the error message in a dynamically allocated wide character buffer that must be released with **QMFree()** when no longer needed.

```vb.net
QMError() as String
If Not QMConnect(Host, -1, UserName, Password, Account) Then   MsgBox(QMError(), MsgBoxStyle.Exclamation, "Failed to connect")End If
```

The **QMBasic Class Module**

```vb.net
Error

session = object("!qmclient")
if not(session->Connect(Host, -1, UserName, Password, Account)) then   stop "Failed to connect - " : session->Error
end
```

The QMError function is implemented as a public property variable in this version of the QMClient API.

```java
Error

if (!qm.Connect(Host, -1, UserName, Password, Account))
{
    System.out.println("Failed to connect - " + qm.Error);
}
```

The QMError function is implemented as a public property variable in this version of the QMClient API.

```python
Error()
if not qm.Connect(Host, -1, UserName, Password, Account):
    print("Failed to connect - ", QMError())
```
The QMEvaluate() function evaluates a dictionary data defining item. It is analogous to the QMBasic ITYPE() function. The QMEvalConv() function is identical except that it applies any conversion code specified in the dictionary item.

The function arguments are:

- **FileNo** is the file number associated with the dictionary of the file, as returned by QMOpen().
- **Name** is the name of the dictionary I-type record. This may be multi-valued as described below.
- **Data** is the data from the record for which the I-type is to be evaluated. This data will be copied into @RECORD on the server.
- **Id** is record id of the record for which the I-type is to be evaluated. This data will be copied into @ID on the server. If the I-type expression does not reference @ID, this is ignored and is best passed as a null string.

Because the QMEvaluate() function must read the dictionary record every time that it is evaluated, applications that execute this function in a loop for many records using the same dictionary item are likely to perform best if record caching is enabled (See the RECCACHE configuration parameter).

The QMEvalConv() function applies conversion codes to the evaluated item but not format codes.

C

```c
char * QMEvaluate(int FileNo, char * Name, char * Data, char * Id)
DataFno = QMOpen("SALES");
DictFno = QMOpen("DICT SALES");
DataRec = QMRead(DataFno, OrderNo, &Err);
OrderValue = QMEvaluate(DictFno, "SALE.VALUE", DataRec, "");
```

The dataRec and OrderValue variables are returned as pointers to dynamically allocated memory that must be released with QMFree() when no longer needed.
QMEvaluate(ByVal FileNo as Integer, ByVal Name as String, ByVal Data as String, ByVal Id as String) as String

DataFno = QMOpen("SALES")
DictFno = QMOpen("DICT SALES")
DataRec = QMRead(DataFno, OrderNo, Err)
OrderValue = QMEvaluate(DictFno, "SALE.VALUE", DataRec, ")"

QMBasic Class Module

Evaluate(FileNo, Name, Data, Id)

DataFno = session->Open("SALES")
DictFno = session->Open("DICT SALES")
DataRec = session->Read(DataFno, OrderNo, Err)
OrderValue = session->Evaluate(DictFno, "SALE.VALUE", DataRec, "")

Java

String Evaluate(int FileNo, String Name, String Data, String Id)

DataFno = qm.Open("SALES");
DictFno = qm.Open("DICT SALES");
DataRec = qm.Read(DataFno, OrderNo, &Err);
OrderValue = qm.Evaluate(DictFno, "SALE.VALUE", DataRec, ");

Python

Evaluate(FileNo, Name, Data, Id)

DataFno = qm.Open("SALES")
DictFno = qm.Open("DICT SALES")
DataRec, Err = qm.Read(DataFno, OrderNo)
OrderValue = qm.Evaluate(DictFno, "SALE.VALUE", DataRec, ")"
The **QMExecute()** function executes a command on the server.

The function arguments are:

- **Cmd**: is the command to be executed.
- **Err**: is an integer variable to receive status information. Some variants of the QMClient API do not have this argument.

The output from the command is returned as a text string. If the command does not produce any output, the C API returns a null pointer.

If the command completes without requesting input, the **Err** variable is set to **SV_OK**.

If the command requests input, any output up to that point is returned and the **Err** variable is set to **SV_PROMPT**. The client may respond to this using the **QMRespond()** function or abort the command using the **QMEndCommand()** function.

On completion of the command, **QMStatus()** will return the value of @SYSTEM.RETURN.CODE.

The executed command may perform most functions of the QM database. Specific restrictions are:

- Input may be requested from the client using the QMBasic **INPUT** and **INPUT@** statements. Use of the **KEYIN()** function is not allowed.
- Testing for input using the QMBasic **KEYREADY()** function or the **INPUT -1** syntax will not show input waiting.
- The length parameter and special options such as HIDDEN and UPCASE in an **INPUT** statement will be ignored if present. Only simple input is supported via **QMExecute()**.
- Execution of a further command from within the executed command may not behave correctly.

The example program fragments below run a program named MYPROG on the server. This program repeatedly outputs data using CRT, DISPLAY or PRINT and prompts for user input. Each output/response pair is handled by the client loop until the program terminates.

### C

```c
char * QMExecute(char * Cmd, int * Err)
S = QMExecute("RUN MYPROG", &Err);
while(Err == SV_PROMPT)
{
    ...Process returned value and get new Response...
    QMFree(S);
    S = QMRespond(Response, &Err);
}...
```

Note that the returned string pointers from both QMExecute() and QMRespond() point to dynamically allocated memory that must be released using **QMFree()** when no longer required.

### VB.Net

```vbnet
QMExecute(ByVal Cmd as String, ByRef Err as Integer) as String
```
```plaintext
S = QMExecute("RUN MYPROG", Err)
While Err = SV_PROMPT
    ...Process returned value and get new Response...
    S = QMRespond(Response, Err)
End While
...Process final response...

<table>
<thead>
<tr>
<th>QMBasic Class Module</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execute</strong>(<em>Cmd</em>, <em>Err</em>)</td>
</tr>
<tr>
<td>S = session-&gt;Execute(&quot;RUN MYPROG&quot;, Err)</td>
</tr>
<tr>
<td>loop</td>
</tr>
<tr>
<td>while Err = SV$PROMPT</td>
</tr>
<tr>
<td>...Process returned value and get new Response...</td>
</tr>
<tr>
<td>S = session-&gt;Respond(&quot;RUN MYPROG&quot;, Err)</td>
</tr>
<tr>
<td>repeat</td>
</tr>
<tr>
<td>...Process final response...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>String Execute(<strong>String Cmd</strong>)</strong></td>
</tr>
<tr>
<td>S = QMExecute(&quot;RUN MYPROG&quot;);</td>
</tr>
<tr>
<td>while(qm.ServerError == SV_PROMPT)</td>
</tr>
<tr>
<td>{}</td>
</tr>
<tr>
<td>...Process returned value and get new Response...</td>
</tr>
<tr>
<td>QMFree(S);</td>
</tr>
<tr>
<td>S = QMRespond(Response);</td>
</tr>
<tr>
<td>}</td>
</tr>
<tr>
<td>...Process final response...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Execute</strong>(<em>Cmd</em>)</td>
</tr>
<tr>
<td>S, Err = qm.Execute(&quot;RUN MYPROG&quot;)</td>
</tr>
<tr>
<td>while(qm.ServerError == SV$PROMPT):</td>
</tr>
<tr>
<td>...Process returned value and get new Response...</td>
</tr>
<tr>
<td>S, Err = qm.Respond(Response)</td>
</tr>
<tr>
<td>...Process final response...</td>
</tr>
</tbody>
</table>

See also:
QMEndCommand, QMRespond()
7.31 QMExitPackage()

The `QMExitPackage()` function exits from a licensed software package. It is analogous to the QMBasic `EXIT.PACKAGE()` function.

The function takes one argument, the package name. It returns 1 if it is successful, 0 if it fails. In the event of failure, the `QMStatus()` function can be used to determine the cause.

The examples below attempt to exit from a registered package named SALESINFO, reporting an error if this fails.

C

```c
int QMExitPackage(name)
    if (QMExitPackage("SALESINFO") == 0)
        {    printf("Error exiting from sales package");
    }
```

VB.Net

```vbnet
QMExitPackage(name As String) As Integer
    if QMExitPackage("SALESINFO") = 0 Then
        MsgBox("Error exiting from sales package", MsgBoxStyle.Exclamation)
    End If
```

QMBasic Class Module

```qmbasic
ExitPackage(name)
    if session->ExitPackage("SALESINFO")= 0 then
        display "Error exiting from sales package"
    end
```

Java

```java
int ExitPackage(String name)
    if (qm.ExitPackage("SALESINFO") == 0) {
        System.out.println("Error exiting from sales package");
    }
```

Python

```python
ExitPackage(name)
    if qm.ExitPackage("SALESINFO") == 0:
        print("Error exiting from sales package")
```

See also:
- Package licensing, `QMEnterPackage()`
7.32 QMExtract()

The QMExtract() function extracts a field, value or subvalue from a dynamic array. It is analogous to the QMBasic field extraction operator or the EXTRACT() function.

The function arguments are:

- **Src** is the dynamic array to be processed
- **Fno** is the number of the field to be extracted. If less than 1, 1 is assumed
- **Vno** is the number of the value to be extracted. If less than 1, the entire field is extracted.
- **Svno** is the number of the subvalue to be extracted. If less than 1, the entire value is extracted.

The QMExtract() function returns the given field, value or subvalue from the source string. If the required item is not found, a null string is returned.

This function is evaluated on the client system and does not require a server connection to be open.

The example program fragments below read the record identified by ClientNo from the file open as fClients and then extracts field 4 from it. A real program should test the status from the read to determine if the action was successful.

**C**

```c
char * QMExtract(char * src, int Fno, int Vno, int Svno)
Rec = QMRead(fClients, ClientNo, Err);
CliAddr = QMExtract(Rec, 4, 0, 0);
```

Note that variables Rec and CliAddr point to dynamically allocated memory areas that must be released using QMFree() when no longer needed.

**VB.Net**

```vbnet
QMExtract(ByVal src as String, ByVal Fno as Integer, ByVal Vno as Integer, ByVal Svno as Integer) as String
Rec = QMRead(fClients, ClientNo, Err)
CliAddr = QMExtract(Rec, 4, 0, 0)
```

**QMBasic Class Module**

This function is not supported by the QMClient class module as it is a QMBasic function.

**Java**

```java
String Extract(String src, int Fno, int Vno, int Svno)
Rec = qm.Read(fClients, ClientNo, Err);
CliAddr = qm.Extract(Rec, 4, 0, 0);
```

**Python**

```python
Extract(src, Fno, Vno, Svno)
```
Rec, Err = qm.Read(fClients, ClientNo)
CliAddr = qm.Extract(Rec, 4, 0, 0)
7.33 QMField()

The QMField() function extracts one or more components of a delimited string. It is analogous to the QMBasic FIELD() function.

The function arguments are:

- **Src** is the string to be processed.
- **Delimiter** is the single character delimiter separating components of the string.
- **Start** is the number from one of the first component of **Src** to be returned.
- **Occurrences** is the number of delimited of components of **Src** to be returned. If less than one, one component is returned.

The QMField() function returns the specified substring components of **Src**.

This function is evaluated on the client system and does not require a server connection to be open.

The example program fragments below take a variable Name holding a person’s first and last names separated by a space and returns just their first name.

**C**

```c
char * QMField(char * Src, char * Delimiter, int Start, int Occurrences)
```

```c
FirstName = QMField(Name, " ", 1, 1);
```

The returned pointer references a dynamically allocated memory area that must be released using QMFree() when no longer needed.

**VB.Net**

```vbnet
QMField(ByVal Src as String, ByVal Delimiter as String, ByVal Start as Integer, Optional ByVal Occurrences as Integer) as String
```

```vbnet
FirstName = QMField(Name, " ", 1)
```

The **Occurrences** argument defaults to 1 if omitted.

**QMBasic Class Module**

This function is not supported by the QMClient class module as it is a QMBasic function.

**Java**

```java
String Field(String Src, char Delimiter, int Start, int Occurrences)
```

```java
FirstName = qm.Field(Name, " ", 1, 1);
```

**Python**

```python
Field(Src, Delimiter, Start, Occurrences)
```

```python
FirstName = qm.Field(Name, " ", 1, 1)
```
7.34  **QMFree()**

The **QMFree()** function releases memory returned by other functions. It is only used with the C API library.

The function takes one argument, *Addr*, as the memory address of the area to be released.

    void QMFree(void * Addr)

Many of the C API functions return dynamically allocated memory. The **QMFree()** function allows a program to release this. The C runtime library free() function should not be used as it may not be compatible with the memory allocator used within the API functions.

To simplify application design where errors may lead to null pointers being returned by QMClient functions, passing a null pointer into **QMFree()** has no effect.

**Example**

```c
Rec = QMReadu(fClients, ClientNo, TRUE, Err);
Rec2 = QMReplace(Rec, 1, Pos, 0, NewData);
QMWrite(fClients, ClientNo, Rec2);
QMFree(Rec);
QMFree(Rec2);
```

The above program fragment reads a record with an update lock, modifies it and then writes it back to the file. A real program should test the Err status from the read operation to determine if it was successful. The **QMFree()** is used to release the dynamic memory allocated to store Rec and Rec2.
The QMGet() function gets data from an instantiated object on the server system.

The function arguments are:

- **Objno** is the numeric identifier returned by a previous use of QMCreateObject().
- **Name** is the name of a public variable or public function within the object.
- **ArgCt** is a count of arguments to be passed into the object. This argument is not present in some variants of the QMClient API.
- **Arg1...** is a series of argument values as strings.

The example program fragments below retrieve the value of the State property in an object previously instantiated with the reference value in Objno. In this example, there are no arguments passed into the object.

**C**

```c
char * QMGet(short int Object, char * Name, short int ArgCt, ...)  
State = QMGet(Objno, "State", 0);
```

The arguments follow the **ArgCt** and must be passed as strings. Any argument values updated by the object can be retrieved using the QMGetArg() function. The returned pointer references a dynamic memory area that must be released with QMFree() when no longer needed.

**VB.Net**

```vbnet
QMGet(ByVal Object as Short, ByVal Name as String, ArgCt as Short, Optional ByRef Arg1 as String, ...) as String  
State = QMGet(Objno, 'State', 0)
```

The arguments follow the **ArgCt** and must be passed as strings. Any argument values updated by the object can be retrieved using the QMGetArg() function.

**QMBasic Class Module**

```qmbasic
Get(Objno, Name {, Arg1, ...})  
State = session->Get(Objno, 'State')
```

**Java**

```java
String QMGet(int Object, String Name, int ArgCt, String Arg1...)  
State = qm.Get(Objno, "State", 0);
```

The arguments follow the **ArgCt** and must be passed as strings. Any argument values updated by the object can be retrieved using the QMGetArg() function.

**Python**

```python
Get(Object, Name, ArgCt, Arg1...)  
State = qm.Get(Objno, "State", 0)
```
The arguments follow the ArgCt and must be passed as strings. Any argument values updated by
the object can be retrieved using the QMGetArg() function.

See also:
QMCreateObject(), QMDestroyObject, QMSet
7.36 **QMGetArg()**

The **QMGetArg()** function retrieves the value of argument variables to **QMCall()**, **QMCallx()**, **QMCreateObject()**, **QMGet()** or **QMSet()** updated by the server.

The function takes a single argument, *ArgNo*, as the argument number (from 1) to be retrieved.

An argument may be retrieved multiple times. Argument data remains accessible until the next use of any of these functions.

In applications that open multiple simultaneous server connections, the argument data is maintained on a per-session basis, reflecting the most recent use of these functions in the currently selected session.

The example program fragments below instantiate a QMBasic object catalogued as MONITOR.CLS on the server. The CREATE.OBJECT subroutine of this object updates its only argument to a status value that is retrieved by the client. This argument is output only and is passed as a null string in the QMCreateObject() function.

C

```c
char * QMGetArg(short int ArgNo)
Obj = QMCreateObject("MONITOR.CLS", 1, "");
State = QMGetArg(1);
```

The returned pointer references a dynamic memory area that must be released with **QMFree()** when no longer needed.

VB.Net

```vbnet
QMGetArg(ByVal ArgNo as Short) as String
Obj = QMCreateObject("MONITOR.CLS", 1, "")
State = QMGetArg(1)
```

QMBasic Class Module

```java
GetArg(ArgNo)
Obj = session->CreateObject("MONITOR.CLS", 1, "")
State = session->GetArg(1)
```

Java

```java
String GetArg(int ArgNo)
Obj = qm.CreateObject("MONITOR.CLS", 1, "");
State = qm.GetArg(1);
```

Python

```python
GetArg(ArgNo)
Obj = qm.CreateObject("MONITOR.CLS", 1, "")
State = qm.GetArg(1)
```

See also: **QMCallx**
7.37  QMGetSession()

The **QMGetSession()** function returns the internal session number associated with the currently selected QMClient session.

In the C or VB.Net versions of QMClient, a single client may open multiple server connections, each identified by a session number. The **QMConnect()**, **QMConnectLocal()** and **QMConnectPool()** functions select an available session number to use for the newly created session which can be retrieved using **QMGetSession()**. All subsequent QMClient function calls relate to this session until an alternative session is selected using **QMSetSession()** or a new connection is opened.

The **QMGetSession()** function has no equivalent in the other variants of the QMClient API as each session is managed as a separate instantiation of the object.

The examples below save the current session number into a variable named OldSesNum, open a new connection (which will create a new session), perform some processing, disconnect the new session and then use **QMSetSession** to revert to the original session.

**C**

```c
int QMGetSession(void)
{
    OldSesNum = QMGetSession();
    if (QMConnectLocal("SALES"))
    {
        ...processing...
        QMDisconnect();
    }
    SetSession(OldSesNum);
}
```

**VB.Net**

```vbnet
    QMGetSession() as Integer
    OldSesNum = QMGetSession()
    If QMConnectLocal("SALES") Then
        ...processing...
        QMDisconnect()
    End If
    SetSession(OldSesNum)
```

See also: **QMSetSession()**
7.38 QMGetVar()

The QMGetVar() function returns the value of an @-variable from the server.

The function takes a single argument, Name, as the name of the @-variable to be returned. The leading @ symbol may be omitted. If the named variable does not exist, a null string is returned.

The examples below return the value of the @SELECTED variable. Note that although this is a numeric value, the returned data is always a character string.

**C**

```c
char * QMGetVar(char * Name)
SelectedCount = QMGetVar("@SELECTED");
```

The returned pointer references a dynamically allocated memory area that must be released using QMFree() when no longer needed.

**VB.Net**

```vbnet
QMGetVar(ByVal Name as String) as String
SelectedCount = QMGetVar("@SELECTED")
```

**QMBasic Class Module**

```qmbasic
GetVar(Name)
SelectedCount = session->GetVar("@SELECTED")
```

**Java**

```java
String GetVar(String Name)
SelectedCount = qm.GetVar("@SELECTED");
```

**Python**

```python
GetVar(Name)
SelectedCount = qm.GetVar("@SELECTED")
```
The **QMIConv()** function converts data to internal form using a conversion code. It is analogous to the QMBasic **ICONV()** function. The **QMIConvs()** function is similar but works on each element of a dynamic array in turn.

The function arguments are:

- **Data** is the data to be converted.
- **Code** is the conversion code to be applied.

The **QMIConv()** function returns a string representing the converted data. The **QMStatus()** function returns zero after a successful conversion. A non-zero value indicates an error as described for **ICONV()**.

This function is evaluated on the server system to ensure access to all conversion codes and hence requires that a server connection is open.

The examples below convert an external form date in the ExtDate variable to an internal day number, storing this in IntDate. Note that the returned value from **QMIConv()** is always a string.

**C**

```c
char * QMIConv(char * Data, char * Code)
IntDate = QMIConv(ExtDate, "D2DMY");
```

The returned pointer references a dynamically allocated memory area that must be released using **QMFree()** when no longer needed.

**VB.Net**

```vbnet
QMIConv(ByVal Data as String, ByVal Code as String) as String
IntDate = QMIConv(ExtDate, "D2DMY")
```

**QMBasic Class Module**

```qmbasic
Iconv(Data, Code)
IntDate = Session->Iconv(ExtDate, "D2DMY");
```

**Java**

```java
String QMIConv(String Data, String Code)
IntDate = qm.IConv(ExtDate, "D2DMY");
```

**Python**

```python
IConv(Data, Code)
IntDate = qm.IConv(ExtDate, "D2DMY")
```

See also: **QMOConv()**
7.40 QMIndices()

The **QMIndices()** function retrieves information about alternate key indices. It is analogous to the QMBasic **INDICES()** function.

The function arguments are:

- \( Fno \) is the file number returned by a previous use of **QMOpen()**.
- \( Index.Name \) is the name of the index to be queried. If given as a null string, a list of all index names is returned.

The **QMIndices()** function has two modes of operation. If \( Index.Name \) is a null string, the function returns a field mark delimited list of index names for the given file. This is equivalent to use of the QMBasic **INDICES()** function with one argument.

If \( Index.Name \) is not a null string, details for the named index are returned in the same form as for the QMBasic **INDICES()** function with two arguments.

The examples below retrieve a list of index names for the file opened as fClients.

- **C**
  ```c
  char * QMIndices(int Fno, char * Index.Name)
  IndexNames = QMIndices(fClients, "");
  ...processing...
  QMFree(IndexNames);
  
  The returned pointer references a dynamically allocated memory area that must be released using **QMFree()** when no longer needed.
  ```

- **VB.Net**
  ```vbnet
  QMIndices(ByVal Fno as Integer, ByVal Index.Name as String) as String
  IndexNames = QMIndices(fClients, ")
  ```

- **QMBasic Class Module**
  ```vbnet
  Indices(Fno, Index.Name)
  IndexNames = session->Indices(fClients, ")
  ```

- **Java**
  ```java
  String Indices(int Fno, String Index.Name)
  IndexNames = qm.Indices(fClients, ")
  ```

- **Python**
  ```python
  Indices(Fno, Index.Name)
  IndexNames = qm.Indices(fClients, ")
  ```
### 7.41 QMIns()

The `QMIns()` function inserts a field, value or subvalue in a dynamic array. It is analogous to the QMBasic `INSERT()` function.

The function arguments are:

- **Src** is the dynamic array to be processed
- **Fno** is the number of the field to be inserted. If less than 1, 1 is assumed.
- **Vno** is the number of the value to be inserted. If less than 1, an entire field is inserted.
- **Svno** is the number of the subvalue to be inserted. If less than 1, an entire value is inserted.
- **NewData** is the new data to form the new dynamic array element.

The `QMIns()` function returns a new dynamic array with the specified field, value or subvalue inserted.

This function is evaluated on the client system and does not require a server connection to be open.

The examples below read a record with an update lock, use `QMIns()` to modify it and then write it back to the file. A real program should test the error status from the read operation to determine if it was successful.

#### C

```c
char * QMIns(char * Src, int Fno, int Vno, int Svno, char * NewData)
Rec = QMReadu(fClients, ClientNo, TRUE, Err);
Rec2 = QMIns(Rec, 1, Pos, 0, NewData);
QMWrite(fClients, ClientNo, Rec2);
QMFree(Rec);
QMFree(Rec2);
```

Note that the function returns a pointer to a newly allocated memory area which must be released using `QMFree()` when no longer needed. A statement such as

```c
Rec = QMIns(Rec, 1, Pos, 0, NewData);
```

will overwrite the Rec pointer without releasing the previously allocated memory.

#### VB.Net

```vbnet
QMIns(ByVal Src as String, ByVal Fno as Integer, ByVal Vno as Integer, ByVal Svno as Integer, ByVal NewData as String) as String
Rec = QMReadu(fClients, ClientNo, True, Err)
Rec = QMIns(Rec, 1, Pos, 0, NewData)
QMWrite fClients, ClientNo, Rec
```

#### QMBasic Class Module

This function is not supported by the QMClient class module as it is a QMBasic function.
String Ins(String Src, int Fno, int Vno, int Svno, String NewData)

Rec = qm.Readu(fClients, ClientNo, true);
Rec2 = qm.Ins(Rec, 1, Pos, 0, NewData);
qm.Write(fClients, ClientNo, Rec2);

Python

Ins(Src, Fno, Vno, Svno, NewData)

Rec, Err = qm.Readu(fClients, ClientNo, true)
Rec2 = qm.Ins(Rec, 1, Pos, 0, NewData)
qm.Write(fClients, ClientNo, Rec2)
7.42 QMIsECS()

The QMIsECS() function determines whether a QMClient session is connected to an ECS mode server.

The examples below test whether a QMClient connection is open in ECS mode and, if not, display an error message.

C

```c
int QMIsECS()
if (!QMIsECS())
{
    printf("Not ECS mode");
    exit(0);
}
```

For non-ECS mode connections, use of the wide character variants of QMClient API functions may fail if characters outside the 8-bit set are referenced.

VB.Net

```vbnet
QMIsECS() as Boolean
If Not QMIsECS() Then
    MsgBox("Not ECS mode", MsgBoxStyle.Exclamation)
End If
```

QMBasic Class Module

```qbasic
IsECS
if not(session->IsECS) then
    stop "Not ECS mode"
end
```

This function is implemented as a property value in the QMBasic class module API.

Java

```java
IsECS
if (!qm.IsECS)
{
    System.out.println("Not ECS mode");
}
```

Python

```python
IsECS()
if not qm.IsEcs():
    print("Not ECS mode")
```
7.43 QMLocate()

The QMLocate() function searches a dynamic array for a field, value or subvalue matching a given string. It is analogous to the LOCATE statement.

The function arguments are:

- **Item** is the item to find.
- **DynArray** is the dynamic array to be processed.
- **Fno** is the number of the field at which the search is to begin. If less than 1, 1 is assumed.
- **Vno** is the number of the value at which the search is to begin. If less than 1, the function searches for a field containing Item.
- **Svno** is the number of the subvalue at which the search is to begin. If less than 1, the function searches for a value containing Item.
- **Pos** is an integer variable to receive the position information. Not present in some variants of the QMClient API.
- **Order** identifies the sort method to the applied. This may be:
  - AL  Ascending, left aligned
  - AR  Ascending, right aligned
  - DL  Descending, left aligned
  - DR  Descending, right aligned
  - If omitted or a null string, no sort order is applied.

The QMLocate() function searches a dynamic array at one of three levels:

- If **Vno** is less than 1, the function searches the dynamic array for a field matching Item, starting at the field position given by **Fno**.
- If **Vno** is given but **Svno** is less than 1, the function searches field **Fno** of the dynamic array for a value matching Item, starting at the value position given by **Vno**.
- If **Vno** and **Svno** are given, the function searches field **Fno**, value **Vno** of the dynamic array for a subvalue matching Item, starting at the value position given by **Svno**.

The Order argument determines the sorting system to be applied during the search:

- If **Order** is a null string, no sort rules are applied. The function scans all applicable dynamic array elements for a match against Item. The **Pos** variable will be returned as the position at which the item was found. If the item is not found, **Pos** will be returned as the position at which a new element could be appended.
- If the first character of **Order** is A, an ascending sort is applied. If the first character of **Order** is D, a descending sort is applied. In either case, the search terminates if an entry is found that would be beyond the correct position for Item. In this case, if the item is not found, **Pos** will be returned as the position at which to insert Item to maintain the correct sort order.
• If the second character of Order is L, a left aligned comparison is performed. Each entry of the dynamic array is compared with Item character by character from the left until a difference is found.

• If the second character of Order is R, a right aligned comparison is performed. If the two items being compared are of different lengths, spaces are added to the front of the shorter item before comparison.

The returned positional information indicates where the item was found or, if not found, where it should be inserted.

This function is evaluated on the client system and does not require a server connection to be open.

The examples below search field 4 of record stored as variable OrderRec for a value that contains the same data as the OrderRec array. If found, the value position is returned in variable Pos and Found is set to true. If not found, Pos will indicate the position at which the item could be inserted to maintain correct sorted order as implied by the AL sort code and Found will be set to False. Note that the QMLocate() function follows the default "Information style" syntax of QM where the field, value and subvalue positions indicate the point at which the search is to start. Thus, these examples specify a value position of 1 to search the entire field.

**C**

```c
int QMLocate(char * Item, char * DynArray, int Fno, int Vno, int Svno, int * Pos, char * Order)
```

```c
Found = QMLocate(PartNo, OrderRec, 4, 1, 0, &Pos, "AL");
```

The QMLocate() function returns True if the item is found, False if it is not found.

**VB.Net**

```vbnet
QMLocate(ByVal Item as String, ByVal DynArray as String, ByVal Fno as Integer, ByVal Vno as Integer, ByVal Svno as Integer, ByRef Pos as Integer, ByVal Order as String) as Boolean
```

```vbnet
Found = QMLocate(PartNo, OrderRec, 4, 1, 0, Pos, "AL")
```

The QMLocate() function returns True if the item is found, False if it is not found.

**QMBasic Class Module**

This function is not supported by the QMClient class module as it is a QMBasic function.

**Java**

```java
int Locate(String Item, String DynArray, int Fno, int Vno, int Svno, String Order)
```

```java
Pos = qm.Locate(PartNo, OrderRec, 4, 1, 0, "AL");
```

The Locate() function returns the position at which the item was found. If not found, the returned value is the negative of the position at which it should be inserted.

**Python**

```python
Locate(Item, DynArray, Fno, Vno, Svno, Order)
```

```python
Found, Pos = qm.Locate(PartNo, OrderRec, 4, 1, 0, "AL");
```
The **Locate()** function returns two values as a list. The first is True if the item is found, False if it is not found. The second value is the positional information.
7.44  QMLogto()

The QMLogto() function moves to an alternative account. It is analogous to the LOGTO command.

The function takes a single argument, Account, as the name of the QM account to be accessed.

If successful, the function returns True. If unsuccessful, the function returns False and the QMError() function can be used to retrieve a text error message identifying the cause of the failure.

If the VOC of the current account contains an executable item named ON.LOGTO, usually a paragraph, this will be executed before moving to the new account.

If the VOC of the new account contains an executable item named LOGIN, this will be executed on arrival in the new account.

A QMClient session can be recognised within these paragraphs by testing the value of @TTY which will be "vbsrvr" for QMClient.

The examples below attempt to move to the account identified Account. If unsuccessful, it displays an error message including the text returned by QMError().

C

```c
int QMLogto(char * Account)
if (!QMLogto(Account))
{
    printf("Cannot move to new account - %s\n", QMError());
    exit;
}
```

VB.Net

```vbnet
QMLogto(ByVal Account as String) as Short
If Not QMLogto(Account) Then
    MsgBox(QMError(), MsgBoxStyle.Exclamation, "Cannot move to new account")
End If
```

QMBasic Class Module

```qmbasic
Logto(Account)
if not(session->Logto(Account)) then
    stop "Cannot move to new account - " : QMError()
end
```

Java

```java
boolean Logto(String Account)
if (!qm.Logto(Account))
{
    System.out.println("Cannot move to new account - " + qm.Error);
    exit;
}
```
Logto(Account)

if not qm.Logto(Account):
    print("Cannot move to new account - ", qm.Error)
### 7.45 QMMarkMapping

The **QMMarkMapping()** function enables or disables mark mapping for a directory file.

The function arguments are:

- **FileNo** is the file number returned by a previous **QMOpen()** call.
- **State** is non-zero to enable mark mapping, zero to disable.

See the QMBasic **MARK.MAPPING** statement for more details.

The examples below open the PDF file and disable mark mapping. A real program should include error handling in case the file cannot be opened.

**C**

```c
void QMMarkMapping(int FileNo, int State)
{
    fPDF = QMOpen("PDF");
    QMMarkMapping(fPDF, FALSE);
}
```

**VB.Net**

```vbnet
QMMarkMapping ByVal FileNo as Integer, ByVal State as Integer
{
    fPDF = QMOpen("PDF")
    QMMarkMapping(fPDF, False)
}
```

**QMBasic Class Module**

```qbasic
MarkMapping(FileNo, State)
{
    fPDF = session->Open('PDF')
    session->MarkMapping(fPDF, @false)
}
```

**Java**

```java
MarkMapping (int FileNo, boolean State)
{
    fPDF = qm.Open("PDF");
    qm.MarkMapping(fPDF, false);
}
```

**Python**

```python
MarkMapping (FileNo, State)
{
    fPDF = qm.Open("PDF")
    qm.MarkMapping(fPDF, False)
}
```
The QMMatch() function matches a string against a pattern template. It is analogous to the QMBasic MATCHES operator.

The function arguments are:

- **Src**: is the string to be processed.
- **Pattern**: is the pattern template to be used.

The QMMatch() function tests whether **Src** matches the **Pattern** template consisting of one or more concatenated items from the following list:

- `...` Zero or more characters of any type
- `0X` Zero or more characters of any type
- `nX` Exactly *n* characters of any type
- `n-mX` Between *n* and *m* characters of any type
- `0A` Zero or more alphabetic characters
- `nA` Exactly *n* alphabetic characters
- `n-mA` Between *n* and *m* alphabetic characters
- `0N` Zero or more numeric characters
- `nN` Exactly *n* numeric characters
- `n-mN` Between *n* and *m* numeric characters
- "string" A literal string which must match exactly. Either single or double quotation marks may be used.

The values *n* and *m* are integers with any number of digits. *m* must be greater than or equal to *n*.

The 0A, nA, 0N, nN and "string" patterns may be preceded by a tilde (~) to invert the match condition. For example, ~4N matches four non-numeric characters such as ABCD (not a string which is not four numeric characters such as 12C4).

A null string matches patterns ..., 0A, 0X, 0N, their inverses (~0A, etc) and "".

The 0X and n-mX patterns match against as few characters as necessary before control passes to the next pattern. For example, the string ABC123DEF matched against the pattern 0X2N0X matches the pattern components as ABC, 12 and 3DEF.

The 0N, n-mN, 0A, and n-mA patterns match against as many characters as possible. For example, the string ABC123DEF matched against the pattern 0X2-3N0X matches the pattern components as ABC, 123 and DEF.

The pattern string may contain alternative templates separated by value marks. The QMMatch() function tries each template in turn until one is a successful match against the string.

This function is evaluated on the client system and does not require a server connection to be open.

The examples below test whether the contents of PartCode is formed from between two and four letters followed by three digits, returning OK as a True/False value.
OK = QMMatch(PartCode, "2-4A3N");

**VB.Net**

```vbnet
QMMatch(ByVal Src as String, ByVal Pattern as String) as Boolean
OK = QMMatch(PartCode, "2-4A3N")
```

**QMBasic Class Module**

This function is not supported by the QMClient class module as it is a QMBasic function.

**Java**

This function is currently not supported by the Java version of the QMClient API.

**Python**

```python
Match(Src, Pattern)
OK = qm.Match(PartCode, "2-4A3N")
```
7.47 QMMatchfield()

The QMMatchfield() function matches a character string against a pattern template and extracts the part corresponding to a specified pattern component. It is analogous to the QMBasic MATCHFIELD() function.

The function arguments are:

- **Src** is the string to be processed.
- **Pattern** is the pattern template to be used.
- **Component** is the pattern template component number for which the corresponding part of **Src** is to be returned.

The QMMatchfield() function matches **Src** against the **Pattern** template as described for the QMMatch() function. If the string matches, the portion corresponding to the specified **Component** is returned. If the string does not match the pattern, a null string is returned.

The examples below return the alphabetic prefix of a value stored in PartCode if it is formed from between two and four letters followed three digits.

This function is evaluated on the client system and does not require a server connection to be open.

**C**

```c
char * QMMatchfield(char * Src, char * Pattern, int Component)
Prefix = QMMatchfield(PartCode, "2-4A3N", 1)
```

The returned pointer references a dynamically allocated memory area that must be released using QMFree() when no longer needed.

**VB.Net**

```vbnet
QMMatchfield(ByVal Src as String, ByVal Pattern as String, ByVal Component as Integer)
Prefix = QMMatchfield(PartCode, "2-4A3N", 1)
```

**QMBasic Class Module**

This function is not supported by the QMClient class module as it is a QMBasic function.

**Java**

This function is currently not supported by the Java version of the QMClient API.

**Python**

```python
Matchfield(Src, Pattern)
Prefix = qm.Matchfield(PartCode, "2-4A3N", 1)
```
7.48 **QMNextPartial()**

The **QMNextPartial()** function returns the next part of a select list built using **QMSelectPartial()**. The function takes one argument, *ListNo* as the select list number (0 to 10).

The **QMNextPartial()** function provides an optimised way to process select lists within a QMClient session. It returns the next part of a list constructed using **QMSelectPartial()**. A null string is returned when the list is exhausted.

**QMNextPartial()** can also be used to return items from a select list built on the server by a program, by execution of a command, or by use of **QMSelect()**. It is essentially the same as a series of **QMReadNext()** operations that merge the ids into a single string before returning it to the server.

See [Select lists in QMClient sessions](#) for a description of the alternative ways to handle select list with QMClient.

The example program fragment below builds select list 1 and uses it to process records from the file open as fClients.

**C**

```c
char * QMNextPartial(int ListNo)
{
    list = QMSelectPartial(fClients, 1);
    while(list != NULL) {
        n = QMDcount(list, FM);
        for(i = 1; i <= n; i++) {
            id = QMExtract(list, i, 0, 0);
            ...processing...
            QMFree(id);
        }
        QMFree(list);
        list = QMNextPartial(1);
    }

    //Note that the returned pointer references a dynamically allocated memory area that must be released using QMFree() when no longer needed as shown in this example.
}
```

**VB.Net**

```vbnet
QMNextPartial(ByVal ListNo as Integer) as String
List = QMSelectPartial(fClients, 1)
While List <> ""
    N = QMDcount(List, FM)
    For I = 1 To N
        Id = QMExtract(List, I, 0, 0)
        ...processing...
    Next I
    List = QMNextPartial(1)
End While
```

**QMBasic Class Module**

```qbasic
NextPartial(ListNo)
```
list = session->SelectPartial(fClients, 1)
loop
while list ≠ ""
   for each id in list
      ...processing...
      next id
      list = session->NextPartial(1)
repeat

Java

String NextPartial(int ListNo)
list = qm.SelectPartial(fClients, 1);
while qm.ServerError == SV_OK) {
    n = qm.Dcount(list, FM);
    for(i = 1; i <= n; i++)
    {
        id = qm.Extract(list, i, 0, 0);
        ...processing...
    }
    list = qm.NextPartial(1);
}

Python

NextPartial(ListNo)
list = qm.SelectPartial(fClients, 1)
while qm.ServerError == SV_OK:
    n = qm.Dcount(list, FM)
    for i in range(1, n+1):
        id = qm.Extract(list, i, 0, 0)
        ...processing...
    list = qm.NextPartial(1)
The `QMOConv()` function converts data to external form using a conversion code. It is analogous to the QMBasic `OCONV()` function. The `QMOConvs()` function is similar but works on each element of a dynamic array in turn.

The function arguments are:

- **Data** is the data to be converted.
- **Code** is the conversion code to be applied.

The `QMOConv()` function returns a string representing the converted data. The `QMStatus()` function returns zero after a successful conversion. A non-zero value indicates an error as described for `OCONV()`.

This function is evaluated on the server system to ensure access to all conversion codes and hence requires that a server connection is open.

The examples below convert an internal form date in the `IntDate` variable to external form, storing this in `ExtDate`. Note that the returned value from `QMOConv()` is always a string.

**C**

```c
char * QMOConv(char * Data, char * Code)
ExtDate = QMOConv(IntDate, "D2DMY");
```

The returned pointer references a dynamically allocated memory area that must be released using `QMFree()` when no longer needed.

**VB.Net**

```vbnet
QMOConv(ByVal Data as String, ByVal Code as String) as String
ExtDate = QMOConv(IntDate, "D2DMY")
```

**QMBasic Class Module**

```qmbasic
Oconv(Data, Code)
ExtDate = session->Oconv(IntDate, "D2DMY");
```

**Java**

```java
String QMOConv(String Data, String Code)
ExtDate = qm.OConv(IntDate, "D2DMY");
```

**Python**

```python
OConv(Data, Code)
ExtDate = qm.OConv(IntDate, "D2DMY")
```

See also: `QMIConv()`
The **QMOpen()** function opens a file. It is analogous to the QMBasic **OPEN** statement.

The function takes a single argument, **FileName**, as the name of the file to be opened. This must correspond to an F or Q-type entry in the VOC of the QM account in which the server is running.

The returned integer value is the file number which must be used in all subsequent operations against this file. If the file cannot be opened, the function returns zero. The **QMStatus()** function can be used to retrieve the error cause.

To open a dictionary, the **FileName** argument should commence with "DICT" and a single space separating this prefix from the file name, for example:

```
DictNo = QMOpen("DICT READERS")
```

There is no practical limit to the number of files that can be open at one time.

The program fragment below opens the CLIENTS file, reads the record identified by ClientNo, and then closes the file. A real program should test the error status from the read to determine if the action was successful.

**C**

```c
int QMOpen(char * FileName)
{
fClients = QMOpen("CLIENTS");
Rec = QMRead(fClients, ClientNo, Err);
QMClose(fClients);

Note that this example leaves variable Rec pointing to a dynamically allocated memory area that must be released using **QMFree()** when no longer needed.
```

**VB.Net**

```vbnet
QMOpen(ByVal FileName as String) as Integer
{
fClients = QMOpen("CLIENTS")
Rec = QMRead(fClients, ClientNo, Err)
QMClose(fClients)
```

**QMBasic Class Module**

```qmbasic
Open(FileName)
{
fClients = session->Open("CLIENTS")
Rec = session->Read(fClients, ClientNo, Err)
session->Close(fClients)
```

**Java**

```java
int Open(String FileName)
{
fClients = qm.Open("CLIENTS");
Rec = qm.Read(fClients, ClientNo);
qmClose(fClients);
```

**Python**

```python
Open(FileName)
```
fClients = qm.Open("CLIENTS")
Rec, Err = qm.Read(fClients, ClientNo)
qmClose(fClients)
The **QMOpenSeq()** function opens a sequential file. It is analogous to the QMBasic **OPENSEQ** statement.

The function arguments are:

- **FileName** is the VOC name of the file to be opened.
- **Id** is the record name within *FileName* of the item to be opened. If given as a null string, *FileName* must be the pathname of the item to be opened.
- **Modes** is an additive value specifying the manner in which the file is to be opened:
  - 0x0002: Return status value ER$LOCKED if locked by another user. If this mode is not present, the operation waits for the lock to be cleared.
  - 0x0010: Open in read-only mode.
  - 0x0100: Open in append mode.
  - 0x0200: Open in overwrite mode, replacing existing content.
  - 0x0400: Unbuffered mode.
  - 0x1000: Open in shared mode.

The returned integer value is the file number which must be used in all subsequent operations against this file. If the file cannot be opened, the function returns zero. The **QMStatus()** function can be used to retrieve the error cause.

If the item does not exist, **QMOpenSeq()** returns a valid file number and the item will be created by subsequent use of any function that writes to the file.

There is no practical limit to the number of files that can be open at one time.

The file should be closed using **QMClose()**.

The examples below open a sequential file in the directory identified by VOC item REPORTS using the value in DATE as the record name. The *modes* value is given in hexadecimal form as the additive value that combines return of an error code if the item is locked and selection of overwrite mode. This could alternatively have been written as the decimal value 514.

### C

```c
int QMOpenSeq(char * FileName, char * Id, int Modes)
{
    fReport = QMOpenSeq("REPORTS", DATE, 0x0202);
}
```

### VB.Net

```vbnet
QMOpenSeq(ByVal FileName as String, ByVal Id as String, ByVal Modes as Integer) as Integer
{
    fReport = QMOpenSeq("REPORTS", DATE, &H0202)
}
```

### QMBasic Class Module

```qmbasic
OpenSeq(FileName, Id, Modes)
{
    fReport = session->OpenSeq("REPORTS", DATE, 0x0202)
}
```
int OpenSeq(String FileName, String Id, int Modes)

fReport = qm.OpenSeq("REPORTS", DATE, 0x0202);

Python

OpenSeq(FileName, Id, Modes)

fReport = qm.OpenSeq("REPORTS", DATE, 0x0202)

See also:
QMReadBlk(), QMReadSeq(), QMSeek(), QMWeofSeq(), QMWriteBlk(), QMWriteSeq()
7.52 QMPoolIdle()

The QMPoolIdle() function terminates a QMClient connection previously established using QMConnectPool(), moving the server process into an idle state waiting for a new connection.

All files opened with QMOpen() are closed and all objects instantiated with QMCreateObject() are destroyed.

If QMPoolIdle() is used with a connection established using QMConnect() or QMConnectLocal(), the server session terminates in the same way as if the QMDisconnect() function had been used.

**C**

```c
QMPoolIdle()
QMPoolIdle();
```

**VB.Net**

```vbnet
QMPoolIdle()
QMPoolIdle()
```

**QMBasic Class Module**

```qbasic
PoolIdle()
session->PoolIdle()
```

**Java**

```java
PoolIdle()
qm.PoolIdle();
```

**Python**

```python
PoolIdle()
qm.PoolIdle()
```

**See also:**
Connection pooling, QMConnect(), QMConnectLocal(), QMConnectPool(), QMDisconnectAll
7.53 QMRead()

The QMRead() function reads a record without locking. It is analogous to the QMBasic READ statement.

The function arguments are:

- **FileNo** is the file number returned by a previous QMOpen() call.
- **Id** is the id of the record to be read.
- **Err** is an integer variable to receive status information. This argument is not present in all variants of the QMClient API.

The QMRead() function requests the server to return the record with key Id from the file opened as FileNo.

If successful, the function returns the record as a dynamic array string and the Err variable is set to SV_OK.

If the record cannot be found, the function returns a null string and the Err variable is set to SV_ELSE. The QMStatus() function can be used to retrieve the error number.

Conditions that would normally cause a QMBasic program to abort or to take the ON ERROR clause of a READ statement return a null string and the Err variable is set to SV_ON_ERROR. The QMStatus() function can be used to retrieve the error number.

The example program fragment below opens the CLIENTS file, reads the record identified by ClientNo, and then closes the file. A real program should test the Err status from the read to determine if the action was successful.

```c
char * QMRead(int FileNo, char * Id, int * Err)

fClients = QMOpen("CLIENTS");
Rec = QMRead(fClients, ClientNo, &Err);
QMClose(fClients);

The returned pointer references a dynamically allocated memory area that must be released using QMFree() when no longer needed. Note that attempting to read a non-existent record returns a pointer to a null string, not a NULL pointer.

VB.Net

QMRead(ByVal FileNo as Integer, ByVal Id as String, ByRef Err as Integer) as String

fClients = QMOpen("CLIENTS")
Rec = QMRead(fClients, ClientNo, Err)
QMClose(fClients)

QMBasic Class Module

Read(FileNo, Id, Err)

fClients = session->Open("CLIENTS")
Rec = session->Read(fClients, ClientNo, Err)
session->Close(fClients)

**Java**

```java
String Read(int FileNo, String Id)
    fClients = qm.Open("CLIENTS");
    Rec = qm.Read(fClients, ClientNo);
    qm.Close(fClients);
```

If successful, the ServerError property will be SV_OK. Other values indicate an error.

**Python**

```python
Read(FileNo, Id)
    fClients = qm.Open("CLIENTS")
    Rec, Err = qm.Read(fClients, ClientNo)
    qm.Close(fClients)
```

The function returns two values as a list; the record data and the error status. If successful, the error status is SV_OK.
7.54 QMReadBlk()

The QMReadBlk() function reads a specified number of bytes from a sequential file. It is analogous to the QMBasic READBLK statement.

The function arguments are:

- **FileNo** is the file number returned by a previous QMOpenSeq() call.
- **Bytes** is the number of bytes to be read.
- **ActualBytes** is an integer variable to receive the actual byte count which may be less than **Bytes** if the end of the data is reached. Note that this argument is not present in all variants of the QMClient API.

The QMReadBlk() function requests the server to return the given number of bytes from the file opened as **FileNo**.

If successful, the function returns the data and the **ActualBytes** variable is set to the number of bytes read.

If the data cannot be read, typically because the end of the file has been reached, the function returns a null string and the **ActualBytes** variable is set to zero. The QMStatus() function can be used to retrieve the error number.

Note that when using the QMClient API in its wide character (Unicode) mode, QMReadBlk is a byte level operation and returns a string in which each character will represent one byte from the file and hence is always in the range char(0) to char(255).

The example program fragments below open a sequential item named STOCK in the IMPORT file, read 100 bytes of data and then closes the file. A real program should test the error status from the read to determine if the action was successful.

### C

```c
char * QMReadBlk(int FileNo, int Bytes, int * ActualBytes)

fData = QMOpenSeq("IMPORT", "STOCK", 0);
Rec = QMReadBlk(fData, 100, &Err);
QMFree(Rec);
QMClose(fData);
```

The returned pointer references a dynamically allocated memory area that must be released using QMFree() when no longer needed. Note that attempting to read beyond the end of the file returns a pointer to a null string, not a NULL pointer.

### VB.Net

```vbnet
QMReadBlk(ByVal FileNo as Integer, ByVal Bytes as Integer, ByRef ActualBytes) as String

fData = QMOpenSeq("IMPORT", "STOCK", 0)
Rec = QMReadBlk(fData, 100, Err)
QMClose(fData)
```

### QMBasic Class Module

```qbasic
ReadBlk(FileNo, Bytes)
```
fData = session->OpenSeq("IMPORT", "STOCK", 0)
Rec = session->ReadBlk(fData, 100)
session->Close(fData)

Java

String ReadBlk(int FileNo, int Bytes)

fData = qm.OpenSeq("IMPORT", "STOCK", 0);
Rec = qm.ReadBlk(fData, 100);
mq.Close(fData);

If successful, the ServerError property will be set to SV$OK. Other values indicate an error.

Python

ReadBlk(FileNo, Bytes)

fData = qm.OpenSeq("IMPORT", "STOCK", 0)
Rec, Err = qm.ReadBlk(fData, 100)
mq.Close(fData)

See also: QMOpenSeq(), QMReadSeq(), QMSeek(), QMWeofSeq(), QMWriteBlk(), QMWriteSeq()
7.55 QMReadl()

The `QMReadl()` function reads a record with a shareable read lock. It is analogous to the QMBasic `READL` statement.

The function arguments are:

- `FileNo` is the file number returned by a previous `QMOpen()` call.
- `Id` is the id of the record to be read.
- `Wait` is a Boolean value indicating the action to be taken if the record is currently locked by another user:
  - `True` wait for the record to become available
  - `False` return an error code of `SV_LOCKED`
- `Err` is an integer variable to receive status information. This argument is not present in all variants of the QMClient API.

The `QMReadl()` function requests the server to return the record with key `Id` from the file opened as `FileNo`. A shareable read lock is applied to the record. Any number of users may hold a shareable read lock on the same record at one time but, while any user has a shareable read lock, no other user can establish an update lock or a file lock.

If the action is blocked by a lock held by another user, the function returns a null string and the `Err` variable is set to `SV_LOCKED`. The `QMStatus()` function can be used to retrieve the user number of the process holding the lock.

If successful, the function returns the record as a dynamic array string and the `Err` variable is set to `SV_OK`. The record is locked by the server process.

If the record cannot be found, the function returns a null string and the `Err` variable is set to `SV_ELSE`. The `QMStatus()` function can be used to retrieve the error number. The record is locked by the server process. If the lock is not required, it should be released using the `QMRelease()` function.

Conditions that would normally cause a QMBasic program to abort or to take the ON ERROR clause of a `READ` statement return a null string and the `Errno` variable is set to `SV_ON_ERROR`. The `QMStatus()` function can be used to retrieve the error number.

The example program fragments below open the CLIENTS file and reads two client records. By using `QMReadl()` instead of `QMRead()`, there is no possibility that the first record has been updated by the time the second record is read. The program therefore sees a guaranteed consistent state of the data. A real program should test the error status from the read operations to determine if they were successful.

```c
char * QMReadl(int FileNo, char * Id, int Wait, int * Err)

fClients = QMOpen("CLIENTS");
Rec1 = QMReadl(fClients, ClientNo, TRUE, &Err);
Rec2 = QMReadl(fClients, OtherClient, TRUE, &Err);
```
The returned pointer references a dynamically allocated memory area that must be released using `QMFree()` when no longer needed. Note that attempting to read a non-existent record returns a pointer to a null string, not a NULL pointer.

**VB.Net**

```vbnet
QMReadl(ByVal FileNo as Integer, ByVal Id as String, ByVal Wait as Integer, ByRef Err as Integer) as String
fClients = QMOpen("CLIENTS")
Rec1 = QMReadl(fClients, ClientNo, True, Err)
Rec2 = QMReadl(fClients, OtherClient, True, Err)
```

**QMBasic Class Module**

```vbnet
Readl(FileNo, Id, Wait, Err)
```

```vbnet
fClients = session->Open("CLIENTS")
Rec1 = session->Readl(fClients, ClientNo, @true, Err)
Rec2 = session->Readl(fClients, OtherClient, @true, Err)
```

**Java**

```java
String QMReadl(int FileNo, String Id, boolean Wait)
```

```java
fClients = qm.Open("CLIENTS");
Rec1 = qm.Readl(fClients, ClientNo, true);
Rec2 = qm.Readl(fClients, OtherClient, true);
```

**Python**

```python
Readl(FileNo, Id, Wait)
```

```python
fClients = qm.Open("CLIENTS")
Rec1, Err = qm.Readl(fClients, ClientNo, True)
Rec2, Err = qm.Readl(fClients, OtherClient, True)
```
7.56 QMReadList()

The QMReadList() function reads a select list into a dynamic array in the client application. It is analogous to the QMBasic READLIST statement.

The function arguments are:

- **ListNo** is the number of the select list to be read in the range 0 to 10.
- **Err** receives an error value indicating the outcome of the request. This argument is not present in all variants of the QMClient API.

If the action is successful, the returned value contains a field mark delimited set of unprocessed entries from the given list. The original list is destroyed by this action.

See Select lists in QMClient sessions for a description of the alternative ways to handle select list with QMClient.

The example program fragments below build select list 1 as a list of all records in the file open as fClients and then transfer this to a dynamic array named List.

**C**

```c
char * QMReadList(int ListNo)
QMSelect(fClients, 1);
List = QMReadList(1);
```

The returned pointer references a dynamically allocated memory area that must be released using QMFree() when no longer needed.

If there is no data to read, the function returns NULL.

**VB.Net**

```vb
QMReadList(ByVal ListNo as Integer) as String
QMSelect fClients, 1
List = QMReadList(1)
```

If there is no data to read, the function returns Nothing.

**QMBasic Class Module**

```vb
ReadList(ListNo, Err)
session->Select(fClients, 1)
List = session->ReadList(1, Err)
```

If there is no data to read, the function returns an empty string.

**Java**

```java
String QMReadList(int ListNo)
qm.Select(fClients, 1);
List = qm.ReadList(1);
```

If successful, the ServerError property value is set to SV$OK. Other values indicate an error in which case the function returns an empty string.
Python

```python
ReadList(ListNo)

qm.Select(fClients, 1)
List = qm.ReadList(1)
```

The function returns the list data as a field mark delimited string. If there is no data to read, the function returns an empty string.
7.57 QMReadNext()

The QMReadNext() function retrieves the next entry from a select list. It is analogous to the QMBasic READNEXT statement.

The function arguments are:

- **ListNo** is the number of the select list to be processed in the range 0 to 10.
- **Err** receives an error value indicating the outcome of the request. This argument is not present in all variants of the QMClient API.

The QMReadNext() function retrieves the next entry from the select list identified by the ListNo argument.

If successful, the function returns the list entry and, in the Visual Basic and QMBasic APIs, Err is set to SV_OK.

See Select lists in QMClient sessions for a description of the alternative ways to handle select list with QMClient.

See also the QMReadList() function for a discussion of the relationship between QMReadNext() and QMReadList().

The example program fragments below builds select list 1 and use it to process records from the file open as fClients.

**C**

```c
char * QMReadNext(short int ListNo)

QMSelect (fClients, 1);  
while((Id = QMReadNext(1)) != NULL) {  
  if (Id == NULL) break;  
  Rec = QMRead(fClients, Id, Err);  
  if (Err == 0)  
    {  
      ...process record...  
      QMFree(Rec);  
    }  
  QMFree(Id);  
}
```

If successful, the function returns a pointer to a dynamically allocated memory area holding the item from the list. This memory must be released using QMFree() when no longer needed. If the function fails, the return value is NULL.

**VB.Net**

```vbnet
QMReadNext(ByVal ListNo as Short) as String

QMSelect fClients, 1
Do  
  Id = QMReadNext(1)  
  If Id Is Nothing Then Exit Do
```
Rec = QMRead(fClients, Id, Err)
If Err = 0 Then
    ...process record...
End If
Loop

If successful, the function returns the item from the list. If the function fails, the return value is Nothing.

QMBasic Class Module

ReadNext(ListNo, Err)

session->Select(fClients, 1)
repeat
    Id = session->ReadNext(1)
until Id = ""
    Rec = session->Read(fClients, Id, Err)
    if Err = 0 then
        ...process record...
    end
end
repeat

If successful, the function returns the item from the list and the Err argument variable is set to SV_OK. If the function fails, the return value is a null string and the Err argument variable will be non-zero.

Java

String ReadNext(int ListNo)

qm.Select(fClients, 1);
while(true)
{
    Id = qm.ReadNext(1);
    if (qm.ServerError != SV$OK) break;
    Rec = qm.Read(fClients, Id);
    if (qm.ServerError == SV$OK)
    { 
        ...process record...
    }
}

Python

ReadNext(ListNo)

qm.Select(fClients, 1)
while True:
    Id = qm.ReadNext(1)
    if Id = ": break
    Rec, Err = qm.Read(fClients, Id)
    ...process record...

If successful, the server error status will be set to SV_OK. If there are no further items to extract from the list, The return id will be a null string and the server error status will be set to SV_ELSE.
7.58 QMReadSeq()

The **QMReadSeq()** function reads a line of text from a sequential file. It is analogous to the QMBasic **READSEQ** statement.

The function arguments are:

- **FileNo** is the file number returned by a previous **QMOpenSeq()** call.
- **Err** is an integer variable to receive status information. This argument is not present in all variants of the QMClient API.

The **QMReadSeq()** function requests the server to return the next line of text from the file opened as **FileNo**.

If successful, the function returns the data, excluding the line terminator.

The example program fragments below open a sequential item named **STOCK** in the **IMPORT** file, read the first line of text and then closes the file. A real program should test the error status from the read to determine if the action was successful.

**C**

```c
char * QMReadSeq(int FileNo, int * Err)

fData = QMOpenSeq("IMPORT", "STOCK", 0);
Rec = QMReadSeq(fData, &Err);
QMFree(Rec);
QMClose(fData);
```

The returned pointer references a dynamically allocated memory area that must be released using **QMFree()** when no longer needed. Note that attempting to read beyond the end of the file returns a pointer to a null string, not a NULL pointer.

**VB.Net**

```vbnet
QMReadSeq(ByVal FileNo as Integer, ByRef Err as Integer) as String

fData = QMOpenSeq("IMPORT", "STOCK", 0)
Rec = QMReadSeq(fData, Err)
QMClose(fData)
```

**QMBasic Class Module**

```qmbasic
ReadSeq(FileNo, Err)

fData = session->OpenSeq("IMPORT", "STOCK", 0)
Rec = session->ReadSeq(fData, Err)
session->Close(fData)
```

**Java**

```java
String ReadSeq(int FileNo)

fData = qm.OpenSeq("IMPORT", "STOCK", 0);
Rec = qm.ReadSeq(fData, &Err);
```

If successful, the ServerError property will be set to SV$OK. Other values indicate an error.
Python

\textbf{ReadSeq}(\textit{FileNo})

\begin{verbatim}
  fData = qm.OpenSeq("IMPORT", "STOCK", 0)
  Rec, Err = qm.ReadSeq(fData)
\end{verbatim}

See also: \texttt{QMOpenSeq}, \texttt{QMReadBlk}, \texttt{QMSeek}, \texttt{QMWeofSeq}, \texttt{QMWriteBlk}, \texttt{QMWriteSeq}
7.59 QMReadu()

The QMReadu() function reads a record with an exclusive update lock. It is analogous to the QMBasic READU statement.

The function arguments are:

- **FileNo** is the file number returned by a previous QMOpen() call.
- **Id** is the id of the record to be read.
- **Wait** is a Boolean value indicating the action to be taken if the record is currently locked by another user:
  - True: wait for the record to become available
  - False: return an error code of SV_LOCKED
- **Err** is an integer variable to receive status information. This argument is not present in all variants of the QMClient API.

The QMReadu() function requests the server to return the record with key *Id* from the file opened as *FileNo*. An exclusive update lock is applied to the record. Only one user may hold an exclusive update lock on any one record at one time. An exclusive update lock also cannot be obtained if another user holds a shareable read lock on the record or a file lock on the entire file.

If the action is blocked by a lock held by another user, the function returns a null string and the *Err* variable is set to SV_LOCKED. The QMStatus() function can be used to retrieve the user number of the process holding the lock.

If successful, the function returns the record as a dynamic array string and the *Err* variable is set to SV_OK. The record is locked by the server process.

If the record cannot be found, the function returns a null string and the *Err* variable is set to SV_ELSE. The QMStatus() function can be used to retrieve the error number. The record is locked by the server process to allow creation of the record. If the lock is not required, it should be released using the QMRelease() function.

Conditions that would normally cause a QMBasic program to abort or to take the ON ERROR clause of a READ statement return a null string and the *Err* variable is set to SV_ON_ERROR. The QMStatus() function can be used to retrieve the error number.

The example program fragments below read a record with an update lock, modify it and then writes it back to the file. A real program should test the error status from the read operations to determine if they were successful.

```c
char * QMReadu(int FileNo, char * Id, int Wait, int * Err)
Rec = QMReadu(fClients, ClientNo, TRUE, &Err);
Rec2 = QMReplace(Rec, 1, Pos, 0, NewData);
QMWrite(fClients, ClientNo, Rec2);
QMFree(Rec);
QMFree(Rec2);
```
The returned pointer references a dynamically allocated memory area that must be released using \texttt{QMFree()} when no longer required. Note that attempting to read a non-existent record returns a pointer to a null string, not a NULL pointer.

\textbf{VB.Net}

\texttt{QMReadu(ByVal FileNo as Integer, ByVal Id as String, ByVal Wait as Integer, ByRef Err as Integer) as String}

\begin{verbatim}
Rec = QMReadu(fClients, ClientNo, True, Err)
Rec = QMReplace(Rec, 1, Pos, 0, NewData)
QMWrite fClients, ClientNo, Rec
\end{verbatim}

\textbf{QMBasic Class Module}

\texttt{Readu(FileNo, Id, Wait, Err)}

\begin{verbatim}
Rec = session->Readu(fClients, ClientNo, @true, Err)
Rec<1, Pos> = NewData
session->Write(fClients, ClientNo, Rec)
\end{verbatim}

\textbf{Java}

\texttt{String Readu(int FileNo, String Id, boolean Wait)}

\begin{verbatim}
Rec = qm.Readu(fClients, ClientNo, true);
Rec = qm.Replace(Rec, 1, Pos, 0, NewData);
qm.Write(fClients, ClientNo, Rec);
\end{verbatim}

\textbf{Python}

\texttt{Readu(FileNo, Id, Wait)}

\begin{verbatim}
Rec, Err = qm.Readu(fClients, ClientNo, True)
Rec = qm.Replace(Rec, 1, Pos, 0, NewData)
qm.Write(fClients, ClientNo, Rec)
\end{verbatim}
7.60 QMRecordlock()

The QMRecordlock() function locks a record. It is analogous to the QMBasic RECORDLOCKL and RECORDLOCKU statements.

The function arguments are:

- **FileNo** is the file number returned by a previous QMOpen() call.
- **Id** is the id of the record to be locked.
- **Update** is a Boolean value specifying the type of lock to be obtained:
  - True  Update lock
  - False Shareable read lock
- **Wait** is a Boolean value indicating the action to be taken if the record is currently locked by another user:
  - True  wait for the record to become available
  - False return an error code of SV_LOCKED

The QMRecordlock function can be used to obtain a lock on a record without reading the record.

The example program fragments below write a new record into the file opened as fClients. To comply with recommended locking rules, a record should never be written unless the program holds an update lock on it. Use of QMRecordlock() gets this lock before the write.

**C**

```c
void QMRecordlock(int FileNo, char * Id, int Update, int Wait)
QMRecordlock(fClients, ClientNo, TRUE, TRUE);
QMWrite(fClients, ClientNo, Rec2);
```

**VB.Net**

```vbnet
QMRecordlock ByVal FileNo as Integer, ByVal Id as String, ByVal Update as Integer, ByVal Wait as Integer
QMRecordlock(fClients, ClientNo, True, True)
QMWrite fClients, ClientNo, Rec
```

**QMBasic Class Module**

```vbnet
Recordlock(FileNo, Id, Update, Wait)
session->Recordlock(fClients, ClientNo, @true, @true)
session->Write(fClients, Id, Rec)
```

**Java**

```java
Recordlock(int FileNo, String Id, boolean Update, boolean Wait)
qm.Recordlock(fClients, ClientNo, true, true);
qmWrite(fClients, ClientNo, Rec);
```

**Python**

```python
Recordlock(FileNo, Id, Update, Wait)
```
qm.Recordlock(fClients, ClientNo, True, True)
qm.Write(fClients, Id, Rec)

See also:
QMRecordlocked()
7.61 QMRecordlocked()

The QMRecordlocked() function queries the lock on a record. It is analogous to the QMBasic RECORDLOCKED() function.

The function arguments are:

- **FileNo** is the file number returned by a previous QMOpen() call.
- **Id** is the id of the record to be locked.

The QMRecordlocked() function returns one of the following values:

<table>
<thead>
<tr>
<th>Value</th>
<th>Token</th>
<th>Lock state</th>
</tr>
</thead>
<tbody>
<tr>
<td>-3</td>
<td>LOCKSOTHER.FILELOCK</td>
<td>Another user holds a file lock</td>
</tr>
<tr>
<td>-2</td>
<td>LOCKSOTHER.READU</td>
<td>Another user holds an update lock</td>
</tr>
<tr>
<td>-1</td>
<td>LOCKSOTHER.READL</td>
<td>Another user holds a read lock</td>
</tr>
<tr>
<td>0</td>
<td>LOCKSNO.LOCK</td>
<td>The record is not locked</td>
</tr>
<tr>
<td>1</td>
<td>LOCKSMY.READL</td>
<td>This user holds a read lock</td>
</tr>
<tr>
<td>2</td>
<td>LOCKSMY.READU</td>
<td>This user holds an update lock</td>
</tr>
<tr>
<td>3</td>
<td>LOCKSMY.FILELOCK</td>
<td>This user holds a file lock</td>
</tr>
</tbody>
</table>

A record may be multiply locked in which case the QMRecordlocked() function reports only one of the current locks. File locks take precedence over read or update locks. If no file lock is set, read or update locks held by the process in which the QMRecordlocked() function is performed take precedence over locks held by other processes.

C

```c
int QMRecordlocked(int FileNo, char * Id)
{
    State = QMRecordlocked(fClients, ClientNo);
}
```

VB.Net

```vbnet
QMRecordlocked(ByVal FileNo as Integer, ByVal Id as String) As Short
    State = QMRecordlocked(fClients, ClientNo)
```

QMBasic Class Module

```vbnet
Recordlocked(FileNo, Id)
    State = session->Recordlock(fClients, ClientNo)
```

Java

```java
int Recordlocked(int FileNo, String Id)
    State = qm.Recordlocked(fClients, ClientNo);
```

Python

```python
Recordlocked(FileNo, Id)
```
State = qm.Recordlock(fClients, ClientNo)

See also:
RECORDLOCKED(), QMRecordlock
The **QMRelease** function releases a record lock. It is analogous to the QMBasic `RELEASE` statement.

The function arguments are:

- **FileNo** is the file number returned by a previous `QMOpen()` call. If zero, all locks are released.
- **Id** is the id of the record to be unlocked. If given as a null string, all locks in the file identified by *FileNo* are released.

The `QMRelease` function can be used to release a lock without writing or deleting the record. One common use of this function is to release the lock obtained by a call to `QMReadl()` or `QMReadu()` where the record was not found and the function returned the SV_ELSE status.

The example program fragments below attempt to read a record with an update lock. If the record is not found, the lock is released.

**C**

```c
void QMRelease(int FileNo, char * Id)
Rec = QMReadu(fClients, ClientNo, TRUE, Err);
if (Err == SV_ELSE)
    {Q
        QMRelease(fClients, ClientNo);
        QMFree(Rec);
    }
```

Note use of `QMFree()` to release the memory dynamically allocated by `QMReadu()` to return the record.

**VB.Net**

```vbnet
QMRelease ByVal FileNo as Integer, ByVal Id as String
Rec = QMReadu(fClients, ClientNo, True, Err)
If Err = SV_ELSE Then
    QMRelease(fClients, ClientNo)
End If
```

**QMBasic Class Module**

```vbnet
Release(FileNo, Id)
Rec = session->Readu(fClients, ClientNo, @true, Err)
if Err = SV_ELSE then
    session->Release(fClients, Id)
end
```

**Java**

```java
void Release(int FileNo, String Id)
Rec = qm.Readu(fClients, ClientNo, true);
if (qm.ServerError == SV_ELSE)
```
{  
    qm.Release(fClients, ClientNo);
  }

Python

**Release(FileNo, Id)**

Rec, Err = qm.Readu(fClients, ClientNo, true)  
if Err = SV_ELSE:  
    qm.Release(fClients, Id)
7.63 QMReplace()

The QMReplace() function replaces the content of a field, value or subvalue in a dynamic array. It is analogous to the QMBasic REPLACE() function.

The function arguments are:

- **Src** is the dynamic array to be processed
- **Fno** is the number of the field to be replaced. If zero, 1 is assumed. If negative, a new field is appended to the dynamic array.
- **Vno** is the number of the value to be replaced. If zero, the entire field is inserted. If negative, a new value is appended to the specified field.
- **Svno** is the number of the subvalue to be replaced. If zero, the entire value is inserted. If negative, a new subvalue is appended to the specified value.
- **NewData** is the new data to form the new dynamic array element.

The QMReplace() function returns a new dynamic array with the specified field, value or subvalue replaced.

This function is evaluated on the client system and does not require a server connection to be open.

The example program fragments below read a record with an update lock, use QMReplace() to modify it and then write it back to the file. A real program should test the error status from the read operations to determine if they were successful.

**C**

```c
char * QMReplace(char * Src, int Fno, int Vno, int Svno, char * NewData)
Rec = QMReadu(fClients, ClientNo, TRUE, Err);
Rec2 = QMReplace(Rec, 1, Pos, 0, NewData);
QMWrite(fClients, ClientNo, Rec2);
QMFree(Rec);
QMFree(Rec2);
```

The returned pointer references a dynamically allocated memory area that must be released using QMFree() when no longer needed. Note that a statement of the form

```c
rec = QMReplace(rec, 2, 0, 0, new_data)
```

will return a pointer to a newly allocated memory area, overwriting the *rec* pointer. The old memory is not freed by this call and it is therefore necessary to retain a pointer to the original *rec* string so that it can be freed later.

**VB.Net**

```vb
QMReplace(ByVal Src as String, ByVal Fno as Integer, ByVal Vno as Integer, ByVal Svno as Integer, ByVal NewData as String) as String
Rec = QMReadu(fClients, ClientNo, True, Err)
Rec = QMReplace(Rec, 1, Pos, 0, NewData)
QMWrite fClients, ClientNo, Rec
```

QMBasic Class Module
This function is not supported by the QMClient class module as it is a QMBasic function.

**Java**

```java
String Replace(String Src, int Fno, int Vno, int Syno, String NewData)
Rec = qm.Readu(fClients, ClientNo, true);
Rec = qm.Replace(Rec, 1, Pos, 0, NewData);
qm.Write(fClients, ClientNo, Rec2);
```

**Python**

```python
Replace(Src, Fno, Vno, Syno, NewData)
Rec, Err = qm.Readu(fClients, ClientNo, True)
Rec = qm.Replace(Rec, 1, Pos, 0, NewData)
qm.Write(fClients, ClientNo, Rec)
```
7.64 **QMRespond()**

The QMRespond() function responds to a request for input from a command executed on the server.

The function arguments are:

- **Response** is the response to be sent.
- **Err** is an integer variable to receive status information. This argument is not present in all variants of the QMClient API.

This function may only be used when an immediately preceding QMExecute() or QMRespond() function has returned a status of SV_PROMPT.

The QMRespond() function returns the given Response to the input request from the server command. Further output from this command is returned as a text string. If the command does not produce any output, the C API returns a null pointer.

If the command completes without requesting input, the Err variable is set to SV_OK.

If the command requests further input, any output up to that point is returned and the Err variable is set to SV_PROMPT. The client may respond to this using the QMRespond() function or abort the command using the QMEndCommand() function.

**C**

```c
char * QMRespond(char * Cmnd, int * Err)
S = QMExecute("RUN MYPROG", &Err);
while(Err == SV_PROMPT)
{  ...
    Process returned value and get new Response...
    QMFree(S);
    S = QMRespond(Response, &Err);
}  ...
Process final response...
```

Note that the returned string pointer from both QMExecute() and QMRespond() points to dynamically allocated memory that must be released using QMFree() when no longer required.

**VB.Net**

```vbnet
QMRespond(ByVal Cmnd as String, ByRef Err as Integer) as String
S = QMExecute("RUN MYPROG", Err)
While Err = SV_PROMPT
    ...
    Process returned value and get new Response...
    S = QMRespond(Response, Err)
End While
    ...
Process final response...
```

**QMBasic Class Module**

```vbnet
Respond(Cmnd, Err)
S = session->Execute("RUN MYPROG", Err)
```
loop
while Err = SV$PROMPT
    ...Process returned value and get new Response...
    S = session->Respond("RUN MYPROG", Err)
repeat
    ...Process final response...

Java

String Respond(String Cmd)
S = QMExecute("RUN MYPROG");
while(qm.ServerError == SV_PROMPT) {
    ...Process returned value and get new Response...
    S = qm.Respond(Response);
}
...Process final response...

Python

Respond(Cmd)
S, Err = qm.Execute("RUN MYPROG")
while(qm.ServerError == SV$PROMPT):
    ...Process returned value and get new Response...
    S, Err = qm.Respond(Response)
...Process final response...
7.65 QMRevision()

The **QMRevision()** function returns the revision level of the client and server components.

This function returns the revision level of the client and server sides of the session in the standard format of QM release numbers (major.minor-build). If executed before a connection is opened, only the client revision is returned. If a connection is open, the data returned is a dynamic array with two values where the first value is the client revision and the second value is the server revision. If the server pre-dates introduction of this function in release 2.9-0, the server revision will appear as 0.0-0.

**C**

```c
char * QMRevision(void)
{
    RevData = QMRevision();
    ServerRevision = QMExtract(RevData, 1, 2, 0);
    printf("Server revision %s\n", ServerRevision);
    QMFree(RevData);
    QMFree(ServerRevision);
}
```

The pointer returned by this function references a dynamically allocated memory area that must be released using **QMFree()** when no longer needed. In this example, there are two dynamically allocated strings that must be released in this way.

**VB.Net**

```
QMRevision() as String
{
    RevData = QMRevision()
    ServerRevision = QMExtract(RevData, 1, 2, 0)
    MsgBox("Server revision ", & ServerRevision)
}
```

**QMBasic Class Module**

```
Revision()
{
    RevData = session->Revision
display "Server revision ": RevData<1,2>
}
```

**Java**

```
String Revision(void)
{
    RevData = qm.Revision();
    ServerRevision = qm.Extract(RevData, 1, 2, 0);
    System.out.printf("Server revision %s\n", ServerRevision);
}
```

**Python**

```
Revision()
{
    RevData = qm.Revision
    print("Server revision ", qm.Extract(RevData, 1, 2, 0))
}
```
The **QMSeek()** function positions in a sequential file. This is analogous to the QMBasic SEEK statement.

The function arguments are:

- **FileNo** is the file number returned by a previous QMOpenSeq() call.
- **Offset** is the desired file position.
- **RelTo** determines where Offset is relative to:
  - 0 Start of file
  - 1 Current position
  - 2 End of file

The example program statements below set the current file position in the file opened to the fLog variable to be at the point specified by the Pos variable relative to the start of the file.

**C**

```c
void QMSeek(int FileNo, int Offset, int RelTo)
QMSeek(fLog, Pos, 0);
```

**VB.Net**

```vbnet
QMSeek ByVal FileNo as Integer, ByVal Offset as Integer, ByVal RelTo as Integer
QMSeek fLog, Pos, 0
```

**QMBasic Class Module**

```sql
Seek(FileNo, Offset, RelTo)
session->Seek(fLog, Pos, 0)
```

**Java**

```java
Seek(int FileNo, int Offset, int RelTo)
qm.Seek(fLog, Pos, 0);
```

**Python**

```python
Seek(FileNo, Offset, RelTo)
qm.Seek(fLog, Pos, 0)
```

See also: QMOpenSeq(), QMReadBlk(), QMReadSeq(), QMWeofSeq(), QMWriteBlk(), QMWriteSeq()
7.67 **QMSelect()**

The **QMSelect** function generates a select list containing the ids of all records in a file. It is analogous to use of the QMBasic **SELECT** statement.

The function arguments are:

- **FileNo** is the file number returned by a previous **QMOpen()** call.
- **ListNo** is the select list number (0 to 10).

The **QMSelect()** function constructs a list of record ids which can subsequently be processed using the **QMReadNext()** function. Select list 0, the default select list, is used automatically by many QM components to control their action and should, therefore, be used with caution. An unwanted or partially processed select list can be cleared using the **QMClearSelect()** function.

See [Select lists in QMClient sessions](#) for a description of the alternative ways to handle select list with QMClient.

The **QMSelect()** function does not provide any method to select only those records that meet specific conditions or to sort the list. These features can be accessed by executing query processor commands using the **QMExecute()** function.

The example program fragments below build select list 1 and use it to process records from the file open as *fClients*.

**C**

```c
void QMSelect(int FileNo, int ListNo)

QMSelect(fClients, 1);
while((Id = QMReadNext(1)) != NULL) {
    Rec = QMRead(fClients, Id, Err);
    if (Err == 0)
    {
        ...process record...
        QMFree(Rec);
    }
    QMFree(Id);
}
```

Note the use of **QMFree()** to release the dynamically allocated memory from some of the functions in this example.

**VB.Net**

```vbnet
QMSelect(ByVal FileNo as Integer, ByVal ListNo as Integer)

QMSelect fClients, 1
Do
    Id = QMReadNext(1, Err)
    If Err <> 0 Then Exit Do
    Rec = QMRead(fClients, Id, Err)
    If Err = 0 Then
        ...process record...
```

End If
Loop

### QMBasic Class Module

**Select**(FileNo, ListNo)

```qmbasic
session->Select(fClients, 1)
loop
    Id = session->ReadNext(1)
until Id = ""
    Rec = session->Read(fClients, Id, Err)
    if Err = 0 then
        ...process record...
    end
repeat
```

### Java

**Select**(int FileNo, int ListNo)

```java
qm.Select(fClients, 1);
while((Id = qm.ReadNext(1)) != NULL) {
    Rec = qm.Read(fClients, Id);
    if (qm.ServerError == SV$OK) {
        ...process record...
    }
}
```

### Python

**Select**(FileNo, ListNo)

```python
qm.Select(fClients, 1)
while True:
    Id = qm.ReadNext(1)
    if Id = "": break
    Rec, Err = qm.Read(fClients, Id)
    ...process record...
```
7.68 QMSelectIndex()

The QMSelectIndex() function generates a select list from an alternate key index.

The function arguments are:

- **FileNo** is the file number returned by a previous QMOpen() call.
- **IndexName** is the name of the alternate key index to be used.
- **IndexValue** is the value to be located in the alternate key index.
- **ListNo** is the select list number (0 to 10).

The QMSelectIndex function constructs a list of record ids from an entry in an alternate key index. This list can subsequently be processed using the QMReadNext() function. Select list 0, the default select list, is used automatically by many QM components to control their action and should, therefore, be used with caution. An unwanted or partially processed select list can be cleared using the QMClearSelect function.

The number of items in the select list can be established by using QMGetVar() to retrieve the @SELECTED variable immediately after the QMSelectIndex().

See the QMReadList() function for a discussion on different ways to process the select list.

The example program fragments below build select list 1 from an index and use it to process records from the file open as fClients.

C

```c
void QMSelectIndex(int FileNo, char * IndexName, char * IndexValue, int ListNo)
QMSelectIndex(fClients, "REGION", RegionNo, 1);
while((Id = QMReadNext(1)) != NULL) {
    Rec = QMRead(fClients, Id, Err);
    if (Err == 0)
        {...process record...}
    QMFree(Rec);
    QMFree(Id);
}
```

Note the use of QMFree() to release the dynamically allocated memory from some of the functions in this example.

VB.Net

```vbnet
QMSelectIndex ByVal FileNo as Integer, ByVal IndexName as String, ByVal IndexValue as
String, ByVal ListNo as Integer
QMSelectIndex fClients, "REGION", RegionNo, 1
Do
    Id = QMReadNext(1, Err)
    If Err <> 0 Then Exit Do
```
Rec = QMRead(fClients, Id, Err)
If Err = 0 Then
    ...process record...
End If
Loop

**QMBasic Class Module**

**SelectIndex(FieldName, IndexName, IndexValue, ListNo)**

session->SelectIndex(fClients, 'REGION', RegionNo, 1)
loop
    Id = session->ReadNext(1)
    Rec = session->Read(fClients, Id, Err)
    if Err = 0 then
        ...process record...
    end
end
repeat

**Java**

**SelectIndex(int FileNo, String IndexName, String IndexValue, int ListNo)**

qm.SelectIndex(fClients, "REGION", RegionNo, 1);
qm.Select(fClients, 1);
while((Id = qm.ReadNext(1)) != NULL) {
    Rec = qm.Read(fClients, Id);
    if (qm.ServerError == SV$OK) {
        ...process record...
    }
}

**Python**

**SelectIndex(FieldName, IndexName, IndexValue, ListNo)**

qm.SelectIndex(fClients, "REGION", RegionNo, 1)
while True:
    Id = qm.ReadNext(1)
    if Id == "": break
    Rec, Err = qm.Read(fClients, Id)
    ...process record...
7.69 QMSelectLeft and QMSelectRight

The QMSelectLeft() and QMSelectRight() functions traverse an alternate key index, creating a select list from the entry to the left or right of the last entry processed. They are analogous to the QMBasic SELECTLEFT and SELECTRIGHT statements.

The function arguments are:

- **Var** is the variable to receive the index key value associated with the returned list.
- **FileNo** is the file number returned by a previous QMOpen() call.
- **IndexName** is the name of the alternate key index to be used.
- **ListNo** is the select list number (0 to 10).

The QMSelectLeft() and QMSelectRight() functions construct a select list from the alternate key index entry to the left or right of the one most recently returned by QMSelectIndex(), QMSelectLeft() or QMSelectRight(). The position of the scan can be set at the extreme left using QMSetLeft() or at the extreme right using QMSetRight().

These operations allow a program to find a specific value and then walk through successive values in the sorted data structure that makes up an alternate key index. The function returns the indexed data value associated with the index entry found. The select list identified by ListNo is set to contain the records ids of the records that have this data value.

If QMSelectIndex() is used to locate a value that does not exist in the index, QMSelectLeft() will return a list of records for the value immediately before the non-existent one and QMSelectRight() will return a list of records for the value immediately after the non-existent one.

The QMStatus() function returns zero if the operation is successful, non-zero if it fails. Although other errors may occur, two useful status values are:

- 3019 ER$AKNF Specified index does not exist
- 3030 ER$EOF No further items at end of index

The number of items in the select list can be established by using QMGetVar() to retrieve the @SELECTED variable immediately after the QMSelectIndex.

The example program fragments below set the scan position for an index on the CODE field to the leftmost item and walk through successive entries, processing each record.

```c
char * QMSelectLeft(int FileNo, char * IndexName, int ListNo)
char * QMSelectRight(int FileNo, char * IndexName, int ListNo)
QMSetLeft(fData, "CODE");
while(1)
{
    Key = QMSelectRight(fData, "CODE", 1);
    if (QMStatus() != 0) break;
    while((Id = QMReadNext(1)) != NULL)
```
{    if (Id == NULL) break;    Rec = QMRead(fData, Id, &Err);    if (Err == 0)    {
        ...process record...
        QMFree(Rec);
    }
    QMFree(Id);
}
QMFree(Key);

The returned pointer references a dynamically allocated memory area that must be released using QMFree() when no longer needed. In this example, there are other dynamic memory areas that must be released in the same way.

VB.Net

QMSelectLeft(ByVal FileNo as Integer, ByVal IndexName as String, ByVal ListNo as Integer) as String

QMSelectRight(ByVal FileNo as Integer, ByVal IndexName as String, ByVal ListNo as Integer) as String

QMSetLeft(fData, "CODE")
Do
    Key = QMSelectRight(fData, "CODE", 1)
    If QMStatus() <> 0 Then Exit Do
    Do
        Id = QMReadNext(1)
        If Id = Nothing Then Exit Do
        Rec = QMRead(fData, Id, Err)
        If Err = 0 Then
            ...process record...
        End If
    Loop
Loop

QMBasic Class Module

SelectLeft(FileNo, IndexName, ListNo)

SelectRight(FileNo, IndexName, ListNo)

session->SetLeft(fData, "CODE")
loop
    Key = session->SelectRight(fData, "CODE", 1)
until session->ServerStatus
loop
    Id = session->ReadNext(1)
until Id = ""
    Rec = session->Read(fData, Id, Err)
    if Err = 0 then
        ...process record...
    end
repeat
repeat

Java
String SelectLeft(int FileNo, String IndexName, int ListNo)

String SelectRight(int FileNo, String IndexName, int ListNo)

gm.SetLeft(fData, "CODE");
while(true)
{
    Key = gm.SelectRight(fData, "CODE", 1);
    if (gm.ServerStatus != 0) break;
    while(true)
    {
        Id = gm.ReadNext(1);
        if (gm.ServerError != SV$OK) break;
        Rec = gm.Read(fData, Id);
        if (gm.ServerError == 0)
        {
            ...process record...
        }
    }
}

Python

SelectLeft(FileNo, IndexName, ListNo)

SelectRight(FileNo, IndexName, ListNo)

gm.SetLeft(fData, "CODE")
while True:
    Key = gm.SelectRight(fData, "CODE", 1)
    if gm.Status() != 0: break
    while True:
        Id = gm.ReadNext(1)
        if Id == ":" : break
        Rec, Err = gm.Read(fData, Id)
        if Err == 0:
            ...process record...
The **QMSelectPartial()** function generates a select list containing the ids of all records in a file using an optimised method for improved performance. It is analogous to use of the QMBasic `SELECT` statement.

The function arguments are:

- **FileNo** is the file number returned by a previous `QMOpen()` call.
- **ListNo** is the select list number (0 to 10).

The `QMSelectPartial()` function constructs a list of record ids in the file open as `FileNo`. The first part of the select list is returned as a field mark delimited string. Further items can then be returned by use of the `QMNextPartial()` function until this returns a null string to indicate that there are no further record ids available.

See [Select lists in QMClient sessions](#) for a description of the alternative ways to handle select list with QMClient.

Select list 0, the default select list, is used automatically by many QM components to control their action and should, therefore, be used with caution. An unwanted or partially processed select list can be cleared using the `QMClearSelect()` function.

The `QMSelectPartial()` function does not provide any method to select only those records that meet specific conditions or to sort the list. These features can be accessed by executing query processor commands using the `QMExecute()` function.

The example program fragments below build select list 1 and use it to process records from the file open as `fClients`.

**C**

```c
char * QMSelectPartial(int FileNo, int ListNo)

list = QMSelectPartial(fClients, 1);
while(list != NULL) {
    n = QMDcount(list, FM);
    for(i = 1; i <= n; i++) {
        id = QMExtract(list, i, 0, 0);
        ...processing...
        QMFree(id);
    }
    QMFree(list);
    list = QMNextPartial(1);
}

\Note that the returned pointer references a dynamically allocated memory area that must be released using `QMFree()` when no longer needed as shown in this example.
```

**VB.Net**

```vbnet
QMSelectPartial(ByVal FileNo as Integer, ByVal ListNo as Integer) as String
```
List = QMSelectPartial(fClients, 1)
While List <> ""
    N = QMDcount(List, FM)
    For I = 1 To N
        Id = QMExtract(List, I, 0, 0)
        ...processing...
        Next I
    List = QMNextPartial(1)
End While

QMBasic Class Module

list = session->SelectPartial(fClients, 1)
loop while list # ""
    for each id in list
        ...processing...
    next id
    list = session->NextPartial(1)
repeat

Java

String SelectPartial(int FileNo, int ListNo)

list = qm.SelectPartial(fClients, 1);
while qm.ServerError == SV_OK) {
    n = qm.Dcount(list, FM);
    for(i = 1; i <= n; i++)
        { id = qm.Extract(list, i, 0, 0);
          ...processing...
        }
    list = qm.NextPartial(1);
}

Python

SelectPartial(FileNo, ListNo)

list = qm.SelectPartial(fClients, 1)
while qm.ServerError == SV_OK:
    n = qm.Dcount(list, FM)
    for i in range(1, n+1):
        id = qm.Extract(list, i, 0, 0)
        ...processing...
    list = qm.NextPartial(1)
7.71 QMSet()

The **QMSet()** function sets data or executes a public subroutine in an instantiated object on the server system.

The function arguments are:

- **Objno** is the numeric identifier returned by a previous use of **QMCreateObject()**.
- **Name** is the name of a public variable or public function within the object.
- **ArgCt** is a count of arguments to be passed into the object. This argument is not present in some variants of the QMClient API.
- **Arg1...** is a series of argument values as strings.

The example program fragments below set the value of the State property to "Online" in an object previously instantiated with the reference value in Objno. In this example, there are no arguments passed into the object.

**C**

```c
QMSet(short int Object, char * Name, short int ArgCt, Arg1,...)
QMSet(Objno, "State", 1, "Online");
```

The arguments follow the **ArgCt** and must be passed as strings. Any argument values updated by the object can be retrieved using the **QMGetArg()** function.

**VB.Net**

```vbnet
QMSet(ByVal Object as Short, ByVal Name as String, ArgCt as Short, Optional ByRef Arg1 as String, ...)
QMSet(Objno, 'State', 1, "Online")
```

The arguments must be passed as strings. Any argument values updated by the object can be retrieved using the **QMGetArg()** function.

**QMBasic Class Module**

```vbscript
Set(Objno, Name {, Arg1,...})
session->Set(Objno, 'State', 'Online')
```

**Java**

```java
QMSet(int Object, String Name, int ArgCt, Arg1,...)
qm.Set(Objno, "State", 1, "Online");
```

The arguments follow the **ArgCt** and must be passed as strings. Any argument values updated by the object can be retrieved using the **QMGetArg()** function.

**Python**

```python
Set(Object, Name, ArgCt, Arg1,...)
qm.Set(Objno, "State", 1, "Online")
```
The arguments follow the $ArgCt$ and must be passed as strings. Any argument values updated by the object can be retrieved using the QMGetArg() function.

See also: QMCreateObject(), QMDestroyObject, QMGet()
7.72  QMSetLeft() and QMSetRight()

The QMSetLeft() and QMSetRight() functions set the scanning position of an alternate key index at the extreme left or right of the data. They are analogous to the QMBasic SETLEFT and SETRIGHT statements.

The function arguments are:

- **FileNo** is the file number returned by a previous QMOpen() call.
- **IndexName** is the name of the alternate key index to be used.

The QMSetLeft and QMSetRight functions are used with QMSelectLeft() and QMSelectRight() to set the scan position to the first or last entry in an alternate key index.

The QMStatus() function returns zero if the operation is successful, non-zero if it fails because the index does not exist.

The example program fragments below use QMSetLeft() to set the scan position for an index on the CODE field to the leftmost item and walk through successive entries, processing each record.

```c
void QMSetLeft(int FileNo, char * IndexName)
void QMSetRight(int FileNo, char * IndexName)
QMSetLeft(fData, "CODE");
while(1) {
    Key = QMSelectRight(fData, "CODE", 1);
    if (QMStatus() != 0) break;
    while((Id = QMReadNext(1)) != NULL) {
        if (Id == NULL) break;
        Rec = QMRead(fData, Id, &Err);
        if (Err == 0) {
            ...process record...
            QMFree(Rec);
        }
        QMFree(Id);
    }
    QMFree(Key);
}
```

The returned pointers from several functions in this example reference dynamically allocated memory areas that must be released using QMFree() when no longer needed.

```vbnet
QMSetLeft(ByVal FileNo as Integer, ByVal IndexName as String)
QMSetRight(ByVal FileNo as Integer, ByVal IndexName as String)
QMSetLeft(fData, "CODE")
Do
    Key = QMSelectRight(fData, "CODE", 1)
```
If QMStatus() <> 0 Then Exit Do
Do
  Id = QMReadNext(1)
  If Id = Nothing Then Exit Do
  Rec = QMRead(fData, Id, Err)
  If Err = 0 Then
    ...process record...
  End If
Loop
Loop

QMBasic Class Module

SetLeft(FileNo, IndexName)
SetRight(FileNo, IndexName)

session->SetLeft(fData, "CODE")
loop
  Key = session->SelectRight(fData, "CODE", 1)
until session->ServerStatus
loop
  Id = session->ReadNext(1)
until Id = ""
Rec = session->Read(fData, Id, Err)
if Err = 0 then
  ...process record...
end
repeat
repeat

Java

void SelectLeft(int FileNo, String IndexName)
void SelectRight(int FileNo, String IndexName)

qm.SetLeft(fData, "CODE");
while(true)
{
  Key = qm.SelectRight(fData, "CODE", 1);
  if (qm.ServerStatus != 0) break;
  while(true)
  {
    Id = qm.ReadNext(1);
    if (qm.ServerError != SV$OK) break;
    Rec = qm.Read(fData, Id);
    if (qm.ServerError == 0)
    {
      ...process record...
    }
  }
}

Python

SetLeft(FileNo, IndexName)
SetRight(FileNo, IndexName)
qm.SetLeft(fData, "CODE")
while True:
    Key = qm.SelectRight(fData, "CODE", 1)
    if qm.Status() != 0: break
    while True:
        Id = qm.ReadNext(1)
        if Id = "": break
        Rec, Err = qm.Read(fData, Id)
        if Err == 0:
            ...process record...
7.73 QMSetSession()

The QMSetSession() function selects an active QMClient session to be referenced by subsequent function calls.

A single client may open multiple QMClient connections, each identified by a session number. The QMSetSession() function determines to which session subsequent QMClient function calls relate.

The return value from this function is True if successful, False if the specified session does not exist.

The QMSetSession() function has no equivalent in the QMBasic class module, Java or Python variants of the QMClient API as each session is managed by a separate instantiation of the object.

The examples below save the current session number into a variable named OldSesNum, open a new connection (which will create a new session), perform some processing, disconnect the new session and then use QMSetSession() to revert to the original session.

C

```c
void QMSetSession(int SesNum)
{
    OldSesNum = QMGetSession();
    if (QMConnectLocal("SALES"))
    {
        ...processing...
        QMDisconnect();
    }
    SetSession(OldSesNum);
}
```

VB.Net

```vbnet
QMSetSession(SesNum as Integer)
{
    OldSesNum = QMGetSession()
    If QMConnectLocal("SALES") Then
        ...processing...
        QMDisconnect()
    End If
    SetSession(OldSesNum)
}
```

See also:
QMGetSession()
7.74 QMStatus()

The **QMStatus()** function returns the value of the QMBasic **STATUS()** function for the last server function executed.

Many server actions set the QMBasic **STATUS()** value and return it to the client process. The **QMStatus()** function retrieves this value. This function does not require passing of a client server message pair as the value is held on the client system.

### C

```c
int QMStatus(void)
{
    IntDate = QMIConv(Date, "D2");
    if (QMStatus() != 0) printf("Invalid date\n");
}
```

### VB.Net

```vbnet
QMStatus() as Integer
IntData = QMIConv(Date, "D2")
If (QMStatus() <> 0) MsgBox("Invalid date")
```

### QMBasic Class Module

```qmbasic
Status
IntDate = session->Iconv(Date, "D2");
if (session->Status # 0) display("Invalid date");
```

The **QMStatus()** function is implemented as a public property variable in this version of the QMClient API.

### Java

```java
Status
IntDate = qm.IConv(Date, "D2")
if (qm.Status != 0) System.out.println("Invalid date");
```

The **QMStatus()** function is implemented as a public property variable in this version of the QMClient API.

### Python

```python
Status
IntDate = qm.IConv(Date, "D2")
if qm.Status() != 0: print("Invalid date")
```

The **QMStatus()** function is implemented as a public property variable in this version of the QMClient API.
7.75 QMSystem()

The QMSystem() function executes a QMBasic SYSTEM() function on the server.

The function takes one argument:

\[ Key \]
identifies the action to be performed.

If the action is successful, the result is returned as a string.

The example program fragments below use the QMSystem() function to retrieve the QMSYS account pathname.

**C**

```c
char * QMSystem(int Key)
Path = QMSystem(32);
```

The returned pointer references a dynamically allocated memory area that must be released using QMFree() when no longer needed.

**VB.Net**

```vbnet
QMSystem(ByVal Key as Integer) as String
Path = QMSystem(32)
```

**QMBasic Class Module**

```qmbasic
System(Key)
Path = session->System(32)
```

**Java**

```java
String QMSystem(int Key)
Path = qm.System(32);
```

**Python**

```python
System(Key)
Path = qm.System(32)
```
7.76 QMTrans(), QMRTrans()

The QMTrans() function performs the equivalent of a QMBasic TRANS() function on the server.

The function arguments are:

- **FileName** is the name of the file from which data is to be read.
- **Id** is the id of the record to be read. This may be a multivalued list of record ids.
- **FieldName** is the dictionary name of the field to be returned. The dictionary record may be any valid data defining item (types A, C, D, E, I, S). Alternatively, **FieldName** may be specified as a numeric field position or -1 to return the entire record.
- **Action** sets the action to be taken if the data cannot be found:
  - C Return the record id.
  - X Return a null value.

In addition,
- **B** Causes the file to be opened in binary mode with mark mapping disabled.

The QMTrans() function causes the server process to open the data file and its dictionary, execute a QMBasic TRANS() function and return the result.

The QMTrans() function returns the specified data with any mark characters lowered by one level (e.g. value marks become subvalue marks).

If record.id is multivalued, the QMTrans() function extracts each requested record and returns a multivalued result with the data from each record separated by a value mark.

The QMRTrans() function is identical to QMTrans() except that it does not lower the mark characters. This makes it impossible to distinguish between the results of retrieving a multivalued field from a single record and retrieving a single valued field from multiple records.

The example program fragments below are based on the QM demonstration database and use the QMTrans() function to fetch the current price for the parts in an order from the STOCK file.

C

```c
char * QMTrans(char * FileName, char * Id, char * FieldName, char * Action)

Prices = QMTrans("STOCK", Items, "PRICE", "X");
```

VB.Net

```vbnet
QMTrans(ByVal FileName as String, ByVal Id as String, ByVal FieldName as String, ByVal Action as String) as String

Prices = QMTrans("STOCK", Items, "PRICE", "X")
```

The returned pointer references a dynamically allocated memory area that must be released using QMFree() when no longer needed. Note that attempting to read non-existent data returns a pointer to a null string, not a NULL pointer.
QMBasic Class Module

Trans(FileName, Id, FieldName, Action)

Prices = session->Trans("STOCK", Items, "PRICE", "X")

Java

String Trans(String FileName, String Id, String FieldName, String Action)

Prices = qm.Trans("STOCK", Items, "PRICE", "X");

Python

Trans(FileName, Id, FieldName, Action)

Prices = qm.Trans("STOCK", Items, "PRICE", "X")

See also:
QMEvaluate()
7.77 QMTrapCallAbort()

The QMTrapCallAbort() function enables or disables client side trapping of aborts in actions that execute QMBasic code on the server.

The function takes a single argument, Mode, evaluating to True or False.

By default, an abort occurring in a subroutine called using QMCall() will display a message box (VB) or display a message. Use of the key value 8 to the QMConnectionType() function before opening the connection can modify this behaviour such that display is suppressed, returning the abort message via QMError().

The QMTrapCallAbort function allows an application to trap the abort and take its own recovery action. When call abort trapping is enabled, use of QMStatus() immediately after return from QMCall() will return zero if the call ran successfully or non-zero if an abort occurred. The actual abort message can be retrieved using QMError().

When call abort trapping is not enabled, the value returned by QMStatus() after QMCall() is the current STATUS() function value on the server.

When operating with multiple server connections from a single client, the state of call abort trapping applies to all connections.

Call abort trapping applies to QMCall(), QMCallx(), QMCreateObject(), QMDestroyObject(), QMGet() and QMSet().

C

void QMTrapCallAbort(int Mode)
QMTrapCallAbort(1);

VB.Net

QMTrapCallAbort ByVal Mode as Integer
QMTrapCallAbort(True)

QMBasic Class Module

TrapCallAbort(Mode)
Session->TrapCallAbort(@true)

Java

boolean TrapCallAbort
qm.TrapCallAbort = true;

The QMTrapCallAbort() function is implemented as a public property variable in this version of the QMClient API.

Python

TrapCallAbort(Mode)
qm.TrapCallAbort(True)
7.78 QMTxn()

The QMTxn() function starts, commits or aborts a transaction on the server.

The function takes a single argument, Mode, as the action to be performed. This may be:

1. Start a transaction. Equivalent to use of the QMBasic TRANSACTION START statement.
2. Commit a durable transaction. Equivalent to use of the QMBasic TRANSACTION COMMIT statement.
3. Abort a transaction. Equivalent to use of the QMBasic TRANSACTION ABORT statement.
4. Commit a non-durable transaction. Equivalent to use of the QMBasic TRANSACTION COMMIT statement in a program compiled with the $MODE compiler directive.

The example program fragments below read two records, apply an update such as transferring funds from one client to the other, and then write both records. By using a transaction, the two updates are applied together. A real program would include error handling.

C

```c
QMTxn(Mode)
QMTxn(1);
Rec1 = QMReadu(fClients, Client1, TRUE, &Err);
Rec2 = QMReadu(fClients, Client2, TRUE, &Err);
...processing...
QMWrite(fClients, Client1, Rec1);
QMWrite(fClients, Client2, Rec2);
QMTxn(2);
QMFree(Rec);
QMFree(Rec2);
```

Note the use of QMFree() to release dynamically allocated memory in this example.

VB.Net

```vbnet
QMTxn(Mode as Integer)
QMTxn(1)
Rec1 = QMReadu(fClients, Client1, True, Err)
Rec2 = QMReadu(fClients, Client2, True, Err)
...processing...
QMWrite fClients, Client1, Rec1
QMWrite fClients, Client2, Rec2
QMTxn(2)
```

QMBasic Class Module

```vbnet
Txn(Mode)
session->Txn(1)
Rec1 = session->Readu(fClients, Client1, @true, Err)
Rec2 = session->Readu(fClients, Client2, @true, Err)
...processing...
```
session->Write(fClients, Client1, Rec1)
session->Write(fClients, Client2, Rec2)
session->Txn(2)

Java

Txn(Mode)

qm.Txn(1);
Rec1 = qm.Readu(fClients, Client1, true);
Rec2 = qm.Readu(fClients, Client2, true);
...processing...
qm.Write(fClients, Client1, Rec1);
qm.Write(fClients, Client2, Rec2);
qm.Txn(2);

Python

Txn(Mode)

qm.Txn(1)
Rec1 = qm.Readu(fClients, Client1, true)
Rec2 = qm.Readu(fClients, Client2, true)
...processing...
qm.Write(fClients, Client1, Rec1)
qm.Write(fClients, Client2, Rec2)
qm.Txn(2)
7.79 QMWeofSeq()

The QMWeofSeq() function truncates a sequential file. This is analogous to the QMBasic WEOFSEQ statement.

The function takes a single argument, FileNo, as the file number returned by a previous QMOpenSeq() call.

The QMWeofSeq function truncates the file opened as FileNo such that any data beyond the current file position is discarded.

The example program fragments below write the line of text in ReportData to the file opened to the fReport variable and then truncate the file to remove any old data beyond that position when overwriting a file.

C

```c
void QMWeofSeq(int FileNo)
QMWriteSeq(fReport, ReportData);
QMWeofSeq(fReport);
```

VB.Net

```vbnet
QMWeofSeq ByVal FileNo as Integer
QMWriteSeq fReport, ReportData
QMWeofSeq fReport
```

QMBasic Class Module

```qmbasic
WeofSeq(FileNo)
session->WriteSeq(fReport, ReportData)
session->WeofSeq(fReport)
```

Java

```java
WeofSeq(int FileNo)
qm.WriteSeq(fReport, ReportData);
qm.WeofSeq(fReport);
```

Python

```python
WeofSeq(FileNo)
qm.WriteSeq(fReport, ReportData)
qm.WeofSeq(fReport)
```

See also: QMOpenSeq(), QMReadBlk(), QMReadSeq(), QMSeek(), QMWriteBlk(), QMWriteSeq()
7.80 QMWrite()

The QMWrite() function writes a record. This is analogous to the QMBasic WRITE statement.

The function arguments are:

- **FileNo** is the file number returned by a previous QMOpen() call.
- **Id** is the id of the record to be written.
- **Rec** is the data to be written to this record.

The QMWrite function writes the given data to the file opened as FileNo. If a record with this Id already exists, it is replaced. If the record does not already exist, it is added.

An application should always obtain an update lock on the record before writing it. This function releases the lock.

When writing a new record to a directory file on a Linux or Unix system, the operating system level file that represents the QM record is created using the umask value specified by the UMASK configuration parameter or, if this parameter is not present, a default of 002. This behaviour can be modified by executing a UMASK command either from the LOGIN paragraph or by using the QMExecute() function.

The example program fragments below read a record with an update lock, modify it and then write it back to the file. A real program should test the error status from the read operations to determine if they were successful.

**C**

```c
void QMWrite(int FileNo, char * Id, char * Rec)

Rec = QMReadu(fClients, ClientNo, TRUE, Err);
Rec2 = QMReplace(Rec, 1, Pos, 0, NewData);
QMWrite(fClients, ClientNo, Rec2);
QMFree(Rec);
QMFree(Rec2);
```

Note the use of QMFree() to release dynamically allocated memory areas in this example.

**VB.Net**

```vbnet
QMWrite ByVal FileNo as Integer, ByVal Id as String, ByVal Rec as String

Rec = QMReadu(fClients, ClientNo, True, Err)
Rec = QMReplace(Rec, 1, Pos, 0, NewData)
QMWrite fClients, ClientNo, Rec
```

**QMBasic Class Module**

```qmbasic
Write(FileNo, Id, Rec)

Rec = session->Readu(fClients, ClientNo, @true, Err)
Rec<1, Pos> = NewData
session->Write(fClients, Id, Rec)
```

**Java**
Write\( (\text{int\ } \text{FileNo}, \text{ String\ } \text{Id}, \text{ String\ } \text{Rec}) \)

\[ \text{Rec} = \text{qm.Readu(fClients, ClientNo, TRUE);} \]
\[ \text{Rec2} = \text{qm.Replace(Rec, 1, Pos, 0, NewData);} \]
\[ \text{qm.Write(fClients, ClientNo, Rec2);} \]

Python

Write\( (\text{FileNo}, \text{ Id, Rec}) \)

\[ \text{Rec, Err = qm.Readu(fClients, ClientNo, true)} \]
\[ \text{Rec2 = qm.Replace(Rec, 1, Pos, 0, NewData)} \]
\[ \text{qm.Write(fClients, ClientNo, Rec2)} \]
The **QMWriteBlk()** function writes a specified number of bytes to a sequential file. This is analogous to the QMBasic WRITEBLK statement.

The function arguments are:

- **FileNo** is the file number returned by a previous **QMOpenSeq()** call.
- **Rec** is the data to be written.
- **Bytes** is the number of bytes to be written. Note that this argument is not present in all variants of the QMClient API as it may be possible to determine the size of **Rec** internally.

The example program statements below write the binary data in LogRec to the file opened to the fLog variable.

**C**

```c
void QMWriteBlk(int FileNo, char * Rec, int Bytes)
QMWriteSeq(fLog, LogRec, LogRecSize);
```

**VB.Net**

```vbnet
QMWriteBlk ByVal FileNo as Integer, ByVal Rec as String, ByVal Bytes as Integer
QMWriteBlk(fLog, LogRec, LogRecSize)
```

**QMBasic Class Module**

```vbnet
WriteBlk(FileNo, Rec)
session->WriteSeq(fLog, LogRec)
```

**Java**

```java
WriteBlk(int FileNo, byte[] Rec)
qm.WriteSeq(fLog, LogRec);
```

**Python**

```python
WriteBlk(FileNo, LogRec)
qm.WriteSeq(fLog, LogRec);
```

See also: **QMOpenSeq()**, **QMReadBlk()**, **QMReadSeq()**, **QMSseek()**, **QMWeofSeq()**, **QMWriteSeq()**
7.82 QMWriteSeq()

The `QMWriteSeq()` function writes a line of text to a sequential file. This is analogous to the QMBasic `WRITESEQ` statement.

The function arguments are:

- `FileNo` is the file number returned by a previous `QMOpenSeq()` call.
- `Rec` is the data to be written.

The example program statements below write the line of text in `ReportData` to the file opened to the `fReport` variable.

C

```c
void QMWriteSeq(int FileNo, char * Rec)
    QMWriteSeq(fReport, ReportData);
```

VB.Net

```vbnet
QMWriteSeq ByVal FileNo as Integer, ByVal Rec as String
    QMWriteSeq(fReport, ReportData)
```

QMBasic Class Module

```qbasic
WriteSeq(FileNo, Rec)
    session->WriteSeq(fReport, ReportData)
```

Java

```java
WriteSeq(int FileNo, String Rec)
    qm.WriteSeq(fReport, ReportData);
```

Python

```python
WriteSeq(FileNo, Rec)
    qm.WriteSeq(fReport, ReportData)
```

See also: `QMOpenSeq()`, `QMReadBlk()`, `QMReadSeq()`, `QMSeek()`, `QMWeofSeq()`, `QMWriteBlk()`
The **QMWriteSeqKey()** function writes a record using an automatically generated sequential numeric key. This is analogous to use of the CREATING.SEQKEY option of the QMBasic **WRITE** statement.

The function arguments are:

- **FileNo** is the file number returned by a previous **QMOpen()** call.
- **Id** is the id of the record.
- **Rec** is the data to be written to this record.

The **QMWriteSeqKey()** function writes the given data to the hashed file opened as **FileNo**. The record id is automatically generated and returned as the value of this function. This operation is not valid for directory files or distributed files.

An update lock is automatically applied to this record for the duration of the internal processing of the write and released before returning from the function.

The example program fragments below write the data in **ClientRec** to the file open as **fClients**. The record id is generated automatically and returned in the **Id** variable.

**C**

```c
char * QMWriteSeqKey(int FileNo, char * Rec)

Id = QMWriteSeqKey(fClients, ClientRec);
printf("New id is '%s'\n", Id);
QMFree(Id);
```

The returned pointer references a dynamically allocated memory area that must be released using **QMFree()** when no longer needed.

**VB.Net**

```vbnet
QMWriteSeqKey(ByVal FileNo as Integer, ByVal Rec as String) as String
Id = QMWriteSeqKey(fClients, ClientRec)
```

**QMBasic Class Module**

```vbs
WriteSeqKey(FileNo, Rec)
Id = session->WriteSeqKey(fClients, ClientRec)
```

**Java**

```java
String WriteSeqKey(int FileNo, String Rec)
Id = qm.WriteSeqKey(fClients, ClientRec);
System.out.printf("New id is '%s'\n", Id);
```

**Python**

```python
WriteSeqKey(FileNo, Rec)
Id = qm.WriteSeqKey(fClients, ClientRec)
```
print("New id is ", Id)
7.84 QMWriteuSeqKey()

The QMWriteuSeqKey() function writes a record using an automatically generated sequential numeric key. This is analogous to use of the CREATING.SEQKEY option of the QMBasic WRITEU statement.

The function arguments are:

- **FileNo** is the file number returned by a previous QMOpen() call.
- **Id** is the id of the record.
- **Rec** is the data to be written to this record.

The QMWriteuSeqKey() function writes the given data to the hashed file opened as FileNo. The record id is automatically generated and returned as the value of this function. This operation is not valid for directory files or distributed files.

An update lock is automatically applied to this record for the duration of the internal processing of the write and retained on return from the function.

The example program fragments below write the data in ClientRec to the file open as fClients. The record id is generated automatically and returned in the Id variable.

**C**

```c
char * QMWriteuSeqKey(int FileNo, char * Rec)

Id = QMWriteuSeqKey(fClients, ClientRec);
printf("New id is '"\%s'\n", Id);
QMFree(Id);
```

The returned pointer references a dynamically allocated memory area that must be released using QMFree() when no longer needed.

**VB.Net**

```csharp
QMWriteuSeqKey(ByVal FileNo as Integer, ByVal Rec as String) as String
Id = QMWriteuSeqKey(fClients, ClientRec)
```

**QMBasic Class Module**

```csharp
WriteuSeqKey(FileNo, Rec)
Id = session->WriteuSeqKey(fClients, ClientRec)
```

**Java**

```java
String WriteuSeqKey(int FileNo, String Rec)
Id = qm.WriteuSeqKey(fClients, ClientRec);
System out.printf("New id is '"\%s'\n", Id);
```

**Python**

```python
WriteuSeqKey(FileNo, Rec)
Id = qm.WriteuSeqKey(fClients, ClientRec)
```
print("New id is ", Id)
7.85 QMWriteu()

The QMWriteu() function writes a record. This is analogous to the QMBasic WRITEU statement.

The function arguments are:

- **FileNo** is the file number returned by a previous QMOpen() call.
- **Id** is the id of the record to be written.
- **Rec** is the data to be written to this record.

The QMWriteu() function writes the given data to the file opened as FileNo, retaining the record lock. If a record with this Id already exists, it is replaced. If the record does not already exist, it is added.

An application should always obtain an update lock on the record before writing it. This function does not release the lock.

When writing a new record to a directory file on a Linux or Unix system, the operating system level file that represents the QM record is created using the umask value specified by the UMASK configuration parameter or, if this parameter is not present, a default of 002. This behaviour can be modified by executing a UMASK command either from the LOGIN paragraph or by using the QMExecute() function.

The example program fragments below read a record with an update lock, modify it and then write it back to the file. A real program should test the error status from the read operations to determine if they were successful.

**C**

```c
void QMWriteu(int FileNo, char * Id, char * Rec)
Rec = QMReadu(fClients, ClientNo, TRUE, Err);
Rec2 = QMReplace(Rec, 1, Pos, 0, NewData);
QMWriteu(fClients, ClientNo, Rec2);
QMFree(Rec);
QMFree(Rec2);
```

Note the use of QMFree() to release dynamically allocated memory areas in this example.

**VB.Net**

```vbnet
QMWriteu ByVal FileNo as Integer, ByVal Id as String, ByVal Rec as String
Rec = QMReadu(fClients, ClientNo, True, Err)
Rec = QMReplace(Rec, 1, Pos, 0, NewData)
QMWriteu fClients, ClientNo, Rec
```

**QMBasic Class Module**

```qbasic
Writeu(FileNo, Id, Rec)
Rec = session->Readu(fClients, ClientNo, @true, Err)
Rec<1, Pos> = NewData
session->Writeu(fClients, Id, Rec)
```

Java

Writeu(int FileNo, String Id, String Rec)

Rec = qm.Readu(fClients, ClientNo, TRUE);
Rec2 = qm.Replace(Rec, 1, Pos, 0, NewData);
qm.Writeu(fClients, ClientNo, Rec2);

Python

Writeu(FileNo, Id, Rec)

Rec, Err = qm.Readu(fClients, ClientNo, true)
Rec2 = qm.Replace(Rec, 1, Pos, 0, NewData)
qm.Writeu(fClients, ClientNo, Rec2)
Part 8
System Administration
8 System Administration

System Configuration
- Configuration Parameters: Parameter descriptions
- UPDATE.LICENCE: Apply a new licence
- The Termino Database: Terminal configuration
- The qmtic Utility: Termino compiler

Account Management
- Accounts: What is an account?
- CREATE.ACCOUNT: Create an account
- DELETE.ACCOUNT: Delete an account
- The Login Process: What happens when a user logs in

System Security
- ADMIN.USER: User administration
- CREATE.USER: Create a user
- DELETE.USER: Delete a user
- LIST USERS: List user name details
- PASSWORD: Password management (Windows USB installations)
- SECURITY: Enable, disable or report security settings

Process Management
- LISTU: Who is logged in?
- PSTAT: Monitoring processes
- LOGOUT: Killing a process

Monitoring the File System
- LIST.FILES: Determining the files in use
- LISTLOCKS: Monitoring task locks
- LIST.READU: Monitoring record and file locks
- UNLOCK: Releasing a lock
- FSTAT: Monitoring file activity
- ANALYSE.FILE: File analysis
- FILE_STAT: Summary report of all files in one or more accounts
8.1 Configuration Parameters

QM has a number of configuration parameters that determine major settings for the system. These are stored in a file named qmconfig in the QMSYS account directory. Only system administrators should have write access to this file. Modifications can be made with the `EDIT.CONFIG` command or any text editor such as `ED` or `SED` from within QM, Notepad on Windows, or `vi` on Linux. Note that copying this file between systems may have unwanted effects as it may also contain other system configuration data.

The file is divided into a number of sections, each with a section title enclosed in square brackets. Only the `[qm]` section must be present. This section contains the configuration parameters described below which may be global or private. Global parameters apply to all users of QM. Private parameters, although initially set to the state defined in the configuration file, may be updated for an individual process using the `CONFIG` command. Private configuration parameters are marked with an asterisk in the table below.

If a line includes a # character, this is treated as the start of a comment and all later text on the line is ignored.

Parameters that take on their default values may be omitted from the qmconfig file.

Configuration parameters that are described below as having additive values can be set in a parameter line or as a series of lines. For example,

```
FILERULE=7
```

is identical in effect to use of three separate parameters

```
FILERULE=1
FILERULE=2
FILERULE=4
```

**AUTOLIC**

If set to a non-zero value, QM will check for licence updates a few seconds after starting the QMSvc service or qmlnxd process and then at intervals of that many days.

**CHARMAP**

Specifies the name of the character map to be loaded on a non-ECS mode system when a process starts. If not present, a default map is used.

**CLEANUP**

Interval in seconds between checks for abnormally terminated QM processes. The value must be in the range 30 to 3600 and defaults to 300 (five minutes) if this parameter is not present. See Lost Processes in Monitoring the System for more information.

**CLIPORT**

The port for incoming QMClient connections. If not present or blank the default port 4243 is used.

Setting this port to zero disables incoming QMClient connections, including use of QMNet.

Take care not to confuse this parameter with the QMCLIENT parameter.

**CMDLOG**

Sets the directory pathname for use with command logging.
CMDSTACK * Determines the default size of the command stack. The value must be in the range 20 to 999 and defaults to 99 if this parameter is not present. If modified with the `CONFIG` command, the change affects only the session in which it is applied.

CODEPAGE * Windows only. Specifies the code page to be used for QMConsole connections. If omitted, QM uses the default console code page. Note that a restriction in Windows requires that the console session is set to use Lucida Console font for this feature to work. This parameter is not applicable to ECS mode systems. See Windows Code Pages for more information.

DEADLOCK If set to 1, QM aborts any program that attempts to wait for a record lock that would result in a deadlock situation. The default value (0) allows deadlocks to occur, potentially making analysis easier. Mode 2 is like mode 1 but saves the current state of all locks in the temp subdirectory of the QMSYS account as a file named deadlock.<i>n</i> where <i>n</i> is the QM user number.

DHCACHE * Determines the size of the dynamic file cache. See Dynamic Files for more details.

DIRSELSZ * Sets the maximum number of directory file items in each stage of a partial select.

DUMPDIR * The pathname of the directory to receive process dump files. If this parameter is null, the QMSYS directory is used. On some systems, users may not have write access to this directory. DUMPDIR may be specified as a full pathname or relative to the account directory.

ECSMAP Defines a comma separated list of ECS character maps to be loaded into shared memory. The first map will be used as the default.

ERRLOG Sets the maximum size in kilobytes of the error log maintained in the errlog file in the QMSYS account directory. When the file reaches this size, the first half of the logged data is discarded. If set to zero, error logging is disabled. The minimum non-zero value is 10. A lower value will be treated as 10.

EURODATE Sets the initial state of European date format (0 = off, 1 = on). The default state if this parameter is not present is off. See also the `DATE.FORMAT` command.

EXCLREM * If set to 1, remote files are omitted from `ACCOUNT.SAVE` unless this exclusion is over-ridden by other mechanisms within the `ACCOUNT.SAVE` command processing.

FEATURE Enable a configurable feature. To enable multiple features, multiple FEATURE parameters must be provided.

The options available are

- **NO.VOC.UPGRADE** Suppress normal check for VOC changes on entry to QM after upgrading to a new release.
  When this feature is enabled, the
**UPDATE.ACCOUNT** command must be used to apply changes.

**TANDEM** Enable use of the **TANDEM** command.

**FILERULE** *

Sets rules for special filename syntax usage. This value is formed by adding together the following options as required:

1. Allow *account*:
2. Allow *server*:
3. Allow PATH:
4. Allow VFS:
5. Allow Pick style filename pathing

The **CONFIG** command can be used to modify this value within an individual process but only to remove options. Thus, the setting of this parameter in the configuration file represents the most powerful set of filename option rules that can be used.

**FIXUSERS**

Reserves a range of user numbers for exclusive use of users specifying a fixed user number when logging in using qm -n where n is the required user number.

The format of this parameter is

```
FIXUSERS=u,n
```

where

- u is the lowest user number in the reserved range.
- n is the number of user numbers to be reserved.

The highest available QM user number is normally 1023. Therefore, the value of u + n must not exceed 1024.

This feature is provided for compatibility with other environments in applications that rely on a fixed user numbers to recognise users. It is recommended that user login names should be used for this purpose in new applications as this gives a more secure system.

**FLTDIFF** *

The FLTDIFF configuration parameter determines how floating point values are compared.

Just as some numbers such as one third cannot be represented accurately in decimal, there are numbers that cannot be represented accurately in the binary notation used in computer systems. Often, numbers that are accurate in one number base, are inaccurate in the other. The inaccuracy is extremely small, typically at about the fourteenth decimal place.

A program that tests a floating point value for equality with some other value must allow for this inaccuracy rather than enforcing a strict equality. The FLTDIFF parameter determines how close two values must be to be considered as equal. The default value, 2.91E-11 (2^-35), is an industry standard but this can be set to any positive value less than one.
The format of the data in the parameter setting may be either a simple number (0.0000000000291) or a number with an exponent (2.91E-11).

FSYNC *

Additive values determining when an fsync operation is performed to flush all updated data to disk:

1. Every time that a file's header is updated. This corresponds to all structural changes within the file (overflow, split, merge, etc) and on closing the file.
2. At transaction commit.
4. After every write. This will have a very severe impact on performance.
8. Flush replication log data on every update. Changing the state of this element of the FSYNC value using the CONFIG command has no impact as it is a system wide setting.

Use of FSYNC can have a severe effect on performance but gives greater resilience to system failures.

File synchronisation always occurs on use of the QMBasic WRITESEQF or FLUSH statements regardless of the setting of this parameter.

GDI *

Setting this parameter non-zero on Windows systems causes the SETPTR command to use GDI mode by default.

GRPSIZE *

Determines the default group size in units of 1kb used when creating a dynamic file. This parameter must be in the range 1 - 8 and defaults to 1. For best performance, it should be a multiple of the operating system disk block size.

INTPREC *

Determines the rounding applied when converting a floating point number to an integer.

Just as there are numbers that cannot be written accurately in decimal such as one third, so there are numbers that cannot be stored accurately in the floating point formats used by computer systems. For example, entering a value of 17.9 will actually result in a stored value of approximately 17.8999999999999986.

The language definition for conversion of floating point values to integers states that the fractional part is discarded. To do so without rounding would mean that

\[
\text{DISPLAY INT(17.9 * 100)}
\]

would display the value 1789 rather than the more intuitively obvious 1790.

To avoid this problem, QM applies rounding to the floating point value based on the INTPREC setting when converting floating point values to integers in the INT() function or any implicit conversion such as dynamic array indices.
The parameter value identifies the decimal place at which rounding is applied. The default value is 11. Setting a value of 0 causes no rounding to be applied.

**INVCASE** *

Setting to 1 enables case inversion for terminal input, setting to 0 disables case inversion. If the parameter is not present, this feature is enabled by default. See also the **PTERM** command.

**IPADDR**

Specifies the IP address on which QM will listen for incoming network connections (telnet, QMClient, QMNet, replication). If omitted, QM listens on all available IP addresses. This parameter is most useful when a server hosts multiple separate instances of QM.

**IPV6**

Setting this parameter to a non-zero value indicates that IPV6 networking is supported by the underlying operating system and should be enabled in QM.

**JNLDIR**

Specifies the pathname of the directory used to store journalling log files. This directory will be created automatically if it does not exist.

**JNLMODE**

Additive value specifying the journalling modes to be supported:

1. Enable before imaging (allows roll back)
2. Enable after imaging (allows roll forward)

**JNLSIZE**

Sets the approximate maximum size of a journalling log file in Mb. The default value is 10. The valid range is 1 to 512.

**LGNWAIT**

Sets the maximum period in tenth of a second units for which a process will wait when attempting to login if the user limit has been reached. This can be useful, for example, with a web server application where peak loads could briefly hit the licensed user limit. The delay period can be retrieved using the QM Basic **SYSTEM(1036)** function, allowing applications to monitor the average delay and determine whether a licence with an increased user count is required. The **SYSTEM(1055)** function can be used to retrieve statistics relating to this parameter.

**LICENCE**

Licence parameters. Use the **UPDATE.LICENCE** command in the QMSYS account to apply new licence parameters.

**LPTRHIGH** *

Determines the default number of lines per page when a print unit is first referenced. This must be in the range 1 to 32767 and may be overridden using the **SETPTR** command or equivalent QM Basic print unit modification functions.

**LPTRWIDE** *

Determines the default number of characters per line when a print unit is first referenced. This must be in the range 1 to 1000 and may be overridden using the **SETPTR** command or equivalent QM Basic print unit modification functions.

**MAXCALL** *

Sets the maximum depth of nested subroutine calls including internal components of QM such as the command processor. If this limit is reached due to, for example, a program error resulting in a recursive call loop, QM will abort the program gracefully rather than failing in...
unpredictable ways when it runs out of memory. The value must be in the range 10 to 1000000 and defaults to 1000.

**MAXIDLEN**

Sets the maximum number of characters allowed in a record id. This must be in the range 63 to 255 and defaults to 63. Increasing this value increases the size of the internal lock tables. It is therefore recommended that the value used should be consistent with the needs of the application.

QM tracks the length of the longest id ever written to a file. Attempting to access a file where this exceeds the value of the MAXIDLEN parameter will cause the operation to fail. The `qmfix` utility and select operations that scan the whole file when no updates occur during the scan will correct the recorded longest id value if records have been deleted from the file.

**MAXRLOCK**

Sets the maximum number of record locks that may be held by a QM process. Attempting to lock a record when at the limit will take the ON ERROR clause of the QMBasic statement trying to acquire the lock or, if no ON ERROR clause is present, will throw a SYS.LOCKS.LIMIT exception or abort. Setting this parameter to zero (the default) disables the lock count check. The maximum value for this parameter is 100000.

**MUSTLOCK**

Setting this parameter to 1 enforces use of locks when writing or deleting records. If a program attempts to write or delete a record when it does not own a record update (READU) lock on the record or a file lock on the file being updated, the program will abort with error ER$NOLOCK. The ON ERROR clause can be used to trap this error. Leaving the parameter at its default value of 0 allows writes or deletes when no lock is in place but can lead to data integrity errors.

**NETDELAY**

Sets the period in seconds for which a connection to a QMNet server will remain open after the last file on that server connection is closed. This can improve performance in applications that repeatedly open files on the same server. This parameter defaults to zero if omitted, resulting in the server process being terminated immediately when the final file on that connection is closed. Connections are also closed on return to the command processor or any other action that flushes the file cache.

**NETFILES**

By default QM does not allow access to files on remote drives. This is because the locking system cannot detect if two systems are accessing the file simultaneously. Where it is certain that a file will never be opened from two systems concurrently, setting this parameter to 1 will enable access to remote files. Incorrect use of this feature can result in corrupt data files.

Setting NETFILES to 2 enables incoming connections from other QM servers accessing files via the QMNet interface.

These two mode settings are additive and can be used together.

**NETWAIT**

Timeout (seconds) in the range 0 to 3600 when opening a QMNet connection. A value of zero implies no limit.
NUMFILES
The maximum number of QM data files that may be opened at one time. This is a system wide limit and has a maximum value of 32767. Use of the same file by multiple users counts as one file. Attempting to open more than this number of files will cause an application to fail. Setting the parameter significantly too high may have a small performance impact. The LIST.FILES command can be used to determine whether the value of this parameter is appropriate.

NUMLOCKS
The maximum number of record locks that can be held at one time as a system wide limit. If the limit is reached, processes attempting to get locks will wait for space to become available in the lock table. Setting the parameter significantly too high may have a small performance impact. The DETAIL option of the LIST.READU command can be used to determine whether the value of this parameter is appropriate. The upper limit for this parameter is 100000.

OBJECTS *
The maximum number of programs which may be loaded into memory before discard is attempted. A program is a candidate to be discarded when it is not part of the call stack and it is not referenced from any subroutine variables from indirect calls. Setting this parameter to zero implies no limit on the number of concurrently loaded programs.

OBJMEM *
The maximum size of all loaded programs in Kb before discard is attempted. Setting this parameter to zero implies no limit on the size of concurrently loaded programs.

OPTIONS
An additive value affecting the QMSvc service (Windows), QMUSBSrvr (Windows USB installations) and qmlinx on other platforms formed from:

1  Log client disconnection. There is a small system overhead for this. (QMSvc only)
2  Do not log authentication errors in the main QM error log file.
4  Controls socket inheritance. Use only if directed by QM support. (QMSvc only)
8  Enables extended diagnostic logging.
16  Do not log successful incoming telnet connection.
32  Do not log successful incoming QMClient connection.
64  Do not log successful incoming serial port connection. (QMSvc only)

PAM
Enable (1) or disable (0) Pluggable Authentication Module support. The default setting varies by platform and is usually correct. If network logins fail, try changing this setting.

PDUMP
Additive flags to configure PDUMP (process dump) features. At this release the only flag value is

1  Ban use of PDUMP to dump processes running under other user names except when performed by a user with administrator rights.
PHANTOMS  Defines a fixed range of user numbers that are reserved for phantom processes started with the USER option to the PHANTOM command.

    PHANTOMS = u,n

where u = lowest user number, n = number of user numbers to be reserved.

POOL  Defines a connection pool.

    POOL = name,limit,timeout

where name is the case insensitive pool name of up to 15 characters, limit is the maximum number of processes to be allowed to be in the idle wait state at one time, and timeout is the period in seconds after which an idle process will terminate. A timeout value of zero implies no timeout. This parameter may be repeated to define multiple pools.

PORT  The port for incoming client telnet connections. If not present or blank, the default port 4242 is used. If QM is the only telnet style service used on the host system, it may be useful to set this to 23, the standard port used by telnet software.

    Setting this port to zero disables incoming telnet client connections. See also the PORTMAP QM configuration parameter.

PORTMAP  Allows users to create a fixed mapping between tcp/ip port numbers and QM user numbers. The format of this parameter is

    PORTMAP = p,u,n

where

    p is the lowest port number to be mapped.
    u is the lowest corresponding user number.
    n is the number of ports to be mapped.

The highest QM user number that may be mapped in this way is 1023. Therefore, the value of u + n must not exceed 1024.

Use of PORTMAP does not prevent users entering QM via the normal shared port defined by the PORT configuration parameter.

This feature is provided for compatibility with other environments in applications that rely on a fixed port number to user number relationship to recognise users. It is recommended that user login names should be used for this purpose in new applications as this gives a more secure system.

This feature is not supported on Windows USB installations of QM.

PWDELAY  The pause in seconds between successive attempts to enter the username and password on network connections. This parameter defaults to 5.

QMCLIENT  Provides additional security control for QMClient sessions. This parameter has one of three values:

    0  No restrictions.
1 Bans use of **QMOpen()** and **QMExecute()**, limiting clients to calling subroutines.

2 In addition to the level 1 restrictions, **QMCall()**, **QMCallx()** and **QMCreateObject()** can only be used to call subroutines or class modules compiled with the **$QMCALL** compiler directive.

This parameter can be modified to a higher level in an individual process using the **CONFIG** command but cannot be taken to a lower level in this way.

**RECCACHE** *

Sets the size of the record cache (default zero, maximum 32). When a QM process reads a record, a copy of this record is retained in the cache. A subsequent read for the same record can find it from the cache rather than requiring an operating system call. The cache mechanism is most likely to benefit an application that makes heavy use of the **TRANS()** function to read the same record many times during a long query, for example when looking up a tax rate that is applied to every record processed. Writing or deleting records will invalidate the affected cache entries in all processes.

**REPLDIR**

Pathname of directory to be used for **replication** log files on the publisher system. This directory will be created automatically when QM is started if it does not already exist.

**REPLMAX**

Sets the limit on the number of simultaneous replication targets that may be active from a publisher. This parameter must be in the range 1 to 255 and defaults to 8.

**REPLPORT**

Port number on which QM will listen for incoming network connections from a **replication** subscriber system. Use of the default port 4244 is recommended as this is also the default port used by the subscriber.

**REPLSIZE**

Sets the approximate maximum size of a replication log file in Mb. The default value is 10. The valid range is 1 to 512.

**REPLSRVR**

Server name of a **replication** subscriber system. This must match the name used on the publisher when setting up replication.

**RETRIES**

The maximum number of attempts allowed for entry of a valid username and password on network connections. This parameter defaults to 3.

**RINGWAIT** *

QM uses a ring buffer to hold type-ahead characters received from the keyboard. If this becomes full, incoming data is thrown away and a bell character is sent back to the terminal. Some applications may need to send a large burst of data which would fill the ring buffer and hence be truncated. Setting the RINGWAIT parameter to 1 causes QM to wait for space to become available in the buffer rather than rejecting input. Enabling this feature could result in the inability to use the break key if the ring buffer is full. The AccuTerm terminal emulator requires this parameter to be set to 1. (This parameter currently only applies to the Windows version of QM).
SAFEDIR *

Setting this parameter to 1 causes QM to adopt a careful update process when writing records to directory files. The new record is written to a temporary file, the old record is deleted (if it exists) and the temporary item is renamed to replace it.

This mechanism results in reduced performance but ensures that the original data is not lost if the write fails because, for example, there is insufficient disk space available.

SECURITY

Additive value controlling system wide security options.

1 Query processor EVAL construct requires write access to the file's dictionary. Note that this is the file's dictionary regardless of possible use of the USING clause to select an alternative dictionary.

2 Do not allow SET.PRIVATE.SERVER to create a definition that has the same name as a public server.

SERIAL

Defines a serial port to be monitored for incoming QM connections. Multiple SERIAL parameters may be specified. The format is

SERIAL=port,rate,parity,bits,stop

where

- port = the port name (e.g. COM1)
- rate = baud rate (e.g. 9600)
- parity = none(0), odd(1), even(2)
- bits = bits per byte (7 or 8)
- stop = number of stop bits (1 or 2)

SH *

(Not Windows) Determines the shell processor and its options to be used when the SH command is used to start an interactive shell. If not set, this parameter defaults to "/bin/sh -i" on FreeBSD and "/bin/bash -i" on all other Unix/Linux based systems.

SH1 *

(Not Windows) Determines the shell processor and its options to be used when the SH command or the QMBasic OS.EXECUTE statement is used to execute a single command. If not set, this parameter defaults to "/bin/sh -c" on FreeBSD and "/bin/bash -c" on all other Unix/Linux based systems.

SORTMEM *

The size in kilobytes at which a sort switches from memory based to disk based. This value is per user and defaults to 4096 (4Mb). Setting values lower than this may lead to poorer performance unless you are severely restricted by memory size. Setting values larger than this will require more memory for large sorts.

SORTMRG *

A disk based sort produces a series of intermediate files that must be merged to produce the final result. The SORTMRG parameter specifies the number of files merged in each pass. This must be in the range 2 to 10 and defaults to 4. The effect of changes to this parameter on sort times is dependant on the relative performance of the disk and processor.
SORTWORK *
The pathname of the directory to hold temporary sort workfiles. These are automatically deleted on normal completion of a sort. If this parameter is null or the specified directory does not exist, the directory defined by the TEMPDIR parameter is used.

SPOOLER *
Sets the name of the default spooler on non-Windows platforms. If this parameter is not specified or is a null string, the lp spooler is used. The name specific may be another standard spooler (e.g. lpr) or a user written program or shell script to perform custom print management.

The qualifying data to this configuration parameter can include other options to be passed to the selected spooler.

See the description of the $SPOOLERS control record in the section on printing.

SRVRLOG
Sets the maximum size of the log file (QMSvc only). A value of zero causes QMSvc to start a new log each time the service is started, saving the previous log as QMSvcLog.old. A non-zero value sets the maximum size of the QMSvc.log file in kb. When this size is reached, the first half of the data in the file is removed.

STARTUP
Sets a command to be executed when QM starts. This may not contain double quotes. See Startup for more details.

TEMPDIR *
The pathname of the directory to hold temporary files. These are normally automatically deleted when no longer required but it is recommended that this parameter points to a directory that is cleared on restart of the system so that any files left behind at a system failure will be deleted. If this parameter is null or the specified directory does not exist, QM uses the TEMP subdirectory of the QMSYS account on Windows or the operating system temporary directory on other platforms. Applications may find the location of this directory using CONFIG("TEMPDIR") or SYSTEM(38) and may use it to store temporary items but developers should be careful not to overwrite items created by other software. Internal parts of QM will never create items that have a leading underscore in their name.

TERMININFO *
The pathname of the directory holding the terminfo database. This defaults to a subdirectory named terminfo under the QMSYS directory.

TIMEOUT
The maximum wait period in seconds allowed during entry of a valid user name and password on network connections. This parameter defaults to 30.

TIMEZONE *
The time zone to be used by the epoch conversion code. The value of this parameter must match the format required for the operating system TZ environment variable. The default value is taken from the TZ environment variable on entering QM. If this variable is not defined, the time zone of the process that started QM is used.

TXCHAR *
Windows only. Determines whether QM uses the OEM character set translation functions on terminal i/o. The default value is 1 (use
translation). A value of zero disables translation. This parameter is ignored on ECS mode systems.

UMASK
Linux and Unix only. Sets the default umask value for network connections to QM that do not enter via the operating system shell (direct telnet, QMClient, QMNet). The mask is specified as an octal value as in the operating system umask command. If this parameter is not present, a default of 002 is used.

USERPOOL
Determines the maximum user number that will be allocated by QM, default value 1023, maximum 16383. The actual maximum user number will be the greater of the value of this parameter and the number of simultaneous processes permitted by the licence. The default behaviour of QM is to rotate through the available user numbers, thus not usually reusing a user number for some time. Setting this parameter to zero causes QM to use the lowest available user number when logging in a new session.

USPLIT
If set to 1, ignores the FIXUSERS, PHANTOMS, PORTMAP and USERPOOL parameters and adopts a system under which interactive terminal users are allocated the lowest available user number and all other processes are allocated the highest available user number in a range that matches the maximum number of concurrent users for the system licence.

VFSCACHE *
Sets the size of the file cache for the Virtual File System.

YEARBASE *
The earliest year in the 100 year range of dates entered with two digit year numbers. This parameter is optional and defaults to 1930.
8.2 The Terminfo Database

Control sequences and other characteristics of terminal devices are defined in the terminfo database. This is normally a subdirectory structure under the QMSYS account but can be moved elsewhere if required by use of the TERMININFO configuration parameter. The structure of the terminfo database closely mimics that found as a standard component on Linux and other operating systems though QM has some private extensions to the internal library format.

Location and Structure

The terminfo directory contains a set of subdirectories named using the first character of the terminal types stored within them. Each of these directories then contains a file for each terminal type. Thus, for example, the definition for a vt100 terminal on a Windows system with the default QMSYS location would be found in c:\qmsys\terminfo\v\vt100.

The definition files are stored in an encoded form that closely reflects the way in which QM uses the data internally. QM provides a utility, qmtic, to compile or decompile terminfo entries.

A master source for a variety of terminals is in the QMSYS account directory as terminfo.src and the entire set of definitions is compiled when QM is installed. To simplify maintenance of a private set of new or modified terminal definitions, the QM installation process will look for a file named terminfo.mods in the QMSYS account directory and, if it exists, will compile it after the standard source, allowing new entries to be added or standard entries to be modified.

Linux users wishing to transfer entries from the standard Linux terminfo database to the QM terminfo database should use the Linux infocmp tool to decompile the Linux definition and then recompile it using qmtic, removing any entries that are not supported on QM.

Source Format

Terminfo entries contain three types of item; Booleans, numbers and strings.

A Boolean item is present in the terminfo entry if the feature or capability that it represents is supported by the terminal. QM currently does not make use of any of the Boolean items.

A number entry holds the value of an integer numeric parameter. For example, the cols item defines the normal number of columns per line.

A string item holds a control string. These may be codes to be sent to the terminal to perform a specific task such as clearing the screen or moving the cursor, or may be a code sent by the terminal when a specific key is pressed by the user. Strings representing commands sent to the terminal device often include parameterised information such as screen positions or counts.

A terminfo source file consists of one or more terminal definitions separated by at least one blank line. Lines commencing with a hash character (#) are comments and are totally ignored during compilation. Each definition consists of a number of comma separated items. A definition can be split over multiple lines by inserting a newline after a comma.

The first line of each entry contains a list of terminal types defined by that entry and a text description. For example:

```
vt100|vt100-am|dec vt100 (w/advanced video),
```

This line must not start with a space or a tab and is separated into a number of fields using the vertical bar (|) character. The last field is the text description. All preceding fields are terminal device
names. Thus, the entry introduced by the line shown above defines the vt100 and vt100-am terminal types. There will be separate compiled files for each of these terminals in the final terminfo database.

The remaining lines of the entry must be indented by at least one space or a tab and define the characteristics of the device. Although the order is not fixed, terminfo entries normally have the Booleans first, followed by the numbers, followed by the strings.

A Boolean entry consists only of its name. A number consists of the name, a hash character, and the value of the parameter. A string consists of the name, an equals character (=), and the value of the parameter.

QM extends the terminfo source format to allow multiple strings in a single capability setting. In this case, the entire set of alternative strings for the capability are enclosed in curly brackets and the individual strings inside the brackets are separated by commas. Each comma may be followed by spaces to improve readability.

For example, part of the vt100 definition is:

```
vt100|vt100-am|dec vt100 (w/advanced video),
am, xenl, msgr, xon,
cols#80, it#8, lines#24, vt#3,
bel=^G, cr=r, csr=\E[%i%p1%d;%p2%dr], tbc=\E[3g,
clear=\E[H\E[J$<50>, el=\E[K$<3>, ed=\E[J$<50>,
cup=\E[%i%p1%d;%p2%dH$<5>, cudl=n, home=\E[H,
kmous={\E[101~, \E[111~}
```

String tokens may contain the following special character sequences:

| b | Backspace (char 8) |
| e | Escape (char 27)  |
| f | Formfeed (char 12) |
| l | Linefeed (char 10) |
| m | Linefeed (char 10) |
| r | Carriage return (char 13) |
| s | Space (char 32) |
| t | Tab (char 9) |
| ^ | Caret (^) |
| \ | Backslash (\) |
| ^x | Parameter action as described below |
| %= | Percent sign (%) |
| $<n> | Insert an n millisecond delay. This code is ignored by QM, removing it from the final string. |

The %= parameter notation performs run time manipulation of the character string, often inserting parameter values. These operations use a stack for intermediate results and are described in terms of their C programming language equivalents:

```%c pop top stack item and print it as a character (like %c in printf())
%d pop top stack item and print it as an integer (like %d in printf())
%s pop top stack item and print it as a string (like %s in printf())
%[[:flags][width][precision]][d][x][X] as in printf, flags are [+-#] and space. The \' is used to avoid making %+ or %- patterns (see below).
%p[1-9] push ith parm
%[a-z] set dynamic variable [a-z] from top stack item
%[a-z] get dynamic variable [a-z] and push it onto stack
%[A-Z] set static variable [A-Z] from top stack item ```
%g[A-Z]  get static variable [A-Z] and push it onto stack
%l    replace topmost stack item with its string length
%c'   push char constant c
%{nn}  push integer constant nn
%+    replace top two stack items with their sum
%-    replace top two stack items with their difference
%*    replace top two stack items with their product
%/    replace top two stack items with their quotient
%m    replace top two stack items with the remainder from division
%&    replace top two stack items with their logical AND
%|    replace top two stack items with their logical OR
%^    replace top two stack items with their logical exclusive OR
%=    replace top two stack items with the result of an equality test
%>    replace top two stack items with the result of a greater than test
%<    replace top two stack items with the result of a less than test
%A %O  logical and & or operations for conditionals
%!    replace top stack item with its logical inverse
%~    replace top stack item with its bitwise inverse
%1    add 1 to first two parms (for ANSI terminals)
%@ expr %t thenpart %e elsepart %;
      if-then-else, %e elsepart is optional.

For those of the above operators which are binary and not commutative, the stack works in the usual way, with
%gx %gy %m
resulting in x mod y, not the reverse.

For example, the QMBasic @(col,row) function translates to the cup (cursor position) terminfo entry. For the vt100 definition shown above this is

cup=\E[%i%p1%d;%p2%dH$<5>

Taking this apart, element by element for a usage as @(10,5):
\E  Escape character
[  [ character
%1  Increment both arguments to allow for positions numbered from 1 rather than 0. The argument values 10 and 5 thus become 11 and 6.
%p1  Push parameter 1 (11) onto the stack
%d  Print top item from stack as an integer
;  ; character
%p2  Push parameter 2 (6) onto the stack
%d  Print top item from stack as an integer
H  H character
$<5> Delay - Ignored by QM.

The end result is thus "Esc[11;6H".

**Mouse Protocols**

The "mouse" numeric token defines the protocol used by the terminal when sending a mouse click. Values are:

1  QMTerm
2  PDA
3  AccuTerm ASCII format
4  XTERM old format (Anzio)
5 AccuTerm ANSI format

The associated "kmous" token defines the prefix that identifies a mouse click. The "mouse-on" and "mouse-off" tokens define the codes to enable and disable the mouse.

Colour Mapping

Different terminal emulators use variations on the numeric values used to represent colours (see the QMBasic @(-37) and @(-38) functions). To enable users to employ a consistent set of colour values in application programs whilst working with different terminal emulators, the terminfo definition may include an optional element named colourmap (British spelling) that provides a translation between internal colour values and the actual colour number transmitted to the terminal. The format of this entry is

```
colourmap=0|1|2|3|7|5|6|4|8|9|10|11|12|13|14|15
```

where the elements correspond to internal colour values zero upwards and the number in each element is the colour value to be sent to the terminal. In this example, colours 4 and 7 have been swapped.

Colours for which the value is not to be changed may be left blank and trailing unchanged values may be omitted. Thus, the above example could be shortened to

```
colourmap=|||7|||4
```

User Definable Entries

The terminfo database includes 10 entries (u0 to u9) for user use. QM pre-defines the function of two of these, the remaining eight are available for any purpose that the user wishes.

<table>
<thead>
<tr>
<th>u0 - u7</th>
<th>@(-100) to @(-107)</th>
<th>Undefined. Users may adopt these for any purpose.</th>
</tr>
</thead>
<tbody>
<tr>
<td>u8</td>
<td>@(-108)</td>
<td>IT$ACMD Asynchronous command execution prefix. This code prefixes a command to be executed on the client system followed by a newline. The QM session is not suspended while the command is executed.</td>
</tr>
<tr>
<td>u9</td>
<td>@(-109)</td>
<td>IT$SCMD Synchronous command execution prefix. This code prefixes a command to be executed on the client system followed by a newline. The QM session is suspended while the command is executed.</td>
</tr>
</tbody>
</table>

The u8 code is used internally by some parts of QM. The remaining codes will only be used as defined in user written application software.

AccuTerm Extensions

The AccuTerm terminal emulator includes support for additional special functions that are not part of the terminal definitions for industry standard terminal types. These extra functions include

- Client side command execution (synchronous and asynchronous)
- Screen region save and restore (used by the QMBasic debugger)
- Mouse click detection
QM ships with extended definitions for some devices using terminal type names with a -at suffix (e.g. vt100-at). Users can easily add similar extensions to other terminal definitions. If the host type in AccuTerm is set to QM, the correct extended terminal type should be selected automatically for direct network connections to QM.
The qmtic Utility

The qmtic tool can be used to compile new terminal definitions or to decompile existing ones. Although it is possible to store separate source definitions of each terminal type, QM includes a single master source file, terminfo.src, in the QMSYS directory. To simplify maintenance of a private set of new or modified terminal definitions, the QM installation process will look for a file named terminfo.mods in the QMSYS account directory and, if it exists, will compile it after the standard source.

The format of the qmtic command to compile terminfo data is:

```
qmtic {options} src...
```

where

- `src...` is a list of one or more source files to be processed.
- `options` are any of the following:
  - `-ppath` Use the terminfo database at the specified path.
  - `-tname` Compile only the specified terminal definition. This option may be repeated to compile several definitions.
  - `-v` Verbose mode. Displays progress information.
  - `-x` Do not overwrite existing entries. Only definitions for terminals not already in the database will be written.

The format of the qmtic command to decompile terminfo data is:

```
qmtic {options} -d name...
```

where

- `name...` is a list of one or more terminal names to be processed.
- `options` are any of the following:
  - `-nname` Show only the named capability. May be repeated.
  - `-ppath` Use the terminfo database at the specified path.
  - `-v` Verbose mode. Displays progress information.

Replacing the `-d` option with `-dall` will decompile all terminfo entries to produce a new master source file.

The format of the qmtic command to display an index of terminal types for which compiled definitions are present is:

```
qmtic {-ppath} -i
```
8.3 Windows Code Pages

This section does not apply to the ECS mode version of QM.

Within Windows QMConsole sessions, single-byte characters are displayed as defined by the current codepage. The default page is set by Windows based on your regional settings but it can be changed at any time from the QM command prompt, within a paragraph, or by EXECUTE within a Basic program by updating the CODEPAGE private configuration parameter or by use of the PTERM command.

Use of code pages allows language localisation. Single-byte characters in the range 127 to 255 are mapped to accented letters (such as é and ô), special characters (such as ç and ß), and symbols (such as the Euro and Trademark signs). There are not enough single-byte characters for all the languages of the world, so different mappings have been set up for different regions. Microsoft calls these mappings code pages.

Word processing software packages such as Notepad handle the choice of code page through the choice of font. Fonts have been built for many languages, some provided with Windows, others available for download from Internet sites. Choose the correct font, and the special characters will display as needed.

This method cannot be used by software such as QM which runs as a Windows console process either directly or via the Windows command prompt. Within processes run in this way, there is just one font (Lucida console) available for use with multiple languages. The characters displayed are determined by which code page has been made current. Any given installation of Windows has a default code page, and several other code pages available. More information can be found on Internet sites that specialise in language issues.

Note that selection of a codepage controls character display within QM – how characters appear at the command prompt, in LIST output, within ED and SED, etc. A file written by a QM program, when viewed in a Windows application such as Notepad or Excel, will show the characters as seen by that Windows application.

Code pages are identified by numbers, usually of 3 or 4 digits. The codepage used by QM can be controlled by use of the CODEPAGE configuration parameter. Setting a value for this parameter in the QM configuration file will determine the default page used. This can be changed within a session by use of CONFIG or PTERM.

Some of the pages that may be available in a Windows system include:

<table>
<thead>
<tr>
<th>Codepage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>850</td>
<td>&quot;Multilingual (Latin-1)&quot; (Western European languages)</td>
</tr>
<tr>
<td>852</td>
<td>&quot;Slavic (Latin-2)&quot; (Eastern European languages)</td>
</tr>
<tr>
<td>855</td>
<td>Cyrillic</td>
</tr>
<tr>
<td>857</td>
<td>Turkish</td>
</tr>
<tr>
<td>1250</td>
<td>East European Latin</td>
</tr>
<tr>
<td>1252</td>
<td>West European Latin</td>
</tr>
<tr>
<td>1253</td>
<td>Greek</td>
</tr>
<tr>
<td>1257</td>
<td>Baltic</td>
</tr>
<tr>
<td>1258</td>
<td>Vietnamese</td>
</tr>
</tbody>
</table>

To find out whether your copy of Windows has a particular code page available, go into the command prompt and enter “chcp codepage”. This will return an error message if codepage is not available.

To view the page in use by a QM session, execute either of the following commands:

PTERM DISPLAY
PTERM CODEPAGE

To change the code page in a QM session, execute either of the following commands:

```bash
CONFIG CODEPAGE page
PTERM CODEPAGE page
```

Use of some code pages may also require the `TXCHAR` configuration parameter to be set appropriately. This parameter controls whether QM uses the OEM character translation modes of Windows. It defaults to 1 (use translation) but can be turned off for the current process using the `CONFIG` command or for all processes by setting parameter to zero in the configuration file.

If there is a need to display special characters that are not supported by any of the available code pages, it is possible to modify the Lucida Console font to display them. This can be done by the following steps:

- Obtain a font editor (e.g. FontCreator) from a supplier who specializes in font support. The font editor needs to have the ability to associate the graphic form of a character - called its "glyph" - with the byte that is to represent that character - called its "mapping".
- Find the existing Lucida Console font, which a file, probably named "lucon.ttf", located in the Fonts folder. The font name is one of the data items within the font file, so is distinct from the name of the file. Windows can see the file as Lucida Console, even though it may have been given another file name.
- Copy the original file to another folder, to keep a backup in case something goes wrong with any of the rest of these steps. Then copy it to another file name, such as "myfont.ttf".
- Look in the character mapping of "myfont.ttf" for characters that are not needed. The "per thousands" sign, dagger, and double dagger are possible examples.
- Using the font editor, replace the glyphs of the characters that are not needed with the glyphs of the characters that are needed.
- Test "myfont.ttf" by copying it into the Fonts folder. Then go into QM Console, do LIST, ED, etc., to confirm that characters display correctly. Any Windows software that gives a choice of fonts should display the same way when Lucida Console is specified. This would include Notepad, word processing, and spreadsheet products.

Like any other software, the font needs to be carefully backed up, and shared with whatever users may need it. Fonts are subject to copyright, so property rights must be respected. It may be necessary to get legal advice.
8.4 Application Level Security

User Registration

QM includes an internal security system that imposes restrictions such that a user can only access QM if their user name is registered using the CREATE.USER or ADMIN.USER command. Users who are system administrators within the underlying operating system are not restricted. When installing a new QM system, the installer will ask whether the security system is to be enabled. This choice may be amended later using the SECURITY command.

On Windows USB installations of QM, the QMUSBSrvr process does not have access to an adequate user authentication mechanism. If security is enabled, users entering QM over a direct network connection will be required to enter a valid user name and password that is private to the QM environment and is created using the CREATE.USER or ADMIN.USER commands. If security is not enabled, network users can connect directly to QM with no authentication. Entry to QM from the operating system command prompt or by use of the QMConsole menu item is unrestricted.

On other installation types, users entering QM directly from a network will always be required to enter a valid user name and password known to the operating system. Furthermore, if security is enabled, this user name must also be in QM's own register of user names. If there is no entry in the user register for the user, the system looks for an entry for DEFAULT (case sensitive) and, if found, uses this to define the user's rights. Entry to QM from the operating system command prompt (or the QMConsole menu option on Windows) will be validated against the user name used to log into the operating system. If security is not enabled, any user with a valid operating system user name will be able to enter QM.

A user can be assigned to a QM user group. User groups are much like operating system groups found in Linux but are a totally separate internal system. The group name of a QM process can be found using the @QM.GROUP variable.

The details set for a user via ADMIN.USER also include options to enable or disabled specific features of QM that may have a bearing on system security. These are:

- **Create/delete accounts**: Controls use of CREATE.ACCOUNT, DELETE.ACCOUNT and entering QM in a directory that is not already set up as an account.
- **Define private servers**: Controls use of SET.PRIVATE.SERVER.
- **Command prompt access**: If this option is not set, the user is denied access to the interactive command prompt unless they are an administrator.

The restrictions listed above apply regardless of whether security is enabled. A user register entry will be created automatically for a user entering QM on a system with security disabled who does not already appear in the user register. This entry will have all features controlled by security options enabled.

Account Restrictions

QM provides the ability for restrictions to be placed on which accounts a user may go into. The ADMIN.USER command can be used for each defined user name to create a list of accounts that is either those accounts that the user may access or those accounts that the user may not access. This mechanism is only meaningful when user level security is active (see the SECURITY command). Users with administrator rights are not subject to account restrictions.
Account restrictions affect:

- Use of the **LOGTO** command.
- Use of the `-A` option to the `QM` command to select an account.
- Direct entry to `QM` from the operating system shell within an account directory.
- `QMClient` connections.
- Use of Q-pointers to access files in other accounts.

For maximum security:

- Non-administrative users should be denied write access to the `ACCOUNTS` file in the QMSYS account otherwise they could set up alternative references to restricted accounts.
- Restricting write access to the `VOC` file prevents creation of pointers to files in other accounts. Note, however, that some applications need write access to the `VOC`. Account barring of Q-pointers should provide effective security.
- Ensure that the **FILERULE** configuration parameter does not allow the user to construct extended syntax filenames that access files in other accounts. The value of this parameter in the configuration file determines the initial rights but it can be reduced (typically in the `LOGIN` paragraph) for specific users or accounts.
- Use the "Ban account creation" option of the **ADMIN.USER** screen to disable creation of new accounts by specific users.

**Application Security**

A well designed application never allows an end user to reach a command prompt. This leaves restriction of what a user may do within the control of the application itself. Where it is necessary to provide differing levels of access to different users, `QM` provides several ways to identify attributes of the current user:

- **@IP.ADDR** User's IP address for network connections. This is also available in QMBasic as **SYSTEM(42)**.
- **@LOGNAME** User's login name.
- **@TTY** Terminal device name.
- **SYSTEM(1017)** Port number for network connection.
- **SYSTEM(27 to 30)** UID, effective UID, GID and effective GID (not Windows)

A user could potentially escape from the controlled environment provided by the application if the application were to abort. This can be avoided by a combination of the following techniques:

Disable the break key. The break key is automatically disabled until completion of the `LOGIN` paragraph unless it is enabled within that paragraph by use of the **BREAK ON** command. There should never be a need for use of the break key in a working application. In the unlikely event of needing to re-enable it for a specific user as a result of an application fault, the system administrator can use an extended form of the **BREAK** command to do this.

Use **OPTION NO.USER.ABORTS** to suppress all options through which a user can cause an abort to occur. This removes the `A` option from the "Press return to continue" prompt, the query processor pagination prompt and the break key options.

Implement the **ON.ABORT** paragraph. Despite the above techniques, an application may still abort as a result of a run time error or use of the **ABORT** statement within the application itself. When an abort occurs, `QM` discards all programs, menus, paragraphs, etc that are running in the process and returns to the lowest level command processor. Before this displays
the command prompt, it checks in the VOC file for an executable item named **ON.ABORT**
and, if this is found, executes it. A typical **ON.ABORT** paragraph terminates the user's session
after, perhaps, logging the incident.

Some users such as application developers may need to be able to reach a command prompt. In this
case, **security subroutines** can be attached to R or V-type VOC entries to provide control over what
can be done.

**See also:**

Permissions
8.5 Permissions

QM uses the underlying operating system to manage processes, files, devices, etc. Therefore, all issues of access permissions ultimately lie with the operating system. This section gives some guidance on setting permissions within a QM system but individual application needs should be taken into account.

As an aid to establishing good security policies on Linux and Unix systems, the `CREATE.FILE` and `CREATE.ACCOUNT` commands include options to set the ownership and permissions of the newly created item.

The QMSYS Account

The only users who should be working in the QMSYS account are system administrators. It is reasonable that these people should have write access to QMSYS. No other user ever needs to create a new item in the QMSYS directory itself. Therefore the directory can be protected so that only administrators can write to it.

System administrators need write access to all items in the QMSYS account. The following table sets out the additional access rights needed for other users.

<table>
<thead>
<tr>
<th>Item</th>
<th>Developers</th>
<th>Others</th>
</tr>
</thead>
<tbody>
<tr>
<td>$FORMS</td>
<td>Form queue definitions created with <code>SET.QUEUE</code> for use with <code>SP.ASSIGN</code></td>
<td>Full</td>
</tr>
<tr>
<td>$HOLD</td>
<td>Hold file for QMSYS account</td>
<td>None</td>
</tr>
<tr>
<td>$HOLD.DIC</td>
<td>Dictionary for $HOLD</td>
<td>None</td>
</tr>
<tr>
<td>$IPC</td>
<td>Inter-process communication file</td>
<td>Full</td>
</tr>
<tr>
<td>$LOGINS</td>
<td>User name database</td>
<td>Full</td>
</tr>
<tr>
<td>$MAP</td>
<td>Catalogue map</td>
<td>Full (note 1)</td>
</tr>
<tr>
<td>$MAP.DIC</td>
<td>Dictionary for $MAP</td>
<td>Read</td>
</tr>
<tr>
<td>$SCREENS</td>
<td>Screens database</td>
<td>Read</td>
</tr>
<tr>
<td>$SERVERS</td>
<td>QMNet server register</td>
<td>Read</td>
</tr>
<tr>
<td>$SVLISTS</td>
<td>$SAVEDLISTS file</td>
<td>None</td>
</tr>
<tr>
<td>$VAULT</td>
<td>Encryption key vault</td>
<td>Read</td>
</tr>
<tr>
<td>ACCOUNTS</td>
<td>Accounts database</td>
<td>Read (note 2)</td>
</tr>
<tr>
<td>ACCOUNTS.DIC</td>
<td>Dictionary for ACCOUNTS</td>
<td>Read</td>
</tr>
<tr>
<td>File</td>
<td>Description</td>
<td>Developers</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------------------------</td>
<td>------------</td>
</tr>
<tr>
<td>audit.log</td>
<td>Encryption audit log</td>
<td>None</td>
</tr>
<tr>
<td>bin</td>
<td>Executable files</td>
<td>Read</td>
</tr>
<tr>
<td>BP</td>
<td>Sample QMBasic items</td>
<td>Read</td>
</tr>
<tr>
<td>cat</td>
<td>Private catalogue of QMSYS account</td>
<td>None</td>
</tr>
<tr>
<td>DICT.DIC</td>
<td>Dictionary for dictionaries</td>
<td>Read</td>
</tr>
<tr>
<td>DIR_DICT</td>
<td>Dictionary for directory files</td>
<td>Read</td>
</tr>
<tr>
<td>DOCS</td>
<td>Documentation (Windows only)</td>
<td>Read</td>
</tr>
<tr>
<td>errlog</td>
<td>Optional error log file</td>
<td>Full (note 3)</td>
</tr>
<tr>
<td>ERRMSG</td>
<td>Pick style error message file</td>
<td>Read (note 4)</td>
</tr>
<tr>
<td>ERRMSG.DIC</td>
<td>Dictionary for ERRMSG</td>
<td>Read</td>
</tr>
<tr>
<td>gcat</td>
<td>Global catalogue</td>
<td>Full</td>
</tr>
<tr>
<td>MESSAGES</td>
<td>Message database</td>
<td>Read</td>
</tr>
<tr>
<td>NEWVOC</td>
<td>Template VOC file</td>
<td>Read</td>
</tr>
<tr>
<td>qmconfig</td>
<td>Configuration parameter file</td>
<td>Read</td>
</tr>
<tr>
<td>qmterm.ini</td>
<td>qmterm configuration parameters</td>
<td>Full</td>
</tr>
<tr>
<td>QM.VOCLIB</td>
<td>VOC extension</td>
<td>Read</td>
</tr>
<tr>
<td>stacks</td>
<td>Command stack repository</td>
<td>None</td>
</tr>
<tr>
<td>SYSCOM</td>
<td>System include records</td>
<td>Read</td>
</tr>
<tr>
<td>temp</td>
<td>Temporary directory</td>
<td>Full</td>
</tr>
<tr>
<td>terminfo</td>
<td>Terminofo database</td>
<td>Read</td>
</tr>
<tr>
<td>terminfo.src</td>
<td>Terminofo definitions</td>
<td>None</td>
</tr>
<tr>
<td>VOC</td>
<td>Vocabulary file</td>
<td>Read</td>
</tr>
<tr>
<td>VOC.DIC</td>
<td>Dictionary for VOC</td>
<td>None</td>
</tr>
<tr>
<td>errlog</td>
<td>Error log</td>
<td>Full</td>
</tr>
</tbody>
</table>
1. Write access to $MAP is only needed by users who execute the MAP command to create a catalogue map with the default destination file name.

2. Any user who is to be allowed to create new accounts will need write access to this file. Restricting write access on this file closes a potential security risk by preventing users creating synonyms to existing accounts that might subvert application level security mechanisms.

3. If error logging is enabled (see the ERRLOG configuration parameter), all users need full access to the optional errlog file. Any user that does not have write access will not log errors.

4. This file contains standard Pick style messages. Although rare, some applications may write to this file.

5. It is possible to restrict access to individual items in the gcat subdirectory. Users need read access (not execute access) to run a compiled QMBasic program.

**Application Accounts**

In general, users should have free access to all files. Taking write access away on the VOC can be used to prevent users modifying its content but beware that some applications modify the VOC as part of their normal operation.

**See also:**

*Application level security*
8.6 Backup and Restore

QM does not provide any special backup and restore utilities but relies instead on use of standard operating system level backup tools. This section sets out some points to consider in planning a backup strategy. Hopefully, you have already thought of these...

- Determine what to backup. If your backup needs to be as quick as possible, remember that it is usually unnecessary to back up system files that can easily be recreated.

- Do not forget that distributed applications sometimes have critical data stored on client PCs. These may need to be backed up at the same time as the server to preserve data integrity across the entire backup.

- How often will you back up? You need to make a sensible trade-off between the time it takes to backup and the difficulty of bringing the system up to date in the event of a restore.

- Consider when to backup in relation to your business routine. It is unsafe to backup a database while it is being used except as described below. You will end up with data integrity problems and, possibly, structural integrity problems in files that were being modified while the backup progressed. The safest approach is to log all users off while the backup is performed. It is not necessary to shutdown QM.

- Use a cycle of backup media rather than continuously overwriting the same media so that you are secure from failures during the backup and also have multiple points in time to which you can revert.

- Ensure that your backup media is kept away from the system that it represents. A fireproof safe or an offsite store is best.

- Think carefully about how long you will keep your backups. There are often legal requirements to be able to restore business data for several years. Remember that technology moves on at a rapid pace. You need to ensure that you still have the appropriate drives to read your old backups.

- Test your backups. Check that you really can restore your data if the need arises.

Backup Tools

The QM file system uses operating system level file structures to represent all aspects of the database. An account is represented by an operating system directory. Files within the account are represented by a further layer of subdirectories though it is possible to relocate these, typically for load balancing across disk drives. For a hashed file, the subdirectories contain files that correspond to the primary, overflow and index portions of the QM file. For a directory file, the subdirectories contain an individual text file for each record. There may also be other subdirectories such as the private catalogue and the savedlists file. The QMSYS account in particular has many additional files and directories. For further information on the file system structure, see The QM File System.

Because QM's file system is visible at the operating system level rather than being stored in raw partitions or similar structures, standard operating system backup tools can be used to archive data. This could be by a simple file to file copy to a separate disk drive or by merging files into a composite save using tools such as Linux's cpio or tar. More advanced tools are available.

QM has versions of the Pick style tools such as T.DUMP, ACCOUNT.SAVE and FILE.SAVE, however use of these for system backup is strongly discouraged because the media format (defined by Pick long ago) is not able to store some aspects of QM files such as information relating to hashed file configuration, triggers, alternate key indices, encryption, replication or Unicode data on an ECS.
mode system. There are also some limitations on saving certain character sequences that may validly appear in binary data such as compiled QMBasic programs or dictionary items. These tools were included in QM primarily as a way to take data back to a Pick style system during migration of an application.

**Adopting a Safe Backup Procedure**

It is essential that the backup procedure creates a valid representation of the state of the system. Unless tools that create a moment in time snapshot of the system are used, it is essential that no users are modifying the database while the backup is in progress otherwise data saved early in the process may be inconsistent with data saved later. For example, backup of a financial application at the point when money is being moved from one account to another could save the two records in a sequence where, if the data is restored, the money appears in both accounts or in neither account. More seriously, similar timing issues may result in structural inconsistencies in the saved file if internal pointers have changed during the backup.

The safest way to avoid this sort of timing issue is to get all users off the system for the duration of the backup but this may not be practical, particularly in a 24x7 application. Even with snapshot backup software, the backup could occur at a time when, for example, the financial application mentioned above has written one record but not yet written the other.

Other possible strategies include:

- **Backup everything twice.** The chances of a structural inconsistency on both saves affecting the same record are very low but it may be much harder to work out which data needs to be restored to recover a consistent view of the system.

- **Suspend updates.** QM includes the ability to suspend disk updates (qm -suspend). When this is used, any application attempting to write to a QM file will stall until updates are re-enabled (qm -resume). This approach guarantees structural integrity of the files but does not guarantee data integrity unless the application uses transactions. The suspension needs to remain in place for the duration of the backup.

- **Break a mirror.** If the database is stored on a system with mirrored disks, use of the suspend option followed by breaking the mirror creates a structurally safe static copy which can be backed up after resuming writes. As with simple suspension, this method does not guarantee data integrity unless transactions are used. The mirror can be reinstated when the backup is complete and will catch up with updates applied while the mirror was broken.

- **Suspend replication.** Broadly similar to breaking a mirror, this applies if the system is replicated to a second server. Suspending updates long enough to complete application of any queued updates and then terminating the subscriber process leaves the subscriber system in a structurally consistent state and, if transactions are used, guarantees data integrity too.

The above options have all highlighted the problem that the application needs to be in a consistent state at the point when the backup is performed. The best way to achieve this is to make full use of transactions but retrofitting this into an existing application is not easy because of the impact on locking strategy. There is no other way to guarantee data consistency without the application having some synchronisation mechanism to bring it to a tidy temporary halt.

**Recovery**

The structure of a QM system is that each account is stored as a separate directory. There is a register of accounts, the QM.ACCOUNTS file, in the QMSYS account. This file is keyed by uppercase account name and has the account pathname in field 1. Optionally, it may have an
account description in field 2 but this is not referenced by QM other than showing it if the file is listed. The QM.ACCOUNTS register should be the only place in the entire system that records the account pathname. If an account is moved to a new location, all that is needed is to update the pathname. The QMSYS account itself is found automatically based on it always being the parent directory of the location of the QM executable. Pathnames in the accounts register may commence with @QMSYS which will be substituted by the QMSYS pathname.

The items that make up QM itself are all within the QMSYS directory and its sub-structures.

The files in an account are located by pathnames stored in the F-type VOC entries. This should be the only place that file pathnames are recorded though it is possible for an application to open a file by pathname (discouraged as it can lead to maintenance problems). The default behaviour of QM’s CREATE.FILE command is to create a subdirectory within the account directory and to store a relative pathname in the VOC item. Moving the entire account to a new location would not require any change to this pathname. It is possible to place files at alternative locations (typically for load balancing across disks). If this is done, moving a file to a new location would require that the pathname in the VOC is updated.

Alternate key indices are usually stored in the directory that represents the file but it is possible to locate them elsewhere. If this is done, the indices should be restored in their original locations or, if this is not possible, the qmidx tool can be used to modify the pointer stored within the file.

Q-pointers should preferably have the target account name in field 2 which will be substituted by the pathname from the corresponding accounts register entry when opening the file. It is possible to reference the account pathname in the Q-pointer but this is discouraged. Ideally, a file should only be referenced by one F-type record and all other references should be via Q-pointers that link to the F-type record.

Restoring the entire system

If the entire system is to be restored, simply restore everything over the top of the previous version. It will be necessary to shutdown QM first, if it is running, so that the executable programs in the bin subdirectory of the QMSYS account can be overwritten. The restore will bring back all saved accounts and, assuming that the QMSYS account was saved, will include restore of the accounts register and all components of QM. Because the VOC is just a standard file, no special actions are needed to restore it.

If the system may have deleted files or records in directory files since the backup was taken, it may be best to delete the old data before starting the restore so that redundant items are not left in place.

If the restore is to a different server or the operating system has been reinstalled, the system id code associated with the licence will have changed. On entry to QM from the operating system command prompt, the licensing screen will be displayed, including the new system id code. A seven day “emergency licence” can be generated by visiting www.openqm.com/emergency.htm and entering the licence number and both the old and new system id codes. The information about the old system is on the licence document which should be stored safely in case it is needed. The emergency licence allows the user to get QM up and running immediately. A new permanent licence will be issued within the seven day period.

A change of system id code will also require the master key to be re-entered if the system uses encrypted data files.

If the version of QM is to be updated as part of the restore, the easiest way is to restore the data and then install the new version of QM over the top. This will ensure that system files are updated but user data such as any globally catalogued items is preserved.
The above process can also be used to create a duplicate of a system but, in this case, a new licence is required instead of an update to the existing system.

**Restoring an entire account**

Simply restore the account over the top of the current version or, if that has been lost, into a newly created directory. In the latter case, if the account pathname differs from the pathname of the saved account, the associated record in the QM.ACCOUNTS file must be modified.

If an account references files stored at locations outside the account directory, these other items must also be restored. If the new pathnames differ from those in the saved data, the corresponding VOC pointers must be updated.

**Restoring a specific file**

Restoring a file to its original location requires no special steps. If the file used relocated indices, these must be restored too and, if necessary, the qmidx tool run as mentioned above.

Restoring a file to a different location requires that the pathname in the VOC F-pointer is updated.

**Restoring specific records**

Follow the procedure to restore the file but to a new location. Then create a temporary VOC F-type record pointing to the restored file. Copy the required records to the original file. As an alternative to use of a temporary F-type record, if the FILERULE configuration parameter has been set to allow it, use a PATH:xxx style filename in the COPY command.

**Restoring to a different system type**

The above procedures assume that the system onto which the data is being restored is of the same architecture as the system from which the data was saved. If this is not true, a little additional work is needed.

If the restore was from Linux to Windows or vice versa, the line terminator in directory file text records will need to be updated. The *FIXDIR catalogued program will do this.

If the restore was to a system with different byte ordering from the one that was saved, the qmconv tool must be used to update all restored files.
8.7 Monitoring the System

QM provides several tools to aid System Administrators in monitoring the system and locating problems.

**LISTU** Displays a list of users currently logged in to QM

**LIST.FILES** Shows the names of all files currently open in the system. This command can also help in determining the optimum value for the NUMFILES configuration parameter.

**LIST.LOCKS** Lists process synchronisation (task) locks.

**LIST.READU** Lists all active record and file locks. Includes a report of who is waiting for locks. The DETAIL option to this command can also help in determining the optimum value for the NUMLOCKS configuration parameter.

**FSTAT** Shows file system performance related data.

**PSTAT** Displays process status information including the command or program being executed.

**HSM** More useful for programmers than administrators, the HSM (hot spot monitor) command shows the processing time spent in each module of an application.

Releasing Locks

Sometimes a QM process may fail to release a lock. In most cases, QM will tidy up automatically if the program or process terminates but there may be times when it is necessary to release a lock manually.

Be careful to consider the implications before releasing a lock. The lock was taken to protect something from simultaneous update. Releasing a lock always carries the risk of data integrity problems.

Record locks, file locks and process synchronisation (task) locks can be released with the UNLOCK command. This command can only be executed from the QMSYS account and requires administrator rights.

Terminating QM Sessions

A System Administrator can terminate a QM session using the LOGOUT command. Also, the qm command has a three special options that may be of use to System Administrators.

qm -k uid Kill process with QM user id uid.

qm -k all Kill all QM processes.

qm -u List all active QM processes

QM will attempt to tidy up, releasing any resources owned by the terminated process. Note that terminating a process carries the risk of data integrity problems if the termination occurs in the middle of an update that affects multiple files.
The Windows Task Manager or the kill command on other platforms with signal number 9 (kill -9) should only be used as a last resort if LOGOUT fails to kill the process. QM cannot catch this event and hence cannot free resources assigned to the terminated process. If this style of process termination is used while a QM process is updating a file, loss of data could occur.

**Lost Processes**

QM periodically compares its internal user table with the processes that are running at the operating system level. If any process registered within QM is no longer running, it must have terminated abnormally. An automated clean up mechanism will remove the user from the internal table, where possible releasing any locks that were held by that process. This automated scan occurs at an interval set with the CLEANUP configuration parameter, defaulting to once every five minutes if this parameter is not used. Setting the interval to a very low value can impact performance.

The RECOVER.USERS command can be used from within the QMSYS account to perform a similar clean up manually. The same effect can be achieved using

```
qm -cleanup
```

from the operating system command prompt.
8.8 Multi-Language Applications

QM includes support for non-English languages in various ways. This section summarises the features available and links to more detailed descriptions elsewhere.

Character Sets

There are two variants of QM offered during installation; one supporting the 8-bit character set, the other (the ECS or Extended Character Set version) supporting all characters in the Unicode Basic Multilingual Plane.

When using the 8-bit version, limited options for handling characters outside the standardised 7-bit ASCII set are provided by alternative definitions of characters 128 to 255 by use of Windows code pages. These can be selected separately in individual QM sessions though this is likely to lead to confusion where the same character value might lead to different display.

The ECS version provides a single standardised representation for characters of many languages and includes a block of character values defined as the Private Use Area to allow mapping of characters outside the Basic Multilingual Plane into the usable character set if needed. The ECS character maps allow selection of features such as sort order such that different users might see the same reported sorted differently to comply with local language conventions.

Both versions support character set encodings such as UTF-8 on external interfaces though the capabilities of this are limited in the 8-bit version.

Message Texts

All message texts used by QM itself are stored in a file named MESSAGES in the QMSYS account. Users can add their own application specific messages to allow multi-language applications.

On first installation of QM, the messages file will be created in either ECS or non-ECS mode depending on the mode of the underlying QM system. Installing an ECS mode version of QM over a non-ECS mode version will automatically reconfigure the messages file to ECS mode. The reverse action, installing a non-ECS mode version of QM over an ECS mode version will delete and recreate the file, populating it only with the standard English messages. Any locally applied alternative messages would need to be reloaded.

The standard message library installed with QM contains English messages texts. Additional language libraries may be available from Zumays or from QM resellers. Users can also perform their own translations of the source text downloaded from the product web site. New message sets are added using the LOAD.LANGUAGE command in the QMSYS account and the language to be used at run time is selected with the SET.LANGUAGE command.

National Language Conventions

QM has limited support for national language conventions such as alternative decimal and thousands separator characters in numbers or local currency symbols. These are selected using the NLS command or the SETNLS QMBasic statement.

See also:
SYSMSG()
8.9 Command Logging

Command logging allows an application to create a detailed audit log of every command executed, including those from within Procs, paragraphs, menus, etc. The QM command processor passes the command and its origin to an optional user supplied logging subroutine, leaving the application designer free to determine what information is actually logged.

The logging is performed by a QMBasic class module globally catalogued as !CLOG. This has a public subroutine named LOG that two arguments; the command and its origin. The source code for the default version of !CLOG is provided in the BP file of the QMSYS account.

The origin is one of the following:

- **User** A command entered by the user at the command prompt, including commands executed from the command stack.
- **S name** Execution of a VOC sentence.
- **PA name@n** The command from line n of a VOC paragraph. For an **IF** command, the conditioned command is separately logged if the condition is true.
- **PQ name@n** The command from line n of the named Proc.
- **M name@n** The command from option n of the named VOC menu.
- **Exec name@n** An **EXECUTE** on line n of a QMBasic program.
- **Chain cmd** A command executed using **CHAIN**.
- **Cli** A command executed by a QMClient session.
- **Auto** Execution of an automatic script (MASTER.LOGIN, LOGIN, ON.LOGTO, ON.EXIT, ON.ABORT).
- **Cmd** A command executed from the operating system shell by supplying it as a command line argument to the QM executable.

Where a command includes an inline prompt construct, the command is logged a second time after expansion of the inline prompt.

Logging may be enabled in two mutually exclusive manners. If the **CMDLOG** configuration parameter is not null, it specifies the pathname of the directory in which logging data is to be recorded. Logging is enabled across all processes and merged into a single log file per day in the specified directory using a name formed from the date in yymmdd format with a .log suffix.

If the **CMDLOG** configuration parameter is null, logging may be enabled in individual processes by use of the **COMMAND.LOGGING ON** command. In this mode, the logged data is written to a file named clog.nnn.log in the QM temporary directory where nnn is the QM user number.

The default command logger shown below is catalogued as part of QM and is also issued in source form in the BP file of the QMSYS account. Users may adapt this to meet their own needs.

```plaintext
class clog
```
$catalogue !clog

$include keys.h

private log.f, log.date, log.dir

public subroutine create.object
log.date = 0
log.dir = config('CMDLOG')
end

public subroutine log(cmd, origin)
tm = epoch()

if not(fileinfo(log.f, FL$OPEN)) then
    if log.dir = '' then ;* Single process logging
        log.path = system(SYS$TEMP) : @ds : 'clog.' :
        @user.no : '.log'
    end else
        dt = mvdate(tm)
        if log.date # dt then ;* New date - switch log
files
            closeseq log.f ;* No effect if file was
not open
        end
        log.date = dt
    end

    log.path = log.dir : @ds : oconv(log.date, 'D2YMD["",""]') : '.log'
end

    openseq log.path shared to log.f else return
end

* Construct and write log data

log.rec = tm
log.rec<-1> = @user.no
log.rec<-1> = @logname
log.rec<-1> = origin
log.rec<-1> = cmd
writeseq change(log.rec, @fm, char(9)) to log.f else null

return
end

This example logger constructs a dynamic array that it then converts to be tab delimited. The logged data is the epoch value (allowing consistent logging across time zones), the QM user number, user name, command origin and the command itself.
8.10 Error Logging

QM includes an error logging system that records brief details of errors that may require investigation by system administrators or application developers. These include:

- Run time program errors (e.g. unassigned variables)
- User authentication errors (failed logins)
- Forced logout
- Internal file system errors

The error log is maintained in a text file named errlog in the QMSYS directory. This file will be created automatically when the first message is logged. On Linux and Unix systems, the file permissions will be set to allow all users full access to the file.

Although the errlog file can be written directly by programs using the sequential file processing statements, this should be avoided as the buffering used by these statements may result in lost messages. Application developers should use the QMBasic LOGMSG statement or the LOGMSG command if they wish to add their own messages to the log file.

On ECS mode systems, the log file is maintained in UTF-8 encoding.

To avoid faulty programs generating very large log files and to remove the need for file maintenance, the ERRLOG configuration parameter sets the maximum size in kilobytes to which the error log file may grow. When this size is reached, the first half of the data in the file is discarded. The minimum acceptable non-zero value of the ERRLOG configuration parameter is 10. A smaller value will be treated as 10. Setting the ERRLOG parameter to zero disables error logging.

Each log message consists of two or more lines of text. The first gives the date and time in the timezone of the user, the QM user number, process id, login name and account name of the user generating the error. The second line gives the actual message, indented by three spaces to make the file more readable. Further lines may be present, also indented by three columns. This format is easy to process using user written tools if required.

Log Filtering

Prior to logging the first message from a QM session, the system checks for the presence of a catalogued subroutine named !ERRLOG.FILTER and, if this is present, calls it for each logged message from that point onwards. The subroutine can be used to filter messages such that certain messages are not logged or to trigger other application events such as emailing run time error messages to support staff.

The subroutine must be declared as

```
SUBROUTINE ERRLOG.FILTER(RESULT, MSG, STATUS, ERR)
```

where

RESULT should be returned as a Boolean value indicating whether the message is to be written to the error log. This defaults to true if not changed by the subroutine.

MSG is the text of the log message. Modifying this text has no effect.

STATUS is the value of the STATUS() function at the point when the log action was initiated. This is only meaningful for some messages.

ERRNO is the value of the OS.ERROR() function at the point when the log action was initiated. This is only meaningful for some messages.
The subroutine must be catalogued with the ! prefix on its name. A log message generated from the filter subroutine such as a run time error is always logged without calling the filter subroutine recursively. Also, log messages from other QM processes such as the replication server (qmrpl) or the network monitor (qmsvc or qmlnxd) are not filtered.
8.11 QM Command Options

The QM executable stored in the bin subdirectory of the QMSYS account has a number of command line options. The command options shown below are all case insensitive. To comply with Linux conventions options may be also used with a double hyphen prefix.

-\( n \)  
Logs the user in as user \( n \) where the value of \( n \) must be in the range of user numbers reserved with the **FIXUSERS** configuration parameter. If the FIXUSERS parameter has not been set, the full range of QM user numbers is available for use in this way. If there is already a user logged in with this user number, the login fails. If the value of \( n \) is not within the reserved range, the parameter is ignored.

-A  
Causes QM to prompt for the account name on entry.

-Aname  
Enters account name unless there is a fixed account name defined for the user name of the user entering QM. Note that there is no space before the account name.

-DL  
Initiate device licensing negotiation for entry to QM from the operating system command prompt.

-K \( n \)  
Kill the QM process for user \( n \).

-K PID \( n \)  
Kill the QM process for operating system process id \( n \).

-K user  
Kill all QM processes for login id user.

-K all  
Kill all QM processes.

-L  
Apply a new licence.

-U  
List current QM users. Append X (qm -ux) for an extended report that shows how many licensed users have been consumed and the login time and account name for each process.

-CLEANUP  
Scans QM’s internal list of users to identify any processes that have terminated abnormally and, if found, tidies up from the lost process.

-HELP  
Display usage help.

-LOCKS  
Show record locks.

-PHANTOM cmd  
Executes cmd as a phantom process.

-QUIET  
Suppresses display of release information, VOC upgrade prompts, etc on entry to QM. Useful in some scripted sessions.

-RESTART  
Restart QM (not Windows).

-RESUME  
Resume database updates.

-START  
Start QM (not Windows).
-STATUS Shows the QM startup time and the current number of users in each process class.

-STDOUT (Windows only) QM normally uses the Windows console APIs to output data to the screen. This option causes it to use the stdout file handle and is of use when capturing the output or piping it into other processes. It should not be used for normal terminal output as some display features may not work with this option.

-STOP Stop QM (not Windows).

-SUSPEND Suspend database updates.

-SYSID Display system id code.

-TERM xxx Sets the initial terminal type. This may be changed later from within the application. If this option is not used, QM defaults to the value of the operating system TERM environment variable or, if this is not defined, vt100.

-UTF8 Sets the terminal mode as UTF-8 (Not Windows).

-VERSION Displays version information.

It is also possible to execute a QM command directly from the operating system command prompt by appending it to the start up of the QM session, after any other command options. For example:

qm RUN OVERNIGHT

Note that quotes may be needed if the QM command contains any characters with special meaning to the operating system. Use of QM in this way can be detected by use of the SYSTEM(1026) function that will return the command (e.g. RUN OVERNIGHT in the above example).

Environment Variables

QMDL If defined, entry to QM from the operating system command prompt behaves as though the -DL option had been used.

QMSYS Normally the location of the QMSYS directory is derived from the pathname of the QM executable as this is installed in the bin subdirectory of the QMSYS account. This variable specifies an alternative default location for the QMSYS account directory. It is also used by QMConnectLocal() if a non-default location is used.

TMP Specifies the location of temporary directory if the TEMP configuration parameter is not set.
The qmfix Utility

The QM file system is designed to be robust, however, there are situations when power failures, hardware failures or abnormal termination of a process might lead to structural integrity problems within a file.

The qmfix utility can be used to check the structural integrity of a file and, if an error is detected, then apply an automated correction. Although qmfix should always result in the file being usable, there are error situations where data will be lost because it was in memory waiting to be flushed to disk.

To use qmfix, firstly ensure that no users have the file(s) to be processed open. It is safest to run qmfix when no users are using QM. The qmfix utility is run from the operating system command prompt, not from within QM. The command line is

qmfix options pathname

where

options are case insensitive option codes from the following set:

- B Suppresses progress bar display. This can be of use when capturing the output for later review as repainting of the progress bars may make the captured data less easy to read.
- C Check file for errors (implied if B, F, Q and R options all absent)
- F Fix errors without querying
- L Log the screen output in qmfix.log
- L path Log the screen output in path
- N Suppresses pagination of displayed output
- Q Query before fixing errors
- R Recover space from unused primary and overflow blocks. Occasional use of this function may improve performance of some large files.
- S Suppresses output to the screen when used with the -L option.

pathname is the pathname of the file to be processed. This may be a list of may include wildcard characters. qmfix will ignore names that do not correspond to QM files. Thus, to check all files in a directory, simply type

qmfix *

Where pathname refers to a directory, either directly or as part of wildcard expansion, qmfix will recurse down one level to allow for directories that represent multi-files.

Do not run qmfix with the -F option without running it to check for errors first.

Note that qmfix may report that a dynamic has an incorrect load value or record count if the file was not closed properly at a system failure. These errors are unlikely to cause any serious problems and will be corrected by qmfix if the -F option is used and automatically by select operations that complete without any intervening file updates.
No automated error recovery tool can ever be 100% accurate in its decisions about the nature of errors so there is a very small risk that qmfix could make the situation worse. **Always backup a file before fixing any errors in it.**

Zumasys aims to provide software of the highest quality. We would be very interested to receive copies of any files that are reported as faulty by qmfix so that we can investigate the cause and improve the resilience of the QM product.
The QM file system supports alternate key indices for retrieval of data based on the content of a non-key field. Normally, these indices reside in the directory that represents the data file. This helps to ensure that the entire file and its indices are handled as a single item when performing backup and restore operations.

Sometimes it may be useful to place the indices elsewhere. This could be to improve load balancing across multiple disk drives or to simplify exclusion of large indices from backups since they can always be recreated from the data file. This separation of the indices from the data can be achieved using the PATHNAME option of the CREATE_INDEX or MAKE_INDEX commands when the first index is created.

The qmidx program is an operating system level command that allows users to report or modify the location of the indices. It has four modes of operation:

```
  qmidx -d data.path  Deletes all indices for the named file.

  qmidx -m data.path index.path  Moves the indices for the named file to the new location specified by index.path. If the index path is omitted, the indices are returned to their default location in the directory identified by data.path.

  qmidx -p data.path index.path  Sets the index path for the given data file. This operation is required after use of an operating system level tool to copy or move a data file to a new location. Failure to perform this step when duplicating a file may result in any changes to the copied file updating the indices of the original file.

  qmidx -q data.path  Displays the location of the indices for the given data file.
```

This program must only be used when the file is not in use. Failure to adhere to this rule may lead to data corruption or process failure.
8.14 The qmconv Utility

QM hashed files and QMBasic object code are sensitive to the byte ordering of the system on which they are used. If a file, program or ECS character map is moved from a system with one byte ordering to a system of the opposite byte ordering, the qmconv utility can be used to update the file or program.

This tool is run from the operating system command prompt. The command line is

```
qmconv  options pathname(s)
```

where

- `options` are case insensitive option codes from the following set:
  - `-B` Convert to "big-endian" format.
  - `-L` Convert to "little-endian" format.

- `pathname` is the pathname of the file to be processed. Multiple pathnames may be given and wildcards may be used. qmconv will ignore names that do not correspond to QM files.

If neither mode option is given, the file is converted to the byte ordering of the system on which the command is run.

Hashed files must be converted when moving to a system with different byte ordering. Conversion is optional for QMBasic object code as programs will be converted automatically on loading into memory, if needed.

In addition to use of qmconv, moving from Windows to Unix/Linux based systems or vice versa requires that pathnames in the VOC are converted to use the appropriate directory delimiter character. This can be achieved by typing

```
*fixpath acc.name
```

at the QM command prompt in the relevant account. Alternatively, an account name or pathname may be given

```
*fixpath acc.name
```
8.15 Tools

QM includes several tools to assist system administrators. These are provided as globally catalogued programs that can be executed by typing their name, including the leading asterisk, at the QM command prompt.

*FIXDIR

The FIXDIR tool updates records in a directory file to use the line terminator appropriate to the operating system. This is a CR/LF pair on Windows and LF on other platforms. It is useful when moving accounts from Windows to Linux or vice versa.

The command is

```
*fixdir filename {OMIT.HIDDEN} {id...}
```

The `filename` may be the VOC entry name for an F or Q-type item referencing a directory file. Alternatively, it may be the pathname of the directory.

If no record ids are given on the command line, the default select list is used. If this list is not active, all records in the file are processed and the OMIT.HIDDEN option will cause hidden files to be omitted.

*FIXPATH

The FIXPATH tool updates pathnames in F, Q and V-type VOC items to use the correct operating system directory delimiter character. It is useful when moving accounts from Windows to Linux or vice versa.

The command is

```
*fixpath
```
to update the current account or

```
*fixpath acc.name
```
to update another account. The `acc.name` may be the account name or its pathname.

*FIXPFX

The FIXPFX tool changes the prefix character on the subfiles within the directory that represents a hashed file from ~ to %. Early releases of QM used the ~ prefix but it was discovered that some PC clean-up tools assumed that anything beginning with a ~ character was a temporary item that could be deleted. QM release 2.7-1 (July 2008) changed the prefix character to % to avoid this problem and automatically renamed the subfiles when opening a file for which write permission was available. This renaming mechanism was removed in release 3.3-0 (April 2014) as there was a small performance impact on all file open operations. Instead, the installer offered an automatic scan to update the subfile prefix in all accounts.

There remains the possibility that a user might need to import an account from a release that used the ~ prefix. The FIXPFX tool will update a named account or all accounts to use the new subfile prefix. The command line is
*fixpfx
to update all accounts or
*fixpfx acc.name
to update a named account (which may be given as a pathname).

**IMPGCAT**

The IMPGCAT tool imports items from the global catalogue of another QM system. It is of use when moving an account or an entire QM system to a new computer.

Simply copying the gcat subdirectory of the QMSYS account does not work unless the two systems are running the same version of QM as the global catalogue includes many items that are part of the QM release and are not compatible across different versions. This tool selectively copies items, transferring user written items and omitting those that are standard components of QM.

The command line is

```
*impgcat {pathname} {options}
```

where

- `pathname` is the pathname of the global catalogue from which items are to be copied.
- `options` are chosen from
  - DISPLAY Show the names of items but do not copy.
  - NO.QUERY Suppress confirmation prompt before copying.
Part 9

Migrating to OpenQM
9 Migrating to OpenQM

This section gives some guidance on the documents and tools that are available to assist in migrating an application to QM.

- **Transferring Data from other Multivalue Systems**
- **Option and Mode Shortcuts**
- **Conversion Tools**

Documents

A document outlining the process of migrating to QM can be found towards the bottom of the QM downloads page on the Zumasys web site. There are two versions of this document; one for migration from Pick style systems (D3 and mvBase) and one for migration from U2 systems (UniVerse and Unidata).

Source Code

The BP file of the QMSYS account includes source code for many Pick style user exits.

Additional source modules that might be useful to developers migrating an application from D3 is available from the QM downloads page as a zip file. The content of this file is:

- **ACCESS** A limited emulation of some features of the D3 ACCESS() function.
- **D3.CREATE.INDEX** A wrapper for the D3 syntax of CREATE.INDEX.
- **D3.CONVERT** A wrapper for the CONVERT() function to use the D3 argument order. Use of $MODE PCONVERT provides a better solution.
- **D3.SORT** An emulation of the D3 SORT() function.

Details of the procedure to move a QM account to a different computer (possibly with a different operating system) can be found in the section entitled "Moving an Account".
9.1 Transferring Data from other Multivalue Systems

QM includes versions of the Pick style data restore tools such as T.LOAD and ACCOUNT.RESTORE however, because the media format used by the corresponding save tools (T.DUMP and ACCOUNT.SAVE) is not publicly documented, some variations have been encountered where restore of a Pick style save into QM fails. To avoid this, QM includes a tool named QMSAVE which is issued in source form and can be compiled and run on a variety of other environments. The QMRESTORE tool can then be used to read the saved data into QM.

QMSAVE

The QMSAVE tool is provided in source form in the BP file of the QMSYS account. It includes conditional compilation constructs ($IFDEF) for the parts that vary according to the environment from which data is being transferred. These constructs can be resolved manually by editing the source code or by compiling and running the preprocessor released as PREPROC in the BP file of the QMSYS account.

The source code has been written in a way that allows it to be moved to the source system simply by running the ED editor on the Pick system, going into insert mode and using the "Paste from File" option of AccuTerm.

A full description of how to modify, compile and run this tool is included in the source code. The extracted data is moved to one or more export files, each with a numeric suffix and limited to a size set in the source code. Unless the exported data is visible from the QM system, the files should then be moved using binary mode FTP, removable hard drive or other data transfer method.

The QMSAVE tool is subject to change. Users who modify it to work in new environments are encouraged to contribute their changes back to QM so that they may be shared with other users.

Versions for UniVerse and Unidata are in development and available in beta test form from Zumasyss support on request.

QMRESTORE

The QMRESTORE tool is used to restore the exported data into QM.

The command line to run this tool is

\[ \text{QMRESTORE} \ \text{pathname} \ (\text{options}) \]

where

- \text{pathname} is the name of the file holding the saved data. The sequence number is added automatically.
- \text{options} are:
  - CLEAR Clear file before restoring
  - NO.CASE Create new accounts with case insensitive VOC
  - NO.QUERY Suppress confirmation queries

The command will prompt for the name of the saved account and the name of the files to be restored. Use of an asterisk as the file name restores all files.
To avoid damaging the QM VOC file, any file named VOC or MD in the saved data is restored as VOC-RESTORE or MD-RESTORE respectively. Any files with names that clash with a QM VOC entry that is not a file will be restored with a "-CLASH" suffix added and must be resolved manually.

**UVIMPORT and UDIMPORT**

The UVIMPORT tool can transfer data directly from UniVerse data files to QM files. The UDIMPORT tool is similar but reads Unidata files. Both tools are in the standard global catalogue but need a VOC entry to be created in the account(s) where they are to be used. For UVIMPORT, this should be:

```
UVIMPORT
1: V
2: CA
3: $UVIMPORT
```

For UDIMPORT, it is similar but with $UDIMPORT in field 3.

The programs can import all or selected files from a U2 account. The source files may be static hashed, dynamic hashed or directory. The file byte ordering will be corrected automatically if the file has come from a system with a different architecture.

These tools do not make changes to the source files, however, it is recommended that a copy of the U2 account is used rather than the live data. It is essential that no U2 users are using the files while the import process is running.

Although the tools will work with a standard 80 x 24 screen, it is useful to enlarge the window to have as many lines as will fit on the screen. For example,

```
TERM 80,43
```

The tools have two modes of operation; interactive mode and command line mode.

**Interactive Mode**

LOGTO the account into which data is to be imported. Create the VOC item described above if it does not already exist.

Type UVIMPORT or UDIMPORT to start the program.

Enter the pathname of the U2 account from which data is to be imported. The program scans the VOC of this account to build a list of importable files and displays this list. Each file is numbered for reference. This process may take some time to complete if there are many files.

Two file status columns are displayed, one for the data file and one for the dictionary. These columns may contain:

- **none** The file component does not exist
- **err** There was an error opening the file
- **sys** The file corresponds to a QM file in the QMSYS account
- **clash** There is an item of this name in the VOC of target account that is not a file
- **COPY** The file is selected for copying

If the column is blank, the file could be copied but is not yet selected.

A help text box is displayed at the bottom of the screen. This display can be toggled by typing a question mark and pressing return.
To select files, type:
   • the line number (e.g. "5") to select a specific file
   • a range of line numbers (e.g. "4-9") to select multiple files
   • "A" to select all files

To deselect files, use the same codes prefixed with "X".

The selection/deselection initially affects both the data and dictionary portions. Type "D" to switch between both, data and dictionary modes. The current mode is displayed at the top right of the screen.

To scroll through the pages of the display (where applicable), type N or P.

Once files have been selected, type Q to quit from this screen. If any files have been selected, the program will prompt for confirmation that import is to continue.

For each selected file, the program displays the file name, source path name and target name.

Press return to import the file, a space to omit the file, or an asterisk to import all files without further prompting.

Two progress bars are displayed, one for files, the other for groups. In a directory type file, each record is considered as a group.

If selected for import with UVIMPORT, the VOC and VOCLIB files will be restored as UV_VOC and UV_VOCLIB. For UDIMPORT, the VOC will be restored as UD_VOC.

**Command Line Operation**

LOGTO the account into which data is to be imported. Create the VOC item described above if it does not already exist.

The command syntax is identical for both tools. Using UVIMPORT as an example, to import to the default location, type

```
UVIMPORT source.path
```

or, to import to a specific location, type

```
UVIMPORT source.path {{DICT} target.file}
```

If the source.path begins "D_", the program assumes that it is a dictionary and imports to the dictionary of the corresponding file.

Alternatively, if the default select list contains a list of filenames to be imported from the U2 account, type

```
UVIMPORT account.path
```

The following additional command line options are supported:

- **ALL** Import all files without prompting.
- **LOCAL** Similar to ALL but imports only local files.
- **CLEAR** Clear files before importing.
- **OVERWRITING** Overwrite files without prompting.
9.2 Option and Mode Shortcuts

This section gives some guidance on the documents and tools that are available to assist in migrating an application to QM.

The **OPTION** command and the QMBasic **$MODE** compiler directive support shortcut names that enable multiple settings in a single operation. These shortcuts provide close compatibility with other multivalue products and may be of use when migrating an application to QM.

The tables below show which options and modes are set by each shortcut name. The shortcut names are:
- PICK.D3
- PICK.MVBASE
- UV.IDEAL
- UV.INFO
- UV.PICK
- UV.REALITY
- UV.PIOPEN
- UDT

The actual options and modes set by these shortcuts may change in future releases.

For compatibility with earlier releases, shortcut names D3 and MVBASE are still supported. These provide a far less accurate emulation of D3 or mvBase and are described with the **OPTION** command and **$MODE** directive.

### OPTION Settings

<table>
<thead>
<tr>
<th>Option</th>
<th>D3</th>
<th>mvBase</th>
<th>UV.IDEAL</th>
<th>UV.Information</th>
<th>UV.Pick</th>
<th>UV.Reality</th>
<th>UV.PI/Open</th>
<th>Udt.ECLTYPE.U</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMPM.UPCASE</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>ASSOC.ALL</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>ASSOC.UNASSOC.MV</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>BACKSLASH.NOT.QUOTE</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>CATALOGUE.LOCAL</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>CHAIN.KEEP.COMMON</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>CHAINED.SELECT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>CLEAR.SELECT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>COMMAND.LEVEL.LOCKS</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>CORRELATIVE.NOCASE</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CORRELATIVE.REUSE</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>Option</td>
<td>D3</td>
<td>mvBase</td>
<td>UV Ideal</td>
<td>UV Information</td>
<td>UV Pick</td>
<td>UV Reality</td>
<td>UV PI/Open</td>
<td>Udt.ECLTYPE</td>
</tr>
<tr>
<td>----------------------------</td>
<td>----</td>
<td>--------</td>
<td>----------</td>
<td>----------------</td>
<td>---------</td>
<td>------------</td>
<td>------------</td>
<td>-------------</td>
</tr>
<tr>
<td>CRDB.UPCASE</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CREATE.DICT.NO.CASE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CREATE.FILE.NO.CASE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEBUG.REBIND.KEYS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEFAULT.MV</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIR.DTM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIR.SEL.OMIT.HIDDEN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DIV.ZERO.WARNING</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>DUMP.ON.ERROR</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ED.NOQUERY.FD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FORCE.RELOAD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INHERIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INHERIT.MFILE.PATH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INHERIT.OWNERSHIP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KEEP_FILENAME.CASE</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>KEEP_OLD_OBJECT</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOCAL.CAT.RELATIVE</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>LOCK.BEEP</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NO_DATE.WRAPPING</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NO.ECHO.DATA</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NO_PHANTOM.NOTIFY</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NO_SEL_LIST_QUERY</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NO_SEL_WHEN_FILTER</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NO_USER_ABORTS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NON_NUMERIC_WARNING</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>NUMERIC.EXPONENT</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>PAGINATE.ON.HEADER</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>PICK_BREAKPOINT</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PICK_BREAKPOINT.U</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PICK_EXPLODE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Option</td>
<td>D3</td>
<td>mvBase</td>
<td>UV Ideal</td>
<td>UV Information</td>
<td>UV Pick</td>
<td>UV Reality</td>
<td>UV PI/Open</td>
<td>Udt ECLTYPE U</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>----</td>
<td>--------</td>
<td>----------</td>
<td>---------------</td>
<td>--------</td>
<td>------------</td>
<td>------------</td>
<td>---------------</td>
</tr>
<tr>
<td>PICK.GRAND.TOTAL</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PICK.IMPLIED.EQ</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PICK.ML.CONV.MASK</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PICK.NULL</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>PICK.PROC</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PICK.WILDCARD</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>PROC.A</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROC.IF.NO.CASE</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QUALIFIED.DISPLAY</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QUERY.FOOTER.PAUSE</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>QUERY.MERGE.PRINT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QUERY.NO.CASE</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QUERY.PRIORITY.AND</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>QUERY.SORT.NO_CASE</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QUERY.STR.COMP.ALL</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QUERY.STRING.COMP</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RUN.NO.PAGE</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SELECT.KEEP.CASE</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SHOW.STACK.ON.ERROR</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SILENT.TANDEM</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SPACE.MCT</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>SPOOL.COMMAND</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STACKED.ACCOUNT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUPPRESS.ABORT.MSG</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>UNASS.WARNING</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>WITH.IMPLIES.OR</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

$MODE Settings
<table>
<thead>
<tr>
<th>Mode</th>
<th>D3</th>
<th>mvBase</th>
<th>UV/Ideal</th>
<th>UV Information</th>
<th>UV Pick</th>
<th>UV Reality</th>
<th>UV PI/Open</th>
<th>Udt B</th>
<th>BASICTYPE U</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTOLOCK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CASE.SENSITIVE</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHANGE.NO.OVERLAP</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPATIBLE.APPEND</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COMPOSITE.READNEXT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CONDITIONAL.STATEMENTS</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>COUNT.OVERLAP</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEBUG.EQUATES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEFAULT.UNASS.ARGS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXCEPTION.SAVE.STACK</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXECUTE.CLEARLIST</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXPLICIT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FOR.STORE.BEFORE.TEST</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HEADING.NO.EJECT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>IMPLIED.STOP</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INDEX.OVERLAP</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NOCASE.STRINGS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NO.ECHO.DATA</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NON.DURABLE.TXN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NUMERIC.BOOLEAN</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPEN.FAIL.ZERO.FVAR</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPTIONAL_FINAL.END</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPTIONAL_THEN.ELSE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PARTIAL_EXPRESSIONS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PICK_CONVERT</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PICK.ENTER</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PICK.ERRMSG</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PICK.JUMP.RANGE</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PICK.MATRPROX</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mode</td>
<td>D3</td>
<td>mvBase</td>
<td>UV.Ideal</td>
<td>UV Information</td>
<td>UV Pick</td>
<td>UV Reality</td>
<td>UV.PI/Open</td>
<td>Udt</td>
<td>BASICTYPE U</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----</td>
<td>--------</td>
<td>----------</td>
<td>----------------</td>
<td>---------</td>
<td>------------</td>
<td>------------</td>
<td>-----</td>
<td>-------------</td>
</tr>
<tr>
<td>PICK.READ</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PICK.READNEXT</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PICK.SELECT</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PICK.SUBSTR</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PICK.SUBSTR.ASSIGN</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PRCLOSE.DEFAULT.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SELECTV</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSELECTV</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STDFIL</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>STDFIL.SHARED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STRICT.EQUATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STRING.LOCATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>X</td>
</tr>
<tr>
<td>TIME.MS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRAP.UNUSED</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRAP.UNUSED.MAIN</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UNASSIGN.COMMON</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UV.LOCATE</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WRITE.DELETE.THEN.ELSE</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
9.3 Conversion Tools

Four tools are available to assist in conversion of programs imported from other multivalue systems:

- **D3CONV** Migration from D3
- **MVCONV** Migration from mvBase
- **UDCONV** Migration from Unidata
- **UVCONV** Migration from UniVerse

The command syntax and general operation of these tools are identical except as explicitly described below.

The conversion tools take a file containing program source code from the originating system and process this to produce a file of the equivalent QM program source code. Although the conversion tools can make many source changes automatically, there are situations where there is a need for manual intervention. These are reported on the screen and also tagged in the QM source text with an easily searchable tag (!!) and a description of the issue.

Using D3CONV as an example, the command syntax is

```
D3CONV source.name target.name {program.name} {options}
```

where

- `source.name` is the name of the QM file holding the original programs.
- `target.name` is the name of the QM file to receive the converted source code.
- `program.name` is one or more names of the programs to be converted. Using an asterisk as the program name converts all programs in the source file. If no program name is given, the default select list is used.
- `options` are chosen from:
  - **BRIEF** Minimal reporting, showing only modules with errors.
  - **DELETING** Delete source program after successful conversion.

In addition, UVCONV has an option that specifies the UniVerse flavour of the origin system as IDEAL, INFORMATION, PICK, PIOPEN or REALITY. For example,

```
UVCONV UVBP QMBP * IDEAL
```

UDCONV has a BASICTYPE option that specifies the compiler language variant for which the program was written (P or U). This defaults to U if not specified. For example,

```
UDCONV UDBP QMBP * BASICTYPE P
```

The convertor performs just enough syntax analysis to be able to determine the role of each source element. It is assumed that the input program is structurally correct and there is minimal attempt to trap program errors. Very incorrect syntax such as misplaced use of a THEN clause may pass through unreported.

The conversion process performs several automated substitutions but does not attempt to rewrite code in any way. Some of the actions performed are:

- Replacement of `SYSTEM()` function key values with token names that are defined in the SYSCOM KEYS.H include record. By using token names rather than simply changing the key values where the QM key is different from that on the originating system, the code is more readable and could still be compiled on the original system if an alternative include record was constructed. `SYSTEM()` functions for which no direct equivalent exists in QM will be tagged for manual inspection.
• The negative mode values in the @() function are replaced by token names that are defined in the SYSCOM KEYS.H include record. Unrecognised @() function mode values are tagged for manual inspection.

• Some @-variable names may be changed to their QM equivalents.

• Some statements and functions may have minor changes to their syntax applied.

Users encountering migration issues that are not resolved by the conversion tools are encouraged to report the details to Zumasys support as a work-around may be available.
Part 10

System Limits
# 10 System Limits

<table>
<thead>
<tr>
<th>Limit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of users</td>
<td>Determined by the licence</td>
</tr>
<tr>
<td>Maximum number of phantoms</td>
<td>Determined by the licence</td>
</tr>
<tr>
<td>Maximum hashed file size</td>
<td>16384Gb</td>
</tr>
<tr>
<td>Maximum sequential file size</td>
<td>Limited only by the operating system</td>
</tr>
<tr>
<td>Maximum records in a file</td>
<td>Limited only by file size</td>
</tr>
<tr>
<td>Maximum record key size</td>
<td>63 bytes, configurable to 255 characters (MAXIDLEN parameter). For a directory file, the record id must not be more than 255 characters after translation of characters that are not allowed in operating system pathnames. A record id may contain any character except the mark characters (chars 251 to 255) or the ASCII NUL character (char 0). Null record ids (zero length) are not allowed in QM. If the system is in ECS mode, characters outside the 8-bit character set are only valid for ECS mode files.</td>
</tr>
<tr>
<td>Maximum record size</td>
<td>Hashed files: A little under 2Gb or available memory space. Directory files: 1Gb.</td>
</tr>
<tr>
<td>Maximum indices per file</td>
<td>32</td>
</tr>
<tr>
<td>Maximum AK index key size</td>
<td>255 bytes</td>
</tr>
<tr>
<td>Maximum trigger function name</td>
<td>32 bytes</td>
</tr>
<tr>
<td>Maximum record locks</td>
<td>Set system wide by NUMLOCKS parameter. Maximum 100000.</td>
</tr>
<tr>
<td>Maximum number of open files</td>
<td>Set system wide by NUMFILES parameter. Maximum 32767. A single file opened by multiple users counts as one in this calculation.</td>
</tr>
<tr>
<td>Maximum QMNet connections</td>
<td>10 separate remote server connections from each QM process. Each connection can open multiple files on the remote server.</td>
</tr>
<tr>
<td>Maximum program size</td>
<td>8Mb per program module</td>
</tr>
<tr>
<td>Maximum character string size</td>
<td>A little under 2Gb or available memory space</td>
</tr>
<tr>
<td>Maximum integer value</td>
<td>32 bits (2147483647). The QM run machine will switch to floating point representation for larger values.</td>
</tr>
<tr>
<td>Maximum floating point value</td>
<td>IEEE 64 bit representation. Upper limit 21024.</td>
</tr>
<tr>
<td>Maximum digits</td>
<td>The maximum number of digits allowed in the character string representation of a number is 307 (limited by IEEE floating point format) though precision is limited to approximately 15 digits.</td>
</tr>
<tr>
<td>Maximum number of public names in a CLASS module</td>
<td>8191 variables, 16383 total</td>
</tr>
</tbody>
</table>
Maximum size of a program for debugging 65535 lines

Maximum pathname length 255 bytes
Part 11

Glossary of Terms
11 Glossary of Terms

Abort An event that occurs at a major application failure. Aborts can be generated by QM internally if it detects a serious error, by application software, or by the end user (though this can be disabled). All programs, menus, paragraphs, etc active in the user’s process are discarded and the user returns to the command prompt. The ON_ABORT VOC item can be used to capture this event and take special action.

Account A collection of database files, programs, etc that form an application. Viewed from outside QM, an account is an operating system directory.

AK See Alternate Key Index

Alias A way to assign an alternative name to a command, perhaps only within specific QM sessions. For example, users migrating from Pick style environments frequently use ALIAS to make COPY run the COPYP command.

Alternate Key Index An index structure that allows applications to identify all records that have a particular value for a secondary key item. The index contains a list of primary key values for each secondary key value.

Association The relationship between two or more multivalued data items where the values belong together. For example, an order processing system might have two associated fields, one holding a list of product codes, the other a list of quantities ordered.

Attribute An alternative name for a field.

B-tree files A file structure used to store alternate key indices. B-tree (balanced tree) files use an internal structure that stores data in sorted order and hence allow efficient access to ranges of key values.

Big endian A term that describes storage of multi-byte numeric values such that the most significant byte comes first.

Byte string A character string in an ECS mode system in which all characters are from the 8-bit set and hence have character values in the range 0 to 255. Typically used to store binary data such as object code or bit mapped image files. In a non-ECS mode system, all character strings are implicitly of this form.

Catalog(ue) A repository for application programs. QM supports three modes of cataloguing; local, private and global. Local and private cataloguing normally restrict access to the program to the one account. Globally catalogued programs can be accessed from all accounts.

Class module A program source module that acts as a container for persistent data definitions and public functions/subroutines in object oriented programming.

Command prompt A prompt (a colon) displayed by the command processor when it is waiting for a command to be entered. This prompt changes to a double colon if the default select list is active.

Command stack A list of the most recent commands executed by a user. The command stack editor allows a user to look back at this historic data, modify commands and repeat commands. (In strict computing terms,
the command stack isn't a stack at all. It's a queue but the incorrect term is widely used.)

Common block
A block of data items used by an application that are to be shared between programs in the one user process.

COMO file
A record in a special file ($COMO) that stores a copy of all output sent to the user's terminal. COMO (command output) trapping is enabled/disabled with the COMO command.

Conversion code
A code describing how data must be transformed from its internal representation within the database before displaying it to a user. For example, dates are usually stored internally as the number of days since 31 December 1967.

Correlative
A rather limited equivalent to I-type dictionary items supported by QM in A and S-type items to simplify migration from other multivalue database products.

Database
A set of data stored in a form that enables rapid access to specific items.

Data collection
An arbitrarily multi-dimensional data item that uses name/value pairs.

Debugger
A development tool for debugging QMBasic programs.

Device licensing
An option that allows multiple connections to QM from a single client to share a licence.

Dictionary
A secondary component attached to most database files that contains a description of the format of records stored in the file and default settings controlling how the query processor will display this data.

Directory files
A simple file structure that makes use of the underlying operating system's files to represent database records. Directory files do not offer high performance but can be accessed from outside QM and are therefore frequently used to exchange data with other software.

Distributed file
A view of a set of separate dynamic files as though they were a single file.

Dynamic array
A character string divided into fields, values and subvalues using the mark characters. A database record is stored in this way but dynamic arrays can be used by application developers for other lists of items.

Dynamic file
A type of hashed file that automatically changes its modulus value to react to changes in the volume of data stored.

ECS
Used to reference a character from the Extended Character Set (16-bit) or in combination as “ECS mode” meaning a QM system that is running in a mode that supports characters beyond the 8-bit set.

Epoch value
A representation of a moment in time in a time zone independent manner.

errlog
An error log file in the QMSYS account. QM writes entries to this log whenever an error is detected. Applications can also write to the log using the QMBasic LOGMSG statement.

Exception
An event, typically related to an error, that can be raised in one part of an application and caught elsewhere.

Field
A column from the tabular representation of a database file.
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>File</td>
<td>A database table. QM supports hashed files for high performance and directory files for data exchange.</td>
</tr>
<tr>
<td>Format code</td>
<td>A code describing the way in which a data item is to be displayed or printed specifying, for example, the number of characters and justification.</td>
</tr>
<tr>
<td>Group</td>
<td>A portion of a hashed file. The number of groups in a file varies automatically according to the volume of data stored in it.</td>
</tr>
<tr>
<td>Group size</td>
<td>The size of a group in multiples of 1024 bytes. QM allows group size values in the range 1 to 8.</td>
</tr>
<tr>
<td>Hashed files</td>
<td>High performance files in which the location of a record is calculated by applying the hashing algorithm.</td>
</tr>
<tr>
<td>Hashing algorithm</td>
<td>A mathematical calculation applied to the characters of a record key to deduce the group in which that record will be stored.</td>
</tr>
<tr>
<td>Hot Spot Monitor</td>
<td>A tool for identifying the number of times each module of an application is executed and the processing time spent in them.</td>
</tr>
<tr>
<td>ID</td>
<td>Another name for the primary key of a record.</td>
</tr>
<tr>
<td>I-descriptor</td>
<td>Another name for an I-type.</td>
</tr>
<tr>
<td>Information style</td>
<td>A reference to the command and programming language style found in the Prime Information multivalue database product and others that adopt this style.</td>
</tr>
<tr>
<td>Inline prompt</td>
<td>A special syntax in a QM command that allows substitution of data into the command. Although the name implies that the construct will prompt the user for this data, there are many variants that retrieve data from elsewhere.</td>
</tr>
<tr>
<td>I-type</td>
<td>A dictionary item that describes a calculation that yields a result that can then be used exactly as though the value was stored in the database.</td>
</tr>
<tr>
<td>Key</td>
<td>A shortened term for the primary key of a record.</td>
</tr>
<tr>
<td>Keyword</td>
<td>A word or symbol affecting the behaviour of a command. Keywords correspond to K-type VOC entries.</td>
</tr>
<tr>
<td>Laws of Normalisation</td>
<td>A set of rules that govern the construction of relational databases. Multivalue databases discard the first law of normalisation, allowing them to store multivalued data. This results in a data model that more accurately reflects the real world than a fully normalised database, usually requires fewer tables and is quicker to develop.</td>
</tr>
<tr>
<td>Link record</td>
<td>A dictionary L-type record that defines the link between two related database files.</td>
</tr>
<tr>
<td>Little endian</td>
<td>A term that describes storage of multi-byte numeric values such that the least significant byte comes first.</td>
</tr>
<tr>
<td>Locking</td>
<td>A mechanism used to ensure that two users cannot update the same data item simultaneously, a situation that would usually result in errors.</td>
</tr>
<tr>
<td>Mark characters</td>
<td>Characters used within the QM to separate field, value and subvalues within a database record or other dynamic array.</td>
</tr>
</tbody>
</table>
Menu
A M-type VOC record that describes a numbered list of options to be displayed to the user. Each option would have an associated sentence to be executed and, optionally, some help text.

Modulus
The number of groups in a hashed file.

Multi-file
A file that is made up from multiple subfiles that share a single dictionary. For example, a sales application might have a multi-file with a subfile for each business region.

Multivalue
Breaking a simple data item (typically a field) into multiple instances of the same type of data.

Overflow
QM uses high performance hashed files in which the location of a record can be deduced from its primary key. If there is insufficient space to store the record at its calculated location, the file system extends the group by adding one or more overflow blocks.

Object
A run time instance of a class module.

Paragraph
A VOC record containing a script of commands to be executed. Paragraphs can include special commands to provide conditional execution, loops, jumps, prompts, etc.

Part file
A dynamic file that forms an element of a distributed file.

Part number
The unique number use to identify a specific part file in a distributed file.

Partial select
A mechanism that breaks a large select operation into smaller sections to optimise performance. See Select Lists.

Partitioning algorithm
A dictionary I-type expression used to transform the record id to a part number in a distributed file.

Phantom
A QM process that runs in the background without a terminal. Phantom processes are started with the PHANTOM command and store a copy of all output that would normally have been displayed on the terminal in the $COMO file.

Phrase
A part of a sentence, excluding the verb. Phrase entries may appear in the VOC or in dictionaries. Phrases are used as short forms in commands, to set defaults for the query processor, and to link fields within an association.

Pick style
A reference to the command and programming language style found in the Pick multivalue database product and others that adopt this style. (The first multivalue database environment was created by Dick Pick).

Primary key
A character string that uniquely identifies a record in a file. QM supports keys from 1 to 255 characters in length but lengths over 63 characters require the MAXIDLEN configuration parameter to be amended.

Proc
The predecessor of paragraphs found in other multivalue database products and supported in QM for ease of migration. Development of new Procs is discouraged.

Process dump file
A text file optionally generated at a run time application error. This file contains a full report on the state of the application at the time of the error.
Publisher | The system that is exporting file updates in a system configured for data replication.
QMBasic | The programming language used to develop QM applications.
QMCClient | An interface that allows access to QM from other languages such as Visual Basic or C.
QMFix | A tool for checking the internal structure of QM files after a system crash.
QMIIdx | A tool for updating internal pointers if an alternate key index file is moved.
qmlnxd | A daemon process that runs on non-Windows systems to perform background system monitoring tasks.
QMNNet | An integrated component of QM that allows access to data stored on another QM server with full support for locking.
Record | An item stored in a database file. The file system accesses data at the record level. Interpretation and manipulation of the content of the record is up to the application and related system tools. A record is usually formed from multiple fields which may in turn be broken down into values and subvalues.
Relational Database | A style of database that represents data in the form of tables (relations).
Replication | A mechanism whereby updates made to selected files on a publisher system are replicated on one or more subscriber systems.
Secondary key | A data item in a record, or a value calculated from data in the record, that is to be used to construct an alternate key index so that records can be selected based on this value.
Select list | A list of items, usually primary keys, to be processed. QM provides memory resident numbered select lists that are private to the process using them and disk based named lists that can be shared or retained for later use.
Sentence | A complete QM command or the start of one. A sentence may be typed at the command prompt or it may be stored in the VOC for later execution simply by typing the name of the VOC record.
String | A sequence of characters stored as a data item.
Subvalue | A subdivision of a value to represent multiple instances of the same type of data within the value. For example, an order processing system might have a pair of associated multivalued fields storing the product number and quantity for each item ordered. If it was necessary to store the serial number of each item shipped, this would be a subvalued field in the same association.
Subscriber | The system that is receiving exported file updates in a system configured for data replication.
Table | Another name for a database file.
Terinfo | An internal database that stores details of the control codes appropriate to all terminal types supported by QM.
Transaction | A related set up updates to a database that must either all happen or none must happen.
<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trigger</td>
<td>A user written QMBasic subroutine that is executed whenever selected operations are performed against a file. Triggers get their name from their use to trigger other related updates but they can also be used for data validation.</td>
</tr>
<tr>
<td>Unicode</td>
<td>A standard character set that, in the form used by QM, allows 65536 different character values.</td>
</tr>
<tr>
<td>Value</td>
<td>A subdivision of a field to represent multiple instances of the same type of data. For example, an order processing system might have multiple values stored in the product number field of an order.</td>
</tr>
<tr>
<td>Verb</td>
<td>The command word in a sentence. This is always the first word and corresponds to a V-type VOC entry.</td>
</tr>
<tr>
<td>Virtual attribute</td>
<td>Another name for an I-type.</td>
</tr>
<tr>
<td>VOC</td>
<td>A file found in all accounts that contains a list of all the words and symbols that can be used in commands and details of how they should be processed. By changing the VOC, it is possible to extend or modify the QM command language.</td>
</tr>
<tr>
<td>Vocabulary</td>
<td>See VOC.</td>
</tr>
</tbody>
</table>
Index

- ! -

! 595
| !ABSPATH() 1467
| !ACCOUNT.RULES() 1468
| !ATVAR() 1469
| !CALLHTTP() 1470
| !CED() 1472
| !CHMOD() 1473
| !DUMP() 1474
| !ERRTEXT() 1475
| !EXC.STACK() 1476
| !FINDPROG() 1477
| !GETPU() 1478
| !LISTU() 1479
| !MULTISORT 1481
| !OSCOPY() 1483
| !PACKAGE() 1484
| !PARSER() 1485
| !PATHTKN() 1487
| !PCL() 1488
| !PHLOG() 1490
| !PICK() 1491
| !PICKLIST() 1493
| !PSTAT() 1494
| !QMCLIENT 1496
| !SCREEN() 1498
| !SETPU() 1499
| !SETVAR() 1500
| !SORT() 1501
| !USERNAME() 1503
| !USERNO() 1504
| !VIEW() 1505
| !VOCREC() 1506

- $ -

$* 855
| $BASIC.OPTIONS 278
| $CATALOG 856
| $CATALOGUE 856
| $COMMAND.STACK 66
| $COPYRIGHT 858
| $DEBUG 859
| $DEBUG.OPTIONS record 1507
| $DEFINE 860
| $ECHO 256
| $EJECT 875
| $ELSE 863
| $ENDIF 863
| $ERROR 861
| $EXECUTE 862
| $IFDEF 863
| $IFNDEF 863
| $INCLUDE 865
| $INSERT 865
| $LIST 866
| $MODE 867
| $MODE shortcuts 1732
| $NO.CATALOG 872
| $NO.CATALOGUE 872
| $NO.XREF 873
| $NOCASE.STRINGS 874
| $PAGE 875
| $PRIVATE.CATALOGUE 286, 348, 461
| $QMCALL 876
| $QUERY.DEFAULTS 640, 642
| $STOP 877
| $UNDEFINE 878
| $WARNING 879

- @ -

@(x,y) function 902
@ABORT.CODE 1458
@ABORT.MESSAGE 1458
@ACALL.FNO 1458
@ACCOUNT 1458
@ANS 1459
@BPV 1459
@COMMAND 1459
@COMMAND.STACK 1459
@CONV 1459
@CRTHIGH 1459
@CRTWIDE 1459
@DATA 1459
@DATA.PENDING 1459
@DATE 1459
@DAY 1459
@DICTRECS 1459
@DRIVE 1459
@DS 1459
@EXCEPTION 1459
@EXCEPTION.DATA 1459
@EXCEPTION.ORIGIN 1459
@EXCEPTION.STACK 1460
@FILE.NAME 1460
@FILENAME 1460
@FMT 1460
@FNAME 1460
@FNO 1460
@GID 1460
@HOSTNAME 1460
@ID 1460
@IP.ADDR 1460
@ITYPE.MODE 1460
@LEVEL 1460
@LINE 1460
@LOGNAME 1460
@LPTRHIGH 1460
@LPTRWIDE 1460
@MBUTTON 1460
@MCOL 1460
@MONTH 1460
@MROW 1461
@NB 1461
@NI 1461
@NULL 1461
@OPTION 1461
@PARASENTENCE 1461
@PATH 1461
@PIB 1461
@POB 1461
@PSTAT 1461
@QM.GROUP 1461
@QMSYS 1461
@RECORD 1461
@SELECTED 1461
@SENTENCE 1461
@SEQNO 1461
@SIB 1461
@SOB 1461
@SOCKET 1461
@SYDFIL 834, 1462
@SYS.BELL 1462
@SYSTEM.RETURN.CODE 1462
@TERM.TYPE 1462
@TIME 1462
@TRANSACTION.ID 1462
@TRANSACTION.LEVEL 1462
@TRIGGER.RETURN.CODE 1462
@TTY 1462
@UID 1462
@USER 1462
@USER.NO 1462
@USER.RETURN.CODE 1462
@USER0 1462
@USER1 1462
@USER2 1462
@USER3 1462
@USER4 1462
@USERNO 1462
@VOC 1462
@WHERE 1462
@WHO 1463
@YEAR 1463
@YEAR4 1463

- ~ -

~ 768

- < -

< 736
<= 732
<> 745

- = -

= 713
<= 732
=> 721
CALC 691
CALCULATE 691
CALL 940
CALL conversion (CALL) 172
CAPTION 722
CASE 924
CASING 944
CATALOG 286
CATALOGUE 286
CATALOGUED() 945
CATCH 1422
CATS() 946
CAUGHT() 947
CD 311
CED 289
CEIL() 948
CHAIN 949
Chained select 649
CHANGE() 950
CHANGE.KEY.PASSWORD 291
CHAR() 951
CHAR.MAP 292
Character conversion (MCx) 188
Character encoding conversions 211
Character encodings 53
Character maps 44
Character maps (non-ECS) 52
Character values 1455
CHARMAP configuration parameter 1681
CHARS() 951
CHECKSUM() 952
CHILD() 954
CLASS 955
CLEAN.ACCOUNT 293
CLEANUP configuration parameter 1681
CLEAR 957
CLEAR COMMON 958
CLEAR.ABORT 294
CLEAR.DATA 295
CLEAR.FILE 296
CLEAR.INPUT 297
CLEAR.LOCKS 298
CLEAR.PROMPTS 299
CLEAR.SELECT 300
CLEAR.STACK 301
CLEARCOMMON 958
CLEARDATA 295, 959
CLEARFILE 960
CLEARINPUT 297, 961
CLEARPROMPTS 299
CLEARSELECT 300, 962
CLIPORE configuration parameter 1681
CLOSE 963
CLOSE.ECI 302
CLOSE.SOCKE 964
CLOSESEQ 965
CLR 303
CMDLOG configuration parameter 1681
CMDSTACK configuration parameter 1682
CNAME 304
Code pages 1699
CODEPAGE 1699
CODEPAGE configuration parameter 1682
COL.HDG 693
COL.HDG.ID 694
COL.HDR.SPACE 695
COL.HDR.SUP 696
COL.HDR.SUPP 696
COL.SPACES 697
COL.SPCS 697
COL.SUP 700
COL1() 966
COL2() 967
COL-HDR-SUP 696
COL-HDR-SUPP 696
Collection editor 289
COLLECTION() 968
COL-SUPP 700
Command editor 68
Command editor options 68
Command line options 1718
Command line parser 1485
Command logging 1714
Command Parsing 64
Command scripts 40
Command stack 66
COMMAND.LOGGING 306
Comment 255
COMMIT 925
COMMON 969
Common blocks (QMBasic) 809
COMO 307
COMPARE() 972
COMPARE.FILES 309
COMPARES() 972
COMPILE_CONSTRAINTS 310
COMPILE.DICT 311
Compiler directives (QMBasic) 854
Concatentation conversion (C) 173
Concordance data 277
CONFIG 312
CONFIG() 974
Configuration parameters 1681
CONFIGURE.FILE 313
CONNECT.PORT() 975
Connection pooling 42
Constants (QMBasic) 799
CONTINUE 976
CONV 698
Conversion codes 166
Conversion Tools 1737
CONVERT 977
CONVERT() 977
COPY 316
COPY.LIST 318
COPY.LISTP 320
COPYI 316
COPYP 322
Correlatives 149
COS() 979
COUNT 659
COUNT() 980
COUNT.SUP 699
COUNTS() 980
CREATE 981
CREATE.ACCOUNT 324
CREATE.FILE 326, 982
CREATE.INDEX 331
CREATE.KEY 333
CREATE.SECURE.KEY 333
CREATE.SERVER.SOCKET 983
CREATE.USER 335
Creating and deleting files 124
Creating and modifying data 145
CROP() 985
CRT 1008
CS 303
CSV 701
CSV.MODE 987
CSVDQ() 986
CT 336
CUMULATIVE 703
CURRENCY 471
CWD 338

- D -

D3CONV 1737

DATA 339, 989
Data collection files 134
Data collections 803
Data Encryption 234
Data field definitions 74
Data Integrity Constraints 231
Data types (QMBasic) 820
DATE 340
Date conversion (D) 174
DATE() 990
DATE.FORMAT 341
Dates 116
DBL.SPC 704
DBL-SPC 704
DCOUNT() 991
DEADLOCK configuration parameter 1682
DEBUG 342, 992
Debugger 1507
DECATALOG 343
DECRYPT() 993
DECRYPT.FILE 344
DEFAULT 705
Default file variable 834
Default listing phrase 646
DEFFUN 995
Deinstallation 28
DEL 997
DELETE 345, 999
DELETE() 997
DELETE.ACCOUNT 347
DELETE.CATALOG 348
DELETE.CATALOGUE 348
DELETE.COMMON 349
DELETE.DEMO 350
DELETE.FILE 351
DELETE.INDEX 352
DELETE.KEY 353
DELETE.LIST 354
DELETE.PRIVATE.SERVER 355
DELETE.SERVER 139, 355
DELETE.USER 356
DELETELIST 1000
DELETESEQ 1001
DELETEU 999
DELIMITER 706
DET.SUP 708
DET-SUPP 708
Device Licensing 35
DFPART() 1002
DHCACHE configuration parameter 1682
Dictionaries 146
Dictionary A and S-type records 147
Dictionary C-type records 155
Dictionary D-type records 156
Dictionary E-type records 157
Dictionary I-type records 158
Dictionary L-type records 159
Dictionary PH-type records 161
Dictionary X-type records 162
DIGEST() 1003
DIM 1004
DIMENSION 1004
DIR() 1006
Directory files 126
DIRSELSZ configuration parameter 1682
DISABLE.INDEX 357
DISABLE.KEY 358
DISABLE.KEY (QMBasic) 1011
DISABLE.PUBLISHING 359
DISINHERIT 1007
DISPLAY 360, 1008
Display clause 646
DISPLAY.LIKE 710
DISPLAY.NAME 693
DISPLAY.WIDTH() 1010
DISPLAY.WIDTHS() 1010
Distributed files 135
DIV() 1012
DIVS() 821
DO.REMOTE 361
DOWNCASE() 1013
DPARSE 1014
DPARSE.CSV 1014
DQUOTE() 1269
DTX() 1016
DUMP 362
DUMPDIR configuration parameter 1682
Dynamic files 130

- E -

EBCDIC() 1017
ECHAR() 1018
ECHO 363, 1019
ECS 44
ECS.MAP 364
ECSMAP configuration parameter 1682
ED 365
EDIT.CONFIG 376
EDIT.LIST 378
EDIT.MAP 379

- F -

FCONTROL() 1043
F-correlative conversion (F) 182
F-Correlatives 151
FEATURE configuration parameter 1682
Field extraction 212
FIELD() 1045
FIELDS() 1045
FIELDSTORE() 1047
FILE 1049
File definitions 75
File processing 824
File translation conversion 207
FILE.EVENT() 1051
FILE.SAVE 388
FILE.STAT 390
FILEINFO() 1053
FILELOCK 1056
FILERULE configuration parameter 1683
Files 122
FILEUNLOCK 1058
FIND 1059
FIND.ACCOUNT 392
FIND.FILE 393
FIND.PROGRAM 394
FINDSTR 1060
FIRST 769
FIX() 1061
FIXUSERS configuration parameter 1683
FLOOR() 948
FLTDIFF configuration parameter 1683
FLUSH 1062
FLUSH.DH.CACHE 1063
FMT 715
FMT() 1064
FMTDW() 1065
FMTDWS() 1065
FMTS() 1064
FOLD() 1066
FOLDDW() 1067
FOLDDWS() 1067
FOLDS() 1066
FOOTER 716
FOOTING 716, 1069
FOR 718, 1071
FORCE 719
FORM.LIST 397
FORMAT 395
Format specifications 213
FORMCSV() 1074
FORMLIST 1075
F-pointers 75
FROM 720
FSTAT 398
FSIZE configuration parameter 1684
FUNCTION 1076

- G -

GDI configuration parameter 1684
GE 721
GENERATE 400
GES() 1078
GET 1079, 1265
GET(ARG.) 1080
GET.LICENSE.UPDATE 401
GET.LICENSE.UPDATE 401
GET.LIST 402
GET.MESSAGES() 1082
GET.PORT.PARAMS() 1083
GET.STACK 403
GETLIST 1084
GETLISTV 1084
GETLOCKS() 1085
GETNLCS() 1085
GETPUS() 1087
GETREM() 1089
Glossary of terms 1744
GO 404
GO TO 1092
GOSUB 1090
GOTO 1092
GRAND.TOTAL 722
GRANT.KEY 405
GREATER 724
Group conversion (G) 183
GROUP.SIZE 313, 326
GROUPSTORE 1093
GRPSIZE configuration parameter 1684
GT 724
GTS() 1095

- H -

HDR.SUP 725
HDR-SUPP 725
HEADER 726
HEADING 726, 1096
HELP 406
Hot Spot Monitor 407
HSM 407
HTML data conversion (HD) 184
HTML URL conversion (HU) 185
- I -

ICONV() 1099
ICONVS() 1099
ID.ONLY 728
ID.SUP 729
IDIV() 1100
ID-SUPP 729
IF 410, 1101
IFS() 1102
IN 730, 1103
INCLUDE 865
INDEX() 1104
INDEXS() 1104
INDICES() 1105
INHERIT 1107
INHIBIT.LOGIN 412
Inline prompts 103
INMAT() 1108
INPUT 1110
INPUT @ 1113
INPUTCLEAR 961
INPUTCSV 1117
INPUTDW 1110
INPUTERR 1257
INPUTFIELD 1118
INPUTFIELDV 1118
INPUTIF 1121
INPUTNULL 1122
INPUTTRAP 1123
INS 1124
INSERT() 1124
Installation 14
Installing on Linux, FreeBSD, AIX or Solaris 16
Installing on Windows 14
INT() 1126
Integer conversion (IS, IL) 186
Interrupting commands 69
INTPREC configuration parameter 1684
Introduction 8
INVCASE configuration parameter 1685
IPADDR configuration parameter 1685
IPV6 configuration parameter 1685
IS.ALNUM() 1127
IS.ALPHA() 1127
IS.DIGIT() 1127
IS.ECS() 1129
IS.GRAPH() 1127
IS.MARK() 1127
IS.NULL() 1130
IS.SPACE() 1127
IS.USER.CHAR() 1127
IS.WIDE() 1127
IS.WIDESTR() 1131
I-type expressions 164
ITYPE() 1132

- J -

JBUILD() 1133
JNLDIR configuration parameter 1685
JNLMODE configuration parameter 1685
JNLSIZE configuration parameter 1685
JPARSE() 1134

- K -

KEEP.SELECT 1135
KEYCODE() 1136
KEYCODEV() 1136
KEYEDIT 1138
KEYEXIT 1140
KEYIN() 1141
KEYINC() 1141
KEYINCV() 1142
KEYINR() 1141
KEYINV() 1142
KEYREADY() 1143
KEYTRAP 1144
Keywords 77
Keywords (query processor) 664

- L -

LABEL 731
Labels (QMBasic) 811
LARGE.RECORD 313, 326
LAST() 1145
LASTS() 1145
LE 732
LEFT() 1146
LEN() 1147
Length conversion (L) 187
LENS() 1147
LES() 1148
LESS 736
LGNWAIT configuration parameter 1685
LICENSE configuration parameter 1685
Licensing 14
LIKE 733
Limits 1740
Limits (QMBasic) 880
Links 640
LIST 653
LIST.COMMON 413
LIST.DF 414
LIST.DIFF 415
LIST.FILES 416
LIST.INDEX 418
LIST.INTER 420
LIST.ITEM 655
LIST.KEYS 421
LIST.LABEL 656
LIST.LOCKS 423
LIST.PHANTOMS 424
LIST.PRINTERS 425
LIST.READU 426
LIST.REPLICATED 428
LIST.SERVERS 139, 429
LIST.TERM.TYPES 430
LIST.TRIGGERS 431
LIST.UNION 432
LIST.USERS 433
LIST.VARS 434
LISTDICT 435
LISTF 436
LISTFL 437
LISTFR 438
LISTINDEX() 1149
LISTK 439
LISTM 440
LISTPA 441
LISTPH 442
LISTPQ 443
LISTQ 444
LISTR 445
LISTS 446
LISTU 447
LISTV 448
LN() 1150
LOAD.LANGUAGE 449, 1713
LOCAL 1151
LOCATE 1153
LOCATE() 1153
LOCK 450, 1157
LOCK.TRACER 452
LOCKING 734
Locks 217
Logging in 35
LOGIN paragraph 40
Login process 38
LOGIN VOC entry 38
LOGIN.PORT 454
LOGMSG 455, 1158
LOGOUT 456
LOGTO 457
LOOP 458, 1159
LOOP / REPEAT 458
LOWER() 1160
LPTR 735
LPTRHIGH configuration parameter 1685
LPTRWIDE configuration parameter 1685
LT 736
LTS() 1161
- M -
MAKE.INDEX 459
MAP 461
MARGIN 737
MARK.MAPPING 1162
Masked decimal conversion (MD, MR) 190
MASTER.LOGIN 38
MASTER.LOGIN paragraph 40
MAT 1163
MAT() 1165
MATBUILD 1166
MATCH operator (QMBasic) 812
MATCHES (Query processor) 733
MATCHES operator (QMBasic) 812
MATCHESS() 1167
MATCHFIELD() 1169
MATCHFIELDS() 1169
MATCHING (Query processor) 733
Matching templates 109
MATPARSE 1172
MATREAD 1174
MATREADCSV 1176
MATREADL 1174
MATREADU 1174
Matrix file i/o 829
MATSTR() 1178
MATWRITE 1179
MATWRITECSV 1181
MATWRITEU 1179
MAX 738
MAX() 1183
MAXCALL configuration parameter 1685
MAXIDLEN configuration parameter 1686
MAXIMUM() 1184
MAXRLOCK configuration parameter 1686
MD5() 1185
MED 462
MEDIAN 739
Menu definitions 78
MERGE.LIST 465
MERGE.LOAD 313, 326
MERGELIST() 1186
MESSAGE 467
Migrating to OpenQM 1728
Migration 1729
MIN 741
MIN() 1188
MINIMUM() 1189
MINIMUM.MODULUS 313, 326
MOD() 1190
MODE 742
MODIFY 468
MODS() 1190
Monitoring the system 1711
MOUSE 1192
Mouse input 1136
Moving an account 32
MULS() 821
MULTI.VALUE 744
Multifiles 326
Multi-tenanting 24
Multivalue functions 821
MULTIVALUED 744
MUSTLOCK configuration parameter 1686
Multi-valued database 10
MVConnect 58, 59
MVCONV 1737
MVDATETIME() 1193
MVDATE.TIME() 1194
MVIEPOCH() 1195
MVTIME() 1196

- N -
NAP 1197
National language support 471
NE 745
NEGI() 1198
NES() 1199
NETDELAY configuration parameter 1686
NETFILES configuration parameter 1686
NETWAIT configuration parameter 1686
Network file access 139
NEW.PAGE 746
NEXT 1071
NLS 471
NO 747
NO.CASE 749
NO.GRAND.TOTAL 750
NO.INDEX 751
NO.MATCH 752
NO.NULLS 753
NO.PAGE 754
NO.SPLIT 755
NOBUF 1200
Non-English messages 1713
NOPAGE 754
NOT 745
NOT (query processor) 748
NOT() 1201
NOT.IN 756
NOT.MATCHING 782
NS() 1205
NSELECT 472
NULL 1202
NUM() 1203
NUM.SUP 655
NUMERIC() 1204
NUMFILES configuration parameter 1687
NUMLOCKS configuration parameter 1687
NV() 1205

- O -
Object orientated programming 836
OBJECT() 1207
OBJECTS configuration parameter 1687
OBJINFO() 1208
OBJMEM configuration parameter 1687
OCONV() 1209
OCONVS() 1209
OFF 503
ON GOSUB 1211
ON GOTO 1212
ON.ABORT paragraph 40
ON.EXIT paragraph 40
ON.LOGTO paragraph 40
ONLY 728
OPEN 1213
OPEN.SOCKEIT() 1222
OPENPATH 1215
OPENSEQ 1217
OPENTEMP 1220
<table>
<thead>
<tr>
<th>Operators (QMBasic)</th>
<th>812</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPTION</td>
<td>473</td>
</tr>
<tr>
<td>OPTION shortcuts</td>
<td>1732</td>
</tr>
<tr>
<td>OPTION()</td>
<td>1224</td>
</tr>
<tr>
<td>OPTIONS configuration parameter</td>
<td>1687</td>
</tr>
<tr>
<td>OR</td>
<td>757</td>
</tr>
<tr>
<td>ORS()</td>
<td>1225</td>
</tr>
<tr>
<td>OS.ERROR()</td>
<td>1226</td>
</tr>
<tr>
<td>OS.EXECUTE</td>
<td>1227</td>
</tr>
<tr>
<td>OSCOPY</td>
<td>480</td>
</tr>
<tr>
<td>OSREAD()</td>
<td>1229</td>
</tr>
<tr>
<td>OSPATH()</td>
<td>1230</td>
</tr>
<tr>
<td>OSDELETE</td>
<td>1233</td>
</tr>
<tr>
<td>OSRENAME()</td>
<td>1234</td>
</tr>
<tr>
<td>OSWRITE</td>
<td>1235</td>
</tr>
<tr>
<td>OUTERJOIN()</td>
<td>1236</td>
</tr>
<tr>
<td>OVERLAY</td>
<td>758</td>
</tr>
</tbody>
</table>

- P -

<table>
<thead>
<tr>
<th>Package licensing</th>
<th>1522</th>
</tr>
</thead>
<tbody>
<tr>
<td>PACKAGE.LICENCE</td>
<td>481</td>
</tr>
<tr>
<td>Packed decimal conversion (MP)</td>
<td>201</td>
</tr>
<tr>
<td>PAGE</td>
<td>1237</td>
</tr>
<tr>
<td>PAGESEQ</td>
<td>759</td>
</tr>
<tr>
<td>PAGINATION</td>
<td>70</td>
</tr>
<tr>
<td>PAM configuration parameter</td>
<td>1687</td>
</tr>
<tr>
<td>PAN</td>
<td>760</td>
</tr>
<tr>
<td>Paragraphs</td>
<td>80</td>
</tr>
<tr>
<td>PARSE()</td>
<td>1238</td>
</tr>
<tr>
<td>PASSWORD</td>
<td>483</td>
</tr>
<tr>
<td>PATHNAME</td>
<td>313, 326</td>
</tr>
<tr>
<td>Pattern matching</td>
<td>109</td>
</tr>
<tr>
<td>Pattern matching conversion (P)</td>
<td>203</td>
</tr>
<tr>
<td>PAUSE</td>
<td>484, 1240</td>
</tr>
<tr>
<td>PCT</td>
<td>761</td>
</tr>
<tr>
<td>PDEBUG</td>
<td>485</td>
</tr>
<tr>
<td>PDUMP</td>
<td>486</td>
</tr>
<tr>
<td>PDUMP (QMBasic)</td>
<td>1241</td>
</tr>
<tr>
<td>PDUMP configuration parameter</td>
<td>1687</td>
</tr>
<tr>
<td>PERCENT</td>
<td>761</td>
</tr>
<tr>
<td>PERCENTAGE</td>
<td>761</td>
</tr>
<tr>
<td>PERFORM</td>
<td>1036</td>
</tr>
<tr>
<td>Permissions</td>
<td>1704</td>
</tr>
<tr>
<td>PHANTOM</td>
<td>487</td>
</tr>
<tr>
<td>PHANTOM (QMBasic)</td>
<td>1242</td>
</tr>
<tr>
<td>PHANTOMS configuration parameter</td>
<td>1688</td>
</tr>
<tr>
<td>Phrase definitions</td>
<td>82</td>
</tr>
<tr>
<td>POOL configuration parameter</td>
<td>1688</td>
</tr>
<tr>
<td>POOL.IDLE</td>
<td>1244</td>
</tr>
</tbody>
</table>

- Q -

<table>
<thead>
<tr>
<th>PORT configuration parameter</th>
<th>1688</th>
</tr>
</thead>
<tbody>
<tr>
<td>PORTMAP configuration parameter</td>
<td>1688</td>
</tr>
<tr>
<td>PRECISION</td>
<td>1245</td>
</tr>
<tr>
<td>PRINT</td>
<td>1246</td>
</tr>
<tr>
<td>PRINTCSV</td>
<td>1247</td>
</tr>
<tr>
<td>PRINTER</td>
<td>490</td>
</tr>
<tr>
<td>PRINTER CLOSE</td>
<td>1249</td>
</tr>
<tr>
<td>PRINTER DISPLAY</td>
<td>1250</td>
</tr>
<tr>
<td>PRINTER FILE</td>
<td>1251</td>
</tr>
<tr>
<td>PRINTER NAME</td>
<td>1252</td>
</tr>
<tr>
<td>PRINTER OFF</td>
<td>1248</td>
</tr>
<tr>
<td>PRINTER ON</td>
<td>1248</td>
</tr>
<tr>
<td>PRINTER RESET</td>
<td>1253</td>
</tr>
<tr>
<td>PRINTER SETTING</td>
<td>1254</td>
</tr>
<tr>
<td>PRINTER.SETTING()</td>
<td>1255</td>
</tr>
<tr>
<td>PRINTERR</td>
<td>1257</td>
</tr>
<tr>
<td>Printing</td>
<td>111</td>
</tr>
<tr>
<td>PRIVATE</td>
<td>1258</td>
</tr>
<tr>
<td>Private vocabulary</td>
<td>71</td>
</tr>
<tr>
<td>Process dump files</td>
<td>1513</td>
</tr>
<tr>
<td>PROCREAD</td>
<td>1259</td>
</tr>
<tr>
<td>PROCs</td>
<td>83</td>
</tr>
<tr>
<td>PROCWRITE</td>
<td>1260</td>
</tr>
<tr>
<td>PROFILER</td>
<td>492</td>
</tr>
<tr>
<td>PROGRAM</td>
<td>1261</td>
</tr>
<tr>
<td>PROMPT</td>
<td>1262</td>
</tr>
<tr>
<td>PSTAT</td>
<td>494</td>
</tr>
<tr>
<td>PTERM</td>
<td>496</td>
</tr>
<tr>
<td>PTERM()</td>
<td>1263</td>
</tr>
<tr>
<td>PUBLIC</td>
<td>1265</td>
</tr>
<tr>
<td>PUBLISH</td>
<td>499</td>
</tr>
<tr>
<td>PUBLISH.ACCOUNT</td>
<td>500</td>
</tr>
<tr>
<td>PVOC</td>
<td>71</td>
</tr>
<tr>
<td>PWDELAY configuration parameter</td>
<td>1688</td>
</tr>
<tr>
<td>PWR()</td>
<td>1268</td>
</tr>
</tbody>
</table>

QM command options 1718
QM Commands 250
QMBasic 796
QMBasic debugger 1507
QMBasic functions 881, 892
QMBasic limits 880
QMBasic overview 797
QMBasic statements 881, 892
QMCall 1547
QMCallx 1549
QMChange() 1551
QMChecksum() 1553
<table>
<thead>
<tr>
<th>Function</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>QMClearFile()</td>
<td>1554</td>
<td></td>
</tr>
<tr>
<td>QMClearSelect</td>
<td>1555</td>
<td></td>
</tr>
<tr>
<td>QMClient</td>
<td>1530</td>
<td></td>
</tr>
<tr>
<td>QMCLIENT configuration parameter</td>
<td>1688</td>
<td></td>
</tr>
<tr>
<td>QMClient security</td>
<td>1540</td>
<td></td>
</tr>
<tr>
<td>QMClose</td>
<td>1557</td>
<td></td>
</tr>
<tr>
<td>QMConnect()</td>
<td>1558</td>
<td></td>
</tr>
<tr>
<td>QMConnected()</td>
<td>1560</td>
<td></td>
</tr>
<tr>
<td>QMConnectionType</td>
<td>1561</td>
<td></td>
</tr>
<tr>
<td>QMConnectLocal()</td>
<td>1563</td>
<td></td>
</tr>
<tr>
<td>QMConnectPool()</td>
<td>1565</td>
<td></td>
</tr>
<tr>
<td>qmconv</td>
<td>1723</td>
<td></td>
</tr>
<tr>
<td>QMCreateObject()</td>
<td>1567</td>
<td></td>
</tr>
<tr>
<td>QMDcount()</td>
<td>1569</td>
<td></td>
</tr>
<tr>
<td>QMDecrypt()</td>
<td>1570</td>
<td></td>
</tr>
<tr>
<td>QMDel()</td>
<td>1571</td>
<td></td>
</tr>
<tr>
<td>QMDelete()</td>
<td>1572</td>
<td></td>
</tr>
<tr>
<td>QMDeleteu()</td>
<td>1574</td>
<td></td>
</tr>
<tr>
<td>QMDestroyObject()</td>
<td>1575</td>
<td></td>
</tr>
<tr>
<td>QMDisconnect</td>
<td>1576</td>
<td></td>
</tr>
<tr>
<td>QMDisconnectAll</td>
<td>1577</td>
<td></td>
</tr>
<tr>
<td>QMEncrypt()</td>
<td>1578</td>
<td></td>
</tr>
<tr>
<td>QMEndCommand</td>
<td>1582</td>
<td></td>
</tr>
<tr>
<td>QMEvalConv()</td>
<td>1583</td>
<td></td>
</tr>
<tr>
<td>QMEvaluate()</td>
<td>1583</td>
<td></td>
</tr>
<tr>
<td>QMExecute()</td>
<td>1585</td>
<td></td>
</tr>
<tr>
<td>QMExitPackage()</td>
<td>1587</td>
<td></td>
</tr>
<tr>
<td>QMExtract()</td>
<td>1588</td>
<td></td>
</tr>
<tr>
<td>QMField()</td>
<td>1590</td>
<td></td>
</tr>
<tr>
<td>qmfix</td>
<td>1720</td>
<td></td>
</tr>
<tr>
<td>QMFrees()</td>
<td>1591</td>
<td></td>
</tr>
<tr>
<td>QMGet()</td>
<td>1592</td>
<td></td>
</tr>
<tr>
<td>QMGetArg()</td>
<td>1594</td>
<td></td>
</tr>
<tr>
<td>QMGetSession()</td>
<td>1595</td>
<td></td>
</tr>
<tr>
<td>QMGetVar()</td>
<td>1596</td>
<td></td>
</tr>
<tr>
<td>QMIConv()</td>
<td>1597</td>
<td></td>
</tr>
<tr>
<td>QMIConvs()</td>
<td>1597</td>
<td></td>
</tr>
<tr>
<td>qmidx</td>
<td>1722</td>
<td></td>
</tr>
<tr>
<td>QMIndices()</td>
<td>1598</td>
<td></td>
</tr>
<tr>
<td>QMIns()</td>
<td>1599</td>
<td></td>
</tr>
<tr>
<td>QMIsECS()</td>
<td>1601</td>
<td></td>
</tr>
<tr>
<td>QM Locate()</td>
<td>1602</td>
<td></td>
</tr>
<tr>
<td>QMLogto()</td>
<td>1605</td>
<td></td>
</tr>
<tr>
<td>QMMarkMapping</td>
<td>1607</td>
<td></td>
</tr>
<tr>
<td>QMMatch()</td>
<td>1608</td>
<td></td>
</tr>
<tr>
<td>QMMatchfield()</td>
<td>1610</td>
<td></td>
</tr>
<tr>
<td>QMNet</td>
<td>139</td>
<td></td>
</tr>
<tr>
<td>QMNextPartial()</td>
<td>1611</td>
<td></td>
</tr>
<tr>
<td>QMOCnv()</td>
<td>1613</td>
<td></td>
</tr>
<tr>
<td>QMOCnv()</td>
<td>1613</td>
<td></td>
</tr>
<tr>
<td>QMOConvs()</td>
<td>1613</td>
<td></td>
</tr>
<tr>
<td>QMOpen()</td>
<td>1614</td>
<td></td>
</tr>
<tr>
<td>QMOpenSeq()</td>
<td>1616</td>
<td></td>
</tr>
<tr>
<td>QMPoolIdle()</td>
<td>1618</td>
<td></td>
</tr>
<tr>
<td>QMRead()</td>
<td>1619</td>
<td></td>
</tr>
<tr>
<td>QMReadBlk()</td>
<td>1621</td>
<td></td>
</tr>
<tr>
<td>QMRead()</td>
<td>1623</td>
<td></td>
</tr>
<tr>
<td>QMReadList()</td>
<td>1625</td>
<td></td>
</tr>
<tr>
<td>QMReadNext()</td>
<td>1627</td>
<td></td>
</tr>
<tr>
<td>QMReadSeq()</td>
<td>1629</td>
<td></td>
</tr>
<tr>
<td>QMRead()</td>
<td>1631</td>
<td></td>
</tr>
<tr>
<td>QMRecordlock</td>
<td>1633</td>
<td></td>
</tr>
<tr>
<td>QMRecordlocked()</td>
<td>1635</td>
<td></td>
</tr>
<tr>
<td>QMRelease</td>
<td>1637</td>
<td></td>
</tr>
<tr>
<td>QMReplace()</td>
<td>1639</td>
<td></td>
</tr>
<tr>
<td>QMRespond()</td>
<td>1641</td>
<td></td>
</tr>
<tr>
<td>QMRESTORE</td>
<td>1729</td>
<td></td>
</tr>
<tr>
<td>QMRevision()</td>
<td>1643</td>
<td></td>
</tr>
<tr>
<td>QMRTrans()</td>
<td>1662</td>
<td></td>
</tr>
<tr>
<td>QMSAVE</td>
<td>1729</td>
<td></td>
</tr>
<tr>
<td>QMSave</td>
<td>1654</td>
<td></td>
</tr>
<tr>
<td>QMSave</td>
<td>1654</td>
<td></td>
</tr>
<tr>
<td>QMSeek()</td>
<td>1644</td>
<td></td>
</tr>
<tr>
<td>QMSelect</td>
<td>1645</td>
<td></td>
</tr>
<tr>
<td>QMSelectIndex</td>
<td>1647</td>
<td></td>
</tr>
<tr>
<td>QMSelectLeft()</td>
<td>1649</td>
<td></td>
</tr>
<tr>
<td>QMSelectPartial</td>
<td>1652</td>
<td></td>
</tr>
<tr>
<td>QMSelectRight()</td>
<td>1649</td>
<td></td>
</tr>
<tr>
<td>QMSet</td>
<td>1654</td>
<td></td>
</tr>
<tr>
<td>QMSetLeft</td>
<td>1656</td>
<td></td>
</tr>
<tr>
<td>QMSetRight</td>
<td>1656</td>
<td></td>
</tr>
<tr>
<td>QMSetSession()</td>
<td>1659</td>
<td></td>
</tr>
<tr>
<td>QMStatus()</td>
<td>1660</td>
<td></td>
</tr>
<tr>
<td>QMSystem()</td>
<td>1661</td>
<td></td>
</tr>
<tr>
<td>qmtic</td>
<td>1698</td>
<td></td>
</tr>
<tr>
<td>QMTrans()</td>
<td>1662</td>
<td></td>
</tr>
<tr>
<td>QMTrapCallAbort</td>
<td>1664</td>
<td></td>
</tr>
<tr>
<td>QMTxn()</td>
<td>1665</td>
<td></td>
</tr>
<tr>
<td>QMWEOFSeq()</td>
<td>1667</td>
<td></td>
</tr>
<tr>
<td>QMWrite</td>
<td>1668</td>
<td></td>
</tr>
<tr>
<td>QMWriteBlk()</td>
<td>1670</td>
<td></td>
</tr>
<tr>
<td>QMWriteSeq()</td>
<td>1671</td>
<td></td>
</tr>
<tr>
<td>QMWriteSeqKey()</td>
<td>1672</td>
<td></td>
</tr>
<tr>
<td>QMWriteu</td>
<td>1676</td>
<td></td>
</tr>
<tr>
<td>QMWriteuSeqKey()</td>
<td>1674</td>
<td></td>
</tr>
<tr>
<td>Q-pointer</td>
<td>96</td>
<td></td>
</tr>
<tr>
<td>QSELECT</td>
<td>502</td>
<td></td>
</tr>
<tr>
<td>Qualified display clauses</td>
<td>646</td>
<td></td>
</tr>
<tr>
<td>Query processing</td>
<td>640</td>
<td></td>
</tr>
<tr>
<td>QUIT</td>
<td>503</td>
<td></td>
</tr>
<tr>
<td>QUOTE()</td>
<td>1269</td>
<td></td>
</tr>
</tbody>
</table>
- R -

Radix conversions (MB, MX) 198
Radix conversions (MCDX, MCXD) 200
RAISE() 1270
RANDOMIZE 1271
RANGE 762
Range check conversion (R) 204
RDIV() 1272
READ 1273
READ.SOCKET() 1275
READBLK 1276
READCSV 1278
READL 1280
READLIST 1282
READNEXT 1284
READSEQ 1286
READU 1288
READV 1290
READVL 1292
READVU 1292
REBUILD.ALL.INDICES 504
RECCACHE configuration parameter 1689
RECORDLOCKED() 1294
RECORDLOCKL 1295
RECORDLOCKU 1295
RECOVER.USERS 1711
REDIM 1004
REDDIMENSION 1004
REDO 505
REFORMAT 658
REGEX() 1296
RELEASE 506, 1297
REM() 1298
REMARK 1299
Remote file pointers 96
Remote pointers 97
REMOVE 1300
REMOVE() 1300
REMOVE.BREAK.HANDLER 1303
REMOVE.DF 507
REMOVEF() 1304
RENAME 304
REPEAT 458, 1159
REPEATING 765
REPLACE() 1306
REPLDIR configuration parameter 1689
Replication 240
REPLMAX configuration parameter 1689
REPLICPORT configuration parameter 1689
REPLSIZE configuration parameter 1689
REPLSRVR configuration parameter 1689
REPORT.SRC 508
REPORT.STYLE 509
REPORTING 766
REQUIRE.INDEX 763
REQUIRE.SELECT 764
RESET.MASTER.KEY 510
RESTORE.ACCOUNTS 511
RESTORE.SCREEN 1308
RETRIES configuration parameter 1689
RETURN 1309
RETURN FROM PROGRAM 1309
RETURN TO 1309
REUSE() 1311
REVOKING() 1313
REVVOKE.KEY 512
REVREMOVE 1300
RIGHT() 1314
RINGWAIT configuration parameter 1689
RMVD() 1315
RND() 1317
ROLLBACK 925
Roman numeral conversion (NR) 202
ROUNDDOWN() 1318
ROUNDUP() 1318
RPL.STATUS 513
RQM 1361
RTRANS() 1411
RUN 514

- S -

SADD() 1320
SAFEDIR configuration parameter 1690
SAID 768
SAMPLE 769
SAMPLED 770
SAVE.LIST 515
SAVE.SCREEN() 1322
SAVE.STACK 516
SAVELIST 1323
SAVING 771
SCAN 648
SCMP() 1325
SCRB 517
SCROLL 772
SDIV() 1320
SEARCH 651
SECURITY 525
Security - application level 1701
Security - permissions 1704
SECURITY configuration parameter 1690
Security subroutines 102
SED 526
SED extension programming 550
SEEK 1326
SEL.RESTORE 567
SELECT 649, 1327
Select lists 220
Select lists in QMClient sessions 1539
SELECT.DIR 568
SELECT.SOCKE() 1329
SELECTE 1331
SELECTINDEX 569, 1332
SELECTINDEXV 1332
SELECTINFO() 1334
Selection clause 644
SELECTLEFT 1335
SELECTLEFTV 1335
SELECTN 1327
SELECTRIGHT 1335
SELECTRIGHTV 1335
SELECTV 1327
Select lists in QMClient sessions 1539
SELECT.DIR 568
SELECT.SOCKE() 1329
SELECTE 1331
SELECTINDEX 569, 1332
SELECTINDEXV 1332
SELECTINFO() 1334
Selection clause 644
SELECTLEFT 1335
SELECTLEFTV 1335
SELECTN 1327
SELECTRIGHT 1335
SELECTRIGHTV 1335
SELECTV 1327
Self-installing applications 1524
SEND 1337
SENTENCE() 1338
Sentences 98
SEQ() 1339
SEQS() 1339
Sequential file i/o 831
Sequential key generation 132
SERIAL configuration parameter 1690
Serial port connection 35
SERVER.ADDR() 1340
SERVER.WINDOW() 1341
SET 570, 1265
SET.ARG 1342
SET.BREAK.HANDLER 1343
SET.DEVICE 572
SET.ECS.MAP() 1344
SET.ENCRYPTION.KEY.NAME 574
SET.EXIT.STATUS 575, 1345
SET.FILE 576
SET.LANGUAGE 577, 1713
SET.MODE 578
SET.PORT.PARAMS() 1347
SET.PRIVATE.SERVER 581
SET.QUEUE 579
SET.SERVER 139, 581
SET.SOCKE.(
SET.SOCKE.MODE() 1348
SET.STATUS 1349
SET.TIMEZONE 1350
SET.TRIGGER 583
SET.VFS.SERVER 584
SET.LEFT 1346
SET.NLS 1351
SET.PORT 586
SET.PTR 587
SET.PU 1353
SET.REM 1355
SET.RIGHT 1346
SET.UP.DEMO 594
SH 595
SH configuration parameter 1690
SH1 configuration parameter 1690
SHARED 1356
SHIFT() 1358
SHOW 661
SHOW.LIST 596
SIGNAL() 1359
Silent command execution 1036, 1227
Simple Web Services 58
SIN() 1360
SINGLE.VALUE 773
SINGLEVALUE 773
SLEEP 597, 1361
SMUL() 1320
Socket inheritance 849
SOCKET.INFO() 1362
Sockets 847
SORT 653
SORT clause 645
SORT() 1364
SORT.COMPARE() 1371
SORT.ITEM 655
SORT.LABEL 656
SORT.LIST 598
SORT.SUP 774
SORT.ADD 1367
SORT.CLEAR 1368
SORT.DA() 1369
SORT.FIRST 775
SORT.SAMPLE 775
Sorting 56
SORT.MEM configuration parameter 1690
SORT.MRG configuration parameter 1690
SORT.NEXT() 1370
SORT.U 447
SORT.WORK configuration parameter 1691
Soundex conversion (S) 205
Index 1765

SOUNDEX() 1373
SP.ASSIGN 599
SP.CLOSE 601
SP.OPEN 601
SP.VIEW 602
SPACE() 1374
SPACES() 1374
SPLICE() 1375
SPLIT.LOAD 313, 326
SPOKEN 768
SPOOL 604
SPOOLER configuration parameter 1691
SQRT() 1376
SQUOTE() 1377
SREFORMAT 658
SRVRLOG configuration parameter 1691
SSELECT 649, 1378
SSELECTN 1378
SSELECTV 1378
SSUB() 1320
Standard subroutines 1464
Startup and shutdown 26
STARTUP configuration parameter 1691
STATUS 605, 1381
STATUS() 1380
STOP 606, 1382
STOPE 1382
STOPM 1382
STRI() 1384
String math 1320
STRINGS 776
STRS() 1384
STYLE 777
Styles (query processing) 640
Stylus input (PDA) 1136
SUBR() 1385
SUBROUTINE 1387
SUBS() 821
SUBSCRIBE 607
SUBSTITUTE() 1389
Substitution conversion (S) 206
SUBSTRDW() 1390
SUBSTRDWS() 1390
SUBSTRINGS() 1391
SUM 660
SUM() 1392
SUMMATION() 1393
SUPP 725
SWAP() 950
SWAPCASE() 1394
SWAPMARKS() 1395
SYSMSG() 1396
System Administration 1680
System id (licensing) 14, 1718
System limits 1740
SYSTEM() 1397

- T -

T.ATT 572
T.DET 611
T.DUMP 608
T.EOD 611
T.FWD 611
T.LOAD 610
T.RDLBL 611
T.READ 611
T.REW 611
T.STAT 611
T.WEOF 611
TAN() 1402
TANDEM 612
TCLREAD 1403
TEMPDIR configuration parameter 1691
TERM 613
Terminal configuration 1693
Terminfo compiler 1698
TERMINFO configuration parameter 1691
Terminfo database 1693
TERMINFO() 1404
TERMINFO.EX() 1404
TESTLOCK() 1405
Text substring conversion 208
THROW 1406
TIME 615
Time conversion (MT) 195
TIME() 1407
TIMEDATE() 1408
TIMEOUT 1409
TIMEOUT configuration parameter 1691
Times 116
TIMEZONE configuration parameter 1691
TO (redirect delimited report) 701, 706
TO (REFORMAT) 780
TO (Selection verbs) 779
Tools 1724
TOTAL 781
TOTAL() 1410
TRANS() 1411
TRANSACTI0N ABORT 1413
TRANSACTI0N COMMIT 1413
TRANSACTION START 1413
Transactions 238
Transferring data 1729
Translation conversion 207
TRANSLITERATE() 1415
Triggers 227
TRIM() 1416
TRIMB() 1418
TRIMBS() 1418
TRIMF() 1419
TRIMFS() 1419
TRIMS() 1420
TRIMW() 1421
TRIMWS() 1421
TRY 1422
TTYGET() 1425
TTYSET 1426
TXCHAR configuration parameter 1691
Type conversion (QMBasic) 820

- U -
UDCONV 1737
UDIMPORT 1729
UMASK 616
UMASK configuration parameter 1692
UNASSIGNED() 1427
UNIQUE 771
UNLIKE 782
UNLOCK 617, 1428
UNLOCK.KEY.VAULT 618
UNTIL 1429
UPCASE() 1430
UPDATE.ACCOUNT 619
UPDATE.LICENSE 620
UPDATE.RECORD 621
UPDATE.RECORD batch mode 624
UPDATE.RECORD visual mode 627
User defined conversions 209
User exits 209
USERPOOL configuration parameter 1692
USING 783
Using socket connections 847
USPLIT configuration parameter 1692
UCONV 1737
UVIMPORT 1729

- V -
VALIDATE() 1431
Variable names (QMBasic) 799
Variables (QMBasic) 802
VARTYPE() 1432
VERIFY.CONSTRAINTS 632
VERSION 313, 326
VERT 784
VERTICALLY 784
VFS 142
VFSCACHE configuration parameter 1692
Virtual File System 142
VOC D-type records 74
VOC file 64, 71
VOC F-type records 75
VOC K-type records 77
VOC M-type records 78
VOC PA-type records 80
VOC PH type records 82
VOC PQ type records 83
VOC Q-type records 96
VOC R-type records 97
VOC S-type records 98
VOC verb definitions 99
VOC V-type records 99
VOC X-type records 100
VOCPATH() 1433
VOID 1435
VSLICE() 1436

- W -
WAIT.FILE.EVENT 1437
WAKE 1439
Web server 1526
WEBSVC 633
WEOFSEQ 1440
WHEN 786
WHERE 637
WHILE 1441
WHO 636
WITH 788
WITHOUT 791
WRITE 1442
WRITE.SOCKET() 1445
WRITEBLK 1446
WRITECVS 1447
WRITELIST 1449
WRITESEQ 1450
WRITESEQF 1450
WRITEU 1442
WRITEV 1452
WRITEVU 1452

- X -

XLATE() 1411
XML 792
XTD() 1454

- Y -

YEARBASE configuration parameter 1692