



# OpenQM

2.12-4

## Reference Manual



# OpenQM

## Reference Manual

---

*Ladybridge Systems Limited*



# OpenQM

## Copyright © Ladybridge Systems, 2011

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**

*Ladybridge Systems Limited  
17b Coldstream Lane  
Hardingstone  
Northampton  
NN4 6DB  
England*

### **Technical Editor**

*Martin Phillips*

### **Cover Graphic**

*Ishimsi*

### **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

<b>Part 1</b>	<b>Introduction to the QM Database</b>	<b>8</b>
<b>Part 2</b>	<b>The Command Environment</b>	<b>40</b>
<b>Part 3</b>	<b>The QM File System</b>	<b>98</b>
<b>Part 4</b>	<b>QM Commands</b>	<b>200</b>
<b>Part 5</b>	<b>Query Processing</b>	<b>546</b>
<b>Part 6</b>	<b>QMBasic</b>	<b>684</b>
<b>Part 7</b>	<b>QMClient API</b>	<b>1260</b>
<b>Part 8</b>	<b>System Administration</b>	<b>1344</b>
<b>Part 9</b>	<b>System Limits</b>	<b>1382</b>
<b>Part 10</b>	<b>Glossary of Terms</b>	<b>1384</b>
	<b>Index</b>	<b>1389</b>



**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 **multivalued 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 multivalued database is often many times quicker than for other methodologies, resulting in lower development costs and simpler maintenance.

OpenQM is the only multivalued database product that is available both as a fully supported closed source commercial product and in open source form for developers who wish to modify the product under the terms of the General Public Licence. In common with all GPL software, the open source version comes with no warranty and no support. This documentation describes the commercial product though most of what is here should apply equally to the open source.

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 multivalued 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 multivalued products to add modern features such as object oriented programming.

### [QMClient API](#)

The QMClient API is a set of functions that can be used from Visual Basic applications to access the QM database. This allows development of applications with a Windows "look and feel". There are variants of this API library for use with other 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

**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.

Items enclosed in curly brackets (e.g. {**FORCE**}) are optional parts of a command or statement. The curly brackets are not part of the data 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 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.

Order no	Date	Customer	Product	Quantity
1001	12 Jan 05	1728	107	4
1002	12 Jan 05	3194	318	2
1003	13 Jan 05	7532	220	1
1004	13 Jan 05	1263	318	2

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.

Order no	Date	Customer	Product	Quantity	Product	Quantity
1001	12 Jan 05	1728	107	4		
1002	12 Jan 05	3194	318	2	452	3
1003	13 Jan 05	7532	220	1		
1004	13 Jan 05	1263	318	2		

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.

Order no	Date	Customer	Product	Quantity	Lines
1001-1	12 Jan 05	1728	107	4	1
1002-1	12 Jan 05	3194	318	2	2
1002-2	12 Jan 05	3194	452	3	2
1003-1	13 Jan 05	7532	220	1	1
1004-1	13 Jan 05	1263	318	2	1

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.

Order no	Date	Customer	Lines
1001	12 Jan 05	1728	1
1002	12 Jan 05	3194	2
1003	13 Jan 05	7532	1
1004	13 Jan 05	1263	1

Detail Ref	Product	Quantity
1001-1	107	4
1002-1	318	2
1002-2	452	3
1003-1	220	1
1004-1	318	2

Things are becoming complex and this is supposed to be a trivial application!

Multivalued 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 "multivalued").

Order no	Date	Customer	Product	Quantity
1001	12 Jan 05	1728	107	4
1002	12 Jan 05	3194	318 452	2 3
1003	13 Jan 05	7532	220	1
1004	13 Jan 05	1263	318	2

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 multivalued 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

Order no	Date	Customer	Product	Quantity	Serial
1002	12 Jan 05	3194	318	2	21222
					21223
			452	3	41272
					41723
					41728

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

```
13527FM3194FM318VM452FM2VM3FM21222SM21223VM41272SM41723SM41728
```

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 multivalued 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:

Item mark	char(255)
Field mark	char(254)
Value mark	char(253)
Subvalue mark	char(252)
Text mark	char(251)

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 Multivalued Databases

The original multivalued 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 TigerLogic 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, originally 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 multivalued 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 PI/open database. At the same time, two start up companies appeared each with their own Unix based multivalued 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, UniVerse and Unidata are now both 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 but not released as a product in its own right until 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 and AIX
- Mac OS X
- A PDA
- Multiple installations on a Linux/Unix system

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

If you purchase a commercial QM licence, you are free to download and install new versions as often as you wish during the free upgrade period (at least one year but this period can be extended). After this period expires, there will be a charge for upgrades. The software comes with free support for the first 60 days beyond which time further support is available on a chargeable basis.

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_2-12-4.exe`, 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.

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 file.
QMTerm	A simple terminal emulator.
QM Online Documentation	Adobe Acrobat style pdf documentation.
QMClient	The Visual Basic API for Windows developers.

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.

Finally, the installer offers to show the readme file.

The installation process does not add QM to the Windows 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.

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 first step is to prepare the memory stick for use with QM. The stick must comply with the USB 2 standards - older USB 1 sticks cannot be used. Download the USBCONFIG tool from the OpenQM website onto the PC that you will use to perform the installation. Alternatively, if you have a Windows version of QM installed on your PC, this tool will already be in the bin subdirectory of the QMSYS account. Run this program, following the on screen instructions. This tool creates a file named memstick on the USB memory stick containing the unique id codes for that stick. The content of this file is only used during licence application, however, you should not amend this file as this may cause QM to fail at a later upgrade.

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.

One or more installations on separate USB memory devices can be run at the same time as a server based installation, however, there are some important rules to observe. Firstly, the configuration parameters must be set such that each installation is listening on different port numbers for incoming network connections (see the PORT and CLIPORT [configuration parameters](#)). 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.

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 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 and AIX

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 `qm_2-12-4` for Linux, `qmf_2-12-4` for FreeBSD, and `qmr_2-12-4` for AIX, where the numeric components identify the release. 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.

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.

It is recommended that the qmInxd daemon should be used to monitor for incoming network connections that are to be routed directly to QM processes (default port 4242 for telnet, 4243 for QMClient, additional ports for port mapping), however, it is possible to use the inetd (etc) network service that is part of the operating system to do this job. In this case, the PORT and CLIPORT configuration parameters should be set to zero and it will be necessary to create a files in the /etc/xinetd.d directory for each port to be monitored. The format of these is similar to the example below.

```
service qmsrvr
{
  id           = qmsrvr
  port        = 4242
  bind        = 0.0.0.0
  type        = UNLISTED
  protocol    = tcp
  flags       = REUSE NAMEINARGS
  socket_type = stream
  wait        = no
  user        = root
  server      = /usr/qmsys/bin/qm
  server_args = /usr/qmsys/bin/qm -n
```

```
log_on_failure += USERID
disable       = no
}
```

For the QMClient port, the `server_args` must also include the `-q` option. It will also be necessary to add lines to the `/etc/services` file to reference these new services if you use `inetd` instead of relying on `qmlnxd` to handle network access.

## Installation on a Power PC 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 `qmm_2-12-4`, where the numeric components identify the release.

Open a terminal window and gain administrative rights (consult your operating system documentation if this concept is new to you). Execute the downloaded file.

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 Mac users may prefer to use `/var/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.

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 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.

## 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_2-12-4`, 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_2-12-4
```

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 Mac users may prefer to use /var/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.

## Installation on a PDA

Note that many devices that run Windows CE or Windows Mobile do not provide the full functionality of a PDA. Users should verify device compatibility.

There are two alternative install files available; qmpda\_2-12-4.exe for Windows Mobile 5.0 and qmpce\_2-12-4.exe for the older Windows CE. The 2-12-4 component of the name identifies the release.

The install file should be executed on a Windows PC to which the PDA is connected via ActiveSync. QM must be installed under the Program Files directory which is the default offered by the installer. If this is an upgrade, you will be asked whether to overwrite or remove existing files. Choose the overwrite option. After the installer has copied the new version to the PDA, complete installation by clicking on the Launch button. This will display a modified version of the licence screen shown below.

The Personal Version of QM is not available on a PDA.

## 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 command prompt by executing QM with the -L option (case insensitive).

The screenshot shows a window titled 'QM' with a sub-header 'LICENCE DETAILS'. The window contains a form with the following fields and values:

```

-----
Licence number   [2930094481]           System id LWXK-QFFW
Max users       [200 ]
Expiry date     [                ]
Authorisation code [HSDSA-FKKND-AIPOE-NNDJD-KLJND]
Security number [40621 ]
Site text [Castle Engineering           ]
-----

Use Return, Tab or Cursor keys to move between fields.
Use Ctrl-X to abort licence data entry.

-----

Enter site text EXACTLY as on your licence information sheet
  
```

You need to enter the details in the boxes surrounded by square brackets as given on your licence paperwork.

Licence number	The unique 10 digit number identifying this licence. If you are using the Personal Version, enter the word Personal and leave all further boxes empty.
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.

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 dealer 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 dealer 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 Questions

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.

### Installing Multiple Separate Instances of QM

On all platforms except Windows and PDAs, QM supports the option to install multiple separate copies that each run in an isolated environment. Because this mode of install does not require root access, it is referred to as "user mode". Each installation must be covered by a separate licence. A user mode installation is initiated by using the -u command line option with the install script described above.

Each user mode installation is identified by a numeric value formed from the device and inode values of 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.

An application can determine whether it is running in a user mode installation by testing the value returned by 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 user mode 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.

When running in user mode, the `qmlnxd` daemon that manages network connections, and all QM processes started by it, run as the user that started QM. Security authentication for users entering by a direct network connection is provided entirely by QM based on information set up with the [CREATE.USER](#) or [ADMIN.USER](#) commands. User authentication can be disabled entirely by use of the [SECURITY](#) command but this is not recommended.

## Upgrades

All QM licences include a free upgrade period during which new versions can be installed to take advantage of new features. The upgrade period is initially either one or ten years and can be extended when it is close to expiry.

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 such as `NEWVOC.MODS` is created to hold these and, on completion of an upgrade, this is copied into `NEWVOC` using a command such as

```
COPY FROM NEWVOC.MODS TO NEWVOC ALL
```

## Compatibility with Other Environments

The various multivalued 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 multivalued 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.

## 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.

This section does not apply to the PDA version of QM.

### Windows Systems

On a Windows system except for Windows 98/ME, the shared memory is created by the QMSvc service. This service must be running for QM to be usable.

Windows 98/ME uses the QMSrvr program in place of QMSvc for network access. QMConsole sessions on Windows 98/ME do not require that this program is running.

QMSvc or QMSrvr 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.

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.

### PDA

Use the File Manager to delete QMSYS from the Program Files directory.

### 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, may not contain spaces and are limited to 32 characters. The format of records in the accounts register is:

Id	Account name
F1	Account pathname
F2	Account description
F3-9	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.

<a href="#">VOC</a>	The vocabulary, a file that controls all aspects of command processing within QM.
BP	Application programs are written using the QMBasic 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 <a href="#">directory file</a> . 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 <a href="#">directory file</a> .
PVOC	The optional <a href="#">private vocabulary</a> 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 <a href="#">directory file</a> .
\$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 <a href="#">directory file</a> when the <a href="#">COMO ON</a> 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.

\$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 <a href="#">SET.QUEUE</a> command and used by the <a href="#">SP.ASSIGN</a> command.
\$HOLD	This is a <a href="#">directory file</a> used to receive output sent to a <a href="#">print unit</a> by a program or standard command that has been set into mode 3 (output to hold file).
\$\$SAVEDLISTS	This is a <a href="#">directory file</a> used to store saved select lists. See the <a href="#">SAVE.LIST</a> and <a href="#">GET.LIST</a> commands for more information.
\$\$SCREENS	This is a <a href="#">dynamic file</a> used to hold screen definitions that are to be shared between accounts. See the description of the <a href="#">SCRIB</a> 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 <a href="#">CATALOGUE</a> 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 <a href="#">VOC</a> 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 <a href="#">LOGTO</a> 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 <a href="#">command stacks</a> when a user exits from QM. On Windows systems, users of QMConsole sessions do not use this file. Instead, the command history is stored in a <a href="#">VOC</a> record named \$COMMAND.STACK.

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.
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 multivalued products and used by the QMBasic <a href="#">STOP</a> , <a href="#">ABORT</a> and <a href="#">ERRMSG</a> 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. (Not present on a PDA)
\$LOGINS	A register of login names of users who may access QM. This file cannot be accessed by directly users. See <a href="#">Application Level Security</a> for more details. (Not present on a PDA)
\$MAP	This file, visible from all accounts, is the default destination for a map of the system catalogue produced with the <a href="#">MAP</a> command.
SYSCOM	The SYSCOM file holds standard definitions for use in QMBasic programs. It also contains versions of the <a href="#">QMClient</a> include record for Visual Basic, C and other languages.
temp	Windows only. This subdirectory holds temporary files that are used to pass control information from the QMSvc service to the QM processes that it starts. All users must have full access to this directory.
terminfo	A subdirectory containing definitions of control data for terminal devices.
terminfo.src	The master source from which the <a href="#">terminfo</a> definitions are built.

Accounts that are no longer needed can be deleted using the [DELETE.ACCOUNT](#) command.

## 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.

PDA users should enter QM by using the QM shortcut in the Programs folder. Some commands may be difficult to use with the default screen size settings. The PDA version can also handle incoming telnet connections using the [NETWAIT](#) command. The remainder of this section is not applicable to PDA systems.

### 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. This is equivalent to opening a Command Prompt window and executing QM from there. 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 other platforms, 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). This technique can also be used from a Command Prompt window on a Windows system. In all cases, 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 using a console session, you can force entry to a specific account using a command line of the form

```
qm -axxx
```

where `xxx` is the account name.

### 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). 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.

You can connect to QM using most terminal emulators. A licence for the AccuTerm emulator from AccuSoft Enterprises 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 AccuSoft website.

On Windows 98/ME, the installation process installs a server program, QMSrvr, 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 later versions of Windows, 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.

### 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 Windows 98/ME.

### Entering QM Directly via a Serial Port

On Windows NT and later, 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 NT and upwards, 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*.

Earlier versions of Windows (98/ME) did not provide 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 other platforms, 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 licence.

Device licensing needs support in the client software. This is included in Windows QMConsole sessions, AccuTerm 2k2 version 5.3c, Winnix version 3, QMTerm and QMClient. With AccuTerm, device licensing must be enabled via a checkbox in the Advanced options of the Tools, Settings, Connection menu.

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 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 it to behave erratically.

## 1.7 The Login Process

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

### User Authentication

On Windows NT and later, users connecting to QM via a network must enter a valid Windows username 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 usernames allowed to use QM. A username 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 username does not appear in the user register, the user runs without administrator rights and an account name prompt is displayed.

On Windows 98/ME, the above mechanism is extended such that QM performs the username and password validation using its own internal user register as these platforms do not provide an adequate user authentication system. The newly created process runs with the Windows user name and access permissions of the user that started the QMServer 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 NT and later apply.

### 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 PDA users, the terminal type is set to pda. 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. On all platforms except the PDA, 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 terminal users, phantom processes and QMClient connections. The [@TTY](#) variable can be tested to determine the user type. The LOGIN paragraph is typically used to set QM [option](#) flags perform security checks, set up printers, set terminal characteristics and enter the application.

#### Example

```
PA
DATE.FORMAT ON
IF @TTY = 'phantom' 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. 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 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. An abort occurring whilst executing the ON.ABORT paragraph will cause a message to be displayed. The paragraph is not re-entered. Aborts occurring in commands started using the QMBasic [EXECUTE](#) statement with the TRAPPING ABORTS option do not execute the ON.ABORT paragraph.

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.

### Example

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

## ON.CONNECT (PDA only)

This item, if present is run when a new incoming telnet connection is established on a PDA using the [NETWAIT](#) command.

## 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
```



**Part**

---



**2**

**The Command Environment**

## 2 The Command Environment

Although applications commonly use the graphical interface capabilities of QM via Visual Basic 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 multivalued 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.

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. 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). They 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.

The stack can be manipulated by commands prefixed by a dot character 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

<b>.An</b> <i>text</i>	Append <i>text</i> to command stack entry <i>n</i> . There must be a space before <i>text</i> . Any additional spaces will be included in the appended data. If <i>n</i> is omitted, the top entry on the stack (position 1) is updated. The text is displayed after modification.
<b>.Cn</b> <i>/old/new/G</i>	Change string <i>old</i> to <i>new</i> in stack entry <i>n</i> . If <i>n</i> is omitted, it defaults to one. The delimiters around <i>old</i> and <i>new</i> may be any non-space character. The space before the first delimiter may be omitted if the delimiter is not a digit. The optional <b>G</b> causes a global replacement, that is, all occurrences of <i>old</i> are replaced by <i>new</i> . If <b>G</b> is not specified, only the first occurrence of <i>old</i> is changed. The text is displayed after modification.
<b>.Dn</b>	Delete stack entry <i>n</i> . If <i>n</i> is omitted, the top stack entry is deleted.
<b>.D</b> <i>name</i>	Delete VOC entry <i>name</i> if it is a sentence or paragraph record. A confirmation prompt is issued prior to deletion.
<b>.In</b> <i>text</i>	Insert <i>text</i> as stack entry <i>n</i> . If <i>n</i> is not specified, <i>text</i> is inserted at the top of the stack. There must be a space before <i>text</i> . Any additional spaces will form part of the inserted entry.
<b>.Ln</b>	List the most recent <i>n</i> commands. The value of <i>n</i> defaults to 20.
<b>.L</b> <i>name</i>	List VOC entry <i>name</i> .
<b>.Rn</b>	Recall stack entry <i>n</i> to the top of the stack without deleting the original copy. If <i>n</i> is omitted, the top entry is duplicated.
<b>.R</b> <i>name</i>	Read VOC entry <i>name</i> to the top of the stack if it is a sentence or paragraph. Field one of the VOC entry is discarded and any continuation lines are merged.
<b>.S</b> <i>name n m</i>	Save stack lines <i>m</i> to <i>n</i> as VOC entry <i>name</i> . The value of <i>m</i> and <i>n</i> may be entered in either order. If <i>m</i> is omitted it defaults to the same value as <i>n</i> . If <i>n</i> 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.
<b>.Un</b>	Convert stack entry <i>n</i> to upper case. <i>n</i> defaults to one if omitted.
<b>.Xn</b>	Execute command <i>n</i> . If <i>n</i> 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. For console users on Windows systems, the command stack will be saved into this record on leaving QM and loaded from it on re-entry. For all other Windows users and on other platforms, 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.

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 and any continuation lines are merged.
- .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:

<b>Ctrl-A</b> or <b>HOME</b>	Move cursor to start of command.
<b>Ctrl-B</b> or <b>Cursor Left</b>	Move cursor left one place.
<b>Ctrl-D</b> or <b>DELETE</b>	Delete character under cursor.
<b>Ctrl-E</b> or <b>END</b>	Move cursor to end of command.
<b>Ctrl-F</b> or <b>Cursor Right</b>	Move cursor right one place.
<b>Ctrl-G</b>	Exit from the command stack and return to a clear command line.
<b>Ctrl-K</b>	Delete all to the right of the cursor.
<b>Ctrl-N</b> or <b>Cursor Down</b>	Display "next" command from command stack.
<b>Ctrl-O</b> or <b>Insert</b>	Toggle insert/overlay mode.
<b>Ctrl-P</b>	Display "previous" command from command stack.
<b>Ctrl-R</b>	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.
<b>Ctrl-T</b>	Interchange characters before cursor.
<b>Ctrl-U</b>	Convert command to uppercase.
<b>Ctrl-Z</b> or <b>Cursor Up</b>	Display "previous" command from command stack.
<b>Backspace</b>	Backspace one place.

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

The command editor operation is 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.
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.

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.

### 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 PVOC with subfiles names that are the uppercase version of the user's login name. The private vocabulary is checked only after attempting 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 .S 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/multivalued 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.

[Click here for a detailed description of dictionaries.](#)

## 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 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. This affects the data part only and may be any compatible combination of the codes listed below. For a multifile, these modes apply to all elements.
  - 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 (dynamic 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.

Applications that make direct access to this field should allow for the presence of codes not in the above list.

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.

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:handler: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 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](#).

When used on a PDA, QM allows selection of menu options using a stylus tap. Clicking on or

below the option line exits from the menu.

### 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 (1) if the menu item is to be allowed, false (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:

```
SUBROUTINE ACCESS(OK, MENU.NAME, KEY)
  BEGIN CASE
    CASE KEY = 'ADMIN'
      OK = (@LOGNAME = 'ADMINISTRATOR')
    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
    END CASE
  RETURN
END
```

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 **In** 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

<b><u>DATA</u></b>	Embedded data for an application
<b><u>IF</u></b>	Conditional execution
<b><u>GO</u></b>	Jumps to labels
<b><u>LOOP</u></b>	Repeated execution of a loop
<b><u>\$ECHO</u></b>	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:

<b>LOGIN</b>	Executed on entry to QM and also when the <b><u>LOGTO</u></b> command is used to switch to a new account.
<b>ON.LOGTO</b>	Executed on use of the <b><u>LOGTO</u></b> command before switching to the new account.
<b>ON.EXIT</b>	Executed on leaving QM by use of the <b><u>QUIT</u></b> command.
<b>ON.ABORT</b>	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 environments.

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:

Token	Buffer	Direct	Indirect
%	Primary input buffer	%1 PIB field 1	##2 PIB field referenced by #2
#	Active output buffer	#1 AOB field 1	##1 AOB field referenced by %1
&	File buffer	&4.2 File 4, field 2	&%1.%2 File %1, field %2
&	Fast file buffer	&1 Field 1	&%2 Field referenced by %2
!	Select list	!5 List 5	!%1 List referenced by %1

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 move back to the previous field, truncating the data at its new position.

**C** Comment

*Ctext*

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.

**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

**DS $n$** 

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**F;***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.

**\*** 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**F-C{LEAR}**  $n$ 

The file buffer for file  $n$  is cleared.

**F-DELETE** Delete a record from an open file**F-D{DELETE}**  $n$ 

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**F-F{REE}** { $n$  {*id*|*ref*}}

The record identified by the file and id associated with file buffer  $n$  is deleted. 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.

**F-OPEN** Open a file

**F-O{PEN}** *n* {**DICT**} {*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

**F-R{EAD}** *n* {*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

**F-U{READ}** *n* {*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

**F-W{RITE}** *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

**FB{U}** ({**DICT**} *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**)

**GO** *label|A-ref|ref|FB*

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

**Sn** 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* = A<sub>VM</sub>B<sub>VM</sub>C GO 10<sub>VM</sub>20<sub>VM</sub>30

and

IF *item* = A<sub>VM</sub>B<sub>VM</sub>C GOSUB 10<sub>VM</sub>GO 20<sub>VM</sub>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*.

### IIH Insert test in the active input buffer

**I{B}H***text|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 though 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{P}**{*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**O***text*{+}

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**}{**L***n*}

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

### **Q***text*

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

**Sf/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**Telement{,element...}**

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,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...L** loop.

**Sn** Emits *n* spaces where *n* may be given as a number or a buffer reference.

**T** Starts a **T...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,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.

**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:

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

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:

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

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

**X** Exit from the PROC**X***text*

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**({DICT} filename {id}) {label}**

The PROC identified by the given *filename* and *id* is executed, starting at *label*, or the first statement if no *label* is specified. If *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**[{DICT} filename {id}] {label}**

The PROC identified by the given *filename* and *id* is executed, starting at *label*, or the first statement if no *label* is specified. If *id* is omitted, the record id is obtained from the field of the active input buffer addressed by the input buffer pointer. If no *filename* or *id* are specified, transfer is controlled to the same Proc at *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
- 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 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](#) }

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** verb) or make copies of standard records to provide synonyms for other verbs.

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)
```

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.R   Default location for new accounts in [CREATE.ACCOUNT](#).  
OOT.DIR

\$BASIC.IGNOR   Contains patten match templates for records to be ignored by the [BASIC](#)  
E                   command.

\$BASIC.OPTIO   Sets default options for the [BASIC](#) command.  
NS

\$COMMAND.S   Controls saving of the [command stack](#) on leaving QM.  
TACK

\$DEBUG.OPTI   Specifies default options for the [QMBasic debugger](#).  
ONS

\$ED.OPTIONS   Sets default modes for the [ED](#) editor.

\$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.CA   Specifies the location of the [private catalogue](#). The American spelling may be  
TALOGUE       used.

\$QUERY.DEFA   Sets default options for the [query processor](#).  
ULTS

\$REPLICATE   Controls [account replication](#).

\$SCRB.FILE    Sets the default name of the screen definitions file for the [SCRB](#) screen builder.

\$VOC.PARSER   Specifies processors associated with user defined [VOC](#) record types.

&SED.OPTION   Sets options for the [SED](#) editor.  
S&

## 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 ' '
  DISPLAY 'Enter security password: ' :
  FOR I = 1 TO 3
    ECHO OFF
    INPUT PASSWORD
    ECHO ON
    IF PASSWORD = 'FSKJJ' THEN RETURN @TRUE
  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 (1) if the command is to be allowed, false (0) if it is to be rejected.
VERB	The name of the R-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. 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

```
<<{control,} text {, check}>>
```

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 control elements.
- text* is the prompt text to be displayed. An equals sign is automatically added to the end of the prompt text.
- check* is used to check whether the response to the prompt is valid. If omitted, no checking is performed.

The *control* option has two parts; the *display control* and the *response control*. Both parts are optional.

The *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).

The *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 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.

**C<sub>n</sub>** 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:

**C<sub>m-n</sub>** Returns tokens *m* to *n*.

**C<sub>n+</sub>** Returns tokens *n* onwards.

**C#** Returns the number of tokens in the command line.

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.

**CHANGE(*str, old, new*)** Change all occurrences of substring *old* in *str* with *new*. Argument values may be enclosed in quotes if necessary to avoid any syntactic ambiguity.

**F(*file, key {,field {, value {, subvalue } } })*** Use record *key* of the data portion of the named *file* as the response to the prompt. The optional *field*, *value* and *subvalue* allow extraction of a particular part of the record.

**ICONV(*str, code*)** Apply input conversion *code* to *str*. Argument values may be enclosed in quotes if necessary to avoid any syntactic ambiguity.

**In** 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.

**Ln** 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.

**ONCV(*str, code*)** Apply output conversion *code* to *str*. Argument values may be enclosed in quotes if necessary to avoid any syntactic ambiguity.

**R** Prompt repeatedly for input, concatenating data with an intervening space until a blank line is entered.

**R(*string*)** Prompt repeatedly for input, concatenating the responses with *string* between responses until a blank line is entered. The *string* may not include field mark characters. An abort will occur if a field mark is specified as part of *string*.

<b><i>Sn</i></b>	Take the <i>n</i> '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 <i>check</i> conditions, an input prompt appears.
<b>SUBR(<i>name</i>)</b>	Execute catalogued QMBasic function <i>name</i> , returning the result of this function as the value of the inline prompt.
<b>SUBR(<i>name</i>, <i>arg1</i>, <i>arg2</i>)</b>	Execute catalogued QMBasic function <i>name</i> , 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.
<b>SYSTEM(<i>n</i>)</b>	Returns the value of the QMBasic <a href="#">SYSTEM(<i>n</i>)</a> function.
<b>U</b>	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.
<b>@<i>var</i></b>	The name of an @-variable, including user defined names (see the <a href="#">SET</a> 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:  <pre>&lt;&lt;@<i>name</i>:<i>value</i>&gt;&gt;</pre> <p>The default value will be applied if the prompt would otherwise return a null string.</p>
<b><i>\$var</i></b>	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:  <pre>&lt;&lt;\$<i>name</i>:<i>value</i>&gt;&gt;</pre> <p>The default value will be applied if the prompt would otherwise return a null string.</p>

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.

The <<@*var*>> construct allows @variables to be inserted into any command, simplifying paragraph structure. For example:

```
SAVE .LIST LIST.<<@USERNO>>
```

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 *check* code performs simple data validation and is either a [pattern match template](#) or an input conversion code. Multiple patterns can be specified separated by the word **OR** with a single space either side. For example,

```
3N' - ' 4N OR 4N' - ' 3N
```

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 [ICONVO](#) 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 [@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 [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 [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

```
IF <<Customer number>> = "" THEN STOP
```

should be written as

```
IF "<<Customer number>>" = "" THEN STOP
```

## Examples

```
PA
SELECT ORDERS SAVING UNIQUE CUST.NO_
WITH DATE AFTER <<Start date>>
LIST CUSTOMERS NAME_
HEADING "Clients ordering after <<Start date>>"
```

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.

```
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

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 one or more concatenated items formed from pairs of lengths and character type:

- 0X       Zero or more characters of any type
- n*X       Exactly *n* characters of any type
- n-m*X     Between *n* and *m* characters of any type
- 0A       Zero or more alphabetic characters
- n*A       Exactly *n* alphabetic characters
- n-m*A     Between *n* and *m* alphabetic characters
- 0N       Zero or more numeric characters
- n*N       Exactly *n* numeric characters
- n-m*N     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 0X code is a wildcard that matches against anything. It has a commonly used synonym:

- ...       Zero or more characters of any type

The 0A, *n*A, 0N, *n*N 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-m*X 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-m*A 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 *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.

The [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 [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 [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.

Windows Mobile and Windows CE used on PDA devices have no built-in printer support. Third party software packages are required to print on these platforms. QM retains support for mode 3 print units as described below.

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.

### Setting Print Unit Characteristics

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 the 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 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 though QM does not provide any functions to generate GDI graphics.

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` file released with QM contains a definition for use of `lpr`. This has

Field 2	<code>lpr</code>
Field 3	<code>-P %</code>
Field 4	<code>-K %</code>
Field 5	<code>-T "% "</code>
Field 6	<code>-Z "% "</code>
Field 7	<code>-Z "landscape"</code>

### 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

```
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

---

<a href="#">CLEAN.ACCOUNT</a>	Clears \$HOLD and other system temporary files.
<a href="#">PRINTER</a>	Various printer control operations
<a href="#">SET.QUEUE</a>	Define a form queue
<a href="#">SETPTR</a>	Display or set print unit characteristics
<a href="#">SP.ASSIGN</a>	Set printer parameters using form queue emulation
<a href="#">SPOOL</a>	Sends record(s) to the printer

### Query Processor Keywords Relating to Printing

<a href="#">FOOTING</a>	Set page footing
<a href="#">HEADING</a>	Set page heading
<a href="#">LPTR</a>	Direct output to a printer (applies to many commands)

### QMBasic Statements and Functions Relating to Printing

<a href="#">FOOTING</a>	Set page footing
<a href="#">HEADING</a>	Set page heading
<a href="#">GETPU()</a>	Retrieve print unit characteristics
<a href="#">PAGE</a>	Force a new page
<a href="#">PRINT</a>	Emit data to a print unit
<a href="#">PRINTER CLOSE</a>	Close a print unit
<a href="#">PRINTER OFF</a>	Direct print unit 0 to the screen
<a href="#">PRINTER ON</a>	Direct print unit 0 to the printer
<a href="#">PRINTER RESET</a>	Resets the default print unit
<a href="#">PRINTER DISPLAY</a>	Direct printer output to the screen
<a href="#">PRINTER FILE</a>	Direct printer output to a file
<a href="#">PRINTER NAME</a>	Direct printer output to a named printer
<a href="#">SETPU</a>	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 they had 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 [Dconversioncode](#) 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 [MTconversioncode](#) 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.TIMZONE 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 multivalued 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.

Epoch values are generally 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 = <a href="#">EPOCH()</a>	Returns the current epoch value.
date = <a href="#">MVDATE(epoch)</a>	Returns the date part of an epoch value as a day number relative to 31 December 1967 in the user's time zone.
time = <a href="#">MVTIME(epoch)</a>	Returns the time part of an epoch value as a number of seconds since midnight in the user's time zone.
time.string = <a href="#">MVTIME.DATE(epoch)</a>	Returns the date and time values as a string with an underscore between the two elements.
epoch = <a href="#">MVEPOCH(time.string)</a>	Returns the epoch value for the given time string in the form returned by <a href="#">MVTIME.DATE()</a> .
epoch = <a href="#">MVEPOCH(date, time)</a>	Returns the epoch value for the given date and time values.

The [E 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.

For a detailed discussion of time zones, see <http://en.wikipedia.org/wiki/Zoneinfo> and [http://en.wikipedia.org/wiki/List\\_of\\_zoneinfo\\_time\\_zones](http://en.wikipedia.org/wiki/List_of_zoneinfo_time_zones).

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 adequate. 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 used by QM are of two types; [directory files](#) and [dynamic 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 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 dynamic 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 multivalued 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. See the [CREATE.FILE](#) command for further details.

Files contain data stored as records which are the basic unit of file access. Records are identified by unique keys which may be any sequence of up to 63 characters. This limit can be increased by the system administrator.

### 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 three 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 of the security of their system before enabling these.

The three extended syntaxes are:

Implicit Q-pointer	<i>account:file</i>
Implicit QMNet pointer	<i>server:account:file</i>
Pathname	<i>PATH:pathname</i>

Note that in the final 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.

These special syntaxes cannot be used with a multifile component name.

For details of file processing from QMBasic programs, see [File Processing](#).

## 3.1 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.

On Windows XP systems, use of mapped drives can assign different physical locations to the same drive letter for different users. 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.

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 DOS SUBST 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 SUBST X: C:\ABC
```

where X: is the mapped drive letter and C:\ABC 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.

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 high performance because the process of searching a directory for a file is, with many operating systems, essentially a linear scan. Locating a record to be read would, therefore, require on average that half of the entries in the directory are examined. Writing a new record would require the entire directory to be processed to verify that the file does not already exist.

Directory files are mainly used for data that is to be processed from outside of QM or for very large records (hundreds of kilobytes) where the operating system file structures may give better performance than the hashed file system. Typical uses include storage of QMBasic programs, [COMO](#) (command output) files, and saved select lists.

When a record is written to a directory file, any field mark characters are converted to the operating system dependent representation of a newline. 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.

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:

*	%A	%	%P
\	%B	"	%Q
,	%C	/	%S
=	%E	+	%V
>	%G	:	%X
	%J	;	%Y
<	%L	?	%Z

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 dynamic hashed file. External applications should avoid writing such items into directory files as they may cause confusion.

The length of a record id after translation of invalid characters as above must not exceed 255 bytes.

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](#).

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.

Depending on the operating system in use, record ids in directory files may be case insensitive.

Note also that the Windows file system does not allow file names that clash with Windows device names such as COM. Also, Windows ignores a trailing period in a file name. Thus files named "ABC." and "ABC" are the same item.

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.

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 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.

### Directory File Dictionaries

Just like any other QM file, a directory file can have a directory 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 username of the owner of the record (not Windows).

### 3.3 Dynamic Files

A **dynamic 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.

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 significantly 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.

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.

### 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 [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.

## 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. The 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.

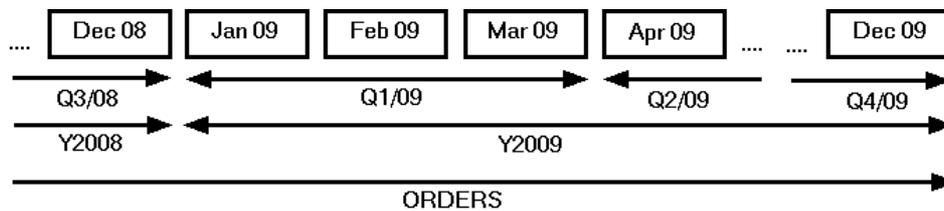
### 3.4 Distributed Files

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 orders 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 gives the application the ability to access 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.

Because the record layout in each part file in a distributed file will be identical, it is normal for a single dictionary to be shared by all part files and the distributed file.

#### 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.

### Distributed File Related Commands

A distributed file is created using the [ADD.DF](#) command:

```
ADD.DF dist.file part.file part.no algorithm
```

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 FL\$TYPE.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 [DEPART\(\)](#) 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.

## 3.5 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. This process will consume a licence. 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. On a secure system, 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.SERVERS](#) 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.
- The [FILEINFO\(\)](#) function will return the file type as FL\$TYPE.NET (6). Some modes of [FILEINFO\(\)](#) are not supported.
- A maximum of 10 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
```

### 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 slave 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 slave 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:**

[DELETE.SERVER](#), [LIST.SERVERS](#), [SET.SERVER](#)

## 3.6 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.

### The VFS Handler Class Module

A VFS handler is a globally catalogued QMBasic class module that intercepts all attempts to access the file. 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.

### Creating a Virtual File System

There are two steps; creating the VFS handler and creating the VOC entries.

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 : *handler*" where *handler* is the name of the globally catalogued VFS handler class module. 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:handler:string*

This third component could be used, for example, when a single *handler* class module is used to access many files. The *string* might be the pathname or other reference to the actual file to be opened by the handler.

### 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 V\$SELECT 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 V\$SELECT, the V\$CONTINUE.SELECT function is called to return the next part of the list. The V\$COMPLETE.SELECT function will be called if the remainder of the list should be returned as a single item and the V\$END.SELECT subroutine is called to terminate generation of a partial list.

---

VFS handlers that do not use partial list construction can omit the V\$CONTINUE.SELECT, V\$COMPLETE.SELECT and V\$END.SELECT entry points.

### **Alternate Key Indices**

The Virtual File System does not support alternate key indices. The [INDICES\(\)](#), [SELECTINDEX](#), [SELECTLEFT](#), [SELECTRIGHT](#), [SETLEFT](#) and [SETRIGHT](#) operations are not valid with a VFS item.

### 3.7 Creating and Modifying Data

QM provides several data entry and modification tools, each designed to fit a particular mode of use.

<a href="#"><u>ED</u></a>	The line editor
<a href="#"><u>MED</u></a>	The menu editor
<a href="#"><u>MODIFY</u></a>	The dictionary driven editor
<a href="#"><u>SED</u></a>	The full screen editor
<a href="#"><u>UPDATE.RECORD</u></a>	The data update utility

## 3.8 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.
- 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, D, 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, 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.

## 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.

An A or S-type record has up to 11 fields:

- 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: Not used
- 6: Not used
- 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+ 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 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 **C;p;q;r** in field 4 where *p*, *q*, *r* (etc) are the field numbers of the associated items. The dependant fields all have **D;n** in field 4 where *n* is the field number of the controlling field. Internally, Internally, QM converts Pick style association definitions into an association name *n*. 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 and 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 does not support 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.

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 dependant 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.

Field 8 of the dictionary may be multi-valued. The first value may be an A or F-correlative or a conversion code. 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 correlative type code 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

```
A;N(TEL)VM"L(3#) 3#-4#"
```

The above correlative takes the contents of the field named TEL and then applies a format mask to it.

```
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 $p$ .
<b>IF</b> $p$ { <b>THEN</b> $q$ } { <b>ELSE</b> $r$ }	Returns $q$ if $p$ is true, $r$ if $p$ is false. Absent elements return a null string.
( <i>conv</i> )	Applies the given conversion code to the expression value. Note that <i>conv</i> is not quoted.
<b>AND</b>	Logical AND
<b>OR</b>	Logical OR.

### Examples

A; 3 \* 2

Multiplies the content of field 3 by the content of field 2.

A; (N(PRICE) + N(TAX)) (MD2)

Adds the PRICE and TAX fields. The result is then converted using an MD2 conversion code.

A; N(PRICE) \* "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) \* "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; DESCRIPTION[ "1" , "20" ]

Extracts the first 20 characters of the DESCRIPTION field.

A; IF N(QTY) < "10" THEN "Re-order" 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
"11"	Push the constant "11" onto the stack
*	Multiply the top two items on the stack, replacing them with the result
"10"	Push the constant "10" onto the stack
/	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 constant. All constants of this style, including numeric values, must be enclosed in single quotes, double quotes or backslashes.

*Cnumber* An integer constant.

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. The field number form can be followed by a conversion code in brackets. If used with **R**, the sequence is *nR(conv)*.

**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 the top two items on the stack, replacing them 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 them 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 up to 8 fields:

- 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/multivalued 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.

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 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.  
  
For compatibility with other multivalued database products a value of 9998 or 9999 will be recognised by the query processor as references to the 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/multivalued 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.

## Dictionary I-type records

An I-type record defines calculated data and has up to 8 fields:

- 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/multivalued 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 [TRANSQ](#) expression for each field. Links can also be used within I-type expressions to minimise the number of explicit [TRANSQ](#) operations.

- 1: L { descriptive text }
- 2: Id expression
- 3: File name

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 D or I-type items in the remote file. They cannot be used to access Pick style A or S-type items.

### Examples

If a library application has two files, BOOKS and TITLES where the record id of BOOKS is formed from the id of the corresponding TITLES record and the copy number separated by a hyphen, the following link placed in the dictionary of the BOOKS file could be used to access the associated TITLES record:

- 1: L
- 2: @ID[ '-', 1, 1 ]
- 3: TITLES

Queries based on the BOOKS file could then reference data from the TITLES file using field names made up from the name of the link record, a % character and the name of the TITLES field to be accessed. For example, if the above line was named TTL, a query such as

```
LIST BOOKS TTL%TITLE TTL%AUTOR
```

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 BOOKS file by use of simply

```
TTL%TITLE
```

as the expression (field 2). This is exactly equivalent to writing

```
TRANS(TITLES, @ID[ '-', 1, 1 ], TITLE, 'X')
```

Use of links in this way removes the need to include the link expression (@ID[ '-', 1, 1 ]) in every reference to items in the TITLES file. Because use of a link causes a compile time substitution of the actual [TRANSQ](#) operation, changing the link record requires all I-types that use it to be recompiled.

## 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.

- |                |   |
|----------------|---|
| <b>@ID</b>     | A D-type record defining the record id.   |
| <b>@</b>       | A phrase record defining the default list of items to be displayed by <a href="#">LIST</a> and <a href="#">SORT</a> in the absence of any other field names.  |
| <b>@LPTR</b>   | A phrase record defining the default list of items to be displayed by <a href="#">LIST</a> and <a href="#">SORT</a> in the absence of any other field names when output is directed to a printer. If this record is not present, the query processor uses the @ record instead. |
| <b>@MODIFY</b> | A phrase record defining the default list of items to be processed by the <a href="#">MODIFY</a> command.   |
| <b>@SHOW</b>   | A phrase record defining the default list of items to be displayed by <a href="#">SHOW</a> 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

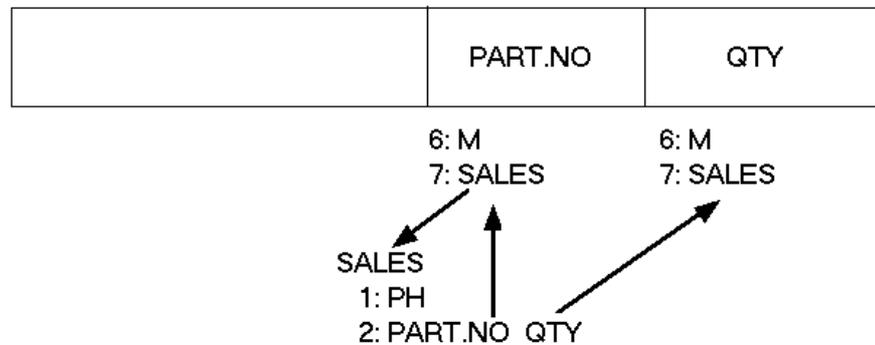
Fields 2 onwards are available for data storage. Users may freely create X-type records for their own purposes but should avoid names containing \$ signs or starting with the @ character as these may clash with system defined records.

## 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 dictionary entry of each field in the association. There is also a [phrase record](#) with this name which contains a space separated list of the fields that make up the association.

Thus, starting from any one element of the association, its dictionary entry can be used to find the phrase record which, in turn, allows the query processor to find all the members of the association.



For compatibility with some other multivalued database products, the ASSOC.UNASSOC.MV mode of the [OPTION](#) command can be used to make the query processor treat all multivalued fields that are not in defined associations as being associated together.

## 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 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:

```
SELL - 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 A, C, D, I or S-type items.

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.

<a href="#">@FILE.NAME</a>	The name of the file being processed by the command.
<a href="#">@FILENAME</a>	Synonym for <a href="#">@FILE.NAME</a> .
<a href="#">@ID</a>	The record id of the current record.
<a href="#">@NB</a>	Break level number.
<a href="#">@NI</a>	Item counter.
<a href="#">@RECORD</a>	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:

<a href="#">TOTAL()</a>	Accumulate totals for use with the query processor <a href="#">CALC</a> keyword.
<a href="#">TRANS()</a>	Fetch data from another file.
<a href="#">SUBR()</a>	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.

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, "D2Y" ); 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.

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](#).

### 3.9 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.

The standard conversion codes are:

<a href="#">Boolean</a>	B
<a href="#">Base64</a>	B64
<a href="#">Concatenation</a>	C
<a href="#">Dates</a>	D
<a href="#">Group</a>	G
<a href="#">Integer</a>	IS, IL
<a href="#">Length</a>	L
<a href="#">Radix</a>	MB, MO, MX
<a href="#">Radix</a>	MCDX, MCXD
<a href="#">Character</a>	MCx
<a href="#">Masked Decimal</a>	MD, ML, MR
<a href="#">Times</a>	MT
<a href="#">Pattern matching</a>	P
<a href="#">Range checking</a>	R
<a href="#">Substitution</a>	S
<a href="#">Text substring</a>	T <i>m,n</i>
<a href="#">Translation</a>	T <i>file</i>
<a href="#">User defined</a>	U
<a href="#">Fields</a>	< <i>f,v,s</i> >

## 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;** *expression*

where

*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 false (0) and true (1) 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 (zero or a null string) or Y for true (all other values).

Used with [ICONV\(\)](#) for input conversion, this conversion code returns false (0) for a null string or a string containing the single character N in upper or lower case. It returns true (1) 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.

### **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.

## 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 F1<sub>FM</sub>F2<sub>FM</sub>F3 and @ID contain 21:

Operation	Result
OCONV('abc', 'C;3;"xxx";1')	F3xxxF1
OCONV('abc', 'C;3;1')	F3F1
OCONV('abc', 'C;"aaa"1"bbb"')	aaaF1bbb
OCONV('abc', 'C;1 2')	F1 F2
OCONV('abc', 'C;1*2')	F1*F2
OCONV('abc', 'C;1**')	F1*abc
OCONV('abc', 'C; 3')	F3
OCONV('abc', 'C;0=1')	21=F1

## 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.
- 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.
  - Q** Quarter number (1 to 4).
  - W** Day of week number. Monday is day 1, Sunday is day 7.
  - WA** 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 *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.

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:

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.

### 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 [DATE.FORMAT](#) setting, both forms are shown.

Code	Result	European date
'D'	01 JUN 1994	
'D2'	01 JUN 94	
'D4'	01 JUN 1994	
'D/'	06/01/1994	01/06/1994
'D '	06 01 1994	01 06 1994
'D2/'	06/01/94	01/06/94
'D/E'	01/06/1994	06/01/94
'D2 E'	01 06 94	06 01 94
'D.YJ'	1994.152	
'D2:JY'	152:94	
'D YMD'	1994 06 01	
'DX'	19940601	
'D MY[A,2]'	JUNE 94	
'D4DOMAYL'	1st June 1994	
'D DMY[,A3,2]'	01 JUN 94	
'D DMY[,A9,2]'	01 JUNE 94	
'D/MDY[Z,Z,2]'	6/1/94	
'D DMYL[,A,]'	01 June 1994	

'DDMYL[Z,A,2]'	1 June 94
'DYMD[2,2,2]'	94 06 01
'DW'	3
'DWA'	WEDNESDAY
'DWAL'	Wednesday
'DMA'	JUNE
'DMAL'	June
'DQ'	2
'D-YIWI'	1994-22
'DISO8601W'	1994W223
'DISO8601W-'	1994-W22-3

### 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, they 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 multivalued 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.

### 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 multivalued 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:

**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.
- 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.

<b>WI</b>	ISO week number.
<b>X</b>	ANSI X12 format date (YYYYMMDD with no separators).
<b>Y</b>	Year.
<b>YI</b>	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.
<b>Z</b>	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

<i>n</i>	Use <i>n</i> characters.
<b>A</b>	Display as alphabetic (applies to month component only).
<b>An</b>	Display as <i>n</i> alphabetic characters (applies to month component only).
<b>Z</b>	Suppress leading zeros.
<b>Zn</b>	Display as <i>n</i> digits with leading zeros replaced by spaces.
<i>"text"</i>	Uses 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-     *yyyy-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.

<b>Code</b>	<b>Result</b>	<b>European date</b>
'E'	13 FEB 2009	
'E2'	13 FEB 09	
'E4'	13 FEB 2009	
'ET'	18:31	
'E/'	02/13/2009	13/02/2009
'E '	02 13 2009	13 02 2009
'E2/'	02/13/09	13/02/09
'E/E'	13/02/2009	02/13/2009
'E2 E'	13 02 09	02 13 09
'E.YJ'	2009.44	
'E2:JY'	44:09	
'E YMD'	2009 02 13	
'E YMDTS'	2009 02 13 18:31:30	
'EX'	20090213	
'E MY[A,2]'	FEBRUARY 09	
'E4DOMAYL'	13th February 2009	
'E4DOMAYLTS[,A3]'	13th Feb 2009 18:31:30	
'E DMY[,A3,2]'	13 FEB 09	
'E DMY[,A9,2]'	13 FEBRUARY 09	
'E/MDY[Z,Z,2]'	2/13/09	
'E DMYL[,A,]'	13 February 2009	
'EDMYL[Z,A,2]'	13 February 09	
'EYMD[2,2,2]'	09 02 13	
'EW'	5	
'EWA'	FRIDAY	
'EWAL'	Friday	
'EMA'	FEBRUARY	
'EMAL'	February	
'EQ'	1	
'E-YIWI'	2009-07	
'EA'	Fri Feb 13 18:31:30 2009	
'EWALMALDTSZY[A3,A3]'	Fri Feb 13 18:31:30 EST 2009	
'EISO8601W'	2009W075	
'EISO8601W-'	2009-W07-5	
'EISO8601T'	20090213T183130	

'EISO8601T-'

2009-02-13T18:31:30

### 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 [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 [YEARBASE](#) configuration parameter. and the order of these defaults to day.

The time must be entered in the form *hh:mm:ss* or *hh:mm* where the colon may be any non-numeric character. If the seconds are omitted, a default of zero is used.

**See also:**

[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**; *expression*

where

*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.

- Gcn** Returns the first  $n$  components of the source data delimited by character  $c$ .
- Gscn** 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

## Integer conversion (IS, IL)

The integer conversion codes convert integer values between numeric form and hardware specific integer representation.

**IS** Short integer (16 bit value).

**IL** Long integer (32 bit value).

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 an 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.
- $L_n$**  Returns the original string if its length is less than or equal to  $n$ . Otherwise it returns a null string.
- $L_{n,m}$**  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.

### Examples

Conversion	Data	Result
L	1234	4
L3	A	A
L3	AB	AB
L3	ABC	ABC
L3	ABCD	
L2,3	A	
L2,3	AB	AB
L2,3	ABC	ABC
L2,3	ABCD	

## Character conversion (MCx)

The character conversion codes perform various character based conversions.

<b>MCA</b>	Delete all non-alphabetic characters
<b>MC/A</b>	Delete all alphabetic characters
<b>MCAN</b>	Delete all non-alphanumeric characters
<b>MC/AN</b>	Delete all alphanumeric characters
<b>MCL</b>	Convert to lower case
<b>MCN</b>	Delete all non-numeric characters
<b>MC/N</b>	Delete all numeric characters
<b>MCP</b>	Replace non-printing characters by periods
<b>MCS</b>	Convert first alphabetic character in each sentence to uppercase
<b>MCT</b>	Capital initial all words (see below)
<b>MCU</b>	Convert to uppercase

These conversion codes behave identically for both input and output conversion.

The MCT conversion code is implemented differently across various multivalued 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

<b>Data</b>	<b>Code</b>	<b>Result</b>
267PS-A17	MCA	PSA
267PS-A17	MC/A	267-17
267PS-A17	MCAN	267PSA17
267PS-A17	MC/AN	-
267PS-A17	MCN	26717
267PS-A17	MC/N	PS-A
Red pencil	MCL	red pencil
Red pencil	MCU	RED PENCIL
Red pencil	MCT	Red Pencil
123 <sub>FM</sub> 456	MCP	123.456
abc. dE. 1f	MCS	Abc. DE. 1F

## 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

<b>MD</b>	Convert without regard to justification
<b>ML</b>	Left justify the converted result
<b>MR</b>	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.

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.

<b>Value</b>	<b>Conversion</b>	<b>Result</b>
0	MD0	'0'
0	MD0Z	"
Sign handling		
12345678	MD0	'12345678'
-12345678	MD0	'-12345678'
12345678	MD0+	'12345678+'
-12345678	MD0+	'12345678-'
12345678	MD0-	'12345678 '
-12345678	MD0-	'12345678-'
12345678	MD0<	' 12345678 '
-12345678	MD0<	'<12345678>'
12345678	MD0(	' 12345678 '
-12345678	MD0(	'(12345678)'
12345678	MD0C	'12345678 '
-12345678	MD0C	'12345678CR'
12345678	MD0D	'12345678 '
-12345678	MD0D	'12345678DB'
Scale factors		
12345678	MD22	'123456.78'
1234567.89	MD22	'12345.68'
-12345678	MD22	'-123456.78'
12345678	MD02	'123457'
12345678	MD02T	'123456'
Thousands separators and currency symbols		
1234567	MD0,	'1,234,567'
12345678	MD2,\$	'\$123,456.78'
Left justification and padding		
0	ML004*	'0****'
12345678	ML0010*	'12345678***'
12345678	ML0+12*	'12345678+****'
-12345678	ML0+12*	'12345678-****'
12345678	ML0-12*	'12345678****'

12345678	ML0(12*	'12345678 ****'
-12345678	ML0(12*	'(12345678)**'
12345678	ML0C12*	'12345678 **'
-12345678	ML0C12*	'12345678CR**'

#### Right justification and padding

0	MR004*	****0'
12345678	MR0010*	***12345678'
12345678	MR0+12*	***12345678+'
-12345678	MR0+12*	***12345678-'
12345678	MR0-12*	***12345678'
12345678	MR0(12*	***12345678'
-12345678	MR0(12*	**(12345678)'
12345678	MR0C12*	**12345678 '
-12345678	MR0C12*	**12345678CR'

### 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.

Value	Conversion	Result
"	MD0	"
'0'	MD0	'0'
Sign handling		
'12345678'	MD0	'12345678'
'-12345678'	MD0	'-12345678'
'12345678-'	MD0	'-12345678'
'<12345678>'	MD0	'-12345678'
'(12345678)'	MD0	'-12345678'
'12345678CR'	MD0	'-12345678'
'12345678DB'	MD0	'-12345678'
Scale factors		
'123456.78'	MD2	'12345678'

'123456.78' MD23 '1234567.8'

Currency symbols and thousands separators

'12,345' MD0, '12345'  
'\$123,456.78' MD2,\$ '12345678'  
 '\$(123456.78)' MD22\$( '-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

**MT** {**M**} {**H**} {**S**} {*c*}

where

- M** specifies that the time value to be converted is in milliseconds (output conversion only).
- H** specifies that the time is to appear in 12-hour format with either "am" or "pm" appended. If **H** is not specified, 24-hour conversion is used for output conversion and assumed for input conversion. If the AMPM.UPCASE mode of the [OPTION](#) command is active, the appended suffix is in uppercase (AM or PM).
- S** specifies that output conversion is to include the seconds field. If the **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 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 M, H and S qualifiers may be in any order.

### Examples

Conversion	Data	Result
MT	0	00:00
MT	31653	08:47
MT	63306	17:35
MTH	0	12:00am
MTH	31653	08:47am
MTH	63306	05:35pm
MTS	31653	08:47:33
MTS	63306	17:35:06
MTHS	63306	05:35:06pm
MTS.	63306	17:35:06
MTMS	33888250	09:24:48.250
MT'h'	63306	17h35
MT"	63306	1735

**See also:**

[Dates, Times and Epoch Values](#)

## 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.

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.

### 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.

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 binary, octal or hexadecimal representation.

### Examples

#### Input conversion:

Source data	Conversion	Result
1110	MB	14
3777777762	MO	-14
97AB	MX	38827
010000010100001001000011	MB0C	ABC
101102103	MO0C	ABC
414243	MX0C	ABC

#### Output conversion:

Source data	Conversion	Result
19	MB	10011
19	MO	23
19	MX	13
ABC	MB0C	010000010100001001000011
ABC	MO0C	101102103
ABC	MX0C	414243

### **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.

### 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*.

## 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

$$\mathbf{P}(\mathit{template})\{;\mathit{template})\dots\}$$

where

*template* is a pattern template as for the [MATCHES](#) operator and must be enclosed in brackets. If more than one *template* is supplied, they may be separated by semicolons (;) or forward slashes (/).

The **P** conversion code tests whether the input data matches *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 *template* is provided, the input value is tested against each in turn.

A null input value always returns a null result.

### Examples

Conversion	Data	Result
P(3A)	123	
P(3A)	ABC	ABC
P(1-2N)	74	74
P(1-2N)	123	
P(2A);(3N)	12	
P(2A);(3N)	AB	AB
P(2A);(3N)	123	123
P(2A)		

## 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

$$\mathbf{R}n,m\{;n,m\dots\}$$

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

Conversion	Data	Result
R3,5	1	
R3,5	3	3
R-2,0	-3	
R-2,0	-1	-1
R1,3;6,8	7	7
R1,3;6,8	5	
R1,3	0002	0002
R1,3		
R1.3	A	A

## 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**;*value1*;*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 ; * ; ' ' vMD4DMY
```

The following table shows a variety of conversions for a data record of "F1<sub>FM</sub>F2" and a record id of "ID".

Conversion	Data	Result
S;1;2	0	F2
S;1;2		F2
S;1;2	5	F1
S;1;2	XX	F1
S;*;'ZZ'	0	ZZ
S;*;'ZZ'	2	2
S;0	0	
S;0	1	ID
S;;0	0	ID

## File translation conversion

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;cv;i;o*

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 [TRANSQ](#) function. Where possible, [TRANSQ](#) should be used in preference to the T conversion code for best performance.

## Text substring conversion

The text substring extraction code returns a portion of the source data.

The format of the conversion code is

**$T_{m,n}$**

**$T_m$**

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.

## User defined conversions

Users may add their own conversion codes to the system by writing a QMBasic subroutine to perform the conversion.

The format of the conversion code is

*Usubrname*  
or  
*Usubrname.extension*

where

*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.

*extension* is optional qualifying information for the conversion code. This can be accessed within the subroutine in the [@CONV](#) variable.

The subroutine should have four arguments:

CONV.SUBR(*result, src, status, oconv*)

where

*result* should be set to the result of the conversion.

*src* is the item to be converted.

*status* is the value to be set for the [STATUS\(\)](#) function.

*oconv* indicates whether this is an input (0) or output (1) conversion.

See the U50BB subroutine in the BP file of the QMSYS account for an example of a Pick user exit routine.

### Example

The following example subroutine converts a combined date/time value as returned by [SYSTEM\(1005\)](#) to a text representation or vice versa.

```
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
```

### 3.10 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 [FMTQ](#) 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 [FMTQ](#) 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 1234567.

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.

Value	Format code	Result
'ABCDE'	'8 L'	'ABCDE '
'ABCDE'	'8 R'	' ABCDE'
'ABCDE'	'8 '* ' L'	'ABCDE***'
'0012345'	'8 R'	' 0012345'
'0012345'	'8 RZ'	' 12345'
'0000000'	'84RZ'	' '
'12345'	'8"0"R'	'00012345'
'1234567'	'15R2'	' 1234567.00'
'1234567'	'15R2\$,'	' \$1,234,567.00'
'12345.67'	'15*R2\$,'	'*****\$12,345.67'
'1234567'	'14L2'	'1234567.00 '
'43'	'L###m'	'43 m'
'43'	'R###m'	' 43m'

' 4 3 '	'"0"R###m'	'043m'
'1234567890'	'L###-#####'	'123-4567890'
'123456789'	'L#3-#3-#3'	'123-456-789'
'12345'	'L#'	' 1 '
'12345'	'R#'	' 5 '
'123456789'	'L#5'	'12345'
'123456789'	'R#5'	'56789'
'12345'	'L#6'	'12345  '
'12345'	'R#6'	' 12345'
'A LONG LINE'	'6T'	'A LONG <sub>TM</sub> LINE  '
'A LONG LINE'	'7T'	'A LONG <sub>TM</sub> LINE  '
'A LONG LINE'	'8T'	'A LONG <sub>TM</sub> LINE  '
'A LONG LINE'	'8R'	'A LONG L <sub>TM</sub> INE  '
'BANANAS'	'3T'	'BAN <sub>TM</sub> ANA <sub>TM</sub> S  '
'1.236'	' 2 '	'1.24'

### 3.11 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.

Although the single user environment of a PDA appears not to require locking, the PDA version of QM retains support for the locking operations for use with [QMNet](#) connections and to minimise changes that need to be made when porting applications from other platforms.

Three types of lock are available on files; file, read 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 lock on the file or 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 record at the same time. An attempt to obtain a read lock will fail if another process has the file is 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 a 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 -1 or, if no **LOCKED** clause is present, waiting for space to become available. A message will be written to the [system error log file](#).

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. 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. This parameter is not supported on the PDA version of QM.

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.

### 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 an 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 applications.

## 3.12 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](#).

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.

### 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.

### 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.

### 3.13 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, 100000 records in it. For simplicity, assume that these records are made up of 10 orders from each of 10000 customers. To locate all the orders placed by a specific customer would require all 100000 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.

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. The only situation where an index should possibly not 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.

Indices can be built on real data stored in the file (dictionary D-type or A/S-type without a correlative) 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. The [CREATE.INDEX](#) and [MAKE.INDEX](#) commands will not allow separate indices to be created for equivalent items.

### Use of Indices in QMBasic

A program can determine whether a file has indices by use of the [FILEINFO\(\)](#) function with action key FL\$AK:

```
IF FILEINFO(FVAR, FL$AK) THEN DISPLAY 'File has indices'
```

The names of the indices that exist for a file can be determined using the [INDICES\(\)](#) function:

```
NAMES = INDICES(FVAR)
```

An extended form of the [INDICES\(\)](#) function can be used to retrieve the details of a specific index by name:

```
INDEX.DATA = INDICES(FVAR, NAME)
```

Programmers can gain access to the index itself using the QMBasic [SELECTINDEX](#) statement and advanced index scanning operations can be performed using [SELECTLEFT](#), [SELECTRIGHT](#), [SETLEFT](#) and [SETRIGHT](#) as described below.

### 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 [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 [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 [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 [SETLEFT](#) or [SETRIGHT](#) statements may be used to position the pointer at the extreme left or right of the indexed values.

The [SELECTRIGHT](#) or [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 [SETLEFT](#) to position at the start of the data and then perform a series of [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:

```
SETLEFT 'CUST.NO' FROM ORD.F
LOOP
  SELECTRIGHT 'CUST.NO' FROM ORD.F SETTING CUST
LOOP
  READNEXT ORD.ID ELSE EXIT
  ...processing...
REPEAT
UNTIL STATUS()
REPEAT
```

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.

### **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.

### 3.14 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 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. Modifying and recataloguing the trigger function will have no effect on processes that have the file open until they close and reopen it. Setting or removing a trigger function while the file is open may not take effect until after the next access to the file.

A trigger function on a dynamic 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.

The interface into a trigger function is:

```
SUBROUTINE name(mode, id, data, on.error, fvar)
```

where

<i>name</i>	is the trigger subroutine name.
<i>mode</i>	indicates the point at which the trigger function is being called:
1	0x0 FL\$TRG.PRE.WRITE before writing a record

1	
2	0x0 FL\$TRG.PRE.DELET before deleting a record
2	E
4	0x0 FL\$TRG.POST.WRIT after writing a record
4	E
8	0x0 FL\$TRG.POST.DELE after deleting a record
8	TE
16	0x1 FL\$TRG.READ after reading a record
0	
32	0x2 FL\$TRG.PRE.CLEAR before clearing the file
0	
64	0x4 FL\$TRG.POST.CLEA after clearing the file
0	R

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.

<i>id</i>	is the id of the record to be written or deleted.
<i>data</i>	is the data. This is a null string for a delete or clearfile action.
<i>on.error</i>	indicates whether the program performing the file operation has used the ON ERROR clause to catch aborts.
<i>fvar</i>	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 the value of *data* is changed by a pre-write trigger function, the modified data is written to the file. Similarly, a read trigger can modify the data that will be returned to the application that requested the read. Changes to the value of *id* will not affect the database update 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 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.

### 3.15 Data Encryption

QM supports three data encryption methods:

**Ad hoc encryption** is provided by two QMBasic functions, [ENCRYPT\(\)](#) and [DECRYPT\(\)](#). These take two arguments; the string to be processed and the encryption key. 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.

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 this style of 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 the index file itself is not encrypted and hence 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. 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.

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.

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 give best security.

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](#).

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.

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.

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. 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.

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.

## 3.16 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 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 rollback. 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, ER\_COMMIT\_FAIL if an error occurred during the commit action, or ER\_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.

### 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.

## 3.17 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.

### Setting up the Publisher

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 severe 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

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 username 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.

### 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 username of the subscriber phantom must have full access to all relevant encryption keys.

### 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 username 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 username 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. 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	Add an element to a distributed file
BUILD.INDEX	Build an alternate key index
CATALOGUE	Add program to the system catalogue. The American spelling may be used.
CREATE.FILE	Create a file

CREATE.INDEX	Create an alternate key index
DELETE.CATALOGUE	Delete a program from the system catalogue. The American spelling may be used.
DELETE.FILE	Delete a file
DELETE.INDEX	Delete an alternate key index
MAKE.INDEX	Create and build an alternate key index
REMOVE.DF	Remove an element of a distributed file
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.

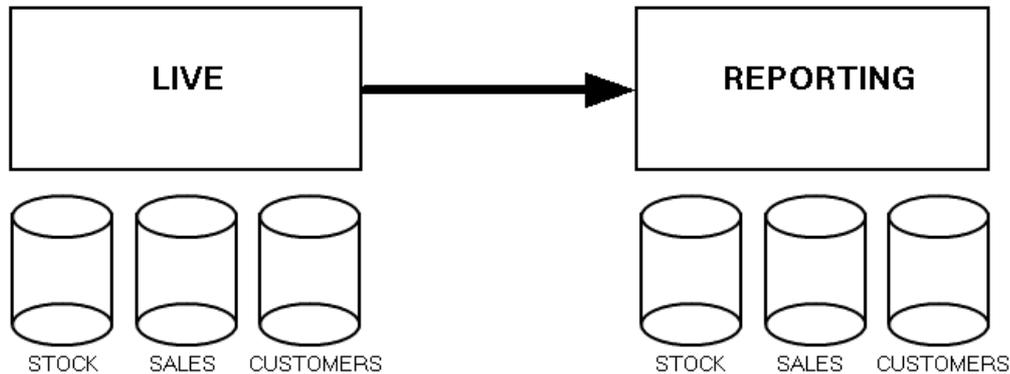
In addition, presence of  
AUTO.PUBLISH

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](#).

## 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 [SETUP.DEMO](#) command, creating a reporting server that mirrors the data from the 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 STORE account from the live server, typically by restoring a backup.
2. Use EDIT.CONFIG or a text editor to set the NETFILES configuration parameter to allow incoming QMNet connections and set REPLSRVR to the name that will be used by the live server to reference the reporting server.

```
NETFILES=2
REPLSRVR=REPORTING
```

3. Restart QM.

### The Live Server

1. Log to the QMSYS account

```
LOGTO QMSYS
```

2. Add a QMNet server definition that allows access to the reporting server using the server name set in the REPLSRVR parameter above.

```
SET.SERVER REPORTING
```

Enter appropriate network and user authentication details.

3. Use EDIT.CONFIG or a text editor to add the REPLDIR configuration parameter that identifies where replication log files will be stored.

```
REPLDIR=/usr/replication
```

4. Log to the account holding the files to be published.

```
LOGTO STORE
```

5. 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.

6. 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.
7. 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 username 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

<a href="#"><u>*</u></a>	Comment
<a href="#"><u>\$ECHO</u></a>	Paragraph tracing
<a href="#"><u>!</u></a>	Synonym for SH
<a href="#"><u>ABORT</u></a>	Abort processing and return to command prompt
<a href="#"><u>ACCOUNT.RESTORE</u></a>	Restore a Pick style ACCOUNT-SAVE tape
<a href="#"><u>ACCOUNT.SAVE</u></a>	Save an account to tape in Pick compatible format
<a href="#"><u>ADD.DF</u></a>	Add an element to a distributed file
<a href="#"><u>ADMIN</u></a>	Administrative command menu
<a href="#"><u>ADMIN.SERVER</u></a>	Manage security rules for QMNet server connections
<a href="#"><u>ADMIN.USER</u></a>	Username administration for network connections
<a href="#"><u>ALIAS</u></a>	Create a temporary alias for a command
<a href="#"><u>ANALYSE.FILE</u></a>	Analyse structure and usage of dynamic file
<a href="#"><u>ANALYZE.FILE</u></a>	Synonym for ANALYSE.FILE
<a href="#"><u>AUTHENTICATE</u></a>	Reduce privileges of a startup process
<a href="#"><u>AUTHKEY</u></a>	Encrypt password for AUTHENTICATE
<a href="#"><u>AUTOLOGOUT</u></a>	Set inactivity timer
<a href="#"><u>BASIC</u></a>	Compile QMBasic programs
<a href="#"><u>BELL</u></a>	Enable or disable audible alarm
<a href="#"><u>BLOCK.PRINT</u></a>	Print text using large characters
<a href="#"><u>BLOCK.TERM</u></a>	Display text using large characters
<a href="#"><u>BREAK</u></a>	Enable, disable or query break key
<a href="#"><u>BUILD.INDEX</u></a>	Build an alternate key index
<a href="#"><u>CATALOG</u></a>	Synonym for CATALOGUE
<a href="#"><u>CATALOGUE</u></a>	Add program to system catalogue
<a href="#"><u>CD</u></a>	Synonym for COMPILE.DICT
<a href="#"><u>CLEAN.ACCOUNT</u></a>	Remove records from \$HOLD, \$COMO and \$SAVEDLISTS
<a href="#"><u>CLEAR.ABORT</u></a>	Clear the abort status in an ON.ABORT paragraph
<a href="#"><u>CLEAR.DATA</u></a>	Clear the data queue
<a href="#"><u>CLEAR.FILE</u></a>	Remove all records from a file
<a href="#"><u>CLEAR.INPUT</u></a>	Clear keyboard type-ahead
<a href="#"><u>CLEAR.LOCKS</u></a>	Release task locks
<a href="#"><u>CLEAR.PROMPTS</u></a>	Clear inline prompt responses
<a href="#"><u>CLEAR.SELECT</u></a>	Clear one or all select lists
<a href="#"><u>CLEAR.STACK</u></a>	Clear the command stack
<a href="#"><u>CLEARDATA</u></a>	Synonym for CLEAR.DATA
<a href="#"><u>CLEARINPUT</u></a>	Synonym for CLEAR.INPUT
<a href="#"><u>CLEARPROMPTS</u></a>	Synonym for CLEAR.PROMPTS
<a href="#"><u>CLEARSELECT</u></a>	Synonym for CLEAR.SELECT
<a href="#"><u>CLR</u></a>	Clear display
<a href="#"><u>CNAME</u></a>	Rename a file or record within a file
<a href="#"><u>COMO</u></a>	Activate or deactivate command output files
<a href="#"><u>COMPILE.DICT</u></a>	Compile I-types in a dictionary
<a href="#"><u>CONFIG</u></a>	Display licence and configuration parameters
<a href="#"><u>CONFIGURE.FILE</u></a>	Change file configuration parameters
<a href="#"><u>COPY</u></a>	Copy records
<a href="#"><u>COPY.LIST</u></a>	Copy a saved select list
<a href="#"><u>COPY.LISTP</u></a>	Copy a saved select list using Pick style syntax
<a href="#"><u>COPYP</u></a>	Copy records using Pick style syntax
<a href="#"><u>COUNT</u></a>	Count records
<a href="#"><u>CREATE.ACCOUNT</u></a>	Make a new QM account

<a href="#"><u>CREATE.FILE</u></a>	Create a file
<a href="#"><u>CREATE.INDEX</u></a>	Create an alternate key index
<a href="#"><u>CREATE.KEY</u></a>	Creates a data encryption key
<a href="#"><u>CREATE.USER</u></a>	Create user name for network connection
<a href="#"><u>CS</u></a>	Synonym for CLR
<a href="#"><u>CT</u></a>	Display records from a file
<a href="#"><u>DATA</u></a>	Add text to the data queue for associated verb or program
<a href="#"><u>DATE</u></a>	Display the date and time
<a href="#"><u>DATE.FORMAT</u></a>	Selects default date format
<a href="#"><u>DEBUG</u></a>	Debug QMBasic program
<a href="#"><u>DELETE</u></a>	Delete records from a file
<a href="#"><u>DELETE.ACCOUNT</u></a>	Delete a QM account
<a href="#"><u>DELETE.CATALOG</u></a>	Synonym for DELETE.CATALOGUE
<a href="#"><u>DELETE.CATALOGUE</u></a>	Delete a program from the system catalogue
<a href="#"><u>DELETE.COMMON</u></a>	Delete a named common block
<a href="#"><u>DELETE.DEMO</u></a>	Delete demonstration files
<a href="#"><u>DELETE.FILE</u></a>	Delete a file
<a href="#"><u>DELETE.INDEX</u></a>	Delete an alternate key index
<a href="#"><u>DELETE.KEY</u></a>	Deletes a data encryption key
<a href="#"><u>DELETE.LIST</u></a>	Delete a saved select list
<a href="#"><u>DELETE.PRIVATE.SERVER</u></a>	Delete a QMNet private server definition
<a href="#"><u>DELETE.SERVER</u></a>	Delete a QMNet server definition
<a href="#"><u>DELETE.USER</u></a>	Delete user name for network connection
<a href="#"><u>DISABLE.INDEX</u></a>	Disable update and use of one or more alternate key indices
<a href="#"><u>DISPLAY</u></a>	Display text
<a href="#"><u>DUMP</u></a>	Display records from a file in hexadecimal and character format
<a href="#"><u>ECHO</u></a>	Disable or enable keyboard echo
<a href="#"><u>ED</u></a>	Line editor
<a href="#"><u>EDIT</u></a>	Synonym for ED
<a href="#"><u>EDIT.LIST</u></a>	Edit a saved select list
<a href="#"><u>ENCRYPT.FILE</u></a>	Applies encryption to a file
<a href="#"><u>FILE.SAVE</u></a>	Save all accounts to tape in Pick compatible format
<a href="#"><u>FILE.STAT</u></a>	Report a summary of all files in one or more accounts
<a href="#"><u>FIND.ACCOUNT</u></a>	Find an account on a FILE.SAVE tape
<a href="#"><u>FIND.PROGRAM</u></a>	Locate program in catalogue
<a href="#"><u>FORMAT</u></a>	Apply conventional formatting to a QMBasic program
<a href="#"><u>FORM.LIST</u></a>	Create a select list from a file record
<a href="#"><u>FSTAT</u></a>	Collect and report file statistics
<a href="#"><u>GENERATE</u></a>	Generate a QMBasic include record from a dictionary
<a href="#"><u>GET.LIST</u></a>	Retrieve a previously saved select list
<a href="#"><u>GET.STACK</u></a>	Restore a saved command stack
<a href="#"><u>GO</u></a>	Jump to a label within a paragraph
<a href="#"><u>GRANT.KEY</u></a>	Grants access to a data encryption key
<a href="#"><u>HELP</u></a>	Display help
<a href="#"><u>HSM</u></a>	Hot Spot Monitor performance monitoring tool
<a href="#"><u>HUSH</u></a>	Disable or enable display output
<a href="#"><u>IF</u></a>	Conditional execution in paragraphs
<a href="#"><u>INHIBIT.LOGIN</u></a>	Enables or disables login of new QM sessions
<a href="#"><u>LIST</u></a>	List records from a file
<a href="#"><u>LIST.COMMON</u></a>	List named common blocks
<a href="#"><u>LIST.DF</u></a>	List elements of a distributed file
<a href="#"><u>LIST.DIFF</u></a>	Form difference of two saved select lists
<a href="#"><u>LIST.FILES</u></a>	List details of open files
<a href="#"><u>LIST.INDEX</u></a>	List details of an alternate key index

<a href="#"><u>LIST.INTER</u></a>	Form intersection of two saved select lists
<a href="#"><u>LIST.ITEM</u></a>	List records from a file in internal format
<a href="#"><u>LIST.KEYS</u></a>	Lists details of encryption keys
<a href="#"><u>LIST.LABEL</u></a>	List records from a file in address label format
<a href="#"><u>LIST.LOCKS</u></a>	List task lock status
<a href="#"><u>LIST.PHANTOMS</u></a>	List currently active phantom processes
<a href="#"><u>LIST.READU</u></a>	List file, read and update locks
<a href="#"><u>LIST.REPLICATEDs</u></a>	Lists files that use replication with the target file details
<a href="#"><u>LIST.SERVERS</u></a>	List defined QMNet server names
<a href="#"><u>LIST.TRIGGERS</u></a>	List trigger functions used in the account
<a href="#"><u>LIST.UNION</u></a>	Form union of two saved select lists
<a href="#"><u>LIST.USERS</u></a>	List user names for network connection
<a href="#"><u>LIST.VARS</u></a>	List user @-variables
<a href="#"><u>LISTDICT</u></a>	List a dictionary in Pick style format
<a href="#"><u>LISTF</u></a>	List all files defined in the VOC
<a href="#"><u>LISTFL</u></a>	List all local files defined in the VOC
<a href="#"><u>LISTFR</u></a>	List all remote files defined in the VOC
<a href="#"><u>LISTK</u></a>	List all keywords defined in the VOC
<a href="#"><u>LISTM</u></a>	List all menus defined in the VOC
<a href="#"><u>LISTPA</u></a>	List all paragraphs defined in the VOC
<a href="#"><u>LISTPH</u></a>	List all phrases defined in the VOC
<a href="#"><u>LISTPQ</u></a>	List all PROCs defined in the VOC
<a href="#"><u>LISTQ</u></a>	List all indirect file references in the VOC
<a href="#"><u>LISTR</u></a>	List all remote items defined in the VOC
<a href="#"><u>LISTS</u></a>	List all sentences defined in the VOC
<a href="#"><u>LISTU</u></a>	List users currently in QM
<a href="#"><u>LISTV</u></a>	List all verbs defined in the VOC
<a href="#"><u>LOCK</u></a>	Set a task lock
<a href="#"><u>LOGIN.PORT</u></a>	Login a serial port from within another QM session
<a href="#"><u>LOGMSG</u></a>	Write a message to the error log
<a href="#"><u>LOGOUT</u></a>	Terminate a phantom process
<a href="#"><u>LOGTO</u></a>	Change to an alternative account
<a href="#"><u>LOOP / REPEAT</u></a>	Defines loop within paragraph
<a href="#"><u>MAKE.INDEX</u></a>	Create and build an alternate key index
<a href="#"><u>MAP</u></a>	Display a list of the catalogue contents
<a href="#"><u>MED</u></a>	Edit a menu definition
<a href="#"><u>MERGE.LIST</u></a>	Create a select list by merging two other lists
<a href="#"><u>MESSAGE</u></a>	Send a message to selected other users
<a href="#"><u>MODIFY</u></a>	Modify records in a file
<a href="#"><u>NETWAIT</u></a>	Wait for incoming telnet connection (PDA only)
<a href="#"><u>NLS</u></a>	Set or report national language support values
<a href="#"><u>NSELECT</u></a>	Remove items from a select list
<a href="#"><u>OFF</u></a>	Synonym for QUIT
<a href="#"><u>OPTION</u></a>	Set, clear or display options
<a href="#"><u>PASSWORD</u></a>	Change user password for network connection
<a href="#"><u>PAUSE</u></a>	Display "Press return to continue" prompt
<a href="#"><u>PDEBUG</u></a>	Runs the phantom debugger
<a href="#"><u>PDUMP</u></a>	Generate a process dump file
<a href="#"><u>PHANTOM</u></a>	Initiate a background process
<a href="#"><u>PRINTER</u></a>	Administer print units
<a href="#"><u>PSTAT</u></a>	Report process status
<a href="#"><u>PTERM</u></a>	Set or display terminal characteristics
<a href="#"><u>PUBLISH</u></a>	Set up, cancel or query publishing of a file for replication
<a href="#"><u>PUBLISH.ACCOUNT</u></a>	Select files to set up or cancel for replication

<a href="#"><u>QSELECT</u></a>	Construct a select list from the content of selected records
<a href="#"><u>QUIT</u></a>	Terminate session or revert to lower command level
<a href="#"><u>REBUILD.ALL.INDICES</u></a>	Rebuild all alternate key indices in an account
<a href="#"><u>REDO</u></a>	Repeat a command at specified intervals
<a href="#"><u>RELEASE</u></a>	Release record or file locks
<a href="#"><u>REMOVE.DF</u></a>	Remove an element of a distributed file
<a href="#"><u>RENAME</u></a>	Synonym for CNAME
<a href="#"><u>REPORT.SRC</u></a>	Display @SYSTEM.RETURN.CODE at command prompt
<a href="#"><u>REPORT.STYLE</u></a>	Sets the default style for query processor reports
<a href="#"><u>RESET.MASTER.KEY</u></a>	Resets the master encryption key
<a href="#"><u>RESTORE.ACCOUNTS</u></a>	Restore all accounts from a FILE.SAVE tape
<a href="#"><u>REVOKE.KEY</u></a>	Removes access to a data encryption key
<a href="#"><u>RUN</u></a>	Run a compiled QMBasic program
<a href="#"><u>SAVE.LIST</u></a>	Save a select list
<a href="#"><u>SAVE.STACK</u></a>	Save the command stack
<a href="#"><u>SCRIB</u></a>	Create or edit a screen definition
<a href="#"><u>SEARCH</u></a>	Search file for records containing string(s)
<a href="#"><u>SECURITY</u></a>	Enable, disable or report system security
<a href="#"><u>SED</u></a>	Screen editor
<a href="#"><u>SEL.RESTORE</u></a>	Selective restore from an ACCOUNT.SAVE or FILE.SAVE tape
<a href="#"><u>SELECT</u></a>	Select records meeting criteria
<a href="#"><u>SELECTINDEX</u></a>	Builds a select list of all indexed values in a specified alternate key index
<a href="#"><u>SET</u></a>	Set a user @variable
<a href="#"><u>SET.DEVICE</u></a>	Attach a tape device
<a href="#"><u>SET.ENCRYPTION.KEY.NAME</u></a>	Updates encryption key names for a file
<a href="#"><u>SET.EXIT.STATS</u></a>	Set final exit status value
<a href="#"><u>SET.FILE</u></a>	Set a Q-pointer to a remote file
<a href="#"><u>SET.PRIVATE.SERVER</u></a>	Define a QMNet private server
<a href="#"><u>SET.QUEUE</u></a>	Define a Pick style form queue
<a href="#"><u>SET.SERVER</u></a>	Define a QMNet server
<a href="#"><u>SET.TRIGGER</u></a>	Set, remove or display trigger function for a dynamic file
<a href="#"><u>SETPORT</u></a>	Set communications parameters of a serial port
<a href="#"><u>SETPTR</u></a>	Set print unit characteristics
<a href="#"><u>SETUP.DEMO</u></a>	Create demonstration files
<a href="#"><u>SH</u></a>	Execute shell command
<a href="#"><u>SHOW</u></a>	Build select list interactively
<a href="#"><u>SLEEP</u></a>	Suspend process until specified time
<a href="#"><u>SORT</u></a>	List records sorted by record key
<a href="#"><u>SORT.ITEM</u></a>	List records sorted by record key in internal format
<a href="#"><u>SORT.LABEL</u></a>	List records in address label format, sorted by record key
<a href="#"><u>SORT.LIST</u></a>	Sort a saved select list
<a href="#"><u>SP.ASSIGN</u></a>	Set printer options using a Pick style form queue
<a href="#"><u>SP.CLOSE</u></a>	Close a print unit previously in "keep open" mode
<a href="#"><u>SP.OPEN</u></a>	Open a print unit in "keep open" mode
<a href="#"><u>SP.VIEW</u></a>	View and print records from \$HOLD or other files
<a href="#"><u>SPOOL</u></a>	Send record(s) to the printer
<a href="#"><u>SSELECT</u></a>	Select records meeting criteria, sorting list by record key
<a href="#"><u>STATUS</u></a>	Display list of active phantom processes
<a href="#"><u>STOP</u></a>	Terminate an active paragraph
<a href="#"><u>SUBSCRIBE</u></a>	Establish replication subscriber data transfer
<a href="#"><u>SUM</u></a>	Report total of named fields
<a href="#"><u>T.ATT</u></a>	Synonym for SET.DEVICE
<a href="#"><u>T.DET</u></a>	Detach a previously assigned tape device

<a href="#"><u>T.DUMP</u></a>	Save a file
<a href="#"><u>T.EOD</u></a>	Position a tape device to the end of the recorded data
<a href="#"><u>T.FWD</u></a>	Move a tape device forward by one file
<a href="#"><u>T.LOAD</u></a>	Restore a file
<a href="#"><u>T.RDLBL</u></a>	Read the label from a tape device
<a href="#"><u>T.READ</u></a>	Read data from a tape device
<a href="#"><u>T.REW</u></a>	Rewind a tape device
<a href="#"><u>T.STAT</u></a>	Report the status of a tape device
<a href="#"><u>T.WEOF</u></a>	Write end of file marker to a tape device
<a href="#"><u>TERM</u></a>	Set or display terminal window size
<a href="#"><u>TIME</u></a>	Display date and time
<a href="#"><u>UMASK</u></a>	Set access rights to be applied at file creation
<a href="#"><u>UNLOCK</u></a>	Unlock a record or file
<a href="#"><u>UNLOCK.KEY.VAULT</u></a>	Enable access to encryption key vault
<a href="#"><u>UPDATE.ACCOUNT</u></a>	Update VOC items from NEWVOC
<a href="#"><u>UPDATE.LICENCE</u></a>	Apply new licence information
<a href="#"><u>UPDATE.RECORD</u></a>	Database maintenance tool
<a href="#"><u>WHO</u></a>	Display user number and account name
<a href="#"><u>WHERE</u></a>	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

```
$ECHO {ON}
$ECHO OFF
```

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 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 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 are trace of each command prior to execution:

```
:CLEAN.BP
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:

<b>BINARY</b>	Suppresses translation of field marks to newlines when restoring directory files. Use this option when restoring binary data.
<b>DET.SUP</b>	Suppresses display of the name of each file as it is restored.
<b>DIRECTORY</b>	Causes new files to be created as directory files. Existing files are not reconfigured.
<b>NO.CASE</b>	Causes new files to be created with case insensitive record ids. Existing files are not reconfigured.
<b>NO.INDEX</b>	Do not create alternate key indices.
<b>NO.OBJECT</b>	Omits restore of object code. This is particularly useful when migrating to QM from other environments.
<b>POSITIONED</b>	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 multivalued 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 dialog. If the name supplied in this dialog 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](#), [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:

- |                       |   |
|-----------------------|---|
| <b>BINARY</b>         | suppresses translation of newlines to field marks when saving directory files. Use this option when saving binary data. |
| <b>DET.SUP</b>        | suppresses display of the names of files saved.   |
| <b>EXCLUDE.REMOTE</b> | causes remote files to be omitted as described below.   |
| <b>INCLUDE.REMOTE</b> | 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. The account transfer tools are intended for use when migrating to QM from other systems. 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 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:

- 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**
- 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.
- 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, 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*

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.

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*.

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
File Statistics Monitor	FSTAT
Data Encryption	
List Encryption Keys	LIST.KEYS
Create Encryption Key	CREATE.KEY
Grant Access to Key	GRANT.KEY
Revoke Access to Key	REVOKE.KEY
Delete Encryption Key	DELETE.KEY
Replication	
Disable Publishing	DISABLE.PUBLISHING
Enable Publishing	ENABLE.PUBLISHING

## 4.8 ADMIN.SERVER

The **ADMIN.SERVER** command allows a system administrator to set security rules on [QMNet](#) public server definitions

### Format

**ADMIN.SERVER** {*name*}

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 slave 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 slave 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 use the same user name and password as the session from which QMNet 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 QMNet slave process based on criteria related to the local user accessing the QMNet file. When creating a QMNet 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 slave process. Changing this name will also prompt for the associated password.                                   |
| 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 <a href="#">ADMIN.USER</a> . 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	Move down to next line
Ctrl-P	Cursor up	Move up to previous line
Ctrl-Z		
	Page down	Move down one page
	Page up	Move up one page

To amend a line, simply type new data or use any of the standard editing keys:

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		Delete all characters after the cursor
	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.
F1		Display help text
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:**

[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.

User management is not applicable to the PDA version of QM.

### 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 98/ME 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 phantoms and QMClient processes. If left blank, they will be asked for an account name after entry of their user name and password.	
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.	
Allow	Allows user to:	
	Create/delete accounts	Use <a href="#">CREATE.ACCOUNT</a> , <a href="#">DELETE.ACCOUNT</a> and create accounts by entering QM in a directory that is not already an account.
	Define private servers	Use <a href="#">SET.PRIVATE.SERVER</a> to create QMNet server definitions that are private to the session in which the command is issued.

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 LevelSecurity](#)

## 4.10 ALIAS

The **ALIAS** command creates a temporary alias for a command.

### Format

<b>ALIAS</b> <i>command target</i>	Create an alias
<b>ALIAS</b> <i>command</i>	Remove an alias
<b>ALIAS</b>	List all defined aliases

where

*command* is the alias name

*target* is the command to which the alias applies

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.

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 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.

<b>LPTR</b> { <i>n</i> }	Directs output to the specified logical print unit. If <i>n</i> is omitted, the default printer is used.
<b>NO.PAGE</b>	Suppresses paging of the output. This option is ignored if <b>LPTR</b> is used.
<b>STATISTICS</b>	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)
```

	Per group:	Minimum	Maximum	Average
Group buffers	:	1	3	1.28
Total records	:	9	28	16.56
Used bytes	:	36	1020	824.19

	Bytes per record:	Minimum	Maximum	Average
All records	:	16	4029	52.70
Normal records	:	16	804	49.82
Large records	:	984	4029	2506.50

The above example shows analysis of a dynamic file.

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 username 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 registered as administrators. 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.

### 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

<i>file.name</i>	is the name of the directory file holding the QMBasic source program. If omitted, the <i>filename</i> defaults to BP.
<i>record.name</i>	is the name of the record within the file. Multiple record names may be specified. An asterisk as the only <i>record.name</i> 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.
<i>options</i>	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

<b>CHANGED</b>	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 it enables all modified programs to be compiled with a single command.
<b>DEBUGGING</b>	Include debugger control information in the compiled program. A program compiled in debug mode can be executed outside the debugger but will be slower than when compiled without the <b>DEBUGGING</b> option.
<b>LISTING</b>	Generate a compiler listing record with a .LIS suffix.
<b>NO.PAGE</b>	Suppress pagination.
<b>NO.QUERY</b>	Suppresses any queries that the compiler may generate such as replacement of a catalogued item, taking the default action.

<b>NOXREF</b>	Omit cross reference tables from compiled program. This results in lower memory usage but run time error messages cannot identify source line numbers or variable names.
<b>XREF</b>	Generates a compiler listing record as for the <b>LISTING</b> option but includes a cross-reference table of all variables and their use. A line number in the cross-reference data will be followed by S on lines where the variable is set and A if it is used as a subroutine or function argument.

The compiler output file, named as the source file but with a .OUT suffix, will be created automatically if it does not already exist.

[@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 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:

<b>CATALOGUE {LOCAL   GLOBAL}</b>	(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. The <b>NO.QUERY</b> keyword can be added to suppress the query prompt that normally occurs if the program is already catalogued in some other mode. The <a href="#">\$NO.CATALOGUE</a> compiler directive can be used in specific program modules to override this action. Alternatively, the <a href="#">\$CATALOGUE</a> compiler directive in a program can set an alternative catalogue name or mode.
<b>DEBUGGING</b>	Compiles the program in debug mode.
<b>DEFINE name {value}</b>	Defines token <i>name</i> in the same way as the <a href="#">\$DEFINE</a> compiler directive. The <i>value</i> may be a number or a quoted string. If omitted, the token is assigned a null string as its value.
<b>LISTING</b>	Generates a listing record in the compiler output file.

<b>MODE</b> <i>option.name</i>	Sets the given compilation mode as described for the <a href="#">\$MODE</a> compiler directive. Multiple MODE lines must be used to set more than one option.
<b>NOCASE.STRINGS</b>	Compiles the program with case insensitive string operations. See the <a href="#">\$NOCASE.STRINGS</a> compiler directive for more details.
<b>NOXREF</b>	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.
<b>NO.PAGE</b> or <b>NOPAGE</b>	Suppress pagination.
<b>WARNINGS.AS.ERRORS</b>	Causes the compiler to treat warning messages as fatal errors.
<b>XREF</b>	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.

### 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
```

### 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

0            after **BELL OFF**

1            after **BELL ON**

-ve          after an error

Printing CHAR(7) to the terminal sounds the audible alarm regardless of the setting of the bell status.

## 4.16 BLOCK.PRINT and 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.

## 4.17 BREAK

The **BREAK** command controls the action taken on use of the break key. It can be used, for example, to suppress quits during critical parts of an application.

### Format

<b>BREAK OFF</b>	To suppress quits
<b>BREAK ON</b>	To enable quits
<b>BREAK CLEAR</b>	To cancel deferred breaks
<b>BREAK COUNT</b>	To report the number of active <b>BREAK OFF</b> commands
<b>BREAK ON USER</b> <i>n</i>	To enable the break key for the specified user

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 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.

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 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:

- |                 |   |
|-----------------|---|
| <b>LOCAL</b>    | The program is to be catalogued in the VOC.   |
| <b>GLOBAL</b>   | The program is to be catalogued for use by all accounts.  |
| <b>NO.PAGE</b>  | Suppresses page end prompts when output exceeds page length (e.g. use of select list).  |
| <b>NO.QUERY</b> | Suppresses all confirmation prompts.  |
| <b>NOXREF</b>   | Catalogues the program without any symbol and line cross-reference information. This results in lower disk and memory usage but prevents full diagnostic messages in case of run time errors. |
| <b>RELATIVE</b> | Used with <b>LOCAL</b> , this causes the pathname stored in the VOC item to be a relative pathname so long as the program file is a subdirectory of the                                       |

account.

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.

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

Without either the **LOCAL** or **GLOBAL** keyword, 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

With the **GLOBAL** keyword, the program is copied to the global catalogue file in the QMSYS directory and is available from all accounts. 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
- \_ Internal undocumented subroutines
- \$ System subroutines, only callable from internal mode programs

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

With the **LOCAL** keyword, an entry is written to the VOC file allowing calls only from the account in which the program is catalogued. This is useful during testing or for functions that are not called frequently. The first call to programs catalogued in this way are slower than to private or globally catalogued programs. Once the program has been loaded into memory, speed will be identical.

---

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.

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.

**See also:**

[BASIC](#), [DELETE.CATALOGUE](#), [FIND.PROGRAM](#), [MAP](#)

## 4.20 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.

## 4.21 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.

### See also:

[ABORT](#) command, QMBasic [ABORT](#) statement

## 4.22 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

## 4.23 CLEAR.FILE

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](#), the file returns to its minimum modulus and all overflow space is released.

## 4.24 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](#)

## 4.25 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](#)

## 4.26 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](#)

## 4.27 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

<b>CLEAR.SELECT</b>	Clears select list 0, the default list
<b>CLEAR.SELECT</b> <i>list.number</i>	Clears the specified list (0 to 10)
<b>CLEAR.SELECT ALL</b>	Clears all select lists

[@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.

## 4.28 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](#)

## 4.29 CLR

The **CLR** command (synonym **CS**) clears the terminal screen.

### Format

**CLR**

[@SYSTEM.RETURN.CODE](#) is not affected by this command.

## 4.30 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.

## 4.31 COMO

The **COMO** command controls recording of terminal output in a como (command output) file.

### Format

<b>COMO ON</b> <i>name</i> ( <b>APPENDING</b> )	Commence recording of terminal output in named record.
<b>COMO SUSPEND</b>	Temporarily suspends logging output in the como file.
<b>COMO RESUME</b>	Resumes logging out output after a previous use of the <b>SUSPEND</b> option.
<b>COMO OFF</b>	Terminate recording of terminal output.

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 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. If *record.name* 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 *name*, 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).

## 4.32 COMPILE.DICT

The **COMPILE.DICT** command (synonym **CD**) is used to compile A, C, I and S-type records in dictionaries.

### Format

**COMPILE.DICT** *file.name* {*name*} ... {**NO.QUERY**} {**NO.PAGE**}

**COMPILE.DICT ALL** {**NO.PAGE**}

**COMPILE.DICT LOCAL** {**NO.PAGE**}

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.

The **COMPILE.DICT ALL** and **COMPILE LOCAL** formats provide an easy way to compile all items in multiple files. 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.

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.

## 4.33 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.

The second form can be used to set parameter *param* to *new.value* in the current process. Only parameters that are maintained on a per-process basis can be modified.

### Examples

```
CONFIG

Version number 2.2-10
Licence number 1961491396, System id LWWK-FTXT
Maximum users 50, available 43
Expiry date 31 DECEMBER 2007
Licensed to Manor Developments Limited

MUSTLOCK 0
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 shows the values of configurable parameters from the QM configuration parameter file.

### 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.

## 4.34 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:

<b>DYNAMIC</b>	Converts file to dynamic hashed type. Ignored if file is already dynamic.
<b>DIRECTORY</b>	Converts file to directory type. Ignored if file is already a directory.
<b>CASE</b>	converts the file to use case sensitive record ids.
<b>GROUP.SIZE</b> <i>n</i>	sets the group size in units of 1024 bytes. Values in the range 1 to 8 are permitted.
<b>LARGE.RECORD</b> <i>bytes</i>	sets the large record size in bytes.
<b>MERGE.LOAD</b> <i>pct</i>	sets the merge load factor for the file.
<b>MINIMUM.MODULUS</b> <i>n</i>	sets the minimum modulus for the file. Any positive non-zero value may be used.
<b>SPLIT.LOAD</b> <i>pct</i>	sets the split load factor for the file.
<b>DEFAULT</b>	resets all parameters to their default values.
<b>NO.CASE</b>	converts the file to use case insensitive record ids.
<b>NO.MAP</b>	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 <b>DICT</b> 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 <a href="#">directory files</a> for more information.
<b>NO.RESIZE</b>	disables file resizing.
<b>RESIZE</b>	enables file resizing.
<b>IMMEDIATE</b>	causes an immediate file resize, if

required.

Parameters which are not specified retain their existing values.

The **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 **REPORTING** option will show progress. Changes only affecting other [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 contain two or more record using keys 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.

### 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.

## 4.35 COPY

The **COPY** command copies selected records from one file to another, or within the same file.

### 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:

<b>ALL</b>	Copy all records from <i>src.file</i> to <i>tgt.file</i> . This option cannot be used with named records or a select list.
<b>BINARY</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 unless the <b>TEXT</b> option is used.
<b>DELETING</b>	Delete the record(s) from <i>src.file</i> after copying.
<b>NO.QUERY</b>	Suppresses confirmation prompt when using a select list. The <b>NO.SEL.LIST.QUERY</b> mode of the <a href="#">OPTION</a> command can be used to imply this option.
<b>OVERWRITING</b>	Overwrite existing record(s) of the same name in <i>tgt.file</i> . Without this option, records which already exist are not copied. This option may not be used with <b>UPDATING</b> .
<b>REPORTING</b>	Display the id of each record as it is copied.
<b>TEXT</b>	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).

**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
```

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 **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.

**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](#)

## 4.36 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

<i>src.list</i>	is the name of the saved select list that is to be copied. If <i>src.list</i> is given as *, all saved select lists in the source file are copied.
<i>tgt.list</i>	is the name to be used for the copied select list. If omitted, the list retains its original name. A <i>tgt.list</i> name cannot be specified when copying all saved lists from the source file.
<i>src.file</i>	is the name of the file holding the <i>src.list</i> to be copied. If omitted, the default saved lists file <b>\$SAVEDLISTS</b> is used.
<i>tgt.file</i>	is the name of the file to receive the copied select list. If omitted, the default saved lists file <b>\$SAVEDLISTS</b> is used.
<i>options</i>	may be any of the following
<b>CRT</b>	Output the select list to the display. Neither <i>tgt.list</i> nor <i>tgt.file</i> may be specified with this option.
<b>DELETING</b>	Delete <i>src.list</i> after copying.
<b>LPTR</b> { <i>n</i> }	Output the select list to logical print unit <i>n</i> . If <i>n</i> is not specified, the default print unit is used.
<b>NO.PAGE</b>	Used with the <b>CRT</b> option, this option suppresses the normal pause between successive pages of output.
<b>OVERWRITING</b>	If <i>tgt.list</i> already exists in <i>tgt.file</i> , 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](#)

## 4.37 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](#)**

## 4.38 COPYP

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.

### 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](#)

## 4.39 CREATE.ACCOUNT

The **CREATE.ACCOUNT** command creates a new QM account.

### Format

```
CREATE.ACCOUNT acc.name pathname {NO.QUERY}  
                {MODE mmm} {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.
- NO.QUERY** suppresses the confirmation prompts when the *pathname* directory already contains a VOC file or when creating a new directory.

The **CREATE.ACCOUNT** command creates a new account and populates the VOC file from the NEWVOC template 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 \$ACCOUNT.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 **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

```
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](#)

## 4.40 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}
```

where

<i>portion</i>	identifies the part of the file to create. This may be <b>DATA</b> to create just the data portion, <b>DICTIONARY</b> to create just the dictionary portion or omitted to create both.
<i>file.name</i>	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 <i>file.name</i> and the directory for the dictionary component of a file is the same but with a .DIC suffix.
<i>subfile</i>	is the name of the subfile to be created in a multifile.
<i>type</i>	specifies the file type as <b>DIRECTORY</b> or <b>DYNAMIC</b> . If omitted, a <a href="#">dynamic file</a> is created by default. Dictionaries are always created as dynamic files regardless of any <i>type</i> argument.

The *configuration* options available when creating a dynamic file to specify the file's configuration and location are:

<b>CASE</b>	creates a file where the record ids will be treated as case sensitive. This is the default action unless the <b>CREATE.FILE.NO.CASE</b> mode of the <a href="#">OPTION</a> command is enabled.
<b>PATHNAME</b> <i>path</i>	specifies the pathname of an existing operating system directory under which the file is to be created.
<b>MINIMUM.MODULUS</b> <i>n</i>	sets the minimum modulus for the file. Any positive non-zero value may be used. The default is 1.
<b>GROUP.SIZE</b> <i>size</i>	sets the group size as a multiple of 1024 bytes. This must be in the range 1 to 8. If omitted, the default group size is taken from the <a href="#">GRPSIZE</a> configuration parameter.
<b>LARGE.RECORD</b> <i>bytes</i>	sets the large record size in bytes. The default is 80% of the group size.
<b>SPLIT.LOAD</b> <i>pct</i>	sets the split load factor for the file. The default is 80%.

<b>MERGE.LOAD</b> <i>pct</i>	sets the merge load factor for the file. The default is 50%.
<b>VERSION</b> <i>vno</i>	allows creation of files with internal formats compatible with older releases of QM.
<b>NO.CASE</b>	creates a file where the record ids will 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. The <b>CREATE.FILE.NO.CASE</b> mode of the <b>OPTION</b> command can be used to make this the default action.
<b>NO.RESIZE</b>	creates the file with resizing disabled. See <a href="#">CONFIGURE.FILE</a> and <a href="#">dynamic files</a> for more information.

The *configuration* options available when creating a directory file are:

<b>PATHNAME</b> <i>path</i>	specifies the pathname of an existing operating system directory under which the file is to be created.
<b>NO.MAP</b>	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 <b>DICT</b> 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 <a href="#">directory files</a> 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.

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.

Case sensitivity of record ids in directory files is dependent on the operating system.

To provide maximum flexibility, QM imposes no restrictions on the file name except that it may not contain the mark characters or the ASCII null character. Use of spaces in file names is not recommended as it is likely to lead to problems with some commands. Characters that are not valid in operating 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.

### 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.

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` parameter.

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. These values are as used on Linux, FreeBSD and AIX where 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](#)

## 4.41 CREATE.INDEX

The **CREATE.INDEX** command creates an [alternate key index](#).

### Format

```
CREATE.INDEX filename field(s) {NO.NULLS} {NO.CASE} {PATHNAME  
index.path}
```

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.

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 D, I, A, S or C-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 [TRANSQ](#) function to access other files.

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.

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 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

```
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](#)

## 4.42 CREATE.KEY

The **CREATE.KEY** command creates a data encryption key. This command can only be executed by users with administrator rights in the QMSYS account.

### Format

```
CREATE.KEY {keyname {algorithm {keystring}}}
```

where

*keyname* is the name for the new encryption key.

*algorithm* is the encryption algorithm to be associated with the key.

*keystring* is the actual encryption key.

The command prompts for items not supplied on the command line.

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 name is case insensitive.

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 off-site in case it is ever necessary to rebuild the key vault.

The **CREATE.KEY** automatically grants access to the key to the user that created it. Other users can be granted access using the **GRANT.KEY** command

### Example

```
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](#), [CREATE.FILE](#), [DELETE.KEY](#), [ENCRYPT.FILE](#), [GRANT.KEY](#),  
[LIST.KEYS](#), [RESET.MASTER.KEY](#), [REVOKE.KEY](#), [SET.ENCRYPTION.KEY.NAME](#),  
[UNLOCK.KEY.VAULT](#)

## 4.43 CREATE.USER

The **CREATE.USER** command creates a new user name in the register of users for security checks.

User management is not applicable to the PDA version of QM.

### 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 for phantoms and QMClient processes. If not specified, an account name prompt will be issued when the user logs in.

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 98/ME and in user mode installations on Linux/Unix, the new 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.

On later versions of Windows and other platforms, 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](#)

## 4.44 CT

The **CT** (Copy to Terminal) command displays the content of record(s) from a file.

### Format

**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:

<b>BINARY</b>	Display the record as a binary data item.
<b>HEX</b>	Display the data in each field in hexadecimal format.
<b>LPTR</b> <i>n</i>	Redirects the output to printer <i>n</i> . If <i>n</i> is omitted, the default printer is used.
<b>NO.QUERY</b>	When using a select list, the confirmation prompt is omitted. The <b>NO.SEL.LIST.QUERY</b> mode of the <a href="#">OPTION</a> command can be used to imply this option.

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.

The **HEX** option, produces a report in which the data is displayed in hexadecimal form, two hexadecimal digits per character.

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.

### 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 Grove Nottingham  
3: 1-1y3-1
```

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: [DUMP](#)

## 4.45 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

**DATA** {*text*}

where

*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

```
PA
* <<History>>
LOOP
  IF <<A,Record name>> = "" THEN GO DONE
  ED BP <<Record name>>
  DATA I * <<History>>
  DATA FI
REPEAT
DONE :
```

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

## 4.46 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

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
```

## 4.47 DATE.FORMAT

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
```

**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.

## 4.48 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.

## 4.49 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.

## 4.50 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](#)

## 4.51 DELETE.CATALOGUE

The **DELETE.CATALOGUE** command (synonym **DELETE.CATALOG**) removes an entry from the system catalogue.

### Format

```
DELETE.CATALOGUE {name...} {GLOBAL | 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.

**LOCAL** indicates that a locally catalogued version of this subroutine is to be removed.

If neither the **GLOBAL** nor the **LOCAL** keyword is present, the **DELETE.CATALOGUE** command deletes entries from the private catalogue unless *name* has a prefix character that identifies a globally catalogued item.

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.

### See also:

[BASIC](#), [CATALOGUE](#), [MAP](#)

## 4.52 DELETE.COMMON

The **DELETE.COMMON** command deletes one or all named common blocks.

### Format

**DELETE.COMMON** *name*

**DELETE.COMMON ALL**

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.

### Examples

```
DELETE.COMMON COM1
```

This command deletes common block COM1.

```
DELETE.COMMON ALL
```

This command deletes all named common blocks.

### See also:

[LIST.COMMON](#)

## 4.53 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](#)

## 4.54 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](#), [LISTE](#), [LISTFL](#), [LISTFR](#)

## 4.55 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](#)

## 4.56 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](#), [CREATE.FILE](#), [CREATE.KEY](#), [ENCRYPT.FILE](#), [GRANT.KEY](#), [LIST.KEYS](#), [RESET.MASTER.KEY](#), [REVOKE.KEY](#), [SET.ENCRYPTION.KEY.NAME](#), [UNLOCK.KEY.VAULT](#)

## 4.57 DELETE.LIST

The **DELETE.LIST** command deletes a previously saved [select list](#).

### Format

**DELETE.LIST** *list.name*

where

*list.name* is the name of the record in \$SAVEDLISTS that is to be deleted.

The **DELETE.LIST** command deletes a previously saved select list from the \$SAVEDLISTS file. It has no effect on any active select lists.

### Example

```
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](#)

## 4.58 DELETE.SERVER

The **DELETE.SERVER** command removes a QMNet 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](#) 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.

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 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:

[QMNet](#), [ADMIN.SERVER](#), [LIST.SERVERS](#), [SET.SERVER](#)

## 4.59 DELETE.USER

The **DELETE.USER** command deletes a user name from the register of users for security checks.

User management is not applicable to the PDA version of QM.

### 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](#)

## 4.60 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](#)

## 4.61 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.

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](#)

## 4.62 DISPLAY

The **DISPLAY** command displays text at the user's terminal.

### Format

```
DISPLAY text {:}
```

```
DISPLAY @(col,row)text {:}
```

The optional colon at the end of the line suppresses the normal newline after the *text* is displayed.

The second format positions the cursor to the given column and row (both numbered from zero) before displaying the *text*. There must be no spaces in the cursor position element. Any spaces following the close bracket are treated as part of the *text*. The QMBasic [@\(\)](#) function variants with a negative value for the first (or only) argument are also supported.

The [@SYSTEM.RETURN.CODE](#) variable is not affected by this command.

### Example

A paragraph containing

```
DISPLAY Hello :  
DISPLAY world
```

would display

```
Hello world
```

## 4.63 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:

- 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 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
```

The above command dumps the record with id 2 from the READERS file.

See also: [CT](#)

## 4.64 ECHO

The **ECHO** command suspends or enables echoing of keyboard input.

### Format

<b>ECHO OFF</b>	Disable echoing of keyboard input
<b>ECHO ON</b>	Enable echoing of keyboard input
<b>ECHO</b>	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.

## 4.65 ED

The **ED** command enters the QM line editor. The synonym **EDIT** can be used.

### Format

**ED** {**DICT**} *file.name* {*record.id*} {**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.

**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. 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 **G***n* (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.

### **PO***n*

The **PO** (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

### **L***n*

The **L***n* (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.

**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.

**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.

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.

### 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, 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**

#### **FD**

Prompts for confirmation and then deletes the entire record from the file.

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

### EX

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 *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.

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 *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.

## X

The **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.

## Miscellaneous Commands

### ?

The **?** command displays status information about the editor and the record being edited. This includes

- The file name and record id
- The current line number
- Block start and end line positions
- The command, if any, which will be reverted by OOPS.
- Non-printing character expansion mode status (^)

Non-printing character entry mode status (NPRINT)  
Block verify mode status (BLOCK)  
The setting of search case sensitivity (CASE)

^

Toggles non-printing character expansion mode. When this mode is enabled, non-printing characters are displayed as *^nnn* where *nnn* is the decimal character number.

### **CASE OFF**

Sets case insensitive mode for the **C**, **F** and **L** commands.

### **CASE ON**

Sets case sensitive mode for the **C**, **F** and **L** commands. This is the default mode of the editor.

### **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.

### **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**

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:

<b>.S</b> <i>item</i>	Save the most recent command
<b>.Sn</b> <i>item</i>	Save command <i>n</i>
<b>.S</b> <i>item n,m</i>	Save commands <i>n</i> to <i>m</i>
<b>.S</b> <i>item n m</i>	Save commands <i>n</i> to <i>m</i>

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 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 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:

<b>.D</b> <i>item</i>	Delete the specified item
<b>.L</b> <i>item</i>	List the item. If the record id is given as an asterisk, a list of stored edit sequences in the file is displayed.
<b>.R</b> <i>item</i>	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 I 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.

**.A**{*n*} *string*

Append *string* to entry *n* of the editor command stack.

**.C**{*n*}/*old.string/new.string*/

Change *old.string* to *new.string* in line *n* of the editor command stack.

**.D**{*n*}

Delete line *n* of the editor command stack.

**.I**{*n*} *string*

Insert *string* as entry *n* of the editor command stack.

**.L**{*n*}

List *n* lines of the editor command stack. The value of *n* defaults to nine.

**.R**{*n*}

Recall line *n* of the editor command stack to the top of the stack.

**.X**{*n*}

Execute line *n* of the editor command stack.

See also **ED Pre-Stored Commands** above for additional command stack features.

### 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.
- 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.
- NPRINT {ON | OFF}** Turn on/off acceptance of non-printing characters on input. Default is OFF.

Unrecognised options or qualifiers are ignored.

## 4.66 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.

The screen display from this command is as shown below.

```

ERRLOG=20
EXCLREM=0
FILERULE=7
FIXUSERS=21,10
FSYNC=0
GDI=0
GRPSIZE=1
INTPREC=13
JNLDIR=\qmjnl
JNLMODE=3
LPTRHIGH=66
LPTRWIDE=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.

F1=Help

```

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	Move down to next line
Ctrl-P	Cursor up	Move up to previous line
Ctrl-Z		
	Page down	Move down one page
	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	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		Delete all characters after the cursor
	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.

## 4.67 EDIT.LIST

The **EDIT.LIST** command invokes the **ED** line editor to edit a saved select list in the `$$AVEDLISTS` 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](#)

## 4.68 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](#)

## 4.69 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*

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, the indexed fields have reduced security.

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](#), [CREATE.FILE](#), [CREATE.KEY](#), [DELETE.KEY](#), [GRANT.KEY](#), [LIST.KEYS](#), [RESET.MASTER.KEY](#), [REVOKE.KEY](#), [SET.ENCRYPTION.KEY.NAME](#)

## 4.70 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:

<b>BINARY</b>	suppresses translation of newlines to field marks when saving directory files. Use this option when saving binary data.
<b>DET.SUP</b>	suppresses display of the names of files saved.
<b>EXCLUDE.REMOTE</b>	causes remote files to be omitted as described below.
<b>INCLUDE.REMOTE</b>	causes remote files to be saved as described below.
<b>NO.QUERY</b>	suppresses the confirmation prompt when using a select list. The <b>NO.SEL.LIST.QUERY</b> mode of the <b>OPTION</b> 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. Because the media format is defined by Pick, it 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:

- Field 5 of the F-type VOC entry can contain
  - Save the dictionary but omit the data element
  - 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](#), [RESTORE.ACCOUNTS](#), [SEL.RESTORE](#), [SET.DEVICE](#), [T.ATT](#), [T.DUMP](#), [T.LOAD](#), [T.xxx](#)

## 4.71 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 or VFS for a virtual 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 dynamic hashed file, this is an approximate count but should be close to the correct value. For a directory file, the <b>FILE.STAT</b> 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).
ST	Is statistics monitoring enabled on this file (Y/N).
NC	Does the file use case insensitive record ids (Y/N).
NR	Is the NO.RESIZE option active for this file (Y/N).
EN	Does the file use encryption (Y/N).
RP	Is replication enabled on this file (Y/N).
JN	Is journalling enabled on this file (Y/N).

Columns that are not applicable to directory files will be left blank for such files.

**See also:**

[ANALYSE.FILE](#)

## 4.72 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 display a list of accounts that are on the tape.

### See also:

[ACCOUNT.RESTORE](#), [ACCOUNT.SAVE](#), [FILE.SAVE](#), [RESTORE.ACCOUNTS](#), [SEL.RESTORE](#), [SET.DEVICE](#), [T.ATT](#), [T.DUMP](#), [T.LOAD](#), [T.xxx](#)

## 4.73 FIND.PROGRAM

The **FIND.PROGRAM** command reports the catalogue mode of a catalogued program.

### Format

**FIND.PROGRAM** *name*

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.

### Example

```
:FIND.PROGRAM DTMOD  
Program is in private catalogue  
Compiled 20 Jan 09 16:24:16  
Subroutine with 3 arguments
```

### See also:

[CATALOGUE](#)

## 4.74 FORMAT

The **FORMAT** command reformats 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
12345678901234567890123456789012345678901234567890123456789
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
12345678901234567890123456789012345678901234567890123456789
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
12345678901234567890123456789012345678901234567890123456789
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
```

## 4.75 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 active select list zero. Any previously active select list zero is discarded. 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](#)

## 4.76 FSTAT

The **FSTAT** command collects and report file access statistics.

### Format

<b>FSTAT ON</b> <i>file.name...</i>	Enable statistic collection
<b>FSTAT</b> <i>file.name...</i> { <b>LPTR</b> }	Report statistics
<b>FSTAT OFF</b> <i>file.name...</i>	Disable statistics collection
<b>FSTAT GLOBAL</b> { <b>LPTR</b> }	Report global system statistics
<b>FSTAT RESET</b>	Clear global statistics counters
<b>FSTAT</b>	Periodic global statistics display

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. <a href="#">READ</a> , <a href="#">READU</a> , <a href="#">READL</a> , <a href="#">MATREAD</a> , <a href="#">MATREADU</a> , <a href="#">MATREADL</a> , 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 <a href="#">SELECT</a> 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. <a href="#">SELECTINDEX</a> ) 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.

### 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           0.0
Reads          273161         143505         4228         4220.7
Writes         258086         134876         3958         3966.9
Deletes        13901          8628           269          253.8
Clears          0             0             0             0.0
Selects         0             0             0             0.0
Splits          4937            10             0             0.3
Merges          0             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

```

## 4.77 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.

## 4.78 GET.LIST

The **GET.LIST** command is used to restore a previously saved [select list](#).

### Format

```
GET.LIST list.name { TO list.no } { COUNT.SUP }
```

where

*list.name* is the name of the record in \$SAVEDLISTS that is to be restored.

*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.

The **GET.LIST** command retrieves a previously saved select list from \$SAVEDLISTS. 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 the number of items in the list.

### 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](#)

## 4.79 GET.STACK

The **GET.STACK** command restores a previously saved [command stack](#).

### Format

```
GET.STACK {stack.name}
```

where

*stack.name* is the name of the saved command stack. A prompt is issued if this name is omitted.

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.

### 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](#)

## 4.80 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
```

## 4.81 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 usernames 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](#), [CREATE.FILE](#), [CREATE.KEY](#), [DELETE.KEY](#), [ENCRYPT.FILE](#), [LIST.KEYS](#), [RESET.MASTER.KEY](#), [REVOKE.KEY](#), [SET.ENCRYPTION.KEY.NAME](#), [UNLOCK.KEY.VAULT](#)

## 4.82 HELP

The **HELP** command provides help on a wide variety of topics. This command is not available on the PDA version of QM.

### 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.hlp` 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 `winhlp32` 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.

## 4.83 HSM

The **HSM** command controls the Hot Spot Monitor performance monitoring tool. This command is not available on the PDA version of QM.

### Format

**HSM ON** {**USER** *n*}            Start monitoring, clearing the counters

**HSM OFF**                            Stop monitoring

**HSM** {**DISPLAY**} {**USER** *n*}    Display performance data

The Hot Spot Monitor records the number of times each program module is called and the processor time in seconds spent in that module.

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.

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... Program
   1      0.000  !SCREEN
  13      0.060  $CPROC
  18      0.000  !PARSER
   1      0.000  PRINT.DEFERED
   4      0.010  $SETPTR
   1      0.000  CHARGE.TOTAL
  19      0.034  CALC.INVOICE.VALUE
  24      0.000  ADD.MONTH
   3      0.000  REVERSE
   1      0.000  C:\QM\BP.OUT\PROC
   6      17.040 C:\QM\BP.OUT\FINANCE
  19      14.010 C:\QM\BP.OUT\INVOICE
   1      0.800  C:\QM\BP.OUT\CHECK_EX
   1      0.000  $HSM
```

## 4.84 HUSH

The **HUSH** command suspends or enables output to the display.

### Format

<b>HUSH ON</b>	Suspend display output
<b>HUSH OFF</b>	Resume display output
<b>HUSH</b>	Toggle hush status

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 [COUNT.SUP](#) option to [SELECT](#) might be a better alternative in this example.

## 4.85 IF

The **IF** command allows conditional execution of sentences within paragraphs.

### Format

```
IF value.1 rel.op value.2 THEN sentence
IF EXISTS filename id THEN sentence
IF NOT.EXISTS filename id THEN sentence
```

where

<i>value.1, value.2</i>	are two items to be compared. These may be inline prompts, constants or @-variables as described below.
<i>rel.op</i>	is the relational operator to be applied to the two values.
<i>sentence</i>	is the sentence to be executed if the condition is true.
<i>filename</i>	is the name of a file in which existence of a record is to be checked.
<i>id</i>	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:

<	LT	BEFORE	LESS
<=	LE	=<	
=	EQ	EQUAL	
>=	GE	=>	
>	GT	AFTER	GREATER

#	NE	<>	><	NOT
LIKE	MATCHES	MATCHING		
UNLIKE	NOT.MATCHES			
~	SAID	SPOKEN		

The function of each relational operator as applied to values of different types is the same as its QMBasic equivalent

The second and third forms of **IF** 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.

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

```
PA
BASIC <<Program name>>
IF @SYSTEM.RETURN.CODE = 1 THEN CATALOGUE <<Program name>>
```

This paragraph compiles a QMBasic 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.

```
IF EXISTS VOC <<@LOGNAME>> THEN <<@LOGNAME>>
```

This example shows how it is possible to produce the effect found in some other multivalued 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.

## 4.86 INHIBIT.LOGIN

The **INHIBIT.LOGIN** command prevents users logging in new QM sessions.

### Format

<b>INHIBIT.LOGIN ON</b>	Prevent further users logging in
<b>INHIBIT.LOGIN OFF</b>	Allow users to login
<b>INHIBIT.LOGIN</b>	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

```
INHIBIT.LOGIN ON
RUN SYSTEM.BACKUP
INHIBIT.LOGIN OFF
```

This sequence prevents new users logging in while the **SYSTEM.BACKUP** program is executed.

## 4.87 LIST.COMMON

The **LIST.COMMON** command lists named common blocks.

### Format

**LIST.COMMON**

The **LIST.COMMON** command displays a list of all currently created QMBasic named common blocks for the process in which the command is executed.

### Example

```
LIST.COMMON  
SYS.FILES  
SCREEN.DATA
```

### See also:

[DELETE.COMMON](#)

## 4.88 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.ALG  
OCONV(@ID['-',1,1], 'DY')
```

<i>Part No.</i>	<i>Pathname</i>
8	C:\SALES\ORDERS-08
9	C:\SALES\ORDERS-09

### See also:

[Distributed files](#), [ADD.DF](#), [REMOVE.DF](#)

## 4.89 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
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 non-major French customers.

### See also:

[LIST.INTER](#), [LIST.UNION](#), [MERGE.LIST](#)

## 4.90 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:\QMSYS\$_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:\QMSYS\$_IPC  
  2 smithm, 7 sales  
C:\QM\BP  
  7 sales  
C:\QM\&SED.EXTENSIONS&  
  7 sales
```

## 4.91 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

<i>filename</i>	is the name of the file to be processed
<i>field(s)</i>	are the names of the fields to be reported. The keyword <b>ALL</b> can be used instead of a list of fields to report all indices on the file.
<b>STATISTICS</b>	reports additional statistical data.
<b>DETAIL</b>	reports statistical data and key reference detail information.
<b>LPTR</b>	directs the report to the default printer (printer 0). The optional <i>n</i> qualifier directs the report to an alternative printer.
<b>NO.PAGE</b>	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 <b>LIST.INDEX</b> 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\*\*](#)

## 4.92 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](#)

## 4.93 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 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
                  bjoness
  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.

```
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, CREATE.FILE, CREATE.KEY, DELETE.KEY, ENCRYPT.FILE,  
GRANT.KEY, RESET.MASTER.KEY, REVOKE.KEY, SET.ENCRYPTION.KEY.NAME

## 4.94 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

```
LIST.LOCKS
 0:      1:      2:      3:      4:      5:      6:      7: 1*
 8:      9:     10:     11:     12:     13:     14:     15:
16:     17:     18:     19: 3    20:     21:     22:     23:
24:     25:     26:     27:     28:     29:     30:     31:
32:     33:     34:     35:     36:     37:     38:     39:
40:     41:     42:     43:     44:     45:     46:     47:
48:     49:     50:     51:     52:     53:     54:     55:
56:     57:     58:     59:     60:     61:     62:     63:
```

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](#)

## 4.95 LIST.PHANTOMS

The **LIST.PHANTOMS** command shows a list of currently running phantom processes.

### Format

#### **LIST.PHANTOMS**

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.

Phantom processes that are running as replication subscribers also show the name of the publisher system from which they are receiving data.

### Example

```
LIST.PHANTOMS
User  Pid      Username      Login time      Account
   2   5001     root          28 May 07:02    QMSYS
   34  5220     jenny         28 May 11:24    SALES
```

### See also:

[CHILD\(\)](#), [PHANTOM](#), [!PHLOG\(\)](#), [STATUS](#)

## 4.96 LIST.READU

The **LIST.READU** command displays details of file and record locks.

### Format

**LIST.READU** {*user*} {**DETAIL**} {**NO.PAGE**} {**LPTR** {*n*}} {**WAIT**}

where

<i>user</i>	is the user number for which the locks are to be reported. If omitted, all locks are displayed.
<b>DETAIL</b>	includes the limit, current count and peak number of record locks.
<b>NO.PAGE</b>	suppresses display pagination.
<b>LPTR</b> { <i>n</i> }	directs output to logical print unit <i>n</i> . If <i>n</i> is omitted, it defaults to zero, the default print unit.
<b>WAIT</b>	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

```
Record lock limit = 400, Current = 3, Peak = 73
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). User 3 is waiting to lock record 17565 in file 4 but is blocked by user 2. Details of users waiting for locks are only shown if the **WAIT** keyword is used.

The first line of the above report is only shown if the **DETAIL** keyword is used. Note that the counts are for active record locks only. 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 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. A type code of WAIT is shown for users waiting for locks.

## 4.97 LIST.REPLICATED

The **LIST.REPLICATED** command displays a summary of all replicated files within an account.

### Format

**LIST.REPLICATED** {**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.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

```
File Name..... Targets.....
ORDERS          REPORTING:SALES:ORDERS
INVOICES        REPORTING:SALES:INVOICES
```

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](#)

## 4.98 LIST.SERVERS

The **LIST.SERVERS** command shows a list of QMNet 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 **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.

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:

[QMNet](#), [ADMIN.SERVER](#), [DELETE.SERVER](#), [SET.SERVER](#)

## 4.99 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

File Name.....	Function.....	Pre	Post
ORDERS	STOCK.CHECK		W D
STAFF	STF.VALIDATE	W	

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.

## 4.100 LIST.UNION

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](#)

## 4.101 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.

User management is not applicable to the PDA version of QM.

### Format

#### LIST.USERS

The **LIST.USERS** command lists the names of all entries in the user name register. This may include an entry for Console, the pseudo name used for QMConsole connections on Windows 98/ME.

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](#).

### Example

```
LIST.USERS
User name..... Login account... Last
login..... Adm Restr
ADMINISTRATOR                22 Oct 06
17:27 Yes No
BERT                          QMTEST                04 Nov 02
10:40 No Yes
Console                        03 Nov 06
13:49 Yes No
GEORGE                        SALES                  03 Nov 06
09:03 No Yes
MARTIN                        01 Nov 06
17:23 No Yes
```

### See also:

[ADMIN.USER](#), [CREATE.USER](#), [DELETE.USER](#), [PASSWORD](#), [SECURITY](#), [Application Level Security](#)

## 4.102 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](#)

## 4.103 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.

## 4.104 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 will be followed by  
      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

Dir    for a directory file.

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 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](#), [LISTFL](#), [LISTFR](#), [LIST.REPLICATED](#)

## 4.105 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 back slash. 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 will be followed by
  - 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
- Dir    for a directory file.
- 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](#)

## 4.106 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 back slash. 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 will be followed by
  - 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
- Dir    for a directory file.
- 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](#)

## 4.107 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.

## 4.108 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.

## 4.109 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.

## 4.110 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.

## 4.111 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.

## 4.112 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.

## 4.113 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.

## 4.114 LISTS

The **LISTS** command lists all sentences defined in the VOC.

### Format

**LISTS** {**LPTR**}

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 **LPTR** keyword will direct the output to the printer. Other query processor keywords can be appended to the command.

## 4.115 LISTU

The **LISTU** command lists the users currently in QM. This command is not available on the PDA version of QM.

### 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.

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.

### Example

```
LISTU
  User  Pid      Puid  Login time      Origin : User name,
Account
*      1   156                15 Apr 11:48  Console: ADMINISTRATOR,
QMSYS
        2   186                15 Apr 11:21  193.118.13.10: JSMITH,
SALES
        4   196                2  15 Apr 12:02  Phantom: ADMINISTRATOR,
QMSYS
```

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
  User  Pid      Puid  Login time      Origin : User name,
Account
*      1   156                15 Apr 11:48  Console: ADMINISTRATOR,
```

---

```
QMSYS
  2 186          15 Apr 11:21 193.118.13.10: JSMITH,
SALES
  4 196          2 15 Apr 12:02 Phantom: ADMINISTRATOR,
QMSYS
```

**See also:**

[LIST.PHANTOMS](#), [STATUS](#)

## 4.116 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.

## 4.117 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:**

[CLEAR.LOCKS](#), [LIST.LOCKS](#), [LOCK](#) (command), [LOCK](#) (QMBasic), [TESTLOCK\(\)](#), [UNLOCK](#)

## 4.118 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:

<b>BAUD</b> <i>rate</i>	Sets the data rate (default 9600).
<b>BITS</b> <i>n</i>	Sets the number of bits per character (default 8).
<b>PARITY</b> <i>mode</i>	Sets the parity mode: NONE (default), ODD or EVEN.
<b>STOP.BITS</b> <i>n</i>	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.

## 4.119 LOGMSG

The **LOGMSG** command adds a line to the system error log. This statement has no effect on the PDA version of QM.

### 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, the **LOGMSG** command will be ignored.

## 4.120 LOGOUT

The **LOGOUT** command terminates a QM process.

### Format

```
LOGOUT {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.

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.

When using a telnet connection to a PDA, this command also terminates the [NETWAIT](#) connection monitor.

### Example

```
LOGOUT 2  
Phantom 2 : forced logout
```

This example shows forced termination of phantom user 2.

**See also:**

[QUIT](#)

## 4.121 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 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.

## 4.122 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
```

## 4.123 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) {NO.NULLS} {NO.CASE} {CONCURRENT} {  
PATHNAME index.path}
```

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.

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 D, I, A, S or C-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 [TRANSQ](#) 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.

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 QMBsaic [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](#)

## 4.124 MAP

The **MAP** command produces a map of the system catalogue.

### Format

```
MAP {ALL} {LPTR {n}} {FILE {file.name}} {NO.PAGE}
```

where

**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.

**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, may be used instead. If both are present, the British spelling takes priority.

### Example

```
MAP ALL LPTR
```

The above command prints a map of the catalogue, including system items.

### See also:

[BASIC](#), [CATALOGUE](#), [DELETE.CATALOGUE](#)

## 4.125 MED

The **MED** command creates or modifies a menu definition.

### Format

```
MED {file.name {menu.name}}
```

where

<i>file.name</i>	identifies the file holding the menu.
<i>menu.name</i>	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:

Ctrl-A	Home	Move to start of line
Ctrl-B	Cursor	Move left one character left
Ctrl-D	Del	Delete character char
Ctrl-E	End	Move to end of line
Ctrl-F	Cursor	Move right one character right
Ctrl-G		Cancel partially entered key sequence
Ctrl-K		Kill line
Ctrl-L		Refresh screen
Ctrl-N	Cursor	Move down one line down
Ctrl-O	Insert	Toggle overlay mode
Ctrl-P	Cursor	Move up one line up
Ctrl-V	Page	Move down one page down
Ctrl-X	Ctrl-X	Exit (some terminal emulators may not permit use of the second form)
C	Ctrl-C	
Ctrl-X	Ctrl-X	Save
S	Ctrl-S	
Ctrl-Y	Esc-Y	Paste data previously removed with Ctrl-K
Esc-V	Page up	Move up one page
Backsp		Backspace ace
F1		Help
F4		Show menu

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.

## 4.126 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](#)

## 4.127 MESSAGE

The **MESSAGE** command sends a message to selected other users. This command is not available on the PDA version of QM.

### 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.

## 4.128 MODIFY

The **MODIFY** command enters the QM record modification processor.

### Format

```
MODIFY {DICT} file.name { field list } { id.list }
```

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.

Items appearing on the command line or in the @MODIFY or @ phrase which are not D-type dictionary entries or phrases are ignored.

Field names may be followed by "**VERIFY filename**". In this case, **MODIFY** will check that data entered into the named field is a record id in the named file.

*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.

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:

*item no* 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.

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.
- 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.

## 4.129 NETWAIT

The **NETWAIT** command is applicable to the PDA version of QM only and waits for an incoming telnet connection.

### Format

#### **NETWAIT**

The **NETWAIT** command waits for an incoming telnet connection, over a network or from a telnet session running on the same PDA. The command may only be executed from command processor level 1, not from an [EXECUTE](#) in a program.

The PDA version of QM does not support multiple processes. Although this may look as though there is a second QM session running, when a connection is established, the telnet connection becomes the keyboard and display for the QM session that executed the **NETWAIT** command, leaving the original screen suspended.

If there is an executable VOC item named ON.CONNECT, this will be run in the newly established telnet session in much the same way as the LOGIN paragraph is executed in normal QM sessions. There is no reason why, if appropriate, the ON.CONNECT item cannot go on to run LOGIN.

The default terminal type, unless changed in the ON.CONNECT paragraph, will be vt100.

The session may be terminated using the [QUIT](#) or [LOGOUT](#) commands. Use of [QUIT](#) will revert to **NETWAIT** to await a new connection. Use of [LOGOUT](#) terminates **NETWAIT** too.

## 4.130 NLS

The **NLS** command sets or reports national language support settings.

### Format

<b>NLS</b>	Report all settings
<b>NLS { DEFAULT }</b>	Set defaults
<b>NLS { <i>key</i> }</b>	Report setting for given parameter
<b>NLS { <i>key value</i> }</b>	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:

Parameter	Default	Notes
CURRENCY	\$	Maximum 8 characters
THOUSANDS	,	Thousands separator character
DECIMAL	.	Decimal separator character
IMPLICIT.DECI MAL	.	Decimal separator character for implicit data conversions. See <a href="#">SETNLS</a> for details.

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](#)

## 4.131 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.

## 4.132 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 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. Also, the [MASTER.LOGIN](#) paragraph is not executed for QMClient sessions.

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.

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.UNASSOC.MV	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.
BACKSLASH.NOT.QUOTE	Causes the command processor to treat backslashes as data characters instead of being recognised as string quotes.
CHAIN.KEEP.COMMON	Retains the unnamed common block and command processor level on use of <a href="#">CHAIN</a> .
CHAINED.SELECT	If set, a query processor select operation that selects at least

	one record will execute any command in the <a href="#">DATA</a> queue on completion. If no records are selected, the data queue is cleared.
CORRELATIVE.NOCASE	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.
CORRELATIVE.REUSE	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.
CRDB.UPCASE	Causes the cr/db suffix displayed by some decimal conversions to appear in uppercase instead of the default lowercase.
CREATE.FILE.NO.CASE	Causes <a href="#">CREATE.FILE</a> to create files with case insensitive record ids by default.
DEBUG.REBIND.KEYS	Causes the <a href="#">QMBasic debugger</a> to rebind the function keys on entry, replacing any user defined bindings with those specified in the terminfo entry for the current terminal type.
DEFAULT.MV	Causes C, D or I-type dictionary items in which the single/multivalued flag is blank to be treated as multivalued instead of defaulting to single valued.
DIR.DTM	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)
DIV.ZERO.WARNING	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.
DUMP.ON.ERROR	Causes generation of a <a href="#">process dump file</a> at a process abort such as a run time fatal error.
ED.NO.QUERY.FD	Suppresses the confirmation prompt in the ED editor when using the FD command or its synonym DELETE.
FORCE.RELOAD	Force reload of QMBasic object code at compilation or catalogue. See the QMBasic <a href="#">CALL</a> statement for more information.
INHERIT	Phantom processes will inherit the option settings of the parent process. Use of an OPTION command in the <a href="#">MASTER.LOGIN</a> or <a href="#">LOGIN</a> paragraphs of the phantom process may modify these settings.
INHERIT.OWNERSHIP	Causes <a href="#">CREATE.FILE</a> to inherit ownership (user id and group id) from the account unless overridden by command options. This option has no effect on Windows systems.
KEEP.FILENAME.CASE	Causes <a href="#">CREATE.FILE</a> to preserve the casing of the QM name of the file when creating the operating system directories that represent this file.
KEEP.OLD.OBJECT	Causes the QMBasic compiler to retain any previous version

---

	of the compiled object code if a compilation fails.
LOCK.BEEP	Emits a beep at the terminal once per second while waiting for a record or file lock.
NO.DATE.WRAPPING	Suppresses rolling of dates with over-large day numbers into the following month on <a href="#">input conversion</a> .
NO.SEL.LIST.QUERY	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.
NO.USER.ABORTS	Suppresses all options that allow a user to generate an abort event. These are: the "Press return to continue" prompt, the pagination prompt when using the <a href="#">SCROLL</a> keyword of the query processor, and the break key options.
NON.NUMERIC.WARNING	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.
PAGINATE.ON.HEADING	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 <a href="#">RUN</a> command can be used to override this.
PICK.BREAKPOINT	Causes the query processor to recognise Pick style syntax for the <a href="#">BREAK.ON</a> and <a href="#">BREAK.SUP</a> keywords where the optional text qualifier follows the field name rather than appearing before it.
PICK.BREAKPOINT.U	Causes the query processor to handle the U breakpoint option differently. See the <a href="#">BREAK.ON</a> and <a href="#">BREAK.SUP</a> keywords for further details.
PICK.EXPLODE	When using <a href="#">BY.EXP</a> , if an associated field has only one value, do not explode this field.
PICK.IMPLIED.EQ	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.
PICK.GRAND.TOTAL	Causes the query processor to display the text of the <a href="#">GRAND.TOTAL</a> keyword on the same line as the total values.
PICK.ML.CONV.MASK	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.
PICK.NULL	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.
PICK.PROC	D3 compatibility for PQ style <a href="#">Procs</a> . Use of this option implies the PROC.A option. Note that the 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 <a href="#">LIKE</a> operator.
PROC.A	Causes the <a href="#">Proc</a> A( <i>n,m</i> ) command not to terminate copying data at the end of the field.
QUALIFIED.DISPLAY	Causes the query processor to recognise Pick style qualified display clauses.
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 <a href="#">AND</a> operator to take priority over the <a href="#">OR</a> operator in query processor commands. This does not affect the behaviour of these operators in QMBasic programs.
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.
RUN.NO.PAGE	Causes the <a href="#">RUN</a> and <a href="#">DEBUG</a> commands to start the program with screen pagination disabled. This option also affects user catalogued programs. If the <a href="#">PAGINATE.ON.HEADING</a> option is set, pagination will be enabled if the application sets a page heading or page footing unless the <a href="#">NO.PAGE</a> 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.
SPACE.MCT	Modifies the behaviour of the <a href="#">MCT</a> 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 <a href="#">EXECUTE</a> 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 <a href="#">ABORT</a> 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**

These *option.name* values set multiple options. Other options that were already set when these are used will not be cleared. These names cannot be used with **ON**, **OFF** or **DISPLAY**.

**D3**

This option gives closer compatibility with D3. The options included are ASSOC.UNASSOC.MV, BACKSLASH.NOT.QUOTE, CHAINED.SELECT, DEFAULT.MV, 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, SPACE.MCT, UNASS.WARNING and WITH.IMPLIES.OR but may change in future releases.

**MVBASE**

This option gives closer compatibility with mvBase. The options included are AMPM.UPCASE, BACKSLASH.NOT.QUOTE, CHAINED.SELECT, CRDB.UPCASE, DEFAULT.MV, KEEP.OLD.OBJECT, LOCK.BEEP, NO.DATE.WRAPPING, PAGINATE.ON.HEADING, 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, SELECT.KEEP.CASE and SPACE.MCT 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.

## 4.133 PASSWORD

The **PASSWORD** command changes the password for a QM user on Windows 98/ME.

User management is not applicable to the PDA version of QM.

### 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](#)

## 4.134 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.

## 4.135 PDEBUG

The **PDEBUG** command runs the phantom debugger. This command is not available on the PDA version of QM.

### Format

**PDEBUG** {*command*}

where

*command* is the command to be executed by the phantom process.

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 **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.

## 4.136 PDUMP

The **PDUMP** command generates a process dump file for a named 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.

## 4.137 PHANTOM

The **PHANTOM** command starts execution of a verb, sentence or paragraph as a background process. This command is not available on the PDA version of QM.

### Format

**PHANTOM** {**DATA**} {**NO.LOG**} {**USER** *n*} *command*

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 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.

Unless the **NO.LOG** keyword is used, the phantom process will automatically create a COMO file named PH*n*\_date\_time exactly as though it had commenced with the command

**COMO ON PH***n*\_date\_time

A QMBasic program can determine this name using the [!PHLOG\(\)](#) subroutine.

Output that would normally be directed to the display is suppressed except for recording in the COMO file. The COMO file may be switched off or redirected as desired from within the phantom process.

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 **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

---

```
IF @TTY = 'phantom' THEN STOP
```

at the relevant point in the paragraph. See [@TTY](#) for more details.

### Example

```
PHANTOM BASIC BP INVOICE
```

This command starts a phantom process to compile the QMBasic program INVOICE in the BP file.

### See also:

[CHILD\(\)](#), [LIST.PHANTOMS](#), [!PHLOG\(\)](#), [STATUS](#)

## 4.138 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.

<b>AT</b> <i>printer.name</i>	Output is directed to the named printer
<b>BOTTOM.MARGIN</b> <i>n</i>	Sets the bottom margin size.
<b>CLOSE</b>	The print unit is closed.
<b>FILE</b> <i>filename recordname</i>	Selects the destination for printed output.
<b>LEFT.MARGIN</b> <i>n</i>	Sets the left margin size.
<b>LINES</b> <i>n</i>	Sets the number of lines per page.
<b>KEEP.OPEN</b>	Keeps the printer open to merge successive printer output.
<b>QUERY</b>	Reports the current settings
<b>RESET</b>	Resets to the default parameter values.
<b>TOP.MARGIN</b> <i>n</i>	Sets the top margin size.
<b>WIDTH</b> <i>n</i>	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. 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:**

[SETPTR](#)

## 4.139 PSTAT

The **PSTAT** command displays the status of one or all QM processes. This command is not available on the PDA version of QM.

### 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.

The *level* parameter specifies extended report features and is formed by adding together the following components:

- 1 Report each program and subroutine in the call stack. If not included, only the currently active program is reported.
- 2 Report internal subroutine calls within each reported program and subroutine. If not included, only external subroutine calls are reported.
- 4 Shows details of QMNet connections to remote servers.

### Examples

```
PSTAT USER 2 LEVEL 3
User Detail
  2 Account: SALES, Username: Joe, Pid: 1383
    Command: RUN INVOICES
      !SCREEN 953 (14E6)
            750 (118E)
            450 (08F3)
            323 (061B)
      D:\LBS\QM\BP.OUT\INVOICES 105 (01BE)
    Command processor
      D:\LBS\QM\BP.OUT\PROC 104 (05B8)
    Command processor
```

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. 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)
          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)
      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)
```

## 4.140 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 MARK { ON | OFF }  
PTERM NEWLINE { CR | LF | CRLF }  
PTERM PROMPT "string1" { "string2" }  
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 more natural as, for historic reasons, the QM command set is all in upper case. In QMBasic programs, case inversion affects [INPUT](#) statements, the [KEYCODE\(\)](#) and [KEYINC\(\)](#) functions but not the [KEYIN\(\)](#) function.

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 **PTERM MARK** command can be used to enable a mark character translation feature

whereby entry of characters 28 - 30 (ctrl-\, ctrl-], ctrl-^) at the keyboard gets translated to field, value and subvalue 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 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
\^ ^
\\ \
^x Ctrl-x
```

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](#)

## 4.141 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](#)

## 4.142 PUBLISH.ACCOUNT

The **PUBLISH.ACCOUNT** command sets up files in the current account for [replication](#).

### Format

**PUBLISH.ACCOUNT** {**DICT**} *server:account*

**PUBLISH.ACCOUNT CANCEL** *server:account*

where

*server:account* 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 *server:account*. 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:

<b>S</b> <i>n</i>	Sets the publish flag for file <i>n</i> .
<b>S</b> <i>m-n</i>	Sets the publish flag for files <i>m</i> to <i>n</i> .
<b>S ALL</b>	Sets the publish flag for all files.
<b>S VISIBLE</b>	Sets the publish flag for all files on the displayed page.
<b>C</b> <i>n</i>	Clears the publish flag for file <i>n</i> .
<b>C</b> <i>m-n</i>	Clears the publish flag for files <i>m</i> to <i>n</i> .
<b>C ALL</b>	Clears the publish flag for all files.
<b>C VISIBLE</b>	Clears the publish flag for all files on the displayed page.
<b>T</b>	Displays the top page.
<b>N</b>	Displays the next page.
<b>P</b>	Displays the previous page.
<b>A</b>	Applies replication to the selected files.
<b>Q</b>	Quits without publishing.

The **ALL** and **VISIBLE** keywords can be abbreviated and do not need a preceding space.

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 *server:account*. 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.PUBLISING](#), [PUBLISH](#), [SUBSCRIBE](#)

## 4.143 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 or a field name. This element 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.

## 4.144 QUIT

The **QUIT** command terminates the current QM session. The synonym **OFF** can be used.

### Format

#### 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.

When using a telnet connection to a PDA, this command returns to NETWAIT to wait for a new connection.

### See also:

[LOGOUT](#)

## 4.145 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 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](#)

## 4.146 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.

## 4.147 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 files remains open when the program terminates. In this case, locks left in place when the program ends will not be released automatically.

## 4.148 REMOVE.DF

The **REMOVE.DF** command removes a part file from a [distributed file](#).

### Format

**REMOVE.DF** *dist.file* *part.file*

**REMOVE.DF** *dist.file* **ALL**

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 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 **ALL** keyword in place of the part file name deletes the entire distributed file. The part files are not deleted by this operation.

### Example

```
REMOVE.DF ORDERS ORDERS-08
```

This command removes ORDERS-08 as a part file within the ORDERS distributed file.

### See also:

[Distributed files](#), [ADD.DF](#), [LIST.DF](#)

## 4.149 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

<b>REPORT.SRC OFF</b>	To turn off display of <a href="#">@SYSTEM.RETURN.CODE</a>
<b>REPORT.SRC ON</b>	To turn on display of <a href="#">@SYSTEM.RETURN.CODE</a>
<b>REPORT.SRC</b>	To toggle display of <a href="#">@SYSTEM.RETURN.CODE</a>

When reporting is enabled, the value of [@SYSTEM.RETURN.CODE](#) is displayed on return to the command prompt.

## 4.150 REPORT.STYLE

The **REPORT.STYLE** command sets the default query processor report style.

### Format

<b>REPORT.STYLE</b> <i>name</i>	To set the default report style
<b>REPORT.STYLE</b>	To display the current setting
<b>REPORT.STYLE OFF</b>	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 setting using [SETPTR](#), the QMBasic [SETPU](#) statement, or the [STYLE](#) option to the query processor.

See the query processor [STYLE](#) option for full details of report styles.

## 4.151 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](#), [CREATE.FILE](#), [CREATE.KEY](#), [DELETE.KEY](#), [ENCRYPT.FILE](#), [GRANT.KEY](#), [LIST.KEYS](#), [REVOKE.KEY](#), [SET.ENCRYPTION.KEY.NAME](#), [UNLOCK.KEY.VAULT](#)

## 4.152 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:

<b>BINARY</b>	Suppresses translation of field marks to newlines when restoring directory files. Use this option when restoring binary data.
<b>DET.SUP</b>	Suppresses display of the name of each file as it is restored.
<b>NO.CASE</b>	Causes new files to be created with case insensitive record ids. Existing files are not reconfigured.
<b>NO.INDEX</b>	Do not create alternate key indices.
<b>NO.OBJECT</b>	Omits restore of object code. This is particularly useful when migrating to QM from other environments.
<b>POSITIONED</b>	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](#), [SEL.RESTORE](#), [SET.DEVICE](#), [T.ATT](#), [T.DUMP](#), [T.LOAD](#), [T.xxx](#)

## 4.153 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 usernames 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](#), [CREATE.FILE](#), [CREATE.KEY](#), [DELETE.KEY](#), [ENCRYPT.FILE](#), [GRANT.KEY](#), [LIST.KEYS](#), [RESET.MASTER.KEY](#), [SET.ENCRYPTION.KEY.NAME](#), [UNLOCK.KEY.VAULT](#)

## 4.154 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

**RUN** {*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 compiler output file is supplied automatically when using the command to execute a QMBasic program.

*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 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.
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 is 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.

## 4.155 SAVE.LIST

The **SAVE.LIST** command is used to save an active select list for future use.

### Format

```
SAVE.LIST list.name {FROM list.no} {COUNT.SUP}
```

where

*list.name* is the name of the record to be created in \$SAVEDLISTS to hold the saved select list.

*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.

The **SAVE.LIST** command copies an active select list to the \$SAVEDLISTS file. This 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.

[@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](#)

## 4.156 SAVE.STACK

The **SAVE.STACK** command saves the current [command stack](#).

### Format

```
SAVE.STACK {stack.name}
```

where

*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 the `$SAVEDLISTS` file as a record named *stack.name*. The file will be created if it does not already exist. Any existing record of the same name will be overwritten.

### 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](#)

## 4.157 SCRIB

The **SCRIB** command runs the screen builder to create or modify a screen definition for use by the QMBasic [!SCREEN\(\)](#) subroutine.

### Format

**SCRIB** {*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, **SCRIB** 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 **SCRIB**. The name of the account's default screen definition file is stored in field 2 of an X-type VOC record named \$SCRIB.FILE and will be used if no file name is given in the **SCRIB** command line. If this record does not exist, **SCRIB** 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 **SCRIB**, the user is invited to enter a screen name unless this has been provided on the command line. **SCRIB** 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

<b><i>Sn</i></b>	Show / edit step <i>n</i>
<b>X</b>	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

<b><i>Cn</i></b>	Copy step <i>n</i> over the currently displayed step
<b><i>Cscrn,n</i></b>	Copy step <i>n</i> of screen <i>scrn</i> over the currently displayed step
<b>D</b>	Delete this step
<b>I</b>	Insert a new step before this step
<b><i>Mn</i></b>	Move currently displayed step to become step <i>n</i>
<b>N</b>	Advance to the next step
<b>P</b>	Move to the previous step
<b>R</b>	Return to the top level screen
<b><i>Sn</i></b>	Move to step <i>n</i>
<b><i>n</i></b>	Edit step definition starting at item <i>n</i>

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.
- G<sub>n</sub>** 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.
- R<sub>n</sub>** 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).

**Field****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:

<b>Ffmt</b>	Apply <a href="#">FMT0</a> using <i>fmt</i> as the format specifier.
<b>Ifile,rec</b>	Execute the I-type named <i>rec</i> in the dictionary of <i>file</i> . against the data

	record.
<b>Sname</b>	Execute catalogued subroutine <i>name</i> , passing in the field value as argument 2 and replacing it with the valued returned through argument 1.
<b>Tfile, fld, code</b>	Apply <a href="#">TRANSQ</a> using the data item as the record id to access field <i>fld</i> of <i>file</i> . The code determines the action if the record is not found. <b>C</b> returns the record id, <b>X</b> returns a null string.
<b>&lt;f,v,s&gt;</b>	Extract the given field, value or subvalue.
<b>Other</b>	All other codes are passed to <a href="#">OCONVQ</a> .

**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:

<b>Y</b>	The field may not be left blank.
<b>N</b>	The field may be left blank.
<b>F</b>	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.
<b>&lt;f&gt;cond</b>	The field is required if field <i>&lt;n&gt;</i> (or <i>&lt;f,v&gt;</i> or <i>&lt;f,v,s&gt;</i> ) 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:

<i>m</i>	Numeric value <i>m</i> . This is a numeric comparison, leading zeros will be ignored.
<i>m-n</i>	A numeric value in the range <i>m</i> to <i>n</i> .
<b>=x</b>	String equality with <i>x</i> . <i>x</i> may be a null string.
<b>D</b>	A valid date.

<b>F</b> <i>filename</i>	A record named as the input data exists in <i>filename</i> .
<b>F</b> <i>filename,n</i>	Similar to the simple <b>F</b> validation but, if the record is found, the input data is replaced by the contents of field <i>n</i> of the record.
<b>M</b> <i>template</i>	The input data matches the specified <i>template</i> .
<b>R</b> <i>file,rec,fld,case,subst</i>	Record <i>rec</i> is read from <i>file</i> . Field <i>fld</i> of this record is scanned for a match with the input data. If <i>case</i> is <b>X</b> , this scan is case insensitive. If <i>subst</i> is specified, the input data is replaced with the content of the corresponding value of field <i>subst</i> of the record. Field <i>fld</i> may be broken down into subvalues to specify alternative strings all of which are replaced by the value (not subvalue) in <i>subst</i> .
<b>@</b> <i>subroutine</i>	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.
<b>X</b> <i>xxx</i>	Inverts the condition <i>xxx</i> . 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:

<b>F</b> <i>fmt</i>	Apply <a href="#">FMT()</a> using <i>fmt</i> as the format specifier.
<b>N</b> <i>n</i>	If the data is numeric, extend it to be right justified in <i>n</i> digits.
<b>S</b> <i>name</i>	Execute catalogued subroutine <i>name</i> , passing in the field value as argument 2 and replacing it with the valued returned through argument 1.
<b>T</b> <i>file,fld,code</i>	Apply <a href="#">TRANS()</a> using the data item as the record id to access field <i>fld</i> of <i>file</i> . The code determines the action if the record is not found. <b>C</b> returns the record id, <b>X</b> returns a null string.
<b>&lt;</b> <i>f,v,s</i> <b>&gt;</b>	Extract the given field, value or subvalue.
Other	All other codes are passed to <a href="#">ICONV()</a> .

**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 backtab key is allowed and may contain **Y**, **N** or blank which is taken as **N**. If backtab 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**<f,v,s>. 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 **SCR**.

**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.

<b>F1</b>	Help
<b>F2</b>	Pick list help
<b>F3</b>	Delete the contents of the current field
<b>F4</b>	Restore the contents of the current field after incorrect entry
<b>Return</b>	Execute the next step action
<b>Tab</b>	Treated identically to the return key
<b>Ctrl-P</b>	Execute the back step action
<b>Exit</b>	Execute the exit key action (Ctrl-X)
<b>Home</b>	Mode to start of field (Ctrl-A)
<b>End</b>	Move to end of field (Ctrl-E)
<b>Cursor left</b>	Move left one character (Ctrl-B)
<b>Cursor right</b>	Move right one character (Ctrl-F)
<b>Delete</b>	Delete character under the cursor (Ctrl-D)

---

<b>Ctrl-K</b>	Delete all to right of the cursor
<b>Backspace</b>	Delete character before cursor
<b>Insert</b>	Toggle overlay mode
<b>Page up</b>	Scroll up in pick list display (Esc-V)
<b>Page down</b>	Scroll down in pick list display (Ctrl-V)

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.

## 4.158 SECURITY

The **SECURITY** command enables, disables or reports the state of QM's internal security system.

User management is not applicable to the PDA version of QM.

### Format

<b>SECURITY ON</b>	Enable security
<b>SECURITY OFF</b>	Disable security
<b>SECURITY</b>	Report the current security setting

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](#)

## 4.159 SED

The **SED** command is a screen based editor, particularly useful for editing QMBasic source programs where many lines (fields) can be seen at once.

### Format

```
SED {{DICT} file.name {record.id ...}}
```

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*.

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 restricted 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.

### 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

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.

	Ctrl-	Esc-	Ctrl-X -	Ctrl-X Ctrl-
<b>A</b>	Start line			
<b>B</b>	Back char	Back word	Goto buffer	*List buffers
<b>C</b>	Repeat	Capital init	*Quit	*Quit
<b>D</b>	Delete char	Delete word	*List records	Dive
<b>E</b>	End line	Run extension	*Execute macro	
<b>F</b>	Forward char	Forward word		*Find record
<b>G</b>	Cancel	Goto line		
<b>H</b>	Backspace			
<b>I</b>	Tab	Align text	Import	
<b>J</b>	Newline			
<b>K</b>	Kill line		Delete buffer	
<b>L</b>	Refresh	Lowercase		Lowercase region
<b>M</b>	Newline	Compile	Compile and run	
<b>N</b>	Down line	Next buffer		Nudge down
<b>O</b>	Overlay		Toggle window	
<b>P</b>	Up line	Previous buffer		Nudge up
<b>Q</b>	Quote char	Quote char	Query replace	
<b>R</b>	Reverse search	Reverse search	Replace	
<b>S</b>	Forward search	Forward search	Save record	Save record
<b>T</b>	Toggle chars			
<b>U</b>	Repeat	Uppercase	Up to parent	Uppercase region
<b>V</b>	Forward screen	Back screen	View	
<b>W</b>	Delete region	Copy region	*Write record	*Write record

<b>X</b>	Ctrl-X prefix	*Command	*Export	Swap mark
<b>Y</b>	Insert kill buffer	Insert kill buffer		
<b>Z</b>	Up line			Nudge up
<b>1</b>			Unsplit window	
<b>2</b>			Split window	
<b>(</b>			*Start macro	
<b>)</b>			*End macro	
<b>=</b>			*Expand char	
<b>.</b>		Set mark		
<b>&lt;</b>		Top		
<b>&gt;</b>		Bottom		
<b>Bkspc</b>	Backspace	Back del word		
<b>Del</b>	Delete char			
<b>Space</b>		Close spaces		

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.



**Cut and Paste**

Set mark	Esc-.	Marks current cursor position
Copy region	Esc-W	Saves a copy of the text between the cursor and the mark
Delete region	Ctrl-W	Deletes the text between the cursor and the mark, saving it
Paste	Ctrl-Y	Inserts text saved by Esc-W, Ctrl-W or Ctrl-K

## SED - Cursor movement and search functions

**Start line** (Ctrl-A)

Moves the cursor to the start of the current line (column 1).

**End line** (Ctrl-E)

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)

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.

**Up line** (Ctrl-P or Ctrl-Z)

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.

**Forward char** (Ctrl-F)

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)

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)

Moves the cursor down by one screen or to the end of the buffer.

**Backscreen** (Esc-V)

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:

- |                          |  |
|--------------------------|--|
| <b>Case sensitive:</b>   | Data will only be found if it matches the search string exactly. This is the default search mode.  |
| <b>Case insensitive:</b> | The case of both the search string and the data is ignored.  |
| <b>Word:</b>             | 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. |
| <b>Basic word:</b>       | 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 be 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 and 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.

Followed by V, S or T, it inserts a value mark, subvalue mark or text mark respectively.

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)

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 (Backspace or Ctrl-H)

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.

### Capitalinit (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.

### 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:

<b>BASIC</b>	Saves the current record and runs the Basic compiler. This command is similar to <b>COMPILE</b> (described below) but it inserts marker lines into the source program where error or warning messages have been produced by the compiler.
<b>BWORD</b>	Sets Basic Word mode for <b>search</b> and <b>replace</b> functions as described for the <b>forward search</b> function.
<b>CASE OFF</b>	Sets Case Insensitive mode for <b>search</b> and <b>replace</b> functions.
<b>CASE ON</b>	Sets Case Sensitive mode for <b>search</b> and <b>replace</b> functions.
<b>COMPILE</b>	Saves the current record and runs the Basic compiler. For compatibility with SED on other platforms, if this record includes a line <i>*\$CATALOG catname</i> the compiled program will be catalogued automatically after successful compilation. The actual cataloguing command issued is <i>CATALOG filename catname recordname</i> It is thus possible to perform either private or global cataloguing. The same action can be performed on QM using the <a href="#">\$CATALOG</a> compiler directive.
<b>EXPAND.TABS</b>	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.
<b>FORMAT</b>	Applies standard format rules to the layout of a QMBasic program.
<b>FUNDAMENTAL</b>	Reverts to the default (fundamental mode) key bindings.
<b>INDENT</b>	Toggles indent mode.
<b>KEYS</b>	Displays the name of the active key binding record.
<b>LNUM</b>	The <b>LNUM</b> 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: <b>OFF</b> Turn off line numbering in the current buffer. <b>ON</b> Turn on line numbering in the current buffer. <b>ALL</b> Turn on line numbering in all buffers. <b>OFF ALL</b> Turn off line numbering in all buffers. <b>ON ALL</b> Synonym for <b>ALL</b> .
<b>LOAD.KEYS</b>	Loads a named key binding record.
<b>OVERLAY</b>	Toggles overlay mode. The <b>OVERLAY</b> function (Ctrl-O) has the same effect.
<b>QUIT</b>	Ends editing of the current record in a similar way to the <b>quit</b> key sequence but also aborts any select list.

---

<b>RELEASE</b>	Releases the update lock on the current record.
<b>RUN</b>	Entering <b>RUN</b> with no qualifying details runs the program in the current buffer (which must have been compiled). If <b>RUN</b> is followed by any qualifying details, the command is passed to the command processor for execution.
<b>SAVE.KEYS</b>	Saves the key bindings as described later.
<b>SPOOL</b>	Spools the contents of the current buffer to print unit zero.  The <b>SPOOL</b> command has optional qualifiers which may be used together if required. <b>LNUM</b> adds a line number prefix to each line printed. <b>REGION</b> prints only the lines between the mark and the cursor (which may be in either order). <b>AT</b> followed by a printer name selects the destination printer.
<b>TABS</b>	Sets the tab interval to be used by the <b>tab</b> function and the <b>EXPAND.TABS</b> command. The value given after <b>TABS</b> must be in the range 1 to 99.
<b>TRIM</b>	Removes trailing spaces from all lines of the current buffer.
<b>WORD</b>	Sets Word mode for <b>search</b> and <b>replace</b> commands.
<b>XEQ</b> { <i>cmd</i> }	Executes QM command <i>cmd</i> . The <b>XEQ</b> command prefix is only required where <i>cmd</i> is also an internal <b>SED</b> command. All commands not recognised by <b>SED</b> are passed to the command processor for execution.

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:

<b>BWORD</b>	Turn on Basic word search mode.
<b>CASE OFF</b>	Turn off search case sensitivity.
<b>CASE ON</b>	Turn on search case sensitivity.
<b>INDENT</b>	Turn on indentation mode.
<b>LNUM</b>	Line numbering is to be on in all windows by default.
<b>OVERLAY</b>	Turn on overlay mode.
<b>TABS <i>n</i></b>	Sets the default tab interval to <i>n</i> columns.
<b>WORD</b>	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

<i>dict.flag</i>	is "DICT" if attempting to write to a dictionary, a null string otherwise.
<i>file.name</i>	is the name of the file to be written.
<i>record.name</i>	is the name of the record to be written.
<i>rec</i>	is the record data.
<i>caller</i>	is 1 to indicate a call from SED, 2 for a call from ED.
<i>write.allowed</i>	should be returned by the subroutine as true (1) if the write may be performed, false (0) if not. This argument is 1 on calling the subroutine.
<i>updated</i>	should be set true if the subroutine has made any changes to the data in <i>rec</i> . 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-termttype* 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 *termttype* 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 \$EXTENSIONS 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:

```
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:

<code>.newline</code>	<code>.start.line</code>	<code>.end.line</code>	<code>.back.char</code>
<code>.fwd.char</code>	<code>.up.line</code>	<code>.down.line</code>	<code>.top</code>
<code>.bottom</code>	<code>.page.up</code>	<code>.page.down</code>	<code>.del.char</code>
<code>.backspace</code>	<code>.kill.line</code>	<code>.save.record</code>	<code>.quit</code>
<code>.overlay</code>	<code>.tab</code>	<code>.goto.line</code>	<code>.toggle.chars</code>
<code>.fwd.search</code>	<code>.replace</code>	<code>.query.replace</code>	<code>.swap.mark</code>
<code>.execute.macro</code>	<code>.nudge.down</code>	<code>.nudge.up</code>	<code>.set.mark</code>
<code>.delete.region</code>	<code>.copy.region</code>	<code>.insert.killed</code>	<code>.forward.word</code>
<code>.delete.word</code>	<code>.import</code>	<code>.reverse.srch</code>	<code>.lowercase</code>
<code>.uppercase</code>	<code>.capital.init</code>	<code>.back.word</code>	<code>.del.back.word</code>
<code>.close.spaces</code>	<code>.next.buffer</code>	<code>.prev.buffer</code>	<code>.goto.buffer</code>
<code>.delete.buffer</code>	<code>.up.to.list</code>	<code>.repeat</code>	<code>.refresh</code>
<code>.quote.char</code>	<code>.list.buffers</code>	<code>.find.record</code>	<code>.write.record</code>
<code>.start.macro</code>	<code>.end.macro</code>	<code>.expand.char</code>	<code>.list.records</code>
<code>.export</code>	<code>.command</code>	<code>.cancel</code>	<code>.run</code>

`.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	@USERNO	

### 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

<code>%file</code>	Returns file name for current buffer. This is prefixed by "DICT " if the buffer is from the dictionary part of the file.
<code>%id</code>	Returns the record id for the current buffer. For an explore buffer or a file list buffer it returns a pseudo id value.
<code>%buffer.type</code>	Returns a type code for the current buffer: 1 : data buffer 2 : explore buffer 3 : file list buffer
<code>%read.only</code>	Returns a logical value indicating whether the current buffer is read only.
<code>(set.read.only <i>n</i>)</code>	Sets the read only status of the current buffer. If <i>n</i> 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.

%changed	Returns a logical value indicating whether the current buffer is marked as having been changed since it was last saved.
(set.changed <i>n</i> )	Sets the status of the buffer changed flag. If <i>n</i> 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.
%overlay	Returns a logical value indicating whether the current buffer is operating in overlay mode.
(set.overlay <i>n</i> )	Sets the overlay status of the current buffer. If <i>n</i> is zero, overlay is turned off, otherwise it is turned on.
%buffer.no	Returns the current buffer number.
(prev.buffer <i>n</i> )	Select the buffer with buffer number <i>n</i> less than the current one. This operation cycles to the highest numbered buffer if necessary.
(next.buffer <i>n</i> )	Select the buffer with buffer number <i>n</i> greater than the current one. This operation cycles to the lowest numbered buffer if necessary.
(goto.buffer <i>n</i> )	Select buffer number <i>n</i> .
(delete.buffer)	Deletes the current buffer. Unlike the SED <b>delete.buffer</b> keyboard function, this operation allows deletion of modified buffers without any confirmation checks.
%current.line	Returns text of current line.
%current.char	Returns character under cursor.
%line.len	Returns length of current line.
%line	Returns current line number.
%col	Returns current column number.
%lines	Returns number of lines in current record.
%mark.line	Returns the line number of the mark position, zero if no mark set.
%mark.col	Returns the column number of the mark position, zero if no mark set.
%scroll	Returns current scroll position. This is the line number of the data displayed on the first line of the screen.
(set.scroll <i>n</i> )	Sets the current scroll position to <i>n</i> . The value of <i>n</i> must be greater than zero and must not be greater than the number of lines in the current buffer. The next screen update ( <b>paint</b> or implicit from some other function) will move the scroll position if the current line is not in the displayed region of the buffer.
%pan	Returns the current pan position. This is the column number (from one) of the leftmost displayed column of the current buffer.
(set.pan <i>n</i> )	Set the current pan position to <i>n</i> . The value of <i>n</i> must be greater than zero. The next screen update ( <b>paint</b> 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.
%tab.interval	Returns the current setting of the tab interval.
(set.tab.interval <i>n</i> )	Sets the tab interval to <i>n</i> . The value of <i>n</i> must be in the range 1 to 99.
%width	Returns the width of the screen display.
%height	Returns the number of lines on the screen display excluding the two editor status lines.
%kill.buffer	Returns the content of the kill buffer.

(make.buffer <i>name</i> )	Makes a new scratch buffer named <i>name</i> . 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 <i>name</i> . This function returns true if successful, false if not. If a scratch buffer of the given <i>name</i> already exists, it makes that buffer current and returns true.
(find.buffer file <i>rec</i> )	Returns the internal number of the buffer holding data from the given <i>file</i> and <i>record</i> . Use of a null string as <i>file</i> allows location of a scratch buffer. This function returns zero if no such buffer exists.
(find.record file <i>rec</i> )	Reads record <i>rec</i> from <i>file</i> 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.
(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.
(write.record file <i>rec</i> )	Writes the current buffer to record <i>rec</i> in <i>file</i> . 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 <b>write.record</b> operation is ignored if the target record is locked by another user.

### Moving the Cursor, Insertion and Deletion

(top)	Move to the start of line 1.
(bottom)	Move to after the final line of the buffer.
(start.line)	Move to the start of the current line.
(home)	Synonym for <b>start.line</b> .
(end.line)	Move to the end of the current line.
(end)	Synonym for <b>end.line</b> .
(back.char <i>n</i> )	Move cursor left <i>n</i> columns.
(left <i>n</i> )	Synonym for <b>back.char</b> .
(fwd.char <i>n</i> )	Move cursor right <i>n</i> columns.
(right <i>n</i> )	Synonym for <b>fwd.char</b> .
(up.line <i>n</i> )	Move cursor up <i>n</i> lines.
(up <i>n</i> )	Synonym for <b>up.line</b> .
(down.line <i>n</i> )	Move cursor down <i>n</i> lines.
(down <i>n</i> )	Synonym for <b>down.line</b> .
(page.up <i>n</i> )	Move cursor up by <i>n</i> display pages.
(page.down <i>n</i> )	Move cursor down by <i>n</i> display pages.
(fsearch <i>mode str</i> )	Search forwards for string <i>str</i> . The <i>mode</i> 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.
(rsearch <i>mode str</i> )	Search backwards for string <i>str</i> . The <i>mode</i> 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.

(del.char <i>n</i> )	Delete <i>n</i> characters.
(backspace <i>n</i> )	Backspace <i>n</i> characters.
(insert <i>s</i> )	Insert string <i>s</i> at current cursor position. Any field marks in the text are treated as newlines. The <b>insert</b> function can take more than one argument, each of which is inserted in turn.
(newline <i>rpt</i> )	Insert <i>rpt</i> newlines at the current cursor position.
(tab <i>rpt</i> )	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 <i>rpt</i> argument specifies how by many tab positions the cursor is to be advanced.
(goto.line <i>a</i> )	Go to column 1 of line <i>a</i> .
(goto.col <i>a</i> )	Go to column <i>a</i> of current line. The line is extended if necessary by adding trailing spaces.
(set.mark)	Set the mark at the current cursor position.
(swap.mark)	Interchange the cursor and mark positions. Has no effect if there is no mark position defined.
(retype <i>s</i> )	Replace the current line by string <i>s</i> .
(set.case <i>mode rpt</i> )	Changes the case of the next <i>rpt</i> words in the current buffer. The <i>mode</i> argument is: <ul style="list-style-type: none"> <li>0 : Set lower case</li> <li>1 : Set upper case</li> <li>2 : Set capital initial casing</li> </ul>
(toggle.chars)	Interchanges the character at the cursor position with that in the preceding column.
(fwd.word <i>n</i> )	Moves the cursor forward by <i>n</i> words.
(back.word <i>n</i> )	Moves the cursor backward by <i>n</i> words.
(del.word <i>n</i> )	Deletes <i>n</i> words from the current cursor position.
(del.back.word <i>n</i> )	Deletes <i>n</i> words backwards from the current cursor position.
(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.
(close.spaces)	Closes multiple spaces around the current cursor position to be just one space.

### Arithmetic, String and Logical Functions

(add <i>a b</i> )	Returns $a + b$ .
(sub <i>a b</i> )	Returns $a - b$ .
(mul <i>a b</i> )	Returns $a * b$ .
(div <i>a b</i> )	Returns $a / b$ .
(rem <i>a b</i> )	Returns the remainder of dividing <i>a</i> by <i>b</i> .

(int <i>a</i> )	Returns the integer portion of value <i>a</i> by truncating any fractional part.
(eq <i>a b</i> )	Test <i>a</i> equal to <i>b</i> .
(ne <i>a b</i> )	Test <i>a</i> not equal to <i>b</i> .
(lt <i>a b</i> )	Test <i>a</i> less than <i>b</i> .
(gt <i>a b</i> )	Test <i>a</i> greater than <i>b</i> .
(le <i>a b</i> )	Test <i>a</i> less than or equal to <i>b</i> .
(ge <i>a b</i> )	Test <i>a</i> greater than or equal to <i>b</i> .
(and <i>a b</i> )	Forms logical relationship <i>a</i> and <i>b</i> . The <b>and</b> function can take more than two arguments, each of which is and'ed in turn.
(or <i>a b</i> )	Forms logical relationship <i>a</i> or <i>b</i> . The <b>or</b> function can take more than two arguments, each of which is or'ed in turn.
(not <i>a</i> )	Returns logical inverse of <i>a</i> .
(max <i>a b</i> )	Returns the maximum of <i>a</i> and <i>b</i> . If either value is not numeric, this function returns the item that appears last in collating sequence order.
(min <i>a b</i> )	Returns the minimum of <i>a</i> and <i>b</i> . If either value is not numeric, this function returns the item that appears first in collating sequence order.
(len <i>s</i> )	Returns the length of string <i>s</i> .
(char <i>n</i> )	Returns the character with ASCII character value <i>n</i> .
(seq <i>c</i> )	Returns the ASCII character sequence number of the first character of <i>c</i> .
(cat <i>a b</i> )	Returns concatenation of strings <i>a</i> and <i>b</i> . The <b>cat</b> function can take more than two arguments, each of which is concatenated in turn.
(substr <i>s a b</i> )	Returns a substring from <i>s</i> starting at character <i>a</i> , <i>b</i> characters long.
(pad <i>s n</i> )	Pad string <i>s</i> with spaces to be <i>n</i> characters long. If string <i>s</i> is already at least <i>n</i> characters long, this function returns the original string.
(trim <i>s</i> )	Returns string <i>s</i> with all leading and trailing spaces removed and all multiple embedded spaces replaced by a single space.
(trimb <i>s</i> )	Returns string <i>s</i> with all trailing spaces removed.
(trimf <i>s</i> )	Returns string <i>s</i> with all leading spaces removed.
(upcase <i>s</i> )	Returns string <i>s</i> converted to upper case.
(downcase <i>s</i> )	Returns string <i>s</i> converted to lower case.
(extract <i>str f v s</i> )	Returns field <i>f</i> , value <i>v</i> , subvalue <i>s</i> of string <i>str</i> . Specify <i>v</i> and <i>s</i> as zero to extract field <i>f</i> , <i>s</i> as zero to extract field <i>f</i> , value <i>v</i> .
(rep <i>str f v s new</i> )	Returns a string formed from <i>str</i> with field <i>f</i> , value <i>v</i> , subvalue <i>s</i> replaced by <i>new</i> . Specify <i>v</i> and <i>s</i> as zero to replace field <i>f</i> , <i>s</i> as zero to replace field <i>f</i> , value <i>v</i> .
(ins <i>str f v s new</i> )	Returns a string formed from <i>str</i> with <i>new</i> inserted before field <i>f</i> , value <i>v</i> , subvalue <i>s</i> . Specify <i>v</i> and <i>s</i> as zero to insert before field <i>f</i> , <i>s</i> as zero to insert before field <i>f</i> , value <i>v</i> .
(del <i>str f v s</i> )	Returns a string formed from <i>str</i> with field <i>f</i> , value <i>v</i> , subvalue <i>s</i> deleted. Specify <i>v</i> and <i>s</i> as zero to delete field <i>f</i> , <i>s</i> as zero to delete field <i>f</i> , value <i>v</i> .
(field <i>str d n count</i> )	Returns a portion of <i>str</i> starting at the <i>n</i> 'th substring delimited by <i>d</i> and extending for <i>count</i> such substrings.

### User Input and Screen Display

(prompt <i>prmt dflt</i> )	Displays <i>prmt</i> prompt text on the upper status line and invites input. <i>Dflt</i> is the default input if the return key is pressed without entering any response.
(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 <b>paint</b> if the cursor should be refreshed at the current position.
%key.ready	Returns a logical value indicating whether there is data waiting from a user key depression.
(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 <b>key tokens</b> above. If the key is a data character, the key type code is <b>.insert</b> and the associated data character can be retrieved using %key.char. See also %prefix.count and %prefix.set.
%key.char	Returns the character from the last <b>get.key</b> if it was an insert action. For other actions, it returns a null string.
%prefix.count	Returns the prefix count for the last use of get.key. Returns 1 if no prefix count was entered. On Initial entry to a procedure started from the keyboard, this function returns any prefix count associated with the keyboard action.
%prefix.set	Returns a logical value indicating whether a prefix count was entered for the last use of get.key.
(status.msg <i>str</i> )	Displays message <i>str</i> on the upper status line. The displayed text may be cleared by using the <b>status.msg</b> operation with a null <i>str</i> .
(paint <i>mode</i> )	Refreshes the screen display and positions the cursor. The <i>mode</i> 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.
(beep)	Sounds the terminal bell.
(wait.input)	Wait for the user to press a key. The actual key pressed remains available for subsequent processing.

### Miscellaneous File Handling

(exists <i>file rec</i> )	Returns a logical value indicating whether record <i>rec</i> exists in <i>file</i> .
(read <i>file rec</i> )	Returns record <i>rec</i> from <i>file</i> . If the record cannot be read, this function returns a null string.
(write <i>file rec str</i> )	Writes <i>str</i> to record <i>rec</i> of <i>file</i> . If the file cannot be opened, this function has no effect
(delete <i>rec</i> )	Deletes <i>rec</i> from <i>file</i> . If the file cannot be opened, this function has no effect

### Conditional Execution and Loops

(if <i>a proc1</i> else <i>proc2</i> )	Executes procedure <i>proc1</i> if logical item <i>a</i> is true, <i>proc2</i> if it is false. The <b>else</b> component is optional. Both <i>proc1</i> and <i>proc2</i> may be one or more procedures.
--	---

(switch <i>val</i> case <i>a proc1</i> case <i>b proc2</i> ... else <i>proc</i> )	Executes one of several procedures depending on the value of <i>val</i> . Items <i>a</i> , <i>b</i> and <i>c</i> (etc.) are values which are compared with <i>val</i> . The <b>else</b> component is optional and is executed only if none of the preceding conditions is met.
(loop <i>proc</i> )	Executes <i>proc</i> repeatedly. <i>proc</i> may be one or more procedures. A loop is terminated by use of <b>exit</b> .
(exit)	Exits from the innermost active <b>loop</b> .
(return)	Returns from the current PROC.
(return <i>n</i> )	Returns <i>n</i> as the value of the current FUNC.
(stop)	Terminates extension program and returns to SED edit mode.
(quit <i>n</i> )	Terminates current edit. If <i>n</i> is zero, SED continues processing a select list (equivalent to the <b>quit</b> editor function). For non-zero values of <i>n</i> , SED terminates the entire sequence (equivalent to the QUIT command).

### Setting Variables

(set <i>var val</i> )	Set local or global variable <i>var</i> to <i>val</i> .
-----------------------	---

### Extension Control

(unload)	Unloads all extensions on exit from outermost extension program.
(bind.command <i>ext name</i> )	Binds extension procedure <i>ext</i> as command <i>name</i> . The extension name is automatically converted to upper case.
(bind.key <i>ext keyseq</i> )	Binds extension procedure <i>ext</i> as key sequence <i>keyseq</i> . The extension name is automatically converted to upper case. This function returns a logical value indicating whether it was successful.
%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.
(xeq <i>cmd</i> )	Executes <i>cmd</i> as though it were entered using the editor <b>command</b> 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.
%macro.state	Returns state of editor macro system: 0 : Not collecting or executing 1 : Collecting macro 2 : Executing macro

**Basic Functions** (Operations that mimic QMBasic programming functions)

(alpha <i>str</i> )	Equivalent to ALPHA( <i>str</i> ) Returns a logical value indicating whether <i>str</i> is an entirely alphabetic string.
(convert <i>old new str</i> )	Equivalent to CONVERT( <i>old, new, str</i> ) For each character in <i>old</i> , this function replaces all occurrences of that character in <i>str</i> by the character in the corresponding position in <i>new</i> . If <i>new</i> is shorter than <i>old</i> , characters in <i>old</i> for which there is no replacement in <i>new</i> are deleted from the returned version of <i>str</i> .
(count <i>str substr</i> )	Equivalent to COUNT( <i>str, substr</i> ) Returns a count of the number of occurrences of <i>substr</i> in <i>str</i> .
(dcount <i>str delim</i> )	Equivalent to DCOUNT( <i>str, delim</i> ) Returns a count of the number of substrings in <i>str</i> delimited by <i>delim</i> . The delimiter must be a single character.
(matches <i>str pattern</i> )	Equivalent to <i>str</i> MATCHES <i>pattern</i> Tests whether <i>str</i> matches the given <i>pattern</i> .
(matchfield <i>str pattern n</i> )	Equivalent to MATCHFIELD( <i>str, pattern, n</i> ) Returns the <i>n</i> 'th component of <i>str</i> when matched against <i>pattern</i> .
(iconv <i>str conv</i> )	Equivalent to ICONV( <i>str, conv</i> ) Returns the result of applying input conversion <i>conv</i> to <i>str</i> .
(index <i>str substr occ</i> )	Equivalent to INDEX( <i>str, substr, occ</i> ) Returns the character position (from one) of the <i>occ</i> 'th occurrence of <i>substr</i> in <i>str</i> . This function returns zero if the specified occurrence is not found.
(num <i>str</i> )	Equivalent to NUM( <i>str</i> ) Returns a logical value indicating whether <i>str</i> can be interpreted as a number.
(oconv <i>str conv</i> )	Equivalent to OCONV( <i>str, conv</i> ) Returns the result of applying output conversion <i>conv</i> to <i>str</i> .

**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.

<b>Editor Function</b>	<b>Equivalent Extension Coding</b>
export	(write <i>file rec</i> %kill.buffer)
import	(insert (read <i>file rec</i> ))
insert killed	(insert %kill.buffer)

**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...
  (
...operations...
  )

```

The *arg1*, *arg2*, etc. items are local variables into which the argument values are to be transferred when the procedure is called.

### 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 once more.

```

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))
        )

        (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 %prefix.count)
      (loop
of block      (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))         * 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
of block      (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 (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

```



```

    )
    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))        * Righthmost 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))
)
(goto.line top)
(goto.col left)
)

```

### 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...
)

```

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.

## 4.160 SEL.RESTORE

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:

<b>BINARY</b>	Suppresses translation of field marks to newlines when restoring directory files. Use this option when restoring binary data.
<b>DET.SUP</b>	Suppresses display of the name of each file as it is restored.
<b>NO.CASE</b>	Causes new files to be created with case insensitive record ids. Existing files are not reconfigured.
<b>NO.INDEX</b>	Do not create alternate key indices.
<b>NO.OBJECT</b>	Omits restore of object code. This is particularly useful when migrating to QM from other environments.
<b>POSITIONED</b>	Assumes that the tape is already positioned at the start of the data to be restored.
<b>NO.QUERY</b>	suppresses the confirmation prompt when restoring using a select list. The NO.SEL.LIST.QUERY mode of the <a href="#">OPTION</a> 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:

[ACCOUNT.RESTORE](#), [ACCOUNT.SAVE](#), [FILE.SAVE](#), [FIND.ACCOUNT](#), [RESTORE.ACCOUNTS](#), [SET.DEVICE](#), [T.ATT](#), [T.DUMP](#), [T.LOAD](#), [T.xxx](#)

## 4.161 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.

## 4.162 SET

The **SET** command sets a value into an @-variable.

### Format

**SET** *variable value*

**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

```
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.

## 4.163 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 <a href="#">ACCOUNT.SAVE</a> 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](#)

## 4.164 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](#), [CREATE.FILE](#), [CREATE.KEY](#), [DELETE.KEY](#), [GRANT.KEY](#), [LIST.KEYS](#), [RESET.MASTER.KEY](#), [REVOKE.KEY](#)

## 4.165 SET.EXIT.STATUS

The **SET.EXIT.STATUS** command sets the final exit status returned by QM to the operating system. This command is not available on the PDA version of QM.

### 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.

## 4.166 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 *server: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.

## 4.167 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*} }

where

<i>queue</i>	is the form queue number in the range 0 to 999.
<i>width</i>	is the page width in characters, excluding any left margin.
<i>depth</i>	is the total page length in lines, including the top and bottom margins. A value of zero implies no pagination of the output data.
<i>top.margin</i>	is the number of lines to be left blank at the top of the page.
<i>bottom.margin</i>	is the number of lines to be left blank at the bottom of the page.
<i>mode</i>	is the print unit mode.
<i>options</i>	qualify the output destination. 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 \$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.

**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](#), [SP.ASSIGN](#)

## 4.168 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 username 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.

*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.

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.

### 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 username (root) has been included in the command but, because the password has been omitted, the command will prompt for this.

### See also:

[QMNet](#), [ADMIN.SERVER](#), [DELETE.SERVER](#), [LIST.SERVERS](#)

## 4.169 SET.TRIGGER

The **SET.TRIGGER** command sets, removes or displays the [trigger function](#) associated with a file.

### Format

<b>SET.TRIGGER</b> <i>file.name</i> <i>function.name</i> { <i>modes</i> }	Set trigger function
<b>SET.TRIGGER</b> <i>file.name</i> ""	Remove trigger function
<b>SET.TRIGGER</b> <i>file.name</i>	Report trigger function

where

*file.name* is the file to be processed.

*function.name* is the name of the catalogued trigger function.

*modes* is any combination of the following tokens indicating when the trigger will be executed.

<b>PRE.WRITE</b>	Before a write operation
<b>PRE.DELETE</b>	Before a delete operation
<b>PRE.CLEAR</b>	Before a clear file operation
<b>POST.WRITE</b>	After a write operation
<b>POST.DELETE</b>	After a delete operation
<b>POST.CLEAR</b>	After a clear file operation
<b>READ</b>	After a read operation

If no *modes* are specified, the default is **PRE.WRITE** and **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 dynamic 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.

## 4.170 SETPORT

The **SETPORT** command sets communications parameters of a serial port. This command is not available on the PDA version of QM.

### 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

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

```
SET.PORT COM1 BAUD 9600 BITS 7 PARITY ODD STOP.BITS 1
```

## 4.171 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

<i>unit</i>	is the print unit number in the range 0 to 255 or the keyword <b>DEFAULT</b> .										
<i>width</i>	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.										
<i>depth</i>	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.										
<i>top.margin</i>	is the number of lines to be left blank at the top of the page.										
<i>bottom.margin</i>	is the number of lines to be left blank at the bottom of the page.										
<i>mode</i>	is the print unit mode: <table> <tr> <td>1</td> <td>Output is sent to a printer.</td> </tr> <tr> <td>3</td> <td>Output is directed to a hold file.</td> </tr> <tr> <td>4</td> <td>Output is directed to stderr (standard error).</td> </tr> <tr> <td>5</td> <td>Output is directed to the terminal auxiliary port.</td> </tr> <tr> <td>6</td> <td>Output is written to a file and also printed.</td> </tr> </table>	1	Output is sent to a printer.	3	Output is directed to a hold file.	4	Output is directed to stderr (standard error).	5	Output is directed to the terminal auxiliary port.	6	Output is written to a file and also printed.
1	Output is sent to a printer.										
3	Output is directed to a hold file.										
4	Output is directed to stderr (standard error).										
5	Output is directed to the terminal auxiliary port.										
6	Output is written to a file and also printed.										
<i>options</i>	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 { 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 *Punit* is used.
- The optional **NEXT** keyword causes QM to attach 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.
- AS PATHNAME *path*** Specifies a destination pathname 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.
- 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.
- 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.
- INFORM** Displays file details when opening a print file in mode 3. 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 to terminate the print job.

<b>LANDSCAPE</b>	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 <code>OPTIONS "landscape"</code> .
<b>LEFT.MARGIN</b> <i>n</i>	Inserts a margin of <i>n</i> spaces to the left of the printed data.
<b>NEWLINE</b> <i>mode</i>	Determines the newline sequence used by the QMBasic <a href="#">PRINT</a> statement. This may be CR, LF or CRLF. The default is LF.
<b>NFMT</b>	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.
<b>NODEFAULT</b>	Omitted options normally take their default values. Use of this keyword leaves the option at its current value.
<b>NOEJECT</b>	Suppresses the normal page throw at the end of a print job. This option applies to Windows only.
<b>OPTIONS</b> <i>xxx</i>	Passes the given option(s) to the underlying operating system print spooler (e.g. <code>OPTIONS "landscape"</code> on non-Windows systems).
<b>OVERLAY</b> <i>subr</i>	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.
<b>PCL</b>	Specifies that this printer supports PCL. This option cannot be used with GDI mode Windows printers.
<b>PORTRAIT</b>	Where supported by the underlying print driver, this keyword specifies that the output is to be printed in portrait format.
<b>PREFIX</b> <i>path</i>	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 <code>SETPTR</code> options. This option is ignored for Windows GDI mode printing.
<b>RAW</b>	Specifies that the non-GDI mode API calls are to be used to initiate printing.
<b>SPOOLER</b> <i>name</i>	Specifies an alternative spooler to be used on non-Windows systems. If not specified, the spooler selected by the <a href="#">SPOOLER</a> configuration parameter is used or, if this is not set, the standard <code>lp</code> spooler is used. The <i>name</i> 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 <i>name</i> with options appropriate to <code>lp</code> 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 a 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.

### 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.

<b>CPI</b> <i>n</i>	Specifies the number of characters per inch. The value may be non-integer.
<b>DUPLEX</b>	Selects duplex (double sided) printing, binding on the long edge.
<b>DUPLEX SHORT</b>	Selects duplex (double sided) printing, binding on the short edge.
<b>LANDSCAPE</b>	Prints the page in landscape format.
<b>LPI</b> <i>n</i>	Specifies the number of lines per inch. The value must be 1, 2, 3, 4, 6, 8, 12, 16, 24 or 48.
<b>PAPER.SIZE</b> <i>xx</i>	Specifies the paper size. Valid size names are A4, LETTER, LEGAL, LEDGER, A3, MONARCH, COM_10, DL, C5, B5.
<b>SYMBOL.SET</b> <i>xx</i>	Specifies the character set. Valid values of <i>xx</i> are ROMAN8 (the default), LATIN1, ASCII, PC8.
<b>WEIGHT</b> <i>xx</i>	Specifies the font weight. Valid values of <i>xx</i> 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 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 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 auxiliary port.

The **SETPTR DISPLAY** command displays a report of the settings of all print unit. The optional **LPTR** keyword directs this report to a printer.

## 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](#)

## 4.172 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\*\*](#)

## 4.173 SH

The **SH** command executes a shell (operating system) command. This command is not available on the PDA version of QM.

### Format

**SH** *command*

**!***command*

**SH**

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. Interactive shells are currently only available for Linux, FreeBSD, AIX and Mac systems when QM is entered from the operating system command processor. Use the shell exit command to return to QM.

### Example

```
SH DIR
```

This command lists the current directory on a Windows system.

## 4.174 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.

## 4.175 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](#)

## 4.176 SP.ASSIGN

The **SP.ASSIGN** command uses a form queue number to set the destination and other options for printer output.

### Format

**SP.ASSIGN** {*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.

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.

**See also:**

[Printing](#), [SETPTR](#), [SET.QUEUE](#)

## 4.177 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](#)

## 4.178 SP.VIEW

The **SP.VIEW** command views and, optionally, prints records from \$HOLD or other files.

### Format

**SP.VIEW** {*file*} {*item*} { **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.

**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.

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
	Q		Quit from program. Requires use of return key to confirm
Return			Dive into selected item

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

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.

## 4.179 SPOOL

The **SPOOL** command sends specified records to the printer.

### Format

```
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

```
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.

## 4.180 STATUS

The **STATUS** command displays a list of active phantom processes. This command is not available on the PDA version of QM.

### 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\(\)](#)

## 4.181 STOP

The **STOP** command terminates the currently active paragraph.

### Format

#### **STOP**

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.

The value of [@SYSTEM.RETURN.CODE](#) is not affected by the **STOP** command.

### See also:

[ABORT](#)

## 4.182 SUBSCRIBE

The **SUBSCRIBE** command starts a phantom process that will poll a replication publisher for updates.

### Format

**SUBSCRIBE** server{:port} username password {options}

where

<i>server</i>	is the network name or ip address of the publisher.				
<i>port</i>	is the port number for connection to the publisher. This must correspond to the value of the <a href="#">REPLPORT</a> configuration parameter on the publisher and defaults to 4244 if omitted.				
<i>username</i>	is the username to be used for the server process established on the publisher.				
<i>password</i>	is the password associated with <i>username</i> . For security, this should be encrypted using the <a href="#">AUTHKEY</a> command and specified with an ENCR: prefix.				
<i>options</i>	are chosen from: <table> <tr> <td><b>INTERVAL</b> <i>n</i></td> <td>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.</td> </tr> <tr> <td><b>RAW</b></td> <td>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.</td> </tr> </table>	<b>INTERVAL</b> <i>n</i>	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.	<b>RAW</b>	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.
<b>INTERVAL</b> <i>n</i>	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.				
<b>RAW</b>	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 username 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](#), [PUBLISHACCOUNT](#)

## 4.183 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

<i>filename</i>	is the name of the file to be saved.
<i>id...</i>	is a list of record ids to be saved.
<b>BINARY</b>	suppresses conversion of newlines to field marks in directory files. Use this mode when saving binary data.
<b>COUNT.SUP</b>	suppresses display of the count of records saved.
<b>DET.SUP</b>	suppresses display of the detailed list of records saved.
<b>FROM</b> <i>listno</i>	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.

**See also:**

[ACCOUNT.RESTORE](#), [ACCOUNT.SAVE](#), [FILE.SAVE](#), [FIND.ACCOUNT](#),  
[RESTORE.ACCOUNTS](#), [SEL.RESTORE](#), [SET.DEVICE](#), [T.ATT](#), [T.LOAD](#), [T.xxx](#)

## 4.184 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](#), [RESTORE.ACCOUNTS](#), [SEL.RESTORE](#), [SET.DEVICE](#), [T.ATT](#), [T.DUMP](#), [T.xxx](#)

## 4.185 T.DET, T.EOD, T.FWD, T.RDLBL, T.READ, T.REW, T.STAT, T.WEOF

The **T.xxx** utility commands perform various functions to a tape device.

### Format

<b>T.DET</b>	Detach device
<b>T.EOD</b>	Position to end of data
<b>T.FWD</b>	Skip forwards one file
<b>T.RDLBL</b>	Read tape label
<b>T.READ</b>	Read tape
<b>T.REW</b>	Rewind tape
<b>T.STAT</b>	Report device status
<b>T.WEOF</b>	Write end of file marker

The tape or pseudo-tape to be processed must first be assigned using the [SET.DEVICE](#) command.

### See also:

[ACCOUNT.RESTORE](#), [ACCOUNT.SAVE](#), [FILE.SAVE](#), [FIND.ACCOUNT](#),  
[RESTORE.ACCOUNTS](#), [SEL.RESTORE](#), [SET.DEVICE](#), [T.ATT](#), [T.DUMP](#), [T.LOAD](#)

## 4.186 TERM

The **TERM** command specifies the terminal display dimensions.

### Format

#### **TERM**

**TERM** { *columns* } { , *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.

*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 command may include either or both of the *columns* and *lines* values to specify the characteristics of the display device. 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* value must be in the range 20 to 32767. The *lines* value must be in the range 10 to 32767.

The *term.type* option selects the terminal device type. The name given must correspond to an entry in the terminfo library or may be QMTERM (case independent) for the QMTerm emulator.

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 **TERM DISPLAY** command displays a list of input key codes and output control sequences for the currently selected terminal type.

**Example**

```
TERM 120,32
```

The above command sets the terminal to be 120 columns wide by 32 lines. A subsequent **TERM** command with no qualifying information would report

```
Page width: 120
Page depth: 32
Device      : QMTERM
```

**See also:**

[PTERM](#)

## 4.187 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

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.

## 4.188 UMASK

The **UMASK** command sets the default access rights for files created within the QM session. This command is not applicable to the PDA version of QM.

### Format

**UMASK** { *rights* }

The **UMASK** command operates in the same way as the `umask` command of Linux, FreeBSD or AIX, 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.

## 4.189 UNLOCK

The **UNLOCK** command, available only in the QMSYS account, releases task, record or file locks set by any process.

### Format

```
UNLOCK { USER user.no } { FILE file.no } { ALL | record.ids... }
```

```
UNLOCK { USER user.no } { FILE file.no } FILELOCK
```

```
UNLOCK TASKLOCK lock.no...
```

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 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.

## 4.190 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](#), [CREATE.FILE](#), [CREATE.KEY](#), [DELETE.KEY](#), [ENCRYPT.FILE](#), [GRANT.KEY](#), [LIST.KEYS](#), [REVOKE.KEY](#), [SET.ENCRYPTION.KEY.NAME](#)

## 4.191 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.

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](#)

## 4.192 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.

The screen layout for the PDA version is different and omits the user limit field.

### Example

```
LICENCE DETAILS
```

```
-----  
Licence number      [1961491396]          System id WLHX-YRTZ  
Max users           [50  ]  
Expiry date         [31 Dec 2000]  
Authorisation code  [YKSAJ-KKRFW-CNDKD-LRRTW-CJRTD]  
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
```

## 4.193 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:

<b>{DICT} file</b>	specifies the file to be updated. The optional <b>DICT</b> keyword indicates that the dictionary portion of the file is to be updated.
<b>USING {DICT} dict</b>	specifies that <i>dict</i> is to be used as the dictionary for <i>file</i> . The optional <b>DICT</b> keyword uses the dictionary portion of <i>dict</i> .

Selection of the records to be updated:

<b>FROM listno</b>	specifies that select list <i>listno</i> (0 to 10) is to be used as the list of records to process.
<b>ALL</b>	specifies that all records in <i>file</i> are to be updated. <b>UPDATE.RECORD</b> 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:

*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.

*value*

identifies the new value to be placed in *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.

**CONV** "*spec*"

Allows use of alternative conversion codes.

**DELETING** *field*

Deletes the specified *field* from the data record, moving all subsequent fields back by one position.

Options controlling the action of **UPDATE.RECORD** in batch mode:

**COUNT.SUP**

suppresses the normal report of the number of records processed.

**VERIFY.SUP**

suppresses the confirmation prior to an update using a select list or the **ALL** keyword.

**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.

**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.

**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.

<b>CREATING</b>	causes <b>UPDATE.RECORD</b> to create records that are not found in the file.
<b>OVERWRITING</b>	allows overwriting of existing records when changing the record id.
<b>REPORTING</b>	produces a detailed report of each update.
<b>LPTR {<i>n</i>}</b>	directs the commentary from <b>UPDATE.RECORD</b> to the specified print unit. Print unit 0 is used if <i>n</i> is omitted.
<b>NO.PAGE</b>	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:

<b>ID.SUP</b>	suppresses display of the field names.
<b>COL.SUP</b>	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 LPTR 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
6: DATE.PAID      : Date Paid       : 12 Feb 98ý17 Mar 98
7: INVOICES       : Invoices        : 001763ý001966
8: LICENCES       : Licences        : 907881792ý1907881802ý1907881808
9: EXPIRED.LICENCES: Expired         :
10: COUNTRY       : Country         :
11: CLASS         : Class          : 3
12: DEALER.SALES  : Sales          :
13: CALLBACK.DATE : Callback        : 01 Jul 98
14: CALLBACK.TEXT : Callback note   : Interested in new product range
15: VAT.NO        : VAT no         : 614 1210 25
16: SITE.TEXT     : Site text      : Acme Software
17: CONTACT       : Contact        : Anne McIntosh
18: POSITION       : Position       :
19: TEL.NO        : Tel no        : 01234-56789ý01234-64526
20: FAX.NO        : Fax no        : 01234-21767
21: MOBILE.NO     : Mobile no     :
22: EMAIL        : E-mail        : acme@mailier.com
23: SALES.TOTAL   : Sales         : £12783.33
24: NOTES        : Notes         :
*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/multivalued 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.

	Ctrl-	Esc-	Ctrl-X -
<b>A</b>	Home		
<b>B</b>	Cursor left		
<b>C</b>	Repeat		Quit
<b>D</b>	Delete char		
<b>E</b>	End		
<b>F</b>	Cursor right		
<b>G</b>	Cancel	Goto	
<b>H</b>	Backspace		
<b>I</b>			
<b>J</b>	(Return)		
<b>K</b>	Kill line		
<b>L</b>	Refresh		
<b>M</b>	(Return)		
<b>N</b>	Cursor down		
<b>O</b>	Overlay		Overlay
<b>P</b>	Cursor up		
<b>Q</b>	Quote char	Quote char	
<b>R</b>			
<b>S</b>			Save
<b>T</b>			
<b>U</b>	Repeat		
<b>V</b>	Page down	Page up	
<b>W</b>	Cut		Copy
<b>X</b>	Ctrl-X prefix	Command	
<b>Y</b>	Paste	Paste	
<b>Z</b>	Cursor up		
.			Mark
=			Expand char
<		Top	
>		Bottom	
Bkspc	Backspace		
Del	Delete char		
Return	Cursor down		

**UPDATE.RECORD** also recognises the following terminal control keys:

Home	End	Cursor left
Cursor right	Cursor up	Cursor down
Insert (Overlay)	Delete	Page up
Page down		

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:

<i>field</i>	go to specified field, value 1, subvalue 1
<i>field, value</i>	go to specified field and value, subvalue 1
<i>field, value, subvalue</i>	go to specified field, value and 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:

<i>,value</i>	go to specified value in current field
<i>,value,subvalue</i>	go to specified value/subvalue in current field
<i>,,subvalue</i>	go to specified subvalue in current field/value
<i>field,,subvalue</i>	go to specified field, value 1, specified subvalue

An asterisk can be use to imply "no change". This is useful when processing associated multivalued fields. For example:

<i>*,value</i>	go to specified value in current field
<i>field,*</i>	go to current value position in specified field
<i>field,*,*</i>	go to current value and subvalue in specified 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 three ways:

Followed by a number of up to three digits, it inserts the character with that decimal ASCII sequence.

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.

## 4.194 WHO

The **WHO** command displays the current user number and account name.

### Format

#### **WHO**

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.

### Example

```
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.

## 4.195 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

<a href="#"><u>LIST</u></a>	List records meeting specified criteria
<a href="#"><u>LIST.ITEM</u></a>	List records meeting specified criteria in internal format
<a href="#"><u>LIST.LABEL</u></a>	List records meeting specified criteria in address label format
<a href="#"><u>REFORMAT</u></a>	Builds a new file from data in the source file
<a href="#"><u>SHOW</u></a>	Interactive select list generation
<a href="#"><u>SORT</u></a>	List records meeting specified criteria in order of record id
<a href="#"><u>SORT.ITEM</u></a>	List records meeting specified criteria in order of record id in internal format
<a href="#"><u>SORT.LABEL</u></a>	List records meeting specified criteria in address label format, in order of record id
<a href="#"><u>SELECT</u></a>	Create a select list of records meeting specified criteria
<a href="#"><u>SREFORMAT</u></a>	Builds a new file from data in the source file, in order of record id
<a href="#"><u>SSELECT</u></a>	Create a select list of records meeting specified criteria in order of record id
<a href="#"><u>SEARCH</u></a>	Create a select list of records meeting specified criteria which include text matching over the entire record
<a href="#"><u>COUNT</u></a>	Count records meeting specified criteria
<a href="#"><u>SUM</u></a>	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

<i>verb</i>	is the query processor verb name
{ <b>DICT</b> } <i>file.name</i>	identifies the file to be processed. The optional <b>DICT</b> keyword indicates that the dictionary part of the file is to be processed. The <b>DICT.DICT</b> file will be used as the dictionary defining the items in the dictionary being reported.
<b>USING</b> { <b>DICT</b> } <i>file.name</i>	indicates that a dictionary other than the one normally associated with the file is to be used.
<i>field.name</i>	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 <b>F</b> <i>n</i> or <b>A</b> <i>n</i> may be used to reference field <i>n</i> and the <a href="#">EVAL</a> keyword may be used to introduce an evaluated expression.
<i>field.qualifier</i>	provides qualifying information about the immediately preceding <i>field.name</i> such as the format in which it is to be displayed.
<a href="#">selection clause</a>	specifies criteria determining which records from the file are to be included.
<a href="#">sort clause</a>	specifies the order in which records are to be processed.
<a href="#">display clause</a>	controls the manner in which data is displayed or printed.
<i>record.id</i>	specifies a particular record id is to be processed. Multiple record ids may be specified.
<b>FROM</b> <i>select.list.no</i>	specifies that a select list is to be used to control which records are processed. If the <b>FROM</b> 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 <i>record.id</i> items, only records that appear in the select list and are named <i>record.ids</i> will be processed.
<b>TO</b> <i>select.list.no</i>	for verbs that produce a select list, specifies which list is to be created. If the <b>TO</b> option is not used, the default select list (list 0) is used.

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 they 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. Field 2 of this record contains 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. Field 3 works in the same way but is applied to [SELECT](#), [SSELECT](#) and [SEARCH](#) commands.

In either case, 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, 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" )
```

## 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, BD1, BD94, BF20, XX90

Right: -6, 1A1, 1B1, +3, 7CX, 10A, 103, 1943, A1A, A1C, AA, BD1, BD94, 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.

The display clause is constructed from the elements in the table below.

Prefix	Data Item	Suffix
AVG	D-type item	CONV " <i>code</i> "
PCT [ <i>n</i> ]	I-type item	FMT " <i>spec</i> "
TOTAL	EVAL " <i>expr</i> " [AS <i>xx</i> ]	COL.HDG " <i>text</i> "
MAX	<i>Fn</i>	ASSOC " <i>name</i> "
MIN		ASSOC.WITH <i>field</i>
BREAK.ON [" <i>text</i> "]		DISPLAY.LIKE <i>field</i>
BREAK.SUP [" <i>text</i> "]		SINGLE.VALUE
ENUM		MULTI.VALUE
CALC		NO.NULLS
CUMULATIVE		

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* data item is recognised by the query processor as a reference to field *n*, treating the data as single valued with a 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 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 {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 {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 multivalued products, if the CHAINED.SELECT mode of the [OPTION](#) command is active, a **SELECT** or **SSELECT** 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
```

## 5.5 SEARCH

The **SEARCH** verb is similar to [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...}
  {FOR string1 string2 ...}
  {FROM select.list.no}
  {STRINGS file.name record.id}
  {SAVING {UNIQUE} {MULTI.VALUE} field.name {NO.NULLS}}
  {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 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.

All other options of the [SELECT](#) verb may be used in **SEARCH**.

### Example

```
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.

### See also:

[ALL.MATCH](#), [NO.CASE](#), [NO.MATCH](#)

## 5.6 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.

## 5.7 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. The [COL.SUP](#) keyword 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.

## 5.8 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 {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 {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 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 **LABEL NO.DEFAULT** in the command.

If no label template has been specified and either there is no @LABEL record or the **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.

It may be necessary to check that a printer font is chosen where the line spacing fits correctly onto the label page. The [SETPTR](#) command top margin may also need to be set to fit the page.

### Example

```
LIST LABEL CUSTOMERS NAME ADDRESS ID.SUP LABEL ADDR LABELS  
LPTR
```

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.

## 5.9 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}
```

**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.

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.

## 5.10 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.

## 5.11 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.

## 5.12 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:

- 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*.

### 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.

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 records marked with an asterisk are entered into the new select list.

QC Quit, clearing any record selection.

R Redisplay the screen. This is useful if a data transmission error causes screen corruption.

^^ Synonym for R.

? Display help text on the error line. Use the return key to walk through this text, line by line. Any key other than the return key will terminate the help display.

*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**).

## 5.13 Query processor keywords

### Field qualifiers

<a href="#"><u>%</u></a>	Synonym for PERCENTAGE
<a href="#"><u>AS</u></a>	Define synonym for field and qualifiers
<a href="#"><u>ASSOC</u></a>	Include field in association
<a href="#"><u>ASSOC.WITH</u></a>	Associate two fields
<a href="#"><u>AVERAGE</u></a>	Report average of field values
<a href="#"><u>AVG</u></a>	Synonym for AVERAGE
<a href="#"><u>BREAK.ON</u></a>	Define field as breakpoint control item.
<a href="#"><u>BREAK-ON</u></a>	Synonym for BREAK.ON
<a href="#"><u>BREAK.SUP</u></a>	Define field as non-displayed breakpoint control item.
<a href="#"><u>BREAK-SUP</u></a>	Synonym for BREAK.SUP
<a href="#"><u>CALC</u></a>	Calculate total of I-type item
<a href="#"><u>CALCULATE</u></a>	Synonym for CALC
<a href="#"><u>COL.HDG</u></a>	Set column heading for displayed item
<a href="#"><u>CONV</u></a>	Specify conversion to be applied to field
<a href="#"><u>CUMULATIVE</u></a>	Report cumulative value of field
<a href="#"><u>DISPLAY.LIKE</u></a>	Display field using attributes of another field
<a href="#"><u>DISPLAY.NAME</u></a>	Synonym for COL.HDG
<a href="#"><u>ENUM</u></a>	Synonym for ENUMERATE
<a href="#"><u>ENUMERATE</u></a>	Count number of values in specified field
<a href="#"><u>EVAL</u></a>	Defines expression to be evaluated
<a href="#"><u>FMT</u></a>	Specify format for display of field
<a href="#"><u>MAX</u></a>	Report maximum value of a field
<a href="#"><u>MIN</u></a>	Report minimum value of a field
<a href="#"><u>MULTI.VALUE</u></a>	Treat field as multivalued
<a href="#"><u>MULTIVALUED</u></a>	Synonym for MULTI.VALUE
<a href="#"><u>NO.NULLS</u></a>	Suppress null fields in MIN and AVG calculations
<a href="#"><u>PCT</u></a>	Synonym for PERCENTAGE
<a href="#"><u>PERCENT</u></a>	Synonym for PERCENTAGE
<a href="#"><u>PERCENTAGE</u></a>	Report percentages
<a href="#"><u>SINGLE.VALUE</u></a>	Treat field as single-valued
<a href="#"><u>SINGLEVALUED</u></a>	Synonym for SINGLE.VALUE
<a href="#"><u>TOTAL</u></a>	Report total of field values

### Selection clause options

<a href="#"><u>FIRST</u></a>	Synonym for SAMPLE
<a href="#"><u>FROM</u></a>	Process only records from given select list
<a href="#"><u>REQUIRE.SELECT</u></a>	Query must have an active select list
<a href="#"><u>SAMPLE</u></a>	Report only specified number of records
<a href="#"><u>SAMPLED</u></a>	Report only a sample of records
<a href="#"><u>WHEN</u></a>	Introduces multivalued field selection criteria
<a href="#"><u>WITH</u></a>	Introduces record selection criteria

### SEARCH options

<a href="#"><u>ALL.MATCH</u></a>	Record must contain all given strings
<a href="#"><u>FOR</u></a>	Specifies search strings on the command line
<a href="#"><u>NO.CASE</u></a>	Case insensitive option for SEARCH
<a href="#"><u>NO.MATCH</u></a>	Record must contain none of the given strings
<a href="#"><u>STRINGS</u></a>	Specifies a file and record containing the search strings

### Selection clause operators used in WITH constructs

<a href="#"><u>#</u></a>	Not equal. Synonym for NE
<a href="#"><u>&amp;</u></a>	Logical AND. Synonym for AND
<a href="#"><u>&lt;</u></a>	Less than. Synonym for LT
<a href="#"><u>&lt;=</u></a>	Less than or equal to. Synonym for LE
<a href="#"><u>&lt;&gt;</u></a>	Not equal. Synonym for NE
<a href="#"><u>=</u></a>	Equals. Synonym for EQ
<a href="#"><u>=&lt;</u></a>	Less than or equal to. Synonym for LE
<a href="#"><u>=&gt;</u></a>	Greater than or equal to. Synonym for GE
<a href="#"><u>&gt;</u></a>	Greater than. Synonym for GT
<a href="#"><u>&gt;&lt;</u></a>	Not equal. Synonym for NE
<a href="#"><u>&gt;=</u></a>	Greater than or equal to. Synonym for GE
<a href="#"><u>~</u></a>	Soundex matching. Synonym for SAID
<a href="#"><u>AFTER</u></a>	Greater than Synonym for GT
<a href="#"><u>AND</u></a>	Logical AND
<a href="#"><u>BEFORE</u></a>	Less than. Synonym for LT
<a href="#"><u>BETWEEN</u></a>	Closed range test
<a href="#"><u>EQ</u></a>	Equals
<a href="#"><u>EQUAL</u></a>	Equals. Synonym for EQ
<a href="#"><u>GE</u></a>	Greater than or equal to
<a href="#"><u>GREATER</u></a>	Greater than. Synonym for GT
<a href="#"><u>GT</u></a>	Greater than
<a href="#"><u>IN</u></a>	Item is in named list
<a href="#"><u>LE</u></a>	Less than or equal to
<a href="#"><u>LESS</u></a>	Less than. Synonym for LT
<a href="#"><u>LIKE</u></a>	Pattern match
<a href="#"><u>LT</u></a>	Less than
<a href="#"><u>MATCHES</u></a>	Pattern match. Synonym for LIKE
<a href="#"><u>MATCHING</u></a>	Pattern match. Synonym for LIKE
<a href="#"><u>NE</u></a>	Not equal
<a href="#"><u>NO</u></a>	Test for null field
<a href="#"><u>NOT</u></a>	Synonym for NE
<a href="#"><u>NOT.IN</u></a>	Item is not in named list
<a href="#"><u>OR</u></a>	Logical OR
<a href="#"><u>SAID</u></a>	Soundex matching
<a href="#"><u>SPOKEN</u></a>	Soundex matching. Synonym for SAID
<a href="#"><u>UNLIKE</u></a>	Inverse pattern match. Opposite of LIKE

**Sort clause options**

<a href="#"><u>BY</u></a>	Sort by ascending field value order
<a href="#"><u>BY.DSND</u></a>	Sort by descending field value order
<a href="#"><u>BY-DSND</u></a>	Synonym for BY.DSND
<a href="#"><u>BY.EXP</u></a>	Ascending exploded multivalued sort
<a href="#"><u>BY-EXP</u></a>	Synonym for BY.EXP
<a href="#"><u>BY.EXP.DSND</u></a>	Descending exploded multivalued sort
<a href="#"><u>BY-EXP-DSND</u></a>	Synonym for BY.EXP.DSND

**Display options**

<a href="#"><u>BOXED</u></a>	Generates a boxed report on a PCL printer
<a href="#"><u>CAPTION</u></a>	Synonym for GRAND.TOTAL
<a href="#"><u>COL.HDG.ID</u></a>	Use field names as default column headings
<a href="#"><u>COL.HDR.SUPP</u></a>	Suppress page and column headings
<a href="#"><u>COL.HDR.SUP</u></a>	Synonym for COL.HDR.SUPP
<a href="#"><u>COL-HDR-SUPP</u></a>	Synonym for COL.HDR.SUPP
<a href="#"><u>COL-HDR-SUP</u></a>	Synonym for COL.HDR.SUPP

<a href="#"><u>COL.SPACES</u></a>	Specifies inter-column spacing
<a href="#"><u>COL.SPCS</u></a>	Synonym for COL.SPACES
<a href="#"><u>COL.SUP</u></a>	Suppress column headings
<a href="#"><u>COL-SUPP</u></a>	Synonym for COL.SUP
<a href="#"><u>COUNT.SUP</u></a>	Suppress record count message at end of report
<a href="#"><u>CSV</u></a>	Specifies a comma separated delimited report
<a href="#"><u>DBL.SPC</u></a>	Output report with double line spacing
<a href="#"><u>DBL-SPC</u></a>	Synonym for DBL.SPC
<a href="#"><u>DELIMITER</u></a>	Specifies delimited report
<a href="#"><u>DET.SUP</u></a>	Suppress detail lines
<a href="#"><u>DET-SUPP</u></a>	Synonym for DET.SUP
<a href="#"><u>FOOTER</u></a>	Synonym for FOOTING
<a href="#"><u>FOOTING</u></a>	Specify page footing
<a href="#"><u>FORCE</u></a>	Force print of headings in empty report
<a href="#"><u>GRAND.TOTAL</u></a>	Specify format for totals line
<a href="#"><u>GRAND-TOTAL</u></a>	Synonym for GRAND.TOTAL
<a href="#"><u>HDR.SUP</u></a>	Suppress default page heading
<a href="#"><u>HDR-SUPP</u></a>	Synonym for HDR.SUP
<a href="#"><u>HEADER</u></a>	Synonym for HEADING
<a href="#"><u>HEADING</u></a>	Specify page heading
<a href="#"><u>ID.ONLY</u></a>	Suppress use of @ phrase
<a href="#"><u>ID.SUP</u></a>	Suppress default inclusion of @ID in report
<a href="#"><u>ID-SUPP</u></a>	Synonym for ID.SUP
<a href="#"><u>LPTR</u></a>	Direct report to a printer
<a href="#"><u>MARGIN</u></a>	Specify width of left margin
<a href="#"><u>NEW.PAGE</u></a>	Display or print each record on a separate page
<a href="#"><u>NO.PAGE</u></a>	Suppress pause between displayed pages
<a href="#"><u>NOPAGE</u></a>	Synonym for NO.PAGE
<a href="#"><u>NO.SPLIT</u></a>	Avoid splitting a record across pages if possible
<a href="#"><u>ONLY</u></a>	Synonym for ID.ONLY
<a href="#"><u>OVERLAY</u></a>	Sets a graphical page overlay
<a href="#"><u>PAN</u></a>	Pan columns of a wide report
<a href="#"><u>REPEATING</u></a>	Repeats single valued items or the final value of a multivalued item
<a href="#"><u>SCROLL</u></a>	Allow scrolling through report pages
<a href="#"><u>STYLE</u></a>	Sets a report style
<a href="#"><u>SUPP</u></a>	Synonym for HDR.SUP
<a href="#"><u>VERT</u></a>	Synonym for VERTICALLY
<a href="#"><u>VERTICALLY</u></a>	Report in vertical (one field per line) format

### SELECT and SSELECT options

<a href="#"><u>NO.NULLS</u></a>	Ignore null fields with SAVING option
<a href="#"><u>SAVING</u></a>	Save field value in place of record id
<a href="#"><u>TO</u></a>	Specifies target select list number
<a href="#"><u>UNIQUE</u></a>	Omit duplicates with SAVING

### Miscellaneous

<a href="#"><u>ABSENT.IGNORE</u></a>	Ignore unfound records
<a href="#"><u>ABSENT.NULL</u></a>	Treats an absent record as a null item rather than an error
<a href="#"><u>DEFAULT</u></a>	Parse the query using default option settings
<a href="#"><u>LABEL</u></a>	Specifies the label template for LIST.LABEL and SORT.LABEL
<a href="#"><u>LOCKING</u></a>	Takes a file lock on the file being processed, preventing updates
<a href="#"><u>NO.INDEX</u></a>	Do not use an alternate key index for record selection
<a href="#"><u>REQUIRE.INDEX</u></a>	Do not perform the query unless an alternate key index can be

[TO](#)

used

Specifies the output file name for REFORMAT. See also CSV and DELIMITER

[USING](#)

Use an alternative dictionary

## 5.14 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.

## 5.15 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).

## 5.16 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](#)

## 5.17 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 multivalued 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
```

## 5.18 AS

The **AS** keyword is a field qualifier which defines a synonym for a field and any qualifying information.

### Format

*field.name* {*field.qualifier*} **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](#)

## 5.19 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

```
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](#)

## 5.20 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* **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

```
LIST ORDERS PART.NO QTY EVAL "SELL * QTY" 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](#)

## 5.21 AVERAGE

The **AVERAGE** field qualifier keyword causes a field to be reported together with its average value. The synonym **AVG** may be used.

### Format

**AVERAGE** *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 **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 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.

The **NO.NULLS** keyword can be used to prevent null values being included in the calculation of the average value.

### Example

The sentence

```
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.

```
LIST INVOICES AVERAGE 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
=====
                £489.49
4 records listed.
```

## 5.22 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.

## 5.23 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.

## 5.24 BREAK.ON

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/footer 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 If the PICK.BREAKPOINT.U mode of the [OPTION](#) command is in effect, this mode inserts a line of hyphens above any subtotals, etc. If this option is not in effect, the line of hyphens is produced unless the U mode is used.
- 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.

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          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
          **                -----
          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.V 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
          Total                -----
          North            1743.86
```

19866	Abbott	465.31
19886	Abbott	397.23
19830	Smith	324.39
	Total	-----
	South	1186.93
		=====
		2930.79

7 records listed.

**See also:**

**[BREAK.SUP](#)**

## 5.25 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/footer 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 If the PICK.BREAKPOINT.U mode of the [OPTION](#) command is in effect, this mode inserts a line of hyphens above any subtotals, etc. If this option is not in effect, the line of hyphens is produced unless the U mode is used.

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
		=====
		2930.79

7 records listed.

**See also:**  
[BREAK.ON](#)

## 5.26 BY

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.

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](#)

## 5.27 BY.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.

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](#)

## 5.28 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.

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.

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.

```
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

```
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
ORDER PART QTY PRICE TOTAL.
24851 107 2 12.50 25.00
24842 216 3 8.00 24.00
24856 319 6 4.50 27.00
24856 372 1 18.75 18.75
24842 648 7 10.00 70.00
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 PART.NO > 300
```

would produce the report below.

```
LIST ORDERS PART.NO QTY PRICE LINE.TOTAL
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](#)**

## 5.29 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

**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.

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.

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.

```
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

```
LIST ORDERS PART.NO QTY LINE.TOTAL BY.EXP.DSND PART.NO
```

applied to the same data would produce the display 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
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](#)**

## 5.30 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 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 works 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.

### Example

We have a file which includes a calculated PROFIT item defined as

```
100 * (SELL - COST) / COST
```

The command

```
LIST PARTS AVG COST AVG SELL AVG PROFIT
```

This might produce a report such as that below

```
LIST PARTS AVG COST AVG SELL AVG PROFIT
Part   Cost.   Sell.   Profit%
101    10.00    13.00    30.00
102    15.00    18.00    20.00
103    14.00    17.00    21.43
=====
        13.00    16.00    23.81
3 records listed.
```

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 * (TOTAL(SELL) - TOTAL(COST)) / TOTAL(COST)
```

or, more simply,

```
100 * TOTAL(SELL - COST) / TOTAL(COST)
```

The command

```
LIST PARTS AVG COST AVG SELL CALC PROFIT
```

now produces

```
LIST PARTS AVG COST AVG SELL CALC PROFIT
Part   Cost.    Sell.   Profit%
 101   10.00    13.00    30.00
 102   15.00    18.00    20.00
 103   14.00    17.00    21.43
      =====
      13.00    16.00    23.08
3 records listed.
```

## 5.31 COL.HDG

The **COL.HDG** keyword defines an alternative column heading for reported data. The synonym **DISPLAY.NAME** may be used.

### Format

*field* **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.

## 5.32 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.
```

### 5.33 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.
```

## 5.34 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.

Tabular reports are automatically adjusted to fit the available page space. 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.

```
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.
```

Use of the **COL.SPACES** keyword could modify this to become

```
LIST INVOICES VALUE CUSTOMER.NAME WITH NO AMT.PAID COL.SPACES
8
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 CONV

The **CONV** keyword defines an alternative conversion for reported data.

### Format

*field* **CONV** *conv.spec*

where

*field* is the field or expression to which the new conversion is to be applied.

*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

```
LIST ORDERS ORDER.DATE CONV "DDMY"
```

This command reports records from the ORDERS file where using a non-default conversion specification for the ORDER.DATE field.

## 5.36 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.

## 5.37 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.
```

## 5.38 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}}
```

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.

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 <sup>^</sup>*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.

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

```
16832,176VM218VM398
```

See also: [DELIMITER](#)

## 5.39 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.
```

## 5.40 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.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
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.PAID 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.
```

## 5.41 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 \$QUERY.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](#)

## 5.42 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 {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 <sup>^</sup>*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.

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.

```
74529,£1712.43,J McTavish
74273,£95.23,County Newspapers
63940,£141.00,R Bryant
74993,£9.29,Write Right Stationery
```

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](#)

## 5.43 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
          **          -----
          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.

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

REGION	ORDER VALUE
North	1743.86
South	1186.93
	=====
	2930.79

7 records listed.

## 5.44 DISPLAY.LIKE

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

*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

```
LIST INVOICES EVAL "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.

## 5.45 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.
```

## 5.46 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** mode 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.

## 5.47 EVAL

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](#)

## 5.48 FMT

The **FMT** keyword defines an alternative format for reported data.

### Format

*field* **FMT** *fmt.spec*

where

*field* is the field or expression to which the new format is to be applied.

*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 *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.

## 5.49 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.
- 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 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.

26 Jun 2000  
Confidential

SALES REPORT

**See also:**

[HEADING](#)

## 5.50 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 PARSE.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](#)

## 5.51 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.

## 5.52 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*

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.

### 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.

## 5.53 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

*field* **GE** {**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 **GE** selection clause operator returns true if *field* is greater than or equal to *value*.

### Example

```
LIST STOCK WITH QTY GE 100
```

This command lists items found on the **STOCK** file with a **QTY** field of 100 or over.

## 5.54 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](#)

## 5.55 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

*field* **GT** {**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 **GT** selection clause operator returns true if *field* is greater than *value*.

### Example

```
LIST STOCK WITH QTY GT 100
```

This command lists items found on the **STOCK** file with a **QTY** field of over 100.

## 5.56 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.
```

## 5.57 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.

```
                SALES REPORT
26 Jun 2000
Confidential 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.

**See also:**

[FOOTING](#)

## 5.58 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 file name in a query processor sentence. For example:

```
LIST ID.ONLY ORDERS
```

## 5.59 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.
```

## 5.60 IN

The **IN** selection clause operator tests whether the content of a field is in a named select list.

### Format

*field* **IN** {**NO.CASE**} *listname*

*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.

## 5.61 LABEL

The **LABEL** keyword specifies the label template record name for [LIST.LABEL](#) and [SORT.LABEL](#).

### Format

**LABEL** *template.name*

**LABEL NO.DEFAULT**

The first format of the **LABEL** keyword specifies the 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.

## 5.62 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

*field* **LE** {**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 **LE** selection clause operator returns true if *field* is less than or equal to *value*.

### Example

```
LIST STOCK WITH QTY LE 100
```

This command lists items found on the **STOCK** file with a **QTY** field of 100 or less.

## 5.63 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

*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

```
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

```
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

```
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](#)

## 5.64 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 snap shot view of the data at the point when the query began.

Files accessed using the [TRANSQ](#) 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.

## 5.65 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.

## 5.66 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

*field* **LT** {**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 **LT** selection clause operator returns true if *field* is less than *value*.

### Example

```
LIST STOCK WITH QTY LT 100
```

This command lists items found on the **STOCK** file with a **QTY** field of less than 100.

## 5.67 MARGIN

The **MARGIN** keyword specifies the width of a blank left margin to appear in the report output.

### Format

**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.

## 5.68 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.
```

## 5.69 MIN

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.

## 5.70 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](#)

## 5.71 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** mode of the [OPTION](#) command can be used to apply string character comparisons to all data.

### Example

```
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.

## 5.72 NEW.PAGE

The **NEW.PAGE** display option keyword causes each record in a report to start on a new page.

### Format

**NEW.PAGE**

## 5.73 NO

The **NO** selection clause operator tests whether a field or evaluated expression is null.

### Format

**WITH NO** *field*

where

*field* is the field or evaluated expression to be tested.

The **NO** selection clause operator returns true if *field* is null.

### Example

```
LIST STOCK WITH NO SUPPLIER.CODE
```

This command lists items found on the **STOCK** file with a null **SUPPLIER.CODE**.

## 5.74 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](#)

## 5.75 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](#)

## 5.76 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.

## 5.77 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](#)

## 5.78 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](#) and [MIN](#) keywords, this keyword suppresses use of null values when calculating averages, enumerations or minimum values.
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 omit records with null field values from the index.

## 5.79 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.

## 5.80 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.

## 5.81 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.

## 5.82 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 multivalued 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.

## 5.83 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.

## 5.84 PAGESEQ

The **PAGESEQ** option identifies a record that controls page numbering.

### Format

**PAGESEQ** *filename id*

where

*filename* is the name of the file containing the control record.

*id* is the record id of the control record.

The **PAGESEQ** 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 **PAGESEQ** 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 PAGESEQ 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.

## 5.85 PAN

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.

## 5.86 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.

### 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 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.

## 5.87 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.

## 5.88 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.

## 5.89 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.

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](#)

## 5.90 SAID

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

*field* **SAID** *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 **SAID** selection clause operator returns true if the Soundex phonetic code for *field* is *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.

0	A E H I O U W Y
1	B F P V
2	C G J K Q S X Z
3	D T
4	L
5	M N
6	R

Other letters 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 EVAL "SOUNDEX('REED')"
```

This command names in the **STAFF** file that sound like Reed (Read, Reid, etc).

## 5.91 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.

### 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.

## 5.92 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.

## 5.93 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 saved 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 treats value marks and subvalue marks as part of the data without applying any special meaning. 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.

## 5.94 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.

## 5.95 SINGLE.VALUE

The **SINGLE.VALUE** keyword is a field qualifier that forces the field to be processed as a single-valued item.

### Format

*field* **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](#)

## 5.96 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](#)

## 5.97 STYLE

The **STYLE** keyword selects the report style to be used, overriding any style selected using the [REPORT.STYLE](#) or [SETPTR](#) commands or the QMBasic [SETPU](#) statement.

### Format

**STYLE** *name*

where

*name* is the name of a VOC style record. Use of **STYLE NONE** will disable use of any style selected using the [REPORT.STYLE](#) or [SETPTR](#) commands or the QMBasic [SETPU](#) statement.

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 VOC record 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:

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

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.

### Example

```
X  
H=Bright Blue,,Bold  
S=Blue  
T=Bright Red,,Bold
```

See also:

[REPORT.STYLE](#)

## 5.98 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.

## 5.99 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.

## 5.100 TOTAL

The **TOTAL** field qualifier keyword causes a field to be reported together with its total value.

### Format

**TOTAL** *field* {*field.qualifiers*}

where

*field* is the field or evaluated expression to be displayed.

*field.qualifiers* 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

```
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.

```
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.
```

## 5.101 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

*field* **UNLIKE** {**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.

### 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](#)

## 5.102 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.

## 5.103 VERTICALLY

The **VERTICALLY** display option keyword causes a vertical format report to be produced. The synonym **VERT** may be used.

### Format

```
VERTICALLY {FMT "format"}
```

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.

The default action of the query processor is to show the field heading text 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.

### 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.
```

## 5.104 WHEN

The **WHEN** keyword introduces a selection clause for a multivalued field.

### Format

**WHEN** *condition*

where

*condition* is *field1 operator field2* to compare two fields  
or *field1 operator value* to compare a field with a literal value

*operator* is any of the query processor operators:

<u><b>EQ</b></u>	=	<b>EQUAL</b>		
<u><b>NE</b></u>	#	<b>NOT</b>	<>	><
<u><b>LT</b></u>	<	<b>LESS</b>	<b>BEFORE</b>	
<u><b>LE</b></u>	<=	=<		
<u><b>GT</b></u>	>	<b>GREATER</b>	<b>AFTER</b>	
<u><b>GE</b></u>	>=	=>		
<u><b>LIKE</b></u>	<b>MATCHES</b>	<b>MATCHING</b>		
<u><b>UNLIKE</b></u>	<b>NOT.MATCHING</b>			
<u><b>SAID</b></u>	<b>SPOKEN</b>	~		
<u><b>NO</b></u>				
<u><b>BETWEEN</b></u>				

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.

### 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 **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.

## 5.105 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:

<u><b>EQ</b></u>	=	<b>EQUAL</b>		
<u><b>NE</b></u>	#	<b>NOT</b>	<>	><
<u><b>LT</b></u>	<	<b>LESS</b>	<b>BEFORE</b>	
<u><b>LE</b></u>	<=	=<		
<u><b>GT</b></u>	>	<b>GREATER</b>	<b>AFTER</b>	
<u><b>GE</b></u>	>=	=>		
<u><b>LIKE</b></u>	<b>MATCHES</b>	<b>MATCHING</b>		
<u><b>UNLIKE</b></u>	<b>NOT.MATCHING</b>			
<u><b>SAID</b></u>	<b>SPOKEN</b>	~		
<u><b>NO</b></u>				
<u><b>BETWEEN</b></u>				

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.

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)
```

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
```

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'
```

## 5.106 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
```

**Part**

---

**6**

**QMBasic**

## 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.

[QMBasic Overview](#)

[Variable Names and Values](#)

[Scalars, Matrices and Dynamic Arrays](#)

[Objects](#)

[Common Blocks](#)

[Labels](#)

[Expressions and Operators](#)

[Assignment Statements](#)

[Type Conversion](#)

[Matrix file i/o](#)

[Sequential file i/o](#)

[Multivalue functions](#)

[Object oriented programming](#)

[Compiler Directives](#)

[Limits](#)

[QMBasic Statements by Name](#)

## 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 in [directory files](#) where 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 directory 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 program module that can be run directly from the command line. It can also be called from other programs using <a href="#">CALL</a> in the same way as a subroutine that has no arguments. A program optionally starts with a <a href="#">PROGRAM</a> 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 <a href="#">CALL</a> . Subroutines usually take <b>arguments</b> , variables that are passed in or out of the subroutine to transfer data between modules. A subroutine module starts with a <a href="#">SUBROUTINE</a> 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 <a href="#">FUNCTION</a> statement.
Class modules	A class module contains the property and method routines that are used for <a href="#">object oriented programming</a> . A class module starts with a <a href="#">CLASS</a> 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 but in a file 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, ;* A subroutine call
            DATA, ;* with each argument
            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.

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,...)
and
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 multivalued database products there is a compiler option to make this final **END** optional.

## 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.

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	GOSUB	ON
BEFORE	GOTO	OR
BY	GT	REPEAT
CAPTURING	IN	RETURNING
CAT	LE	SETTING
DO	LOCKED	STEP
ELSE	LT	THEN
EQ	MATCH	TO
FROM	MATCHES	TRAPPING
GE	NE	UNTIL
GO	NEXT	WHILE

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 was performed many times in a loop, it may be worth forcing a type conversion to prevent repeated temporary conversions. For this reason, QMBasic programs often contain apparently redundant looking statements of the form

```
A = A + 0      ;* Convert to numeric form
or
S = S : ""     ;* Convert to string form
```

Numeric values may be 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 multivalued 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 is typically many megabytes. Although QMBasic avoids copying strings unnecessarily whenever it can, operations involving very

large strings are likely to have a detrimental effect on performance.

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 or strings.

**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.

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.

The compiler imposes no limit on the length of a string literal value though it may not extend from one line to the next. 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.

## QMBasic - Scalars, matrices and dynamic arrays

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.

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 6<sub>FM</sub>8<sub>VM</sub>10<sub>FM</sub>12. 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.

## 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 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. 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 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,...etc...
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 redimensioned 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[ <i>s,n</i> ]	Use <i>n</i> character substring starting at character <i>s</i> of <i>var</i>
var[ <i>n</i> ]	Use last <i>n</i> characters of string <i>var</i>
var[ <i>d,s,n</i> ]	Use <i>n</i> consecutive sections of string <i>var</i> delimited by character <i>d</i> , starting at section <i>s</i>
var< <i>f</i> >	Use field <i>f</i> of dynamic array variable
var< <i>f,v</i> >	Use field <i>f</i> , value <i>v</i> of dynamic array variable
var< <i>f,v,s</i> >	Use field <i>f</i> , value <i>v</i> , subvalue <i>s</i> of dynamic array variable
func( <i>args</i> )	Use value of 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 index values.

There is also a special conditional item of the form

```
IF conditional.expr THEN expr.1 ELSE expr.2
```

where *conditional.expression* 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 that any value other than zero or a null string is treated as true, zero or a null string being treated as false. An expression returning a boolean value returns the integer value 1 for true, zero for false. The boolean values are available as @TRUE and @FALSE for use in programs.

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 = "abcdefghi jkl "  
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.

The trailing substring extraction operation *x[n]* extracts the last *n* characters of the string *x*. Thus

```
A = "abcdefghi jkl "
```

```
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 *var*[*d,s,n*]. It treats *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 [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>           extracts field 2
x<2 , 7>       extracts field 2, value 7
x<2 , 7 , 3>   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 with the exception of the exponentiation operator which is processed right to left. Round brackets may be used to alter the order of execution or to improve readability of complex expressions.

+	Unary plus	0
-	Unary minus	0
~	Unary bitwise not	0
< >	Dynamic array extraction	1
[ ]	Substring extraction	1
** or ^	Exponentiation (raising to power)	2
*	Multiplication	3
/	Division	3
//	Integer division	3
+	Addition	4
-	Subtraction	4
	Implicit format (See <a href="#">FMT()</a> function)	5
:	Concatenation	6
<	Less than	7
>	Greater than	7
=	Equal to	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.

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.

As an example of the **MATCHES** operator, the statement

```
IF S MATCHES "1N0N'-'1N0N" 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 `1N0N` to ensure that at least one digit is present.

The content of two dimensioned matrices may be compared using

```
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.

**See also:**

[Pattern Matching](#)

## QMBasic - Assignment statements

Variables may be assigned values by statements of the following forms

<code>var op expr</code>	Assign <i>expr</i> to <i>var</i>
<code>var[s,n] = expr</code>	Assign <i>expr</i> to substring of <i>var</i>
<code>var[n] = expr</code>	Assign <i>expr</i> to trailing substring of <i>var</i>
<code>var[d,i,n] = expr</code>	Assign <i>expr</i> to delimited substring of <i>var</i>
<code>var&lt;f&gt; = expr</code>	Assign <i>expr</i> to field <i>f</i> of <i>var</i>
<code>var&lt;f,v&gt; = expr</code>	Assign <i>expr</i> to field <i>f</i> , value <i>v</i> of <i>var</i>
<code>var&lt;f,v,s&gt; = expr</code>	Assign <i>expr</i> to field <i>f</i> , value <i>v</i> , subvalue <i>s</i> of <i>var</i>

In all cases, *var* may be a dimensioned matrix element.

The `var op expr` format allows the following operators.

<code>=</code>	Simple assignment
<code>+=</code>	Add <i>expr</i> to original value
<code>-=</code>	Subtract <i>expr</i> from original value
<code>*=</code>	Multiply original value by <i>expr</i>
<code>/=</code>	Divide original value by <i>expr</i>
<code>:=</code>	Concatenate <i>expr</i> as string to original value
<code>&amp;=</code>	Bitwise AND with original value
<code>!=</code>	Bitwise OR with original value

Additionally, many other statements set values into variables.

### Substring Assignment

The substring assignment operator is

$$\text{var}[s,n] = \text{expr}$$

In the default compiler modes, substring assignment overlays an existing portion of a string. 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 multivalued environments. Use of the `PICK.SUBSTR` option 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 `PICK.SUBSTR.ASSIGN` option 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 or *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

$$\text{var}[n] = \text{expr}$$

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.

### 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. When adding a new field at the end of a string, the syntax

$$Z<-1> = \text{expr}$$

can be used. The QMBasic language will add a new field to receive the result. Similarly, the operations

$$\begin{aligned} Z<5,-1> &= \text{expr} \\ Z<5,3,-1> &= \text{expr} \end{aligned}$$

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  $S = \text{"ABC}_{\text{FM}}\text{DEF}_{\text{VMFM}}\text{XYZ}_{\text{FM}}\text{"}$

$S<-1> = \text{"ghi"}$	sets S to $\text{"ABC}_{\text{FM}}\text{DEF}_{\text{VMFM}}\text{XYZ}_{\text{FM}}\text{GHI"}$
$S<1,-1> = \text{"ghi"}$	sets S to $\text{"ABC}_{\text{VM}}\text{GHI}_{\text{FM}}\text{DEF}_{\text{VMFM}}\text{XYZ}_{\text{FM}}\text{"}$
$S<2,-1> = \text{"ghi"}$	sets S to $\text{"ABC}_{\text{FM}}\text{DEF}_{\text{VMVM}}\text{GHI}_{\text{FM}}\text{XYZ}_{\text{FM}}\text{"}$

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 the how other multivalue database products work and results in

```
S<-1> = "ghi"      sets S to "ABCFMDEFVMFMXYZFMGHI"
S<1,-1> = "ghi"    sets S to "ABCVMGHIFMDEFVMFMXYZFM"
S<2,-1> = "ghi"    sets S to "ABCFMDEFVMGHIFMXYZFM"
```

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

```
PART.NO<-1> = part number
DESCRIPTION<-1> = description
COMMENT<-1> = 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:

```
PART.NO<N> = part number
DESCRIPTION<N> = description
COMMENT<N> = text
```

where N is the dynamic array filed 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.

## 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

```
NUM.INVOICES = 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.

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..

## 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	<i>account:file</i>
Implicit QMNet pointer	<i>server:account:file</i>
Pathname	<i>PATH:pathname</i>

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 of 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 would be stored in ORD.ID and records would 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 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 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.

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 number 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. 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.ID
UNTIL CLI.ID = ''
  READ CLI.REC FROM CLI.F, CLI.ID THEN
    DISPLAY CLI.REC<2>
  END ELSE
    DISPLAY 'Client not found'
  END
REPEAT
```

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 1 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 multivalued 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
REPEAT
```

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
  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
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. For programs that perform a significant amount of processing of the data in a record, it may be worth the cost of breaking the fields into separate elements of a dimensioned matrix using [MATREAD](#) and [MATWRITE](#).

These statements 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:

```
AFMB1VMB2FMC
```

using [MATREAD](#) to read this into a three element (plus the zero element) matrix would result in:

0	
1	A
2	B1 <sub>VM</sub> B2
3	C

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:

0	
1	A
2	B1 <sub>VM</sub> B2
3	C
4	
5	

The [INMAT\(\)](#) function can be used to determine how many fields the record had. 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 a record with five fields and an array with three elements:

0	D <sub>FME</sub>
1	A
2	B <sub>1vM</sub> B <sub>2</sub>
3	C

The **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:

1	A
2	B <sub>1vM</sub> B <sub>2</sub>
3	C <sub>FM</sub> D <sub>FME</sub>

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 [STATUSQ](#) 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:

<a href="#">READSEQ</a>	Read text line by line
<a href="#">READBLK</a>	Read a given number of bytes
<a href="#">WRITESEQ</a>	Write text line by line
<a href="#">WRITESEQF</a>	Write text line by line, flushing to disk before continuing
<a href="#">WRITEBLK</a>	Write a given number of bytes
<a href="#">READCSV</a>	Read comma separated variable (CSV) format data
<a href="#">WRITECSV</a>	Write comma separated variable format data
<a href="#">SEEK</a>	Position within the sequential item
<a href="#">NOBUF</a>	Suppress buffering
<a href="#">WEOFSEQ</a>	Write end of file (truncate the item)
<a href="#">CLOSESEQ</a>	Close the sequential item, flushing buffers and releasing the lock.

**Examples**

```

OPENSEQ 'C:\PRICES' TO SEQ.F ELSE STOP 'Cannot open price data'
OPEN 'STOCK' TO STK.F ELSE STOP 'Cannot open STOCK'
LOOP
  READSEQ TEXT FROM SEQ.F ELSE EXIT
  STK.ID = TEXT[1,5]
  READU STK.REC FROM STK.F, STK.ID THEN
    STK.REC<STK.PRICE> = ICONV(TEXT[6,8], 'MD2')
    WRITE STK.REC TO STK.F, STK.ID
  END ELSE
    RELEASE STK.F, STK.ID
    DISPLAY 'Stock item ' : STK.ID : ' not found'
  END
REPEAT

```

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.

```

OPENSEQ 'C:\IMPORT.CSV' TO SEQ.F ELSE STOP 'Cannot open import
file'
OPEN 'STOCK' TO STK.F ELSE STOP 'Cannot open STOCK'
LOOP
  READCSV FROM SEQ.F TO STK.ID, PRICE ELSE EXIT
  READU STK.REC FROM STK.F, STK.ID THEN
    STK.REC<STK.PRICE> = ICONV(PRICE, 'MD2')
    WRITE STK.REC TO STK.F, STK.ID
  END ELSE
    RELEASE STK.F, STK.ID
    DISPLAY 'Stock item ' : STK.ID : ' not found'
  END
REPEAT

```

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.

```

OPENSEQ 'C:\EXPORT.CSV' OVERWRITE TO SEQ.F ELSE
  IF STATUS() THEN STOP 'Cannot open export file'
END
OPEN 'STOCK' TO STK.F ELSE STOP 'Cannot open STOCK'
SELECT STK.F TO 1
LOOP
  READNEXT STK.ID FROM 1 ELSE EXIT
  READ STK.REC FROM STK.F, STK.ID THEN
    WRITECSV STK.ID, OCONV(STK.REC<STK.PRICE>, 'MD2'),
      STK.REC<STK.QOH> TO SEQ.F ELSE
    STOP 'Write error'
  END

```

```
END  
REPEAT
```

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)*}

OPENPATH *pathname* {READONLY} {ON ERROR *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.

## Multivalued 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 multivalued operations differs across multivalued 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  
and  
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

<a href="#"><u>CATS()</u></a>	Concatenate elements of a dynamic array
<a href="#"><u>COUNTS()</u></a>	Multivalued variant of <a href="#"><u>COUNT()</u></a>
<a href="#"><u>FIELDS()</u></a>	Multivalued variant of <a href="#"><u>FIELD()</u></a>
<a href="#"><u>FMTS()</u></a>	Format elements of a dynamic array
<a href="#"><u>ICONVS()</u></a>	Perform input conversion on a dynamic array
<a href="#"><u>INDEXS()</u></a>	Multivalued equivalent of <a href="#"><u>INDEX()</u></a>
<a href="#"><u>MATCHES()</u></a>	Multivalued application of pattern matching ( <a href="#"><u>MATCHES</u></a> operator)
<a href="#"><u>NUMS()</u></a>	Multivalued variant of <a href="#"><u>NUM()</u></a>
<a href="#"><u>OCONVS()</u></a>	Perform output conversion on a dynamic array
<a href="#"><u>SPACES()</u></a>	Multivalued variant of <a href="#"><u>SPACE()</u></a>
<a href="#"><u>STRS()</u></a>	Multivalued variant of <a href="#"><u>STR()</u></a>
<a href="#"><u>SUBSTRINGS()</u></a>	Multivalued substring extraction
<a href="#"><u>TRIMBS()</u></a>	Multivalued variant of <a href="#"><u>TRIMB()</u></a>
<a href="#"><u>TRIMFS()</u></a>	Multivalued variant of <a href="#"><u>TRIMF()</u></a>
<a href="#"><u>TRIMS()</u></a>	Multivalued variant of <a href="#"><u>TRIM()</u></a>

For compatibility with other multivalued products, QM supports four multivalued arithmetic functions that are equivalent to the implied value by value execution of the arithmetic operators. For

example,

$$X = \text{ADDS}(A, B)$$

is equivalent to

$$X = A + B$$

<b>ADDS()</b>	Add corresponding elements of two dynamic arrays
<b>DIVS()</b>	Divide corresponding elements of two dynamic arrays
<b>MULS()</b>	Multiply corresponding elements of two dynamic arrays
<b>SUBS()</b>	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  $11_{\text{FM}}0_{\text{VM}}17_{\text{VM}}\text{PQR}_{\text{FM}2}$   
and B contains  $12_{\text{FM}}0_{\text{VM}}14_{\text{VM}}\text{ACB}_{\text{FM}2}$

$$C = \text{GTS}(A, B)$$

Returns C as  $0_{\text{FM}}0_{\text{VM}}1_{\text{VM}}1_{\text{FM}}0$

The multivalued logical functions are

<b><u>ANDSQ</u></b>	Multivalued logical AND
<b><u>EQSQ</u></b>	Multivalued equality test
<b><u>GESQ</u></b>	Multivalued greater than or equal to test
<b><u>GTSQ</u></b>	Multivalued greater than test
<b><u>LESQ</u></b>	Multivalued less than test
<b><u>LTSQ</u></b>	Multivalued less than or equal to test
<b><u>NESQ</u></b>	Multivalued inequality test
<b><u>NOTSQ</u></b>	Multivalued logical NOT
<b><u>ORSQ</u></b>	Multivalued logical OR

The **IFSQ** function returns a dynamic array constructed from elements chosen from two other dynamic arrays depending on the content of a third dynamic array.

$$\text{IFS}(\text{control.array}, \text{true.array}, \text{false.array})$$

where

*control.array* is a dynamic array of true / false values.

<i>true.array</i>	holds values to be returned where the corresponding element of <i>control.array</i> is true.
<i>false.array</i>	holds values to be returned where the corresponding element of <i>control.array</i> 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 1 0 0 1 1 1 0

B contains 6 2 3 4 9 6 3

C contains 2 8 5 0 3 1 3

D = IFS(A, B, C)

D now contains 6 8 5 4 9 6 3

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)))

## 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 [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, **instantiated** by use of the [OBJECT\(\)](#) function

```
OBJ = OBJECT("MYCLASS")
```

where "MYCLASS" is the catalogue name of the class module. The OBJ variable becomes a reference to an **instance** of the class.

A second use of the [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.

In other program types, data is stored either in local variables that are discarded on return from the program, or in common blocks that persist and may be shared by many programs. A class module has the additional concept of persistent data that is related to the particular instance of the object and is preserved across repeated entry to the object. If an object is instantiated more than once, each instantiation has its own version of the persistent data.

Persistent data is defined using the [PRIVATE](#) or [PUBLIC](#) statements:

```
PRIVATE A, B(5)
PUBLIC C, D(2,3)
```

These statements must appear at the start of the class module, before any executable program statements. Data items defined as private are only accessible by program statements within the class module. Data items defined as public can be accessed from outside of the class module (subject to rules set out below). 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.

**PRIVATE** and **PUBLIC** variables are set to unassigned when the object is instantiated.

### 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

```
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

```

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,

```

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 )
```

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.

All object, property and public routine names are case insensitive.

### 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
```

### 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 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](#).

## Syntax Summary

```
CLASS name {INHERITS class1, class2...}
  PUBLIC A {READONLY}, 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
```

---

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](#).

## 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.

### 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 193.118.13.11) 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 fulfil 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.

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.SOCKET\(\)](#) 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\(\)](#).

```
DATA = READ.SOCKET(SKT, 100, SKT$BLOCKING, 0)
N = 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 QMBasic variables, if a program terminates and a socket variable is discarded, the socket will be closed automatically.

```
CLOSE.SOCKET SKT
CLOSE.SOCKET SRVR.SKT
```

The client program that wants to connect to this server would be of the form

```
SKT = OPEN.SOCKET("193.118.13.14", 4000, SKT$BLOCKING)
IF STATUS() THEN STOP 'Cannot open socket'
N = WRITE.SOCKET(SKT, DATA, 0, 0)
REPLY = READ.SOCKET(SKT, 100, SKT$BLOCKING, 0)
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:

```
SKT = CREATE.SERVER.SOCKET("", 4000, SKT$DGRM+ SKT$UDP)
IF STATUS() THEN STOP 'Cannot initialise socket'
DATA = READ.SOCKET(SKT, 100, SKT$BLOCKING, 0)
N = WRITE.SOCKET(SKT, DATA, 0, 0)
CLOSE.SOCKET SKT
```

The client program becomes:

```
SKT = OPEN.SOCKET("193.118.13.14", 4000, SKT$DGRM + SKT$UDP +
SKT$BLOCKING)
IF STATUS() THEN STOP 'Cannot open socket'
N = WRITE.SOCKET(SKT, DATA, 0, 0)
REPLY = READ.SOCKET(SKT, 100, SKT$BLOCKING, 0)
```

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.

#### 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\(\)](#)

## 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](#)

[\\$DEBUG](#)

[\\$DEFINE](#)

[\\$EXECUTE](#)

[\\$IFDEF](#) and [\\$IFNDEF](#) (also [\\$ELSE](#) and [\\$ENDIF](#))

[\\$INCLUDE](#)

[\\$INSERT](#)

[\\$LIST](#)

[\\$MODE](#)

[\\$NO.CATALOGUE](#)

[\\$NO.XREF](#)

[\\$NOCASE.STRINGS](#)

[\\$PAGE](#)

[\\$QMCALL](#)

[\\$STOP](#)

## **\$CATALOGUE compiler directive**

The **\$CATALOGUE** directive (or the American spelling **\$CATALOG**) causes automatic cataloguing of a program after successful compilation.

### **Format**

```
$CATALOGUE name {GLOBAL | 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 **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](#)

## **\$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.

If the defined *name* is also a built-in function or statement, that function or statement becomes inaccessible.

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, QMMAC, QMAIX or QMPDA 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](#)

## **\$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 program 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.

**\$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](#)

## **\$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 dynamic 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 compiler will always look in SYSCOM if no file name is given and the include record is not found in the same file as the source program.

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 must not contain any 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 multivalued 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.
CASE.SENSITIVE	Enables case sensitivity for names of labels, variables and user defined functions.
COMPATIBLE.APPEND	Modifies the behaviour of the append modes of the <a href="#">S&lt;f,v,sv&gt;</a> assignment operator, the <a href="#">INS</a> statement and the <a href="#">INSERT()</a> and <a href="#">REPLACE()</a> functions to match that of other multivalued products.
COMPOSITE.READNEXT	Modifies how a <a href="#">READNEXT</a> statement handles exploded select lists.
CONDITIONAL.STATEMENTS	Allow most statements that have a <b>THEN/ELSE</b> clause to be used as conditional elements in <a href="#">WHILE</a> or <a href="#">UNTIL</a> .
DEFAULT.UNASS.ARGS	Enables substitution of a default value for unassigned arguments in a <a href="#">SUBROUTINE</a> or <a href="#">FUNCTION</a> statement.
FOR.STORE.BEFORE.TEST	Stores the new value of the control variable in a <a href="#">FOR/NEXT</a> construct before testing the end condition.
HEADING.NO.EJECT	Makes the <b>NO.EJECT</b> mode of the QMBasic <a href="#">HEADING</a> statement the default, suppressing the automatic page throw on setting a new heading.
IMPLIED.STOP	The QMBasic compiler normally inserts an implied <a href="#">RETURN</a> at the end of a program, subroutine or function. Setting this option inserts an implied <a href="#">STOP</a> for compatibility with other multivalued products. Class modules always insert a <a href="#">RETURN</a> regardless of the setting of this option.
NO.ECHO.DATA	Suppresses echo of data for <a href="#">INPUT</a> if it is taken from the <a href="#">DATA</a> queue.
OPTIONAL.FINAL.END	Suppresses the warning message if there is no <a href="#">END</a> statement at the end of the program.
PICK.ENTER	Pick style processing of <a href="#">ENTER</a> .

PICK.ERRMSG	Pick style syntax for <a href="#">STOP</a> and <a href="#">ABORT</a> .
PICK.JUMP.RANGE	Causes the <a href="#">ON GOSUB</a> and <a href="#">ON GOTO</a> statements to continue at the next statement if the index value is out of range.
PICK.MATRIX	Create Pick style matrices. See the <a href="#">COMMON</a> and <a href="#">DIMENSION</a> statements for a discussion of the implications of this mode.
PICK.READ	Causes <a href="#">READ</a> , <a href="#">READL</a> , <a href="#">READU</a> , <a href="#">READV</a> , <a href="#">READVU</a> and <a href="#">READVL</a> statements to take on the Pick style behaviour in which the target variable is left unchanged if the record is not found.
PICK.SELECT	Changes the behaviour of <a href="#">SELECTV</a> (and <a href="#">SELECT</a> when <code>\$MODE SELECTV</code> 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 <a href="#">Assignment Statements</a> . Setting this mode overrides the PICK.SUBSTR mode.
PRCLOSE.DEFAULT.0	<a href="#">PRINTER CLOSE</a> defaults to printer 0 rather than closing all printers.
SELECTV	Changes the action of <a href="#">SELECT</a> to be as for <a href="#">SELECTV</a> .
SSELECTV	Changes the action of <a href="#">SSELECT</a> to be as for <a href="#">SSELECTV</a> .
STDFIL	Enables use of the <a href="#">default file variable</a> .
STDFIL.SHARED	Enables use of the shared default file variable.
STRING.LOCATE	Numeric data in right aligned <a href="#">LOCATE</a> is not treated as a special case.
TRAP.UNUSED	Displays a warning about variables that are assigned a value but never used.
UNASSIGNED.COMMON	Variables in <a href="#">common</a> blocks are created unassigned instead of being initialised to zero.
UV.LOCATE	UniVerse Ideal / Reality flavour style <a href="#">LOCATE</a> .

Prefixing a mode name (other than DEFAULT) with a minus sign turns off the named option.

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.

## Examples

```
$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**

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 multivalued function equivalents ([EQS\(\)](#), [NES\(\)](#), [LTS\(\)](#), [GES\(\)](#), [LES\(\)](#), [GES\(\)](#)), the [CHANGE\(\)](#), [CONVERT\(\)](#), [COUNT\(\)](#), [DCOUNT\(\)](#), [FIELD\(\)](#), [INDEX\(\)](#) and [SWAP\(\)](#) functions and the [CONVERT](#), [FIND](#), [FINDSTR](#) and [LOCATE](#) statements.

## **\$PAGE Compiler Directive**

The **\$PAGE** directive inserts a page break in the compiler listing record.

### **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 via QMClient.

### **Format**

**\$QMCALL**

QMClient includes the ability to restrict which catalogued subroutines may be called directly by the client interface.

For processes running with the [QMCLIENT](#) configuration parameter set to 2, this directive makes the subroutine in which it appears available for calling using the QMClient API [QMCall](#) function. This subroutine may then call other subroutines 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**

```
$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**

```
$IFDEF QM  
  $STOP Program not applicable to QM  
$ENDIF
```

The above example appears in the QMSAVE program in the BP file of the QMSYS account. This program is issued in source form for use in environments other than QM.

## 6.3 QMBasic Limits

Number of local variables or elements in a matrix  
Dependant on system swap file size.

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 though long string constants are broken into fragments of no more than 255 characters by the compiler and reassembled at run time.

Maximum arguments to a subroutine  
255

Maximum labels in an ON GOSUB or ON GOTO  
65535

Maximum characters in a catalogue call name  
63 (32 for trigger functions)

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

### Program Structure, Declaration and Assignment

<a href="#"><u>CLASS</u></a>	Declare a class module
<a href="#"><u>CLEAR</u></a>	Set all local variables to zero
<a href="#"><u>CLEARCOMMON</u></a>	Set all unnamed common variables to zero
<a href="#"><u>COMMON</u></a>	Define a common block
<a href="#"><u>DEFFUN</u></a>	Define a function
<a href="#"><u>DIM</u></a>	Synonym for DIMENSION
<a href="#"><u>DIMENSION</u></a>	Set matrix dimensions
<a href="#"><u>DISINHERIT</u></a>	Disinherit an object
<a href="#"><u>END</u></a>	Terminate program or statement group
<a href="#"><u>EQUATE</u></a>	Define a symbolic name for a constant or matrix element
<a href="#"><u>FUNCTION</u></a>	Declare function name and arguments
<a href="#"><u>GET</u></a>	Synonym for PUBLIC FUNCTION
<a href="#"><u>INHERIT</u></a>	Inherit an object
<a href="#"><u>MAT</u></a>	Matrix initialisation or copy
<a href="#"><u>PRIVATE</u></a>	Declare private variables in a local subroutine or a class module
<a href="#"><u>PROGRAM</u></a>	Declare program name
<a href="#"><u>PUBLIC</u></a>	Declare public properties in a class module
<a href="#"><u>PUT</u></a>	Synonym for PUBLIC SUBROUTINE
<a href="#"><u>SUBROUTINE</u></a>	Declare subroutine name and arguments
<a href="#"><u>VOID</u></a>	Discard the result of evaluating an expression

### Program Control

<a href="#"><u>ABORT</u></a>	Abort to command prompt
<a href="#"><u>ABORTE</u></a>	Abort to command prompt with Pick style message handling
<a href="#"><u>ABORTM</u></a>	Abort to command prompt with Information style message handling
<a href="#"><u>CALL</u></a>	Call an external subroutine
<a href="#"><u>CASE</u></a>	Perform statements according to multiple conditions
<a href="#"><u>CHAIN</u></a>	Terminate program and execute a command
<a href="#"><u>CONTINUE</u></a>	Continue next iteration of a loop
<a href="#"><u>ENTER</u></a>	Synonym for CALL
<a href="#"><u>EXECUTE</u></a>	Execute a command
<a href="#"><u>EXIT</u></a>	Leave a loop
<a href="#"><u>FOR / NEXT</u></a>	Iterative loop construct
<a href="#"><u>GET(ARG.)</u></a>	Retrieve command line arguments
<a href="#"><u>GO / GOTO</u></a>	Jump to a label
<a href="#"><u>GOSUB</u></a>	Enter an internal subroutine
<a href="#"><u>IF / THEN / ELSE</u></a>	Perform conditional statements
<a href="#"><u>LOCAL</u></a>	Declares an internal subroutine or function that has private local variables
<a href="#"><u>LOOP / REPEAT</u></a>	Define a loop to be repeated
<a href="#"><u>NAP</u></a>	Suspend program for a short period
<a href="#"><u>ON GOSUB</u></a>	Jump to one of a list of labels selected by value
<a href="#"><u>ON GOTO</u></a>	Enter one of a list of internal subroutines selected by value
<a href="#"><u>OS.EXECUTE</u></a>	Execute an operating system command
<a href="#"><u>PAUSE</u></a>	Pause execution until awoken by another process
<a href="#"><u>PERFORM</u></a>	Synonym for EXECUTE
<a href="#"><u>REMOVE.BREAK.HANDLER</u></a>	Deactivate a break handler subroutine
<a href="#"><u>RETURN</u></a>	Return from CALL or GOSUB

<a href="#"><u>RETURN FROM PROGRAM</u></a>	Return from CALL
<a href="#"><u>RETURN TO</u></a>	Return from program or subroutine to a specific label
<a href="#"><u>RQM</u></a>	Synonym for SLEEP
<a href="#"><u>SET.BREAK.HANDLER</u></a>	Establish a break handler subroutine
<a href="#"><u>SLEEP</u></a>	Suspend program to / for given time
<a href="#"><u>STOP</u></a>	Terminate program
<a href="#"><u>STOPE</u></a>	Terminate program with Pick style message handling
<a href="#"><u>STOPM</u></a>	Terminate program with Information style message handling
<a href="#"><u>SUBR()</u></a>	Call a subroutine as a function
<a href="#"><u>UNTIL</u></a>	Leave loop if condition is met
<a href="#"><u>WAKE</u></a>	Restart execution of a process on a PAUSE
<a href="#"><u>WHILE</u></a>	Leave loop unless condition is met

### Mathematical and Logical Functions

<a href="#"><u>ABS()</u></a>	Absolute value
<a href="#"><u>ABSS()</u></a>	Multivalued absolute value
<a href="#"><u>ACOS()</u></a>	Arc-cosine
<a href="#"><u>ANDS()</u></a>	Multivalued logical AND
<a href="#"><u>ASIN()</u></a>	Arc-sine
<a href="#"><u>ATAN()</u></a>	Arc-tangent
<a href="#"><u>BITAND()</u></a>	Bitwise logical AND operation
<a href="#"><u>BITNOT()</u></a>	Bitwise logical NOT operation
<a href="#"><u>BITOR()</u></a>	Bitwise logical OR operation
<a href="#"><u>BITRESET()</u></a>	Turn off specified bit
<a href="#"><u>BITSET()</u></a>	Turn on specified bit
<a href="#"><u>BITTEST()</u></a>	Test specified bit
<a href="#"><u>BITXOR()</u></a>	Bitwise logical exclusive OR operation
<a href="#"><u>COS()</u></a>	Cosine
<a href="#"><u>DIV()</u></a>	Divide
<a href="#"><u>EQS()</u></a>	Multivalued equality test
<a href="#"><u>EXP()</u></a>	Exponential
<a href="#"><u>GES()</u></a>	Multivalued greater than or equal to test
<a href="#"><u>GTS()</u></a>	Multivalued greater than test
<a href="#"><u>IDIV()</u></a>	Integer division
<a href="#"><u>IFS()</u></a>	Multivalued conditional expression
<a href="#"><u>INT()</u></a>	Truncate value to integer
<a href="#"><u>LES()</u></a>	Multivalued less than test
<a href="#"><u>LN()</u></a>	Natural log
<a href="#"><u>LTS()</u></a>	Multivalued less than or equal to test
<a href="#"><u>MAX()</u></a>	Returns the greater of two values
<a href="#"><u>MAXIMUM()</u></a>	Find the greatest value in a dynamic array
<a href="#"><u>MIN()</u></a>	Returns the lesser of two values
<a href="#"><u>MINIMUM()</u></a>	Find the lowest value in a dynamic array
<a href="#"><u>MOD()</u></a>	Modulus value from division
<a href="#"><u>MODS()</u></a>	Multivalued modulus value from division
<a href="#"><u>NEG()</u></a>	Arithmetic inverse
<a href="#"><u>NEGS()</u></a>	Multivalued arithmetic inverse
<a href="#"><u>NES()</u></a>	Multivalued inequality test
<a href="#"><u>NOT()</u></a>	Logical NOT
<a href="#"><u>NOTS()</u></a>	Multivalued logical NOT
<a href="#"><u>ORS()</u></a>	Multivalued logical OR
<a href="#"><u>PWR()</u></a>	Raise value to power
<a href="#"><u>RDIV()</u></a>	Rounded integer division

<a href="#">RANDOMIZE</a>	Set random number seed value
<a href="#">REM()</a>	Remainder value from division
<a href="#">REUSE()</a>	Reuse element of numeric arrays in mathematical functions
<a href="#">RND()</a>	Generate random number
<a href="#">ROUNDDOWN()</a>	Round a number towards zero in a specified increment
<a href="#">ROUNDUP()</a>	Round a number away from zero in a specified increment
<a href="#">SHIFT()</a>	Perform bit shift
<a href="#">SIN()</a>	Sine
<a href="#">SQRT()</a>	Square root
<a href="#">SUM()</a>	Sum lowest level elements of a numeric array
<a href="#">SUMMATION()</a>	Sum all elements of a numeric array
<a href="#">TAN()</a>	Tangent

### String Handling

<a href="#">ALPHA()</a>	Test if string holds only alphabetic characters
<a href="#">ASCII()</a>	Convert an EBCDIC string to ASCII
<a href="#">CATS()</a>	Concatenate elements of a dynamic array
<a href="#">CHANGE()</a>	Replace substring in a string
<a href="#">CHAR()</a>	Get ASCII character for a given collating sequence value
<a href="#">COL1()</a>	Start of substring position from FIELD()
<a href="#">COL2()</a>	End of substring position from FIELD()
<a href="#">COMPARE()</a>	Compare strings
<a href="#">COMPARES()</a>	Multivalued variant of COMPARE()
<a href="#">CONVERT</a>	Substitute characters with replacements
<a href="#">CONVERT()</a>	Substitute characters with replacements
<a href="#">COUNT()</a>	Count occurrences of substring in string
<a href="#">COUNTS()</a>	Multivalued variant of COUNT()
<a href="#">CROP()</a>	Remove redundant mark characters
<a href="#">CSV.DQ()</a>	Dequote a CSV string
<a href="#">CSV.MODE</a>	Set CSV conversion mode
<a href="#">DCOUNT()</a>	Count delimited substrings in string
<a href="#">DECRYPT()</a>	Decrypt text
<a href="#">DEL</a>	Delete a field, value or subvalue
<a href="#">DELETE()</a>	Delete a field, value or subvalue
<a href="#">DOWNCASE()</a>	Convert string to lowercase
<a href="#">DPARSE</a>	Split elements of a delimited string
<a href="#">DPARSE.CSV</a>	Split elements of a CSV format delimited string
<a href="#">DQUOTE()</a>	Synonym for QUOTE()
<a href="#">DQUOTES()</a>	Synonym for QUOTES()
<a href="#">EBCDIC()</a>	Convert an ASCII string to EBCDIC
<a href="#">ENCRYPT()</a>	Encrypt data
<a href="#">EXTRACT()</a>	Extract a field, value or subvalue
<a href="#">FIELD()</a>	Extract delimited fields
<a href="#">FIELDS()</a>	Multivalued variant of FIELD()
<a href="#">FIELDSTORE()</a>	Replace or insert delimited fields
<a href="#">FIND</a>	Find a string in a dynamic array element
<a href="#">FINDSTR</a>	Find a substring in a dynamic array element
<a href="#">FMT()</a>	Format a string
<a href="#">FMTS()</a>	Format a dynamic array
<a href="#">FOLD()</a>	Break a string into sections, splitting at spaces where possible
<a href="#">FOLDS()</a>	Multivalued variant of FOLD()
<a href="#">GETREM()</a>	Get remove pointer position
<a href="#">ICONV()</a>	Perform input conversion

<a href="#"><u>ICONVS()</u></a>	Perform input conversion on a dynamic array
<a href="#"><u>INDEX()</u></a>	Locate occurrence of substring within a string
<a href="#"><u>INDEXS()</u></a>	Multivalued equivalent of INDEX()
<a href="#"><u>INS</u></a>	Insert a field, value or subvalue
<a href="#"><u>INSERT()</u></a>	Insert a field, value or subvalue
<a href="#"><u>LEN()</u></a>	Return length of a string
<a href="#"><u>LENS()</u></a>	Multivalued equivalent of LEN()
<a href="#"><u>LISTINDEX()</u></a>	Return position of an item in a delimited list
<a href="#"><u>LOCATE</u></a>	Locate string in dynamic array
<a href="#"><u>LOCATE()</u></a>	Locate string in dynamic array
<a href="#"><u>LOWER()</u></a>	Convert delimiters to lower level
<a href="#"><u>MATBUILD</u></a>	Build a dynamic array from matrix elements
<a href="#"><u>MATCHES()</u></a>	Matches a string against a pattern template
<a href="#"><u>MATCHESS()</u></a>	Matches each element of a dynamic array against a pattern template
<a href="#"><u>MATCHFIELD()</u></a>	Return portion of string matching pattern
<a href="#"><u>MATPARSE</u></a>	Break a dynamic array into matrix elements
<a href="#"><u>MD5()</u></a>	Convert a string to its 32 digit message digest value.
<a href="#"><u>NUM()</u></a>	Test if string holds a numeric value
<a href="#"><u>NUMS()</u></a>	Multivalued variant of NUM()
<a href="#"><u>OCONV()</u></a>	Perform output conversion
<a href="#"><u>OCONVS()</u></a>	Perform output conversion on a dynamic array
<a href="#"><u>QUOTE()</u></a>	Enclose a string in double quotes
<a href="#"><u>QUOTES()</u></a>	Enclose each element of a dynamic array in double quotes
<a href="#"><u>RAISE()</u></a>	Convert delimiters to higher level
<a href="#"><u>REMOVE</u></a>	Remove an item from a dynamic array
<a href="#"><u>REMOVE()</u></a>	Remove an item from a dynamic array
<a href="#"><u>REMOVEF()</u></a>	Extract data from a delimited character string
<a href="#"><u>REPLACE()</u></a>	Replace a field, value or subvalue
<a href="#"><u>SEQ()</u></a>	Get collating sequence value for a given ASCII character
<a href="#"><u>SETREM</u></a>	Set remove pointer position
<a href="#"><u>SORT.COMPARE()</u></a>	Compare items according to sort rules
<a href="#"><u>SOUNDEX()</u></a>	Form a soundex code value for a string
<a href="#"><u>SOUNDEXS()</u></a>	Multivalued variant of SOUNDEX()
<a href="#"><u>SPACE()</u></a>	Create a string of spaces
<a href="#"><u>SPACES()</u></a>	Multivalued variant of SPACE()
<a href="#"><u>SPLICE()</u></a>	Concatenates elements of two dynamic arrays, inserting a string between the items.
<a href="#"><u>SQUOTE()</u></a>	Enclose a string in single quotes
<a href="#"><u>SQUOTES()</u></a>	Enclose each element of a dynamic array in single quotes
<a href="#"><u>STR()</u></a>	Create a string from a repeated substring
<a href="#"><u>STRS()</u></a>	Multivalued variant of STR()
<a href="#"><u>SUBSTITUTE()</u></a>	Multivalued substring replacement
<a href="#"><u>SUBSTRINGS()</u></a>	Multivalued substring extraction
<a href="#"><u>SWAP()</u></a>	Synonym for CHANGE()
<a href="#"><u>SWAPCASE()</u></a>	Invert case of alphabetic characters in a string
<a href="#"><u>TRIM()</u></a>	Trim characters from string
<a href="#"><u>TRIMB()</u></a>	Trim spaces from back of string
<a href="#"><u>TRIMBS()</u></a>	Multivalued variant of TRIMB()
<a href="#"><u>TRIMF()</u></a>	Trim spaces from front of string
<a href="#"><u>TRIMFS()</u></a>	Multivalued variant of TRIMF()
<a href="#"><u>TRIMS()</u></a>	Multivalued variant of TRIM()
<a href="#"><u>UPCASE()</u></a>	Convert string to uppercase
<a href="#"><u>VSLICE()</u></a>	Extract a value or subvalue slice from a dynamic array

**Date and Time**

<a href="#">DATE()</a>	Return the current date as a day number
<a href="#">EPOCH()</a>	Return time and date as an epoch value
<a href="#">MVDATE()</a>	Extract the multivalued style date from an epoch value
<a href="#">MVDATE.TIME()</a>	Extract the multivalued style date and time from an epoch value
<a href="#">MVEPOCH()</a>	Convert a multivalued style date and time to an epoch value
<a href="#">MVTIME()</a>	Extract the multivalued style time from an epoch value
<a href="#">SET.TIMEZONE</a>	Set time zone for use by the epoch conversion code
<a href="#">TIME()</a>	Return the current time
<a href="#">TIMEDATE()</a>	Return the date and time as a string

**File Handling**

<a href="#">BEGIN TRANSACTION</a>	Start a new transaction
<a href="#">CLEARFILE</a>	Clear a file, deleting all records and releasing disk space
<a href="#">CLOSE</a>	Close a file
<a href="#">CLOSESEQ</a>	Close a record opened for sequential access
<a href="#">COMMIT</a>	Commit transaction updates
<a href="#">CREATE</a>	Create an empty sequential file record
<a href="#">CREATE.FILE</a>	Create a file
<a href="#">DELETE</a>	Delete record from a file
<a href="#">DELETESEQ</a>	Delete an operating system file
<a href="#">DELETEU</a>	Delete record from a file preserving locks
<a href="#">DFPART()</a>	Access a part file within a distributed file
<a href="#">DIR()</a>	Return the contents of a directory
<a href="#">END TRANSACTION</a>	Terminate a transaction
<a href="#">FCONTROL()</a>	Perform control action on an open file
<a href="#">FILE</a>	Open a file and access data by field name
<a href="#">FILE.EVENT()</a>	Create a file event monitoring variable
<a href="#">FILEINFO()</a>	Return information about an open file
<a href="#">FILELOCK</a>	Lock a file
<a href="#">FILEUNLOCK</a>	Unlock a file
<a href="#">FLUSH</a>	Flush sequential file data to disk
<a href="#">FLUSH.DH.CACHE</a>	Flush dynamic file cache
<a href="#">GET.PORT.PARAMS()</a>	Get serial port parameters
<a href="#">MARK.MAPPING</a>	Control field mark translation in directory files
<a href="#">MATREAD</a>	Read a record, parsing into a matrix
<a href="#">MATREADCSV</a>	Read a CSV format text item into a matrix
<a href="#">MATREADL</a>	Read a record setting a read lock, parsing into a matrix
<a href="#">MATREADU</a>	Read a record setting an update lock, parsing into a matrix
<a href="#">MATWRITE</a>	Write a record from matrix elements
<a href="#">MATWRITEU</a>	Write a record from matrix elements, retaining any lock
<a href="#">NOBUF</a>	Turn off buffering for a record opened using OPENSEQ
<a href="#">OPEN</a>	Open a file
<a href="#">OPENPATH</a>	Open a file by pathname
<a href="#">OPENSEQ</a>	Open a record for sequential access
<a href="#">OSDELETE</a>	Delete a file by pathname
<a href="#">OSREAD</a>	Read a file by pathname
<a href="#">OSWRITE</a>	Write a file by pathname
<a href="#">OUTERJOIN()</a>	Fetch data from a file using an "outer join"
<a href="#">READ</a>	Read a record from a file
<a href="#">READBLK</a>	Read bytes from a sequential file

<a href="#">READCSV</a>	Read a CSV format text item
<a href="#">READL</a>	Read a record from a file, setting a read lock
<a href="#">READSEQ</a>	Read from a sequential file
<a href="#">READU</a>	Read a record from a file, setting an update lock
<a href="#">READV</a>	Read a field from a record in a file
<a href="#">READVL</a>	Read a field from a record in a file, setting a read lock
<a href="#">READVU</a>	Read a field from a record in a file, setting an update lock
<a href="#">RECORDLOCKED()</a>	Test if record is locked
<a href="#">RECORDLOCKL</a>	Set a read lock on a record
<a href="#">RECORDLOCKU</a>	Set an update lock on a record
<a href="#">RELEASE</a>	Release record or file locks
<a href="#">ROLLBACK</a>	Discard transaction updates
<a href="#">RTRANS()</a>	Fetch data from a file
<a href="#">SEEK</a>	Position a sequential file
<a href="#">SET.PORT.PARAMS()</a>	Set serial port parameters
<a href="#">STATUS</a>	Returns a dynamic array of information about an open file
<a href="#">TIMEOUT</a>	Sets a timeout for READBLK and READSEQ
<a href="#">TRANS()</a>	Fetch data from a file
<a href="#">TRANSACTION ABORT</a>	Abort a transaction
<a href="#">TRANSACTION COMMIT</a>	Commit a transaction
<a href="#">TRANSACTION START</a>	Start a new transaction
<a href="#">VOCPATH()</a>	Resolve a filename to its corresponding pathname
<a href="#">WAIT.FILE.EVENT()</a>	Wait for a file monitoring event to occur
<a href="#">WEOFSEQ</a>	Write end of file position to sequential file
<a href="#">WRITE</a>	Write a record to a file
<a href="#">WRITEBLK</a>	Write bytes to a sequential file
<a href="#">WRITECSV</a>	Write CSV format data to a sequential file
<a href="#">WRITESEQ</a>	Write to sequential file
<a href="#">WRITESEQF</a>	Write to sequential file, flushing to disk
<a href="#">WRITEU</a>	Write a record to a file, retaining any lock
<a href="#">WRITEV</a>	Write a field to a record in a file
<a href="#">WRITEVU</a>	Write a field to a record in a file, retaining any lock
<a href="#">XLATE()</a>	Synonym for TRANS()

### Printer, Keyboard and Display Handling

<a href="#">@(x,y)</a>	Terminal cursor movement and control
<a href="#">BINDKEY()</a>	Set, remove, query, save or restore key bindings
<a href="#">BREAK</a>	Enable or disable break key handling
<a href="#">CLEARDATA</a>	Clear DATA queue
<a href="#">CLEARINPUT</a>	Clear keyboard type-ahead
<a href="#">CRT</a>	Synonym for DISPLAY
<a href="#">DATA</a>	Save text in DATA queue
<a href="#">DISPLAY</a>	Output to the display
<a href="#">ECHO</a>	Enable or disable input echo
<a href="#">ERRMSG</a>	Display a Pick style message from the ERRMSG file
<a href="#">FOOTING</a>	Set footing text
<a href="#">FORMCSV()</a>	Transforms a string to a CSV standard compliant item
<a href="#">GETPU()</a>	Get a characteristic of a print unit
<a href="#">HEADING</a>	Set heading text
<a href="#">HUSH</a>	Suppress or enable display output
<a href="#">IN</a>	Read a single byte from the terminal with an optional timeout
<a href="#">INPUT</a>	Input a string from the keyboard or data queue
<a href="#">INPUT @</a>	Input a string from the keyboard or data queue

<a href="#"><u>INPUTCLEAR</u></a>	Synonym for CLEARINPUT
<a href="#"><u>INPUTCSV</u></a>	Input CSV format data
<a href="#"><u>INPUTERR</u></a>	Synonym for PRINTERR
<a href="#"><u>INPUTFIELD</u></a>	Input a string with function key handling
<a href="#"><u>KEYCODE()</u></a>	Input a single keystroke from the keyboard with terminfo translation
<a href="#"><u>KEYEDIT</u></a>	Define editing keys for INPUT @
<a href="#"><u>KEYEXIT</u></a>	Define exit keys for INPUT @
<a href="#"><u>KEYIN()</u></a>	Input a single keystroke from the keyboard
<a href="#"><u>KEYINC()</u></a>	Input a single keystroke from the keyboard with case inversion
<a href="#"><u>KEYINR()</u></a>	Input a single keystroke from the keyboard in raw mode (no internal processing)
<a href="#"><u>KEYREADY()</u></a>	Test for keyboard input
<a href="#"><u>KEYTRAP</u></a>	Define trap keys for INPUT @
<a href="#"><u>PAGE</u></a>	Start a new page
<a href="#"><u>PRINT</u></a>	Output to a logical print unit
<a href="#"><u>PRINTER CLOSE</u></a>	Close a print unit
<a href="#"><u>PRINTER DISPLAY</u></a>	Associate a print unit with the display
<a href="#"><u>PRINTER FILE</u></a>	Associate a file with a print unit
<a href="#"><u>PRINTER NAME</u></a>	Associate a print device with a print unit
<a href="#"><u>PRINTER OFF</u></a>	Disable print unit zero
<a href="#"><u>PRINTER ON</u></a>	Enable print unit zero
<a href="#"><u>PRINTER RESET</u></a>	Reset default print unit and display
<a href="#"><u>PRINTER SETTING</u></a>	Set a print unit parameter
<a href="#"><u>PRINTER.SETTING()</u></a>	Set or retrieve a print unit parameter
<a href="#"><u>PRINTCSV</u></a>	Print data in comma separated variable format
<a href="#"><u>PRINTERR</u></a>	Display an error message
<a href="#"><u>PROMPT</u></a>	Set the input prompt character
<a href="#"><u>PTERM()</u></a>	Set/clear/query terminal settings
<a href="#"><u>RESTORE.SCREEN</u></a>	Restore screen image data
<a href="#"><u>SAVE.SCREEN()</u></a>	Save screen image data
<a href="#"><u>SETPU</u></a>	Set a characteristic of a print unit
<a href="#"><u>TERMINFO()</u></a>	Retrieve information from the terminfo database
<a href="#"><u>TTYGET()</u></a>	Get current terminal mode settings
<a href="#"><u>TTYSET</u></a>	Set terminal modes

### Select Lists and Alternate Key Indices

<a href="#"><u>CLEARSELECT</u></a>	Clear one or all select lists
<a href="#"><u>DELETELIST</u></a>	Delete a saved select list
<a href="#"><u>FORMLIST</u></a>	Create a numbered select list from a dynamic array
<a href="#"><u>FORMLISTV</u></a>	Create a select list variable from a dynamic array
<a href="#"><u>GETLIST</u></a>	Restore a saved select list
<a href="#"><u>INDICES()</u></a>	Return information about alternate key indices
<a href="#"><u>READLIST</u></a>	Save a select list in a dynamic array
<a href="#"><u>READNEXT</u></a>	Read a record id from a select list
<a href="#"><u>SAVELIST</u></a>	Save a select list in the \$SAVEDLISTS file
<a href="#"><u>SELECT</u></a>	Build a select list of all records in an open file
<a href="#"><u>SELECTE</u></a>	Transfer select list 0 to a select list variable
<a href="#"><u>SELECTINDEX</u></a>	Build a select list from an alternate key index
<a href="#"><u>SELECTINFO()</u></a>	Return information regarding a select list
<a href="#"><u>SELECTLEFT</u></a>	Scan left through an alternate key index
<a href="#"><u>SELECTN</u></a>	Build a numbered select list of all records in an open file
<a href="#"><u>SELECTRIGHT</u></a>	Scan right through an alternate key index

<a href="#"><u>SELECTV</u></a>	Build a select list variable of all records in an open file
<a href="#"><u>SETLEFT</u></a>	Set alternate key index scan position to leftmost
<a href="#"><u>SETRIGHT</u></a>	Set alternate key index scan position to rightmost
<a href="#"><u>SSELECT</u></a>	Build a sorted select list of all records in an open file

### Socket Interface

<a href="#"><u>ACCEPT.SOCKET.CONNECTION()</u></a>	Accept an incoming connection on a server socket
<a href="#"><u>CLOSE.SOCKET</u></a>	Close a socket
<a href="#"><u>CREATE.SERVER.SOCKET()</u></a>	Open a server socket
<a href="#"><u>OPEN.SOCKET()</u></a>	Open a socket connection
<a href="#"><u>READ.SOCKET()</u></a>	Read data from a socket
<a href="#"><u>SELECT.SOCKET()</u></a>	Monitor events on multiple sockets
<a href="#"><u>SERVER.ADDR()</u></a>	Find the IP address for a given server name
<a href="#"><u>SET.SOCKET.MODE()</u></a>	Set mode of a socket
<a href="#"><u>SOCKET.INFO()</u></a>	Retrieve information about a socket
<a href="#"><u>WRITE.SOCKET()</u></a>	Write data to a socket

### Miscellaneous

<a href="#"><u>ARG()</u></a>	Returns an argument variable based on its argument list position
<a href="#"><u>ARG.COUNT()</u></a>	Returns the number of arguments passed into a subroutine
<a href="#"><u>ARG.PRESENT()</u></a>	Test for presence of an argument in a subroutine or function
<a href="#"><u>ASSIGNED()</u></a>	Test whether variable is assigned
<a href="#"><u>CATALOGUED()</u></a>	Check catalogue entry
<a href="#"><u>CHECKSUM()</u></a>	Calculate a checksum value for a supplied data item
<a href="#"><u>CHILD()</u></a>	Test if a phantom process is still running
<a href="#"><u>CONFIG()</u></a>	Returns the value of a configuration parameter
<a href="#"><u>CONNECT.PORT()</u></a>	Connect a serial port to a phantom process
<a href="#"><u>DEBUG</u></a>	Enter debugger
<a href="#"><u>DTX()</u></a>	Convert a number to hexadecimal
<a href="#"><u>ENV()</u></a>	Retrieve an operating system environment variable
<a href="#"><u>GET.MESSAGES()</u></a>	Retrieve messages from the message queue
<a href="#"><u>GETNLS()</u></a>	Get national language support parameter value
<a href="#"><u>INMAT()</u></a>	Return status of matrix operations
<a href="#"><u>ITYPE()</u></a>	Execute a compiled I-type
<a href="#"><u>LOCK</u></a>	Set task lock
<a href="#"><u>LOGMSG</u></a>	Add an entry to the system error log
<a href="#"><u>NULL</u></a>	No operation
<a href="#"><u>OBJECT()</u></a>	Instantiates an object
<a href="#"><u>OBJINFO()</u></a>	Returns information about an object variable
<a href="#"><u>OS.ERROR()</u></a>	Return operating system error information
<a href="#"><u>PRECISION</u></a>	Set number of decimal places in numeric conversion
<a href="#"><u>PROCREAD</u></a>	Read data from the PROC primary input buffer
<a href="#"><u>PROCWRITE</u></a>	Write data to the PROC primary input buffer
<a href="#"><u>REMARK</u></a>	Alternative syntax for comments
<a href="#"><u>SENTENCE()</u></a>	Returns the command line that started the current program
<a href="#"><u>SET.ARG</u></a>	Sets an argument variable based on its argument list position
<a href="#"><u>SET.EXIT.STATUS</u></a>	Set final exit status value
<a href="#"><u>SETNLS()</u></a>	Set national language support parameter value
<a href="#"><u>STATUS()</u></a>	Return status from previous operation
<a href="#"><u>SYSTEM()</u></a>	Return system information
<a href="#"><u>TCLREAD</u></a>	Returns the sentence that started the current program
<a href="#"><u>TESTLOCK</u></a>	Test state of a task lock

[UNASSIGNED\(\)](#)  
[UNLOCK](#)  
[XTD\(\)](#)

Test whether variable is unassigned  
Release task lock  
Convert a hexadecimal number

## 6.5 QMBasic Statements and Functions by Name

<a href="#"><u>@(x,y)</u></a>	Terminal cursor movement and control
<a href="#"><u>ABORT</u></a>	Abort to command prompt
<a href="#"><u>ABORTE</u></a>	Abort to command prompt with Pick style message handling
<a href="#"><u>ABORTM</u></a>	Abort to command prompt with Information style message handling
<a href="#"><u>ABS()</u></a>	Absolute value
<a href="#"><u>ABSS()</u></a>	Multivalued absolute value
<a href="#"><u>ACCEPT.SOCKET.CONNECTION()</u></a>	Accept an incoming connection on a server socket
<a href="#"><u>ACOS()</u></a>	Arc-cosine
<a href="#"><u>ALPHA()</u></a>	Test if string holds only alphabetic characters
<a href="#"><u>ANDS()</u></a>	Multivalued logical AND
<a href="#"><u>ARG()</u></a>	Returns an argument variable based on its argument list position
<a href="#"><u>ARG.COUNT()</u></a>	Returns the number of arguments passed into a subroutine
<a href="#"><u>ARG.PRESENT()</u></a>	Test for presence of an argument in a subroutine or function
<a href="#"><u>ASCII()</u></a>	Convert an EBCDIC string to ASCII
<a href="#"><u>ASIN()</u></a>	Arc-sine
<a href="#"><u>ASSIGNED()</u></a>	Test whether variable is assigned
<a href="#"><u>ATAN()</u></a>	Arc-tangent
<a href="#"><u>BEGIN TRANSACTION</u></a>	Start a new transaction
<a href="#"><u>BINDKEY()</u></a>	Set, remove, query, save or restore key bindings
<a href="#"><u>BITAND()</u></a>	Bitwise logical AND operation
<a href="#"><u>BITNOT()</u></a>	Bitwise logical NOT operation
<a href="#"><u>BITOR()</u></a>	Bitwise logical OR operation
<a href="#"><u>BITRESET()</u></a>	Turn off specified bit
<a href="#"><u>BITSET()</u></a>	Turn on specified bit
<a href="#"><u>BITTEST()</u></a>	Test specified bit
<a href="#"><u>BITXOR()</u></a>	Bitwise logical exclusive OR operation
<a href="#"><u>BREAK</u></a>	Enable or disable break key handling
<a href="#"><u>CALL</u></a>	Call an external subroutine
<a href="#"><u>CASE</u></a>	Perform statements according to multiple conditions
<a href="#"><u>CATALOGUED()</u></a>	Check catalogue entry
<a href="#"><u>CATS()</u></a>	Concatenate elements of a dynamic array
<a href="#"><u>CHAIN</u></a>	Terminate program and execute a command
<a href="#"><u>CHANGE()</u></a>	Replace substring in a string
<a href="#"><u>CHAR()</u></a>	Get ASCII character for a given collating sequence value
<a href="#"><u>CHECKSUM()</u></a>	Calculate a checksum value for a supplied data item
<a href="#"><u>CHILD()</u></a>	Test if a phantom process is still running
<a href="#"><u>CLASS</u></a>	Declare a class module
<a href="#"><u>CLEAR</u></a>	Set all local variables to zero
<a href="#"><u>CLEARCOMMON</u></a>	Set all unnamed common variables to zero
<a href="#"><u>CLEARDATA</u></a>	Clear DATA queue
<a href="#"><u>CLEARFILE</u></a>	Clear a file, deleting all records and releasing disk space
<a href="#"><u>CLEARINPUT</u></a>	Clear keyboard type-ahead
<a href="#"><u>CLEARSELECT</u></a>	Clear one or all select lists
<a href="#"><u>CLOSE</u></a>	Close a file
<a href="#"><u>CLOSESEQ</u></a>	Close a record opened for sequential access
<a href="#"><u>CLOSE.SOCKET</u></a>	Close a socket
<a href="#"><u>COL1()</u></a>	Start of substring position from FIELD()
<a href="#"><u>COL2()</u></a>	End of substring position from FIELD()
<a href="#"><u>COMMIT</u></a>	Commit transaction updates
<a href="#"><u>COMMON</u></a>	Define a common block

<a href="#"><u>COMPARE()</u></a>	Compare strings
<a href="#"><u>COMPARES()</u></a>	Multivalued variant of COMPARE()
<a href="#"><u>CONFIG()</u></a>	Returns the value of a configuration parameter
<a href="#"><u>CONNECT.PORT()</u></a>	Connect a serial port to a phantom process
<a href="#"><u>CONTINUE</u></a>	Continue next iteration of a loop
<a href="#"><u>CONVERT</u></a>	Substitute characters with replacements
<a href="#"><u>CONVERT()</u></a>	Substitute characters with replacements
<a href="#"><u>COS()</u></a>	Cosine
<a href="#"><u>COUNT()</u></a>	Count occurrences of substring in string
<a href="#"><u>COUNTS()</u></a>	Multivalued variant of COUNT()
<a href="#"><u>CREATE</u></a>	Create an empty sequential file record
<a href="#"><u>CREATE.FILE</u></a>	Create a file
<a href="#"><u>CREATE.SERVER.SOCKET()</u></a>	Open a server socket
<a href="#"><u>CROP()</u></a>	Remove redundant mark characters
<a href="#"><u>CRT</u></a>	Synonym for DISPLAY
<a href="#"><u>CSV.DQ()</u></a>	Dequote a CSV string
<a href="#"><u>CSV.MODE</u></a>	Set CSV conversion mode
<a href="#"><u>DATA</u></a>	Save text in DATA queue
<a href="#"><u>DATE()</u></a>	Return the current date as a day number
<a href="#"><u>DCOUNT()</u></a>	Count delimited substrings in string
<a href="#"><u>DEBUG</u></a>	Enter debugger
<a href="#"><u>DECRYPT()</u></a>	Decrypt text
<a href="#"><u>DEFFUN</u></a>	Define a function
<a href="#"><u>DEL</u></a>	Delete a field, value or subvalue
<a href="#"><u>DELETE</u></a>	Delete record from a file
<a href="#"><u>DELETE()</u></a>	Delete a field, value or subvalue
<a href="#"><u>DELETELIST</u></a>	Delete a saved select list
<a href="#"><u>DELETESEQ</u></a>	Delete an operating system file
<a href="#"><u>DELETEU</u></a>	Delete record from a file preserving locks
<a href="#"><u>DFPART()</u></a>	Access a part file within a distributed file
<a href="#"><u>DIM</u></a>	Synonym for DIMENSION
<a href="#"><u>DIMENSION</u></a>	Set matrix dimensions
<a href="#"><u>DISINHERIT</u></a>	Disinherit an object
<a href="#"><u>DISPLAY</u></a>	Output to the display
<a href="#"><u>DIR()</u></a>	Return the contents of a directory
<a href="#"><u>DIV()</u></a>	Divide
<a href="#"><u>DOWNCASE()</u></a>	Convert string to lowercase
<a href="#"><u>DPARSE</u></a>	Split elements of a delimited string
<a href="#"><u>DPARSE.CSV</u></a>	Split elements of a CSV format delimited string
<a href="#"><u>DQUOTE()</u></a>	Synonym for QUOTE()
<a href="#"><u>DQUOTES()</u></a>	Synonym for QUOTES()
<a href="#"><u>DTX()</u></a>	Convert a number to hexadecimal
<a href="#"><u>EBCDIC()</u></a>	Convert an EBCDIC string to ASCII
<a href="#"><u>ECHO</u></a>	Enable or disable input echo
<a href="#"><u>ENCRYPT()</u></a>	Encrypt data
<a href="#"><u>END</u></a>	Terminate program or statement group
<a href="#"><u>END TRANSACTION</u></a>	Terminate a transaction
<a href="#"><u>ENTER</u></a>	Synonym for CALL
<a href="#"><u>ENV()</u></a>	Retrieve an operating system environment variable
<a href="#"><u>EPOCH()</u></a>	Return time and date as an epoch value
<a href="#"><u>EQS()</u></a>	Multivalued equality test
<a href="#"><u>EQUATE</u></a>	Define a symbolic name for a constant or matrix element
<a href="#"><u>ERRMSG</u></a>	Display a Pick style message from the ERRMSG file
<a href="#"><u>EXECUTE</u></a>	Execute a command

<a href="#"><u>EXIT</u></a>	Leave a loop
<a href="#"><u>EXP()</u></a>	Exponential
<a href="#"><u>EXTRACT()</u></a>	Extract a field, value or subvalue
<a href="#"><u>FCONTROL()</u></a>	Perform control action on an open file
<a href="#"><u>FIELD()</u></a>	Extract delimited fields
<a href="#"><u>FIELDS()</u></a>	Multivalued variant of FIELD()
<a href="#"><u>FIELDSTORE()</u></a>	Replace or insert delimited fields
<a href="#"><u>FILE</u></a>	Open a file and access data by field name
<a href="#"><u>FILE.EVENT()</u></a>	Create a file event monitoring variable
<a href="#"><u>FILEINFO()</u></a>	Return information about an open file
<a href="#"><u>FILELOCK</u></a>	Lock a file
<a href="#"><u>FILEUNLOCK</u></a>	Unlock a file
<a href="#"><u>FIND</u></a>	Find a string in a dynamic array element
<a href="#"><u>FINDSTR</u></a>	Find a substring in a dynamic array element
<a href="#"><u>FLUSH</u></a>	Flush sequential file data to disk
<a href="#"><u>FLUSH.DH.CACHE</u></a>	Flush dynamic file cache
<a href="#"><u>FMT()</u></a>	Format a string
<a href="#"><u>FMTS()</u></a>	Format a dynamic array
<a href="#"><u>FOLD()</u></a>	Break a string into sections, splitting at spaces where possible
<a href="#"><u>FOLDS()</u></a>	Multivalued variant of FOLD()
<a href="#"><u>FOOTING</u></a>	Set footing text
<a href="#"><u>FOR / NEXT</u></a>	Iterative loop construct
<a href="#"><u>FORMCSV()</u></a>	Transforms a string to a CSV standard compliant item
<a href="#"><u>FORMLIST</u></a>	Create a numbered select list from a dynamic array
<a href="#"><u>FORMLISTV</u></a>	Create a select list variable from a dynamic array
<a href="#"><u>FUNCTION</u></a>	Declare function name and arguments
<a href="#"><u>GES()</u></a>	Multivalued greater than or equal to test
<a href="#"><u>GET</u></a>	Synonym for PUBLIC FUNCTION
<a href="#"><u>GET(ARG.)</u></a>	Retrieve command line arguments
<a href="#"><u>GET.MESSAGES()</u></a>	Retrieve messages from the message queue
<a href="#"><u>GET.PORT.PARAMS()</u></a>	Get serial port parameters
<a href="#"><u>GETLIST</u></a>	Restore a saved select list
<a href="#"><u>GETNLS()</u></a>	Get national language support parameter value
<a href="#"><u>GETPU()</u></a>	Get a characteristic of a print unit
<a href="#"><u>GETREM()</u></a>	Get remove pointer position
<a href="#"><u>GO / GOTO</u></a>	Jump to a label
<a href="#"><u>GOSUB</u></a>	Enter an internal subroutine
<a href="#"><u>GTS()</u></a>	Multivalued greater than test
<a href="#"><u>HEADING</u></a>	Set heading text
<a href="#"><u>HUSH</u></a>	Suppress or enable display output
<a href="#"><u>ICONV()</u></a>	Perform input conversion
<a href="#"><u>ICONVS()</u></a>	Perform input conversion on a dynamic array
<a href="#"><u>IDIV()</u></a>	Integer division
<a href="#"><u>IF / THEN / ELSE</u></a>	Perform conditional statements
<a href="#"><u>IFS()</u></a>	Multivalued conditional expression
<a href="#"><u>IN</u></a>	Read a single byte from the terminal with an optional timeout
<a href="#"><u>INDEX()</u></a>	Locate occurrence of substring within a string
<a href="#"><u>INDEXS()</u></a>	Multivalued equivalent of INDEX()
<a href="#"><u>INDICES()</u></a>	Return information about alternate key indices
<a href="#"><u>INHERIT</u></a>	Inherit an object
<a href="#"><u>INMAT()</u></a>	Return status of matrix operations
<a href="#"><u>INPUT</u></a>	Input a string from the keyboard or data queue
<a href="#"><u>INPUT @</u></a>	Input a string from the keyboard or data queue
<a href="#"><u>INPUTCLEAR</u></a>	Synonym for CLEARINPUT

<a href="#"><u>INPUTCSV</u></a>	Input CSV format data
<a href="#"><u>INPUTERR</u></a>	Synonym for PRINTERR
<a href="#"><u>INPUTFIELD</u></a>	Input a string with function key handling
<a href="#"><u>INS</u></a>	Insert a field, value or subvalue
<a href="#"><u>INSERT()</u></a>	Insert a field, value or subvalue
<a href="#"><u>INT()</u></a>	Truncate value to integer
<a href="#"><u>ITYPE()</u></a>	Execute a compiled I-type
<a href="#"><u>KEYCODE()</u></a>	Input a single keystroke from the keyboard with terminfo translation
<a href="#"><u>KEYEDIT</u></a>	Define editing keys for INPUT @
<a href="#"><u>KEYEXIT</u></a>	Define exit keys for INPUT @
<a href="#"><u>KEYIN()</u></a>	Input a single keystroke from the keyboard
<a href="#"><u>KEYINC()</u></a>	Input a single keystroke from the keyboard with case inversion
<a href="#"><u>KEYINR()</u></a>	Input a single keystroke from the keyboard in raw mode (no internal processing)
<a href="#"><u>KEYREADY()</u></a>	Test for keyboard input
<a href="#"><u>KEYTRAP</u></a>	Define trap keys for INPUT @
<a href="#"><u>LEN()</u></a>	Return length of a string
<a href="#"><u>LENS()</u></a>	Multivalued equivalent of LEN()
<a href="#"><u>LES()</u></a>	Multivalued less than test
<a href="#"><u>LISTINDEX()</u></a>	Return position of an item in a delimited list
<a href="#"><u>LN()</u></a>	Natural log
<a href="#"><u>LOCAL</u></a>	Declares an internal subroutine or function that has private local variables
<a href="#"><u>LOCATE</u></a>	Locate string in dynamic array
<a href="#"><u>LOCATE()</u></a>	Locate string in dynamic array
<a href="#"><u>LOCK</u></a>	Set task lock
<a href="#"><u>LOGMSG</u></a>	Add an entry to the system error log
<a href="#"><u>LOOP / REPEAT</u></a>	Define a loop to be repeated
<a href="#"><u>LOWER()</u></a>	Convert delimiters to lower level
<a href="#"><u>LTS()</u></a>	Multivalued less than or equal to test
<a href="#"><u>MARK.MAPPING</u></a>	Control field mark translation in directory files
<a href="#"><u>MAT</u></a>	Matrix initialisation or copy
<a href="#"><u>MATBUILD</u></a>	Build a dynamic array from matrix elements
<a href="#"><u>MATCHES()</u></a>	Matches a string against a pattern template
<a href="#"><u>MATCHESS()</u></a>	Matches each element of a dynamic array against a pattern template
<a href="#"><u>MATCHFIELD()</u></a>	Return portion of string matching pattern
<a href="#"><u>MATPARSE</u></a>	Break a dynamic array into matrix elements
<a href="#"><u>MATREAD</u></a>	Read a record, parsing into a matrix
<a href="#"><u>MATREADCSV</u></a>	Read a CSV format text item into a matrix
<a href="#"><u>MATREADL</u></a>	Read a record setting a read lock, parsing into a matrix
<a href="#"><u>MATREADU</u></a>	Read a record setting an update lock, parsing into a matrix
<a href="#"><u>MATWRITE</u></a>	Write a record from matrix elements
<a href="#"><u>MATWRITEU</u></a>	Write a record from matrix elements, retaining any lock
<a href="#"><u>MAX()</u></a>	Returns the greater of two values
<a href="#"><u>MAXIMUM()</u></a>	Find the greatest value in a dynamic array
<a href="#"><u>MD5()</u></a>	Convert a string to its 32 digit message digest value.
<a href="#"><u>MIN()</u></a>	Returns the lesser of two values
<a href="#"><u>MINIMUM()</u></a>	Find the lowest value in a dynamic array
<a href="#"><u>MOD()</u></a>	Modulus value from division
<a href="#"><u>MODS()</u></a>	Multivalued modulus value from division
<a href="#"><u>MVDATE()</u></a>	Extract the multivalued style date from an epoch value
<a href="#"><u>MVDATE.TIME()</u></a>	Extract the multivalued style date and time from an epoch value

<a href="#"><u>MVEPOCH()</u></a>	Convert a multivalued style date and time to an epoch value
<a href="#"><u>MVTIME()</u></a>	Extract the multivalued style time from an epoch value
<a href="#"><u>NAP</u></a>	Suspend program for a short period
<a href="#"><u>NEG()</u></a>	Arithmetic inverse
<a href="#"><u>NEGS()</u></a>	Multivalued arithmetic inverse
<a href="#"><u>NES()</u></a>	Multivalued inequality test
<a href="#"><u>NOBUF</u></a>	Turn off buffering for a record opened using OPENSEQ
<a href="#"><u>NOT()</u></a>	Logical NOT
<a href="#"><u>NOTS()</u></a>	Multivalued logical NOT
<a href="#"><u>NULL</u></a>	No operation
<a href="#"><u>NUM()</u></a>	Test if string holds a numeric value
<a href="#"><u>NUMS()</u></a>	Multivalued variant of NUM()
<a href="#"><u>OBJECT()</u></a>	Instantiates an object
<a href="#"><u>OBJINFO()</u></a>	Returns information about an object variable
<a href="#"><u>OCONV()</u></a>	Perform output conversion
<a href="#"><u>OCONVS()</u></a>	Perform output conversion on a dynamic array
<a href="#"><u>ON GOSUB</u></a>	Jump to one of a list of labels selected by value
<a href="#"><u>ON GOTO</u></a>	Enter one of a list of internal subroutines selected by value
<a href="#"><u>OPEN</u></a>	Open a file
<a href="#"><u>OPENPATH</u></a>	Open a file by pathname
<a href="#"><u>OPENSEQ</u></a>	Open a record for sequential access
<a href="#"><u>OPEN.SOCKET()</u></a>	Open a socket connection
<a href="#"><u>ORS()</u></a>	Multivalued logical OR
<a href="#"><u>OS.ERROR()</u></a>	Return operating system error information
<a href="#"><u>OS.EXECUTE</u></a>	Execute an operating system command
<a href="#"><u>OSDELETE</u></a>	Delete a file by pathname
<a href="#"><u>OSREAD</u></a>	Read a file by pathname
<a href="#"><u>OSWRITE</u></a>	Write a file by pathname
<a href="#"><u>OUTERJOIN()</u></a>	Fetch data from a file using an "outer join"
<a href="#"><u>PAGE</u></a>	Start a new page
<a href="#"><u>PAUSE</u></a>	Pause execution until awoken by another process
<a href="#"><u>PERFORM</u></a>	Synonym for EXECUTE
<a href="#"><u>PRECISION</u></a>	Set number of decimal places in numeric conversion
<a href="#"><u>PRINT</u></a>	Output to a logical print unit
<a href="#"><u>PRINTCSV</u></a>	Print data in comma separated variable format
<a href="#"><u>PRINTER CLOSE</u></a>	Close a print unit
<a href="#"><u>PRINTER DISPLAY</u></a>	Associate a print unit with the display
<a href="#"><u>PRINTER FILE</u></a>	Associate a file with a print unit
<a href="#"><u>PRINTER NAME</u></a>	Associate a print device with a print unit
<a href="#"><u>PRINTER OFF</u></a>	Disable print unit zero
<a href="#"><u>PRINTER ON</u></a>	Enable print unit zero
<a href="#"><u>PRINTER RESET</u></a>	Reset default print unit and display
<a href="#"><u>PRINTER SETTING</u></a>	Set a print unit parameter
<a href="#"><u>PRINTER.SETTING()</u></a>	Set or retrieve a print unit parameter
<a href="#"><u>PRINTERR</u></a>	Display an error message
<a href="#"><u>PRIVATE</u></a>	Declare private variables in a local subroutine or a class module
<a href="#"><u>PROCREAD</u></a>	Read data from the PROC primary input buffer
<a href="#"><u>PROCWRITE</u></a>	Write data to the PROC primary input buffer
<a href="#"><u>PROGRAM</u></a>	Declare program name
<a href="#"><u>PROMPT</u></a>	Set the input prompt character
<a href="#"><u>PTERM()</u></a>	Set/clear/query terminal settings
<a href="#"><u>PUBLIC</u></a>	Declare public properties in a class module
<a href="#"><u>PUT</u></a>	Synonym for PUBLIC SUBROUTINE
<a href="#"><u>PWR()</u></a>	Raise value to power

<a href="#"><u>QUOTE()</u></a>	Enclose a string in quotes
<a href="#"><u>QUOTES()</u></a>	Enclose each element of a dynamic array in double quotes
<a href="#"><u>RAISE()</u></a>	Convert delimiters to higher level
<a href="#"><u>RANDOMIZE</u></a>	Set random number seed value
<a href="#"><u>RDIV()</u></a>	Rounded integer division
<a href="#"><u>READ</u></a>	Read a record from a file
<a href="#"><u>READ.SOCKET()</u></a>	Read data from a socket
<a href="#"><u>READBLK</u></a>	Read bytes from a sequential file
<a href="#"><u>READCSV</u></a>	Read a CSV format text item
<a href="#"><u>READL</u></a>	Read a record from a file, setting a read lock
<a href="#"><u>READLIST</u></a>	Save a select list in a dynamic array
<a href="#"><u>READNEXT</u></a>	Read a record id from a select list
<a href="#"><u>READSEQ</u></a>	Read from a sequential file
<a href="#"><u>READU</u></a>	Read a record from a file, setting an update lock
<a href="#"><u>READV</u></a>	Read a field from a record in a file
<a href="#"><u>READVL</u></a>	Read a field from a record in a file, setting a read lock
<a href="#"><u>READVU</u></a>	Read a field from a record in a file, setting an update lock
<a href="#"><u>RECORDLOCKED()</u></a>	Test if record is locked
<a href="#"><u>RECORDLOCKL</u></a>	Set a read lock on a record
<a href="#"><u>RECORDLOCKU</u></a>	Set an update lock on a record
<a href="#"><u>RELEASE</u></a>	Release record or file locks
<a href="#"><u>REM()</u></a>	Remainder value from division
<a href="#"><u>REMARK</u></a>	Alternative syntax for comments
<a href="#"><u>REMOVE</u></a>	Remove an item from a dynamic array
<a href="#"><u>REMOVE()</u></a>	Remove an item from a dynamic array
<a href="#"><u>REMOVEF()</u></a>	Extract data from a delimited character string
<a href="#"><u>REMOVE.BREAK.HANDLER</u></a>	Deactivate a break handler subroutine
<a href="#"><u>REPLACE()</u></a>	Replace a field, value or subvalue
<a href="#"><u>RESTORE.SCREEN</u></a>	Restore screen image data
<a href="#"><u>RETURN</u></a>	Return from CALL or GOSUB
<a href="#"><u>RETURN FROM PROGRAM</u></a>	Return from CALL
<a href="#"><u>RETURN TO</u></a>	Return from program or subroutine to a specific label
<a href="#"><u>REUSE()</u></a>	Reuse element of numeric arrays in mathematical functions
<a href="#"><u>RND()</u></a>	Generate random number
<a href="#"><u>ROUNDDOWN()</u></a>	Round a number towards zero in a specified increment
<a href="#"><u>ROUNDUP()</u></a>	Round a number away from zero in a specified increment
<a href="#"><u>ROLLBACK</u></a>	Discard transaction updates
<a href="#"><u>RQM</u></a>	Synonym for SLEEP
<a href="#"><u>RTRANS()</u></a>	Fetch data from a file
<a href="#"><u>SAVE.SCREEN()</u></a>	Save screen image data
<a href="#"><u>SAVELIST</u></a>	Save a select list in the \$SAVEDLISTS file
<a href="#"><u>SEEK</u></a>	Position a sequential file
<a href="#"><u>SELECT</u></a>	Build a select list of all records in an open file
<a href="#"><u>SELECT.SOCKET()</u></a>	Monitor events on multiple sockets
<a href="#"><u>SELECTE</u></a>	Transfer select list 0 to a select list variable
<a href="#"><u>SELECTINDEX</u></a>	Build a select list from an alternate key index
<a href="#"><u>SELECTINFO()</u></a>	Return information regarding a select list
<a href="#"><u>SELECTLEFT</u></a>	Scan left through an alternate key index
<a href="#"><u>SELECTN</u></a>	Build a numbered select list of all records in an open file
<a href="#"><u>SELECTRIGHT</u></a>	Scan right through an alternate key index
<a href="#"><u>SELECTV</u></a>	Build a select list variable of all records in an open file
<a href="#"><u>SENTENCE()</u></a>	Returns the command line that started the current program
<a href="#"><u>SEQ()</u></a>	Get collating sequence value for a given ASCII character
<a href="#"><u>SERVER.ADDR()</u></a>	Find the IP address for a given server name

<a href="#"><u>SET.ARG</u></a>	Sets an argument variable based on its argument list position
<a href="#"><u>SET.BREAK.HANDLER</u></a>	Establish a break handler subroutine
<a href="#"><u>SET.EXIT.STATUS</u></a>	Set final exit status value
<a href="#"><u>SET.PORT.PARAMS()</u></a>	Set serial port parameters
<a href="#"><u>SET.SOCKET.MODE()</u></a>	Set mode of a socket
<a href="#"><u>SET.TIMEZONE</u></a>	Set time zone for use by the epoch conversion code
<a href="#"><u>SETLEFT</u></a>	Set alternate key index scan position to leftmost
<a href="#"><u>SETNLS()</u></a>	Set national language support parameter value
<a href="#"><u>SETPU</u></a>	Set a characteristic of a print unit
<a href="#"><u>SETRIGHT</u></a>	Set alternate key index scan position to rightmost
<a href="#"><u>SETREM</u></a>	Set remove pointer position
<a href="#"><u>SHIFT()</u></a>	Perform bit shift
<a href="#"><u>SIN()</u></a>	Sine
<a href="#"><u>SLEEP</u></a>	Suspend program to / for given time
<a href="#"><u>SOCKET.INFO()</u></a>	Retrieve information about a socket
<a href="#"><u>SORT.COMPARE()</u></a>	Compare items according to sort rules
<a href="#"><u>SOUNDEX()</u></a>	Form a soundex code value for a string
<a href="#"><u>SOUNDEXS()</u></a>	Multivalued variant of SOUNDEX()
<a href="#"><u>SPACE()</u></a>	Create a string of spaces
<a href="#"><u>SPACES()</u></a>	Multivalued variant of SPACE()
<a href="#"><u>SPLICE()</u></a>	Concatenates elements of two dynamic arrays, inserting a string between the items.
<a href="#"><u>SQRT()</u></a>	Square root
<a href="#"><u>SQUOTE()</u></a>	Enclose a string in single quotes
<a href="#"><u>SQUOTES()</u></a>	Enclose each element of a dynamic array in single quotes
<a href="#"><u>SSELECT</u></a>	Build a sorted select list of all records in an open file
<a href="#"><u>STATUS()</u></a>	Return status from previous operation
<a href="#"><u>STATUS</u></a>	Returns a dynamic array of information about an open file
<a href="#"><u>STOP</u></a>	Terminate program
<a href="#"><u>STOPE</u></a>	Terminate program with Pick style message handling
<a href="#"><u>STOPM</u></a>	Terminate program with Information style message handling
<a href="#"><u>STR()</u></a>	Create a string from a repeated substring
<a href="#"><u>STRS()</u></a>	Multivalued variant of STR()
<a href="#"><u>SUBR()</u></a>	Call a subroutine as a function
<a href="#"><u>SUBROUTINE</u></a>	Declare subroutine name and arguments
<a href="#"><u>SUBSTITUTE()</u></a>	Multivalued substring replacement
<a href="#"><u>SUBSTRINGS()</u></a>	Multivalued substring extraction
<a href="#"><u>SUM()</u></a>	Sum lowest level elements of a numeric array
<a href="#"><u>SUMMATION()</u></a>	Sum all elements of a numeric array
<a href="#"><u>SWAP()</u></a>	Synonym for CHANGE()
<a href="#"><u>SWAPCASE()</u></a>	Invert case of alphabetic characters in a string
<a href="#"><u>SYSTEM()</u></a>	Return system information
<a href="#"><u>TAN()</u></a>	Tangent
<a href="#"><u>TCLREAD</u></a>	Returns the sentence that started the current program
<a href="#"><u>TERMINFO()</u></a>	Retrieve information from the terminfo database
<a href="#"><u>TESTLOCK()</u></a>	Test state of a task lock
<a href="#"><u>TIME()</u></a>	Return the current time
<a href="#"><u>TIMEDATE()</u></a>	Return the date and time as a string
<a href="#"><u>TIMEOUT</u></a>	Sets a timeout for READBLK and READSEQ
<a href="#"><u>TRANS()</u></a>	Fetch data from a file
<a href="#"><u>TRANSACTION ABORT</u></a>	Abort a transaction
<a href="#"><u>TRANSACTION COMMIT</u></a>	Commit a transaction
<a href="#"><u>TRANSACTION START</u></a>	Start a new transaction
<a href="#"><u>TRIM()</u></a>	Trim characters from string

---

<a href="#"><u>TRIMB()</u></a>	Trim spaces from back of string
<a href="#"><u>TRIMBS()</u></a>	Multivalued variant of TRIMB()
<a href="#"><u>TRIMF()</u></a>	Trim spaces from front of string
<a href="#"><u>TRIMFS()</u></a>	Multivalued variant of TRIMF()
<a href="#"><u>TRIMS()</u></a>	Multivalued variant of TRIM()
<a href="#"><u>TTYGET()</u></a>	Get current terminal mode settings
<a href="#"><u>TTYSET</u></a>	Set terminal modes
<a href="#"><u>UNASSIGNED()</u></a>	Test whether variable is unassigned
<a href="#"><u>UNLOCK</u></a>	Release task lock
<a href="#"><u>UNTIL</u></a>	Leave loop if condition is met
<a href="#"><u>UPCASE()</u></a>	Convert string to uppercase
<a href="#"><u>VOCPATH()</u></a>	Resolve a filename to its corresponding pathname
<a href="#"><u>VOID</u></a>	Discard the result of evaluating an expression
<a href="#"><u>VSLICE()</u></a>	Extract a value or subvalue slice from a dynamic array
<a href="#"><u>WAIT.FILE.EVENT()</u></a>	Wait for a file monitoring event to occur
<a href="#"><u>WAKE</u></a>	Restart execution of a process on a PAUSE
<a href="#"><u>WEOFSEQ</u></a>	Write end of file position to sequential file
<a href="#"><u>WHILE</u></a>	Leave loop unless condition is met
<a href="#"><u>WRITE</u></a>	Write a record to a file
<a href="#"><u>WRITE.SOCKET()</u></a>	Write data to a socket
<a href="#"><u>WRITEBLK</u></a>	Write bytes to a sequential file
<a href="#"><u>WRITECSV</u></a>	Write CSV format data to a sequential file
<a href="#"><u>WRITESEQ</u></a>	Write to sequential file
<a href="#"><u>WRITESEQF</u></a>	Write to sequential file, flushing to disk
<a href="#"><u>WRITEU</u></a>	Write a record to a file, retaining any lock
<a href="#"><u>WRITEV</u></a>	Write a field to a record in a file
<a href="#"><u>WRITEVU</u></a>	Write a field to a record in a file, retaining any lock
<a href="#"><u>XLATE()</u></a>	Synonym for TRANS()
<a href="#"><u>XTD()</u></a>	Convert a hexadecimal number

## @(x,y) Function

The @(x, y) function is used to control the format of displayed output.

### Format

@( <i>col</i> {, <i>line</i> })	Cursor movement
@( <i>mode</i> {, <i>arg</i> })	Device control functions

where

<i>col</i>	evaluates to a display column position.
<i>line</i>	evaluates to a display line position.
<i>mode</i>	evaluates to a mode value as described below.
<i>arg</i>	provides qualifying information for use with some <i>mode</i> 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).

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 qmterm 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 return 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 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, 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,

```
DISPLAY @(-1):@(-15):'ABC':@(-16):'DEF'
```

on a terminal that stores the attribute for each character position would result in a display of

```
ABCDEF
```

whereas a terminal that uses a character cell to store the attribute would display

```
ABC DEF
```

Furthermore, if the program then executed

```
DISPLAY @(3,0):'X'
```

the first terminal would display

```
ABCXEF
```

but the second would display

```
ABCXDEF
```

with the underline extending to the end of the screen or the next cell holding an attribute setting.

Value	Token	Function	Argument
-1	IT\$CS	Clear screen	
-2	IT\$CAH	Cursor home	
-3	IT\$CLEOS	Clear to end of screen	
-4	IT\$CLEOL	Clear to end of line	
-5	IT\$SBLINK	Start flashing text	
-6	IT\$EBLINK	End flashing text	
-7	IT\$SPA	Start protected area	
-8	IT\$EPA	End protected area	
-9	IT\$CUB	Backspace	No of characters (default 1)
-10	IT\$CUU	Cursor up	No of lines (default 1)
-11	IT\$SHALF	Start half brightness	
-12	IT\$EHALF	End half brightness	

-13	IT\$SREV	Start reverse video	
-14	IT\$EREV	End reverse video	
-15	IT\$SUL	Start underline	
-16	IT\$EUL	End underline	
-17	IT\$IL	Insert line	No of lines (default 1)
-18	IT\$DL	Delete line	No of lines (default 1)
-19	IT\$ICH	Insert character	No of characters (default 1)
-22	IT\$DCH	Delete character	No of characters (default 1)
-23	IT\$AUXON	Turn on printer	
-24	IT\$AUXOFF	Turn off printer	
-29	IT\$E80	Set 80 column mode	
-30	IT\$E132	Set 132 column mode	
-31	IT\$RIC	Reset inhibit cursor	
-32	IT\$SIC	Inhibit cursor	
-33	IT\$CUD	Cursor down	No of lines (default 1)
-34	IT\$CUF	Cursor forward	No of characters (default 1)
-37	IT\$FGC	Set foreground colour	Colour
-38	IT\$BGC	Set background colour	Colour
-54	IT\$SLT	Set line truncation	
-55	IT\$RLT	Reset line truncation	
-58	IT\$SBOLD	Set bold mode	
-59	IT\$EBOLD	Reset bold mode	
-100 to -107	User definable via the u0 to u7 terminfo keys		
-108	IT\$ACMD	Asynchronous command	Command to execute
-109	IT\$SCMD	Synchronous command	Command to execute
-250	IT\$STYLUS	Enable/disable stylus taps	0 = disable, 1 = enable (PDA)
-251	IT\$KEYS	Display/hide screen keyboard	0 = hide, 1 = display (PDA)

## Descriptions

### IT\$CS (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).

### IT\$CAH (Cursor home)

The cursor is positioned at the top left of the screen (position 0,0).

### 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.

**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.

**IT\$SBLINK (Start flashing text)**

Displays subsequent text in flashing mode.

**IT\$EBLINK (End flashing text)**

Terminates flashing mode.

**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.

**IT\$EPA (End protected area)**

Terminates protected mode.

**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.

**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.

**IT\$SHALF (Start half brightness)**

Displays subsequent data in half brightness (dim) mode.

**IT\$EHALF (End half brightness)**

Terminates half brightness (dim) mode.

**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.

**IT\$EREV (End reverse video)**

If in reverse video mode, this operation reverts to the normal display colours for subsequent text output.

**IT\$SUL (Start underline)**

Displays subsequent data in underline mode.

**IT\$EUL (End underline)**

Terminates underline mode.

**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.

**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.

**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.

**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.

**IT\$AUXON (Turn on printer)**

For terminals with attached printers, this mode directs output to the printer. The mc5 [terminfo](#) entry must be set correctly for this to work.

**IT\$AUXOFF (Turn off printer)**

For terminals with attached printers, this mode turns off output to the printer. The mc4 [terminfo](#) entry must be set correctly for this to work.

**IT\$E80 (Set 80 column mode)**

The display window is set to be 80 characters wide.

**IT\$E132 (Set 132 column mode)**

The display window is set to be 132 characters wide.

**IT\$RIC (Reset inhibit cursor)**

The cursor is displayed if it was previously inhibited.

**IT\$SIC (Inhibit cursor)**

Display of the cursor is inhibited. All cursor positioning functions continue to work whilst the cursor is not visible.

**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.

**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.

**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.

IT\$BLACK	0
IT\$BLUE	1
IT\$GREEN	2
IT\$CYAN	3
IT\$RED	4
IT\$MAGENTA	5
IT\$BROWN	6
IT\$WHITE	7
IT\$GREY	8
IT\$BRIGHT.BLUE	9

IT\$BRIGHT.GREEN	10
IT\$BRIGHT.CYAN	11
IT\$BRIGHT.RED	12
IT\$BRIGHT.MAGENTA	13
IT\$YELLOW	14
IT\$BRIGHT.WHITE	15

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\$BGC (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.

#### **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.

#### **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.

#### **IT\$SBOLD (Start bold)**

Displays subsequent data in bold mode.

#### **IT\$EBOLD (End bold)**

Terminates bold mode.

#### **IT\$ACMD (Asynchronous command)**

Executes the given command on the client system without suspending the QM session. This operation depends on correct setting of the [terminfo](#) u8 token.

#### **IT\$SCMD (Synchronous command)**

Executes the given command on the client system, suspending the QM session until the command completes. This operation depends on correct setting of the [terminfo](#) u9 token.

#### **IT\$STYLUS (Enable/disable stylus taps)**

Applicable only to QM on a PDA, this operation 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.

#### **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.

### **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 multivalued database products.

### Format

**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

```
$MODE PICK.ERRMSG
```

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

**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.ERRMSG option.

**Examples**

```
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

**ABS**(*expr*)

where

*expr* evaluates to a number or a numeric array.

The **ABS()** function returns the absolute value of *expr*. If *expr* is positive or zero, the value of **ABS**(*expr*) is *expr*. If *expr* is negative, the value of **ABS**(*expr*) is *-expr*.

If *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 *expr*.

### Examples

```
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
```

```
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

**ACCEPT.SOCKET.CONNECTION**(*srvr.skt*, *timeout*)

where

*srvr.skt* is the server socket created by an earlier use of [CREATE.SERVER.SOCKET](#).

*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.

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

```
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](#), [CLOSE.SOCKET](#), [CREATE.SERVER.SOCKET\(\)](#), [OPEN.SOCKET\(\)](#), [READ.SOCKET\(\)](#), [SELECT.SOCKET\(\)](#), [SERVER.ADDR\(\)](#), [SET.SOCKET.MODE\(\)](#), [SOCKET.INFO\(\)](#), [WRITE.SOCKET\(\)](#)

## 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 (1) if *string* contains only alphabetic characters (A to Z, a to z). The function returns false (0) for a null string or a string that contains non-alphabetic characters.

Use of **ALPHA()** is 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\(\)](#)

## ANDS()

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. Any value other than zero or a null string is treated as 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\(\)](#)

## ARG ()

The **ARG()** function returns an 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.

### 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.

### 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

**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 (1) if it is assigned or (0) 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 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 rollback 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 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.

## BINDKEY()

The **BINDKEY()** function sets, removes, queries, saves or restores key bindings.

### Format

**BINDKEY**(*key.string*, *action*)

where

<i>key.string</i>	The character sequence for the key to be bound, unbound or queried.
<i>action</i>	identifies the action to be performed: <ul style="list-style-type: none"> <li>&gt;= 0 Bind key to this code (0 to 255)</li> <li>-1 Remove binding for <i>key.string</i>.</li> <li>-2 Query binding for <i>key.string</i>.</li> <li>-3 Save bindings.</li> <li>-4 Restore bindings from <i>key.string</i>.</li> <li>-5 Disables lone Esc key handling in <a href="#">KEYCODE()</a>.</li> <li>-6 Re-enables lone Esc key handling in <a href="#">KEYCODE()</a>.</li> </ul>

The **BINDKEY()** function used with an *action* value in the range 0 to 255 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 (1) and the [STATUS\(\)](#) function would return zero. If an error occurs, the function returns false(0) 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

An *action* value of -1 removes any defined binding for *key.string*. Used in this mode, the function always returns true (1) 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 (1) if lone Esc key handling was previously enabled, or false (0) 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\(\)](#), [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 logical inverse of A.

### See also:

[BITAND\(\)](#), [BITOR\(\)](#), [BITRESET\(\)](#), [BITSET\(\)](#), [BITTEST\(\)](#), [BITXOR\(\)](#), [SHIFT\(\)](#)

## BITOR()

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 (1) if it is set and false (0) 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\(\)](#)

## BREAK

The **BREAK** statement allows the action of the break key to be disabled during program execution.

### Format

```
BREAK {KEY} OFF  
BREAK {KEY} ON  
BREAK {KEY} CLEAR  
BREAK {KEY} expr
```

where

*expr* 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 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

```
BREAK OFF  
GOSUB UPDATE.FILES  
BREAK ON
```

This program fragment inhibits use of the break key while the internal subroutine UPDATE.FILES is executed.

### See also:

[BREAK command](#), [SET.BREAK.HANDLER](#), [REMOVE.BREAK.HANDLER](#)

## 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 allowed. 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**,

```
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

```
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. 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

```
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

```
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](#)

## CASE

The **CASE** statement provides conditional execution dependant on the result of expression evaluation.

### Format

```
BEGIN CASE
  CASE expr
    statement(s)
  CASE expr
    statement(s)
END CASE
```

where

*expr* is an expression which can be resolved to a numeric 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

### **CASE 1**

which makes use of the fact that 1 is the value of the boolean True. This must be the final element of the **CASE** statement.

### Example

```
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.

## 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 as 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

**CATS**(*string1*, *string2*)

where

*string1* is the string to which *string2* is to be concatenated.

*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

```
S1 = "ABC":@fm:"DEF"  
S2 = "123":@vm:"456":@fm:"789"  
X = CATS(S1,S2)
```

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<sub>vm</sub>456<sub>fm</sub>DEF789"

```
S1 = "ABC":@fm:"DEF"  
S2 = "123":@fm:"456":@fm:"789"  
X = CATS(S1,S2)
```

This example concatenates elements of the two strings yielding a result in X of "ABC123<sub>fm</sub>DEF456<sub>fm</sub>789"

```
S1 = "ABC":@fm:"DEF"  
S2 = "123":@fm:"456":@fm:"789"  
X = CATS(REUSE(S1),S2)
```

Adding the **REUSE()** function to the previous example gives a result in X of "ABC123<sub>fm</sub>DEF456<sub>fm</sub>DEF789"

## CHAIN

The **CHAIN** statement terminates the current program and executes a command.

### Format

**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

```
CHAIN "RUN PROGRAM2 "
```

This program fragment terminates the current program and executes PROGRAM2 in its place.

## CHANGE()

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

<i>string</i>	is the string in which the replacement is to occur.
<i>old</i>	evaluates to the substring to be replaced.
<i>new</i>	evaluates to the substring with which <i>old</i> is to be replaced.
<i>occurrence</i>	evaluates to the number of occurrences of <i>old</i> to be replaced. If omitted or specified as a value of less than one, all occurrences are replaced.
<i>start</i>	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. If *new* is a null string, all occurrences of *old* are removed.

If the [\\$NOCASE.STRINGS](#) compiler directive is used, matching of *old* against *string* is case insensitive.

### Examples

```
PRINT CHANGE ("ABRACADABRA", "A", "a", 3, 2)
```

This statement results in printing the string "ABRaCaDaBRA".

**See also:**

[CONVERT\(\)](#)

## 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.

### 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:

[SEQ\(\)](#)

## CHECKSUM()

The **CHECKSUM()** function returns a checksum value for supplied data.

### Format

**CHECKSUM**(*data*)

where

*data* is the data for which a checksum is to be calculated.

The **CHECKSUM()** function calculates a checksum value for the supplied *data* item. Note that the algorithm used by QM may be different from that of other multivalued products and hence may yield a different result for the same data.

### Example

```
DISPLAY CHECKSUM( REC )
```

This statement displays a checksum value for the data in the REC variable.

### See also:

[MD5Q](#)

## 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 (1) 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 (0). 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.

**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 *name* is omitted, all variables in the unnamed common area are set to zero. Other variables are not affected.

If *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. All records are deleted from the file, contracting the file to its minimum modulus size and releasing disk space.

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 file variable will be closed. 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.SOCKET

The **CLOSE.SOCKET** statement closes a socket.

### Format

**CLOSE.SOCKET** *skt*

where

*skt* is the socket variable corresponding to the socket to be closed.

The socket previously opened using [ACCEPT.SOCKET.CONNECTION\(\)](#), [CREATE.SERVER.SOCKET\(\)](#) or [OPEN.SOCKET\(\)](#) is closed.

### See also:

[Using Socket Connections](#), [ACCEPT.SOCKET.CONNECTION](#), [CREATE.SERVER.SOCKET\(\)](#), [OPEN.SOCKET\(\)](#), [READ.SOCKET\(\)](#), [SELECT.SOCKET\(\)](#), [SERVER.ADDR\(\)](#), [SET.SOCKET.MODE\(\)](#), [SOCKET.INFO\(\)](#), [WRITE.SOCKET\(\)](#)

## 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.

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\(\)](#) function.

### Format

**COL1()**

The **COL1()** function is used after a [FIELD\(\)](#) 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 the [FIELD\(\)](#) and **COL1()** functions, the value returned relates to the last use of [FIELD\(\)](#) in the current program. Any [FIELD\(\)](#) functions executed by the subroutine will have no effect on the **COL1()** value.

**COL1()** returns zero if

No [FIELD\(\)](#) 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*EEEE"
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\(\)](#)

## COL2()

The **COL2()** function returns the character position immediately following the substring extracted by the last [FIELD\(\)](#) function.

### Format

**COL2()**

The **COL2()** function is used after a [FIELD\(\)](#) 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 the [FIELD\(\)](#) and **COL2()** functions, the value returned relates to the last use of [FIELD\(\)](#) in the current program. Any [FIELD\(\)](#) functions executed by the subroutine will have no effect on the **COL2()** value.

**COL2()** returns zero if

No [FIELD\(\)](#) 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*EEEE"
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\(\)](#)

## COMMON

The **COMMON** statement declares variables in a common block.

### Format

```
COMMON {/name/} var1 {,var2...}
```

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
```

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

```
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

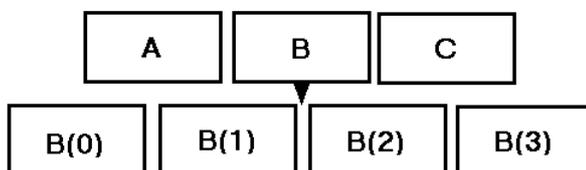
```
COMMON /MYCOMMON/  MAT1 ( 5 , 3 )
```

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 resizable. A common block declared as

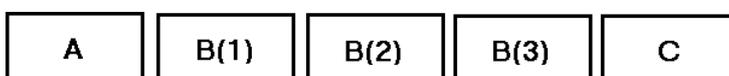
```
COMMON  A , B ( 3 ) , C
```

has three variables, one of which is a matrix:



A matrix of this type in a common block can be redimensioned 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:



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.

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 [\\$NOCASE.STRINGS](#) compiler directive and can therefore be used to force a case sensitive comparison in otherwise case insensitive programs..

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.

**Examples**

```

DIM ITEM(100)
ITEMS = 0
LOOP
  INPUT NEW.ITEM
WHILE LEN(NEW.ITEM)
  * Find position to insert 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.

```

A = 'AAA':@VM:'BBB':@FM:'CCC'
X = COMPARES(A, 'BBB')

```

The above program fragment returns X as "-1VM0FM1". Note how the mark characters used to delimit A have been replicated in X.

**See also:**

[\*\*SORT.COMPARE\(\)\*\*](#)

## 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 (1) if successful, false (0) 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 COM01 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...
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.string* **TO** *to.string* **IN** *var*

**CONVERT**(*from.string*, *to.string*, *source.string*)

where

*from.string* evaluates to a string containing the characters to be replaced.

*to.string* evaluates to a string containing the replacement characters.

*var* is the variable in which the replacement is to occur.

*src.string* is the string in which replacement is to occur.

The statement

```
S = CONVERT(X, Y, S)
```

is equivalent to

```
CONVERT X TO Y IN S
```

Characters taken from the *from.string* and *to.string* define character translations to be performed. Each occurrence of a character from *from.string* in *var* (or *src.string*) is replaced by the character in the same position in *to.string*. If *to.string* is shorter than *from.string*, characters for which there is no replacement character are deleted. If *to.string* is longer than *from.string* the surplus characters are ignored.

If a character appears more than once in *from.string* only the first occurrence is used.

If the [\\$NOCASE.STRING](#)s compiler directive is used, matching of *from.string* against *var* is case insensitive.

### Examples

```
S = "ABCDEFGHGIJK"
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.string* 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

**COS**(*expr*)

where

*expr* 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

ADJ = HYP \* 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:

[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

**COUNT**(*string*, *substring*)

**COUNTS**(*string*, *substring*)

where

*string* evaluates to the string in which substrings are to be counted.

*substring* evaluates to the substring to count.

The **COUNT()** function counts occurrences of *substring* within *string*. Substrings may not overlap, thus

```
S = "ABABABABABAB"  
N = COUNT(S, "ABA")
```

sets N to 3.

If *substring* is null, **COUNT()** returns the length of *string*.

Programs compiled with the [\\$NOCASE.STRINGS](#) compiler directive use case insensitive string comparisons in the **COUNT()** and **COUNTS()** functions.

### Example

```
MARKS = COUNT(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.SOCKET

The **CREATE.SERVER.SOCKET()** function creates a server socket on which a program may wait for incoming connections.

### Format

**CREATE.SERVER.SOCKET**(*addr, port, flags*)

where

<i>addr</i>	is the address on which to listen for incoming connections. This may be an IP address or a host name. A null string implies listen on any local address.
<i>port</i>	is the port number on which to listen for incoming connections.
<i>flags</i>	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:
	SKT\$STREAM      A stream connection (default)
	SKT\$DGRM      A datagram type connection
	Protocol, one of:
	SKT\$TCP      Transmission Control Protocol (default)
	SKT\$UDP      User Datagram Protocol

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.SOCKET()** 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.SOCKET(), READ.SOCKET(), SELECT.SOCKET(), SERVER.ADDR(), SET.SOCKET.MODE(), SOCKET.INFO(), WRITE.SOCKET()

## **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.

## CSV DQ()

The **CSV DQ()** function dequotes a CSV (comma separated variable) string.

### Format

**CSV DQ**(*string* {, *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 **CSV DQ()** function dequotes 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).

### Example

```
S = 'ABC, "DEF", "GHI, JKL", "MN" "O" '  
DISPLAY CSV DQ(S)
```

The above program fragment would display

```
ABCFMDEFFMGHI, JKLFMMN"O
```

ABC was unquoted and remains unchanged.

DEF was quoted. The quotes have been removed.

GHI,JKL contains a comma that has been preserved after removal of the quotes.

MN"O contains an embedded quote that has been preserved.

### See also:

[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.

### 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"
```

---

```
S = 'ABC':@FM:'123':@FM:'A"B':@FM:'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 separated by item marks. 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

**DCOUNT**(*string*, *delimiter*)

where

*string* evaluates to the string in which delimited substrings are to be counted.

*delimiter* evaluates to the single character substring delimiter.

The **DCOUNT()** function counts substrings delimited by *delimiter* within *string*. Substrings may not overlap, thus

```
S = "ABABABABABABA "  
N = DCOUNT ( S , "BAB " )
```

sets N to 4.

If *string* is null, **DCOUNT()** returns zero. In all other cases, **DCOUNT()** returns a value one greater than the **COUNT()** function applied to the same *string*.

If *substring* is null, **DCOUNT()** returns the length of *string*.

Programs compiled with the [\\$NOCASE.STRINGS](#) compiler directive use case insensitive string comparisons in the **DCOUNT()** function.

### Example

```
FIELDS = DCOUNT ( REC , @FM )
```

This statement counts the fields in 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 compiled 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 is 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 useable 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.

### Example

```

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 "subr" | LOCAL} {VAR.ARGS} {KEY  
key}
```

where

*name* is the name of the function.

*arg1*, *arg2*... are the function arguments.

*subr* is the catalogue name of the subroutine if different from *name*.

*key* is a key value to be passed into the subroutine as described below.

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** keyword is 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.

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.

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. 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 **VAR.ARGS** 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 local 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 the 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.

### Format

```
DEL dyn.array<field {, value {, subvalue } }>  
DELETE(dyn.array, field {, value {, subvalue } })
```

where

<i>dyn.array</i>	is the dynamic array from which the item is to be deleted.
<i>field</i>	evaluates to the number of the field to be deleted.
<i>value</i>	evaluates to the number of the value to be deleted. If omitted or zero, the entire field is deleted.
<i>subvalue</i>	evaluates to the number of the subvalue to be deleted. If omitted or zero, the entire value is deleted.

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*.

### Example

```
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.

### See also:

[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

**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 [STATUS\(\)](#) function can be used to determine the cause of execution of the **ON ERROR** clause. A fatal error occurring when no **ON ERROR** clause is present will cause an abort to occur.

### Example

```
DELETE STOCK, ITEM.ID
```

This statement deletes the record whose id is in **ITEM.ID** from the file associated with file variable **STOCK**.

## DELETEDLIST

The **DELETEDLIST** statement deletes a select list from the \$SAVEDLISTS file.

### Format

**DELETEDLIST** *name*

where

*name* is the name of the \$SAVEDLISTS entry to be deleted.

The **DELETEDLIST** statement restores the previously saved select list identified by *name* from the \$SAVEDLISTS file. No error occurs if the list does not exist.

## 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

<i>filename</i>	evaluates to the VOC name of the directory file holding the record to be deleted.
<i>id</i>	evaluates to the name of the record to be deleted.
<i>pathname</i>	evaluates to the operating system pathname of the record to be deleted.
<i>statement(s)</i>	are statement(s) to be executed depending on the outcome of the <b>DELETESEQ</b> 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.

## 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

```
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.

```
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.

## DIMENSION

The **DIMENSION** statement is used to set the dimensions of a matrix. The short form **DIM** may be used in place of **DIMENSION**.

### Format

```
DIMENSION mat(rows {, cols})
```

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.

A matrix variable is a one or two dimensional array of values. Matrices must be declared by use of the **DIMENSION** statement. The **DIMENSION** 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 redimensioned at any time by a further **DIMENSION** statement though the number of dimensions cannot be changed. Existing values of matrix elements will be retained in the redimensioned matrix by copying elements on a row by row basis. If the matrix is enlarged, the newly created elements will be unassigned. If it is smaller than before, any values in the excess elements are discarded.

The **\$MODE** compiler directive can be used to select Pick style matrices which do not have a zero element and cannot be redimensioned.

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 ABORT "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.

### 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
```

### See also:

[INMAT\(\)](#), [MAT](#), [MATBUILD](#), [MATREAD](#), [MATREADCSV](#), [MATPARSE](#), [MATWRITE](#)

## 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.

```
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](#)

## 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 two items separated by value marks:

- Value 1     The item name. On platforms with case insensitive file names, the text returned preserves the casing used by the underlying operating system.
- Value 2     The item type. This is F for a file or D for a directory.
- Value 3     File modes. On Windows and PDA 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.

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.

### 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.

## DIV()

The **DIV()** function returns the quotient from a division operation.

### Format

**DIV**(*dividend*, *divisor*)

where

*dividend* evaluates to a number.

*divisor* evaluates to a number.

The **DIV()** function returns the quotient from dividing *dividend* by *divisor*.

## 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.

### Example

```
REF.NO = "A12F635"  
PRINT DOWNCASE(REF.NO)
```

This program fragment prints the string "a12f635".

### See also:

[UPCASE\(\)](#)

## DPARSE and DPARSE.CSV

The **DPARSE** 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.

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.

## DTX()

The **DTX()** function converts a number to hexadecimal.

### Format

**DTX**(*expr* [, *expr*])

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.

### Examples

In each example, the [DISPLAY](#) statement is followed by the output that it would produce. Note that **DTX()** treats the value as an unsigned 32 bit quantity and hence negative numbers will always appear as 8 hexadecimal digits.

```
DISPLAY DTX(87)  
57
```

```
DISPLAY DTX(87,4)  
0057
```

```
DISPLAY DTX(-21,4)  
FFFFFFEB
```

**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\(\)](#)

## 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 @](#)

## ENCRYPT()

The **ENCRYPT()** function encrypts 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.

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 useable 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.

### Example

```

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'
    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](#), [DECRYPT\(\)](#)

## 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 multivalued 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.

## ENV()

The **ENV()** function retrieves an operating system environment variable.

### Format

**ENV**(*var.name*)

where

*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.

This function always returns a null string on the PDA version of QM.

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 11<sub>FM</sub>0<sub>VM</sub>14<sub>VM</sub>ABC<sub>FM</sub>2

B contains 12<sub>FM</sub>0<sub>VM</sub>14<sub>VM</sub>ACB<sub>FM</sub>2

C = EQS(A, B)

C now contains 0<sub>FM</sub>1<sub>VM</sub>1<sub>VM</sub>0<sub>FM</sub>1

**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 constant or reference to a matrix element.

### Format

**EQUATE** *name* **LITERALLY** "*string*"  
**EQUATE** *name* **TO** *value*  
**EQUATE** *name* **TO** *variable*  
**EQUATE** *name* **TO** *matrix*(*index1* {, *index2*)  
**EQUATE** *name* **TO** *dyn.array*<*field* {, *value* {, *subvalue* } }>  
**EQUATE** *name* **TO** **CHAR**(*seq*)

where

*name* is the name to be attached to the value or matrix reference.  
*value* is a number or a quoted string.  
*variable* is the name of a variable (scalar or matrix) or an [@-variable](#).  
*matrix* is the name of a matrix.  
*index1* is a number representing the first index to matrix.  
*index2* is a number representing the second index to a two dimensional matrix.  
*dyn.array* is the name of a variable holding a dynamic array.  
*field* is a number representing the field position within *dyn,array*.  
*value* is a number representing the value position within *dyn,array*.  
*subvalue* is a number representing the subvalue position within *dyn,array*.  
*seq* is a character sequence number.

The **EQUATE** statement defines a name that can be used later within the program as a reference to a constant, matrix element, dynamic array element, character value or QMBasic source construct. **EQUATE** statements are often placed in [include records](#) so that the same values can be incorporated into many programs without duplication.

All variants of the **EQUATE** statement except **LITERALLY** can alternatively be written using the [\\$DEFINE](#) compiler directive.

**EQUATE** *name* **LITERALLY** "*language elements*"

This form of **EQUATE** defines a *name* that is replaced by the *language elements*. These may be any QMBasic construct and must be enclosed in quotes. The **LITERALLY** keyword may be abbreviated to **LIT**.

**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)**

If the equated *name* is also a built-in function or statement, that function or statement becomes inaccessible.

### 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](#)

## 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**, [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.
- S***n* 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](#)

## 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}  
                {PASSLIST {src.list }}  
                {RTNLIST tgt.list }  
                {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. For compatibility with other multivalue environments, the **PASSLIST** clause is ignored if *src.list* is omitted.
- 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.

**PASSLIST** and **RTNLIST** are used with Pick style select list variables. The **PASSLIST** clause passes the list in the named select list variable into the executed command as list 0. The **RTNLIST** clause returns the content of the default select list in the named variable, removing the active

numbered select list.

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](#).

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

```
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.

```
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](#), [UNITL](#)

## 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:

[LNQ](#)

## 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

**FCONTROL**(*fvar*, *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 reports the state of forced writes for a dynamic hashed file. See synchronous (forced write) mode in the section that discussed [dynamic files](#) for more details.

If *qualifier* is false (0), forced writes are disabled. If *qualifier* is true (1), forced writes are enabled. For all values of *qualifier*, the previous state of forced writes is returned as the result of the function.

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.

### Example

```
OPEN "AUDIT" TO AUD.F ELSE STOP "Cannot open AUDIT file"  
VOID FCONTROL(AUD.F, FC$SYNC, @TRUE)
```

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 [OPEN](#) statement.

### See also:

[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.

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 form, 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.

Use of the [\\$NOCASE.STRINGS](#) compiler directive makes the *delimiter* case insensitive.

### Example

```
A = "1*2*3*4*5"  
S = FIELD(A, '*', 2, 3)
```

This program fragment assigns the string "2\*3\*4" to variable S.

**See also:**

[COL1\(\)](#), [COL2\(\)](#), [FIELDSTORE\(\)](#)

## FIELDSTORE()

The **FIELDSTORE()** function provides delimited substring assignment.

### Format

**FIELDSTORE**(*string*, *delimiter*, *i*, *n*, *rep.string*)

where

<i>string</i>	evaluates to the string in which replacement is to occur. If <i>string</i> 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 <b>FIELDSTORE()</b> function appears.
<i>delimiter</i>	evaluates to a string, the first character of which is used as the delimiter separating the substrings within <i>string</i> . A null <i>delimiter</i> string will cause a run time error.
<i>i</i>	evaluates to the position of the first substring to be replaced. Substring positions are numbered from one.
<i>n</i>	evaluates to the number of substrings to be replaced.
<i>rep.string</i>	evaluates to the new data to be inserted in <i>string</i> .

The value returned by the **FIELDSTORE()** function is the result of the replacement.

A statement of the form

```
S = FIELDSTORE(S, d, i, n, rep.string)
```

is equivalent to the delimited substring assignment operation

```
S[d, i, n] = 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 [\\$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 <i>rep.string</i> has only 2 substrings
B = 1*A*B*C*5	<i>rep.string</i> replaces substrings 2 to 5
C = 1*A*B*C*5	Final substring of <i>rep.string</i> is not inserted
D = 1*A*B*2*3*4*5	No substrings are deleted as <i>n</i> is zero
E = 1*A*B*5	Three substrings are deleted and <i>rep.string</i> is inserted

### See also:

[FIELD\(\)](#)

## 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:

FE\$FILE.NAME	Any file name change in the watched directory or sub-tree. Changes include renaming, creating, or deleting a file name.
FE\$DIR.NAME	Any directory-name change in the watched directory or sub-tree. Changes include creating or deleting a directory.
FE\$ATTRIBUTES	Any attribute change in the watched directory or sub-tree.
FE\$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.
FE\$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.
FE\$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.
FE\$CREATION	Any change to the creation time of file in the watched directory or sub-tree.
FE\$SECURITY	Any security-descriptor change in the watched directory or subtree causes a change notification wait operation to return.
FE\$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	FL\$OPEN	Check if file is open. Returns true (1) if <i>file.var</i> is associated with an open file, false (0) if it is not.
1	FL\$VOCNAME	Returns the VOC name used to open the file. For a file opened via QMNet, this is the <i>server:account:file</i> form of the file name.
2	FL\$PATH	Returns pathname of open file.
3	FL\$TYPE	File type. Returns one of FL\$TYPE.DH (3) - Dynamic file FL\$TYPE.DIR (4) - Directory file FL\$TYPE.SEQ (5) - Sequential file FL\$TYPE.VFS (6) - Virtual file system FL\$TYPE.DIST (7) - Distributed file
5	FL\$MODULUS	File modulus (dynamic files only)
6	FL\$MINMOD	Minimum modulus (dynamic files only)
7	FL\$GRPSIZE	Group size in multiples of 1024 bytes (dynamic files only)
8	FL\$LARGEREC	Large record size (dynamic files only)
9	FL\$MERGE	Merge load percentage (dynamic files only)
10	FL\$SPLIT	Split load percentage (dynamic files only)
11	FL\$LOAD	Current load percentage (dynamic files only)
13	FL\$AK	File has AK indices (dynamic files only)
14	FL\$LINE	Line to read or write next (sequential files only)
15	FL\$PARTS	Field mark delimited list of part file numbers (distributed files)
1000	FL\$LOADBYTES	Current load bytes (dynamic files only)
1001	FL\$READONLY	Returns true (1) if file is read-only
1002	FL\$TRIGGER	Returns trigger function name, null if none
1003	FL\$PHYSBYTES	Returns total size of file, excluding indices

1004	FL\$VERSION	Internal file version (dynamic files only)
1005	FL\$STATS.QUERY	Returns true (1) 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 (1) if the file uses case insensitive ids.
1009	FL\$FILENO	Returns the internal file number for <i>file.var</i> . 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 <b>READ</b> statement.
1013	FL\$STATUS	Returns a dynamic array as for the <b>STATUS</b> 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 <a href="#">dynamic files</a> 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 <b>FILEINFO()</b> function will not appear in this list if <i>file.var</i> 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.
1025	FL\$NOMAP	Returns true if this is a directory file with character translation of record ids suppressed.

**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](#), [DELETEDU](#), [MATREAD](#), [MATWRITE](#), [MATWRITEU](#), [READ](#), [READV](#), [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, ER\$LCK 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

<i>string</i>	evaluates to the item to be located.
<i>dyn.array</i>	is the dynamic array in which searching is to occur.
<i>occurrence</i>	is the occurrence of <i>string</i> to be found. If omitted, the first occurrence is located.
<i>field</i>	is the variable to receive the field number at which <i>string</i> is found.
<i>value</i>	is the variable to receive the value number at which <i>string</i> is found. If omitted, only the field position is returned
<i>subvalue</i>	is the variable to receive the subvalue number at which <i>string</i> is found. If omitted, only the field and, optionally, value positions are returned
<i>statement(s)</i>	are statements to be executed depending on the outcome of the <b>FIND</b> 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 [\\$NOCASE.STRINGS](#) compiler directive makes the comparison case insensitive.

### Example

Variable X contains A<sub>FM</sub>B<sub>VM</sub>C<sub>VM</sub>D<sub>FM</sub>E<sub>VM</sub>F<sub>SM</sub>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

<i>string</i>	evaluates to the item to be located.
<i>dyn.array</i>	is the dynamic array in which searching is to occur.
<i>occurrence</i>	is the occurrence of <i>string</i> to be found. If omitted, the first occurrence is located.
<i>field</i>	is the variable to receive the field number at which <i>string</i> is found.
<i>value</i>	is the variable to receive the value number at which <i>string</i> is found. If omitted, only the field position is returned
<i>subvalue</i>	is the variable to receive the subvalue number at which <i>string</i> is found. If omitted, only the field and, optionally, value positions are returned
<i>statement(s)</i>	are statements to be executed depending on the outcome of the <b>FINDSTR</b> 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 [\\$NOCASE.STRINGS](#) compiler directive makes the comparison case insensitive.

### Example

Variable X contains ABC<sub>FM</sub>DEF<sub>VM</sub>GHI<sub>VM</sub>JKL<sub>FM</sub>MNO<sub>VM</sub>PQR<sub>SM</sub>STU

```
FINDSTR 'KL' IN X SETTING F, V, S
```

The above FINDSTR would return F = 2, V = 3, S = 1

### See also:

[DEL](#), [DELETE\(\)](#), [EXTRACT\(\)](#), [FIND](#), [INS](#), [INSERT\(\)](#), [LISTINDEX\(\)](#), [LOCATE](#),  
[LOCATE\(\)](#), [REPLACE\(\)](#)

## 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:

[CLOSESEQ](#), [NOBUF](#), [READBLK](#), [READCSV](#), [READSEQ](#), [SEEK](#), [WEOFSEQ](#), [WRITEBLK](#), [WRITECSV](#), [WRITESEQ](#), [WRITESEQF](#)

## FLUSH.DH.CACHE

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 (but not in 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'

## 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

**FOLD**(*string*, *width* {, *delim*})

**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)  
LOOP  
    CRT REMOVE(X, CODE)  
WHILE CODE  
REPEAT
```

The above program fragment prints

```
The quick  
brown fox  
jumps over  
the lazy  
dog
```

## 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** Centres 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}  
    statement(s)  
NEXT {var}
```

```
FOR var = value1, value2, value3...  
    statement(s)  
NEXT {var}
```

```
FOR EACH var IN string  
    statement(s)  
NEXT {var}
```

where

<i>var</i>	is the loop control variable.
<i>start.expr</i>	evaluates to the value to be placed in <i>var</i> for the first iteration of the loop.
<i>limit.expr</i>	evaluates to the value beyond which <i>var</i> must not pass.
<i>step.expr</i>	evaluates to the value by which <i>var</i> should be incremented between iterations of the loop. If omitted, <i>step.expr</i> defaults to one.
<i>value1,value2...</i>	are values to be assigned to <i>var</i> .
<i>string</i>	is an expression that evaluates to a mark character delimited list of values to be assigned to <i>var</i> .
<i>statement(s)</i>	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 *var* should not be changed within the loop as this may lead to unexpected results.

Use of non-integer values for *start.expr*, *limit.expr* and *step.expr* is not recommended where rounding errors in incrementing *var* and the loop termination comparison may lead to unexpected effects.

For best performance, *limit.expr* and *step.expr* should be constants or simple variable references as they are evaluated for every iteration of the loop. In the unlikely case that *limit.expr* or *step.expr* use the value of *var*, note that these expressions are evaluated before the first cycle of the loop assigns the start value to *var*.

The second form of the **FOR / NEXT** statement executes the loop for each of the supplied values of *var* in turn. The value of *var* on leaving the loop will be the final value for which the loop was executed. The values to be assigned to *var* may be constants or expressions.

The third form of the **FOR / NEXT** statement executes the loop for each mark character delimited value in *string*. This form of **FOR / NEXT** is broadly equivalent to using the [REMOVE](#) statement to extract successive values from *string*. The value of *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 *var* on leaving the loop will be the final value for which the loop was executed.

If *var* is present in the **NEXT** statement it must be the same *var* as in the **FOR** statement at the head of the loop. Use of *var* in the **NEXT** statement aids program readability and is checked by the compiler for correct matching of **FOR** and **NEXT** statements.

The [WHILE](#), [UNTIL](#) and [EXIT](#) statements can be used with the **FOR / NEXT** loop to provide another loop termination control. The [CONTINUE](#) statement causes a jump to the start of the next iteration.

**FOR / NEXT** loops may be nested to any depth. Jumping to a label within a **FOR / NEXT** loop from outside may have undefined effects.

The **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

```
FOR I = 1 TO 10
```

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 `FOR.STORE.BEFORE.TEST` option of the [\\$MODE](#) compiler directive can be used to modify the behaviour of **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

```
FOR I = 1 TO 10 STEP 2
  DISPLAY I
NEXT I
```

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 through 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

**FORMCSV**(*string*)

where

*string* is the string to be transformed

The **FORMCSV()** function converts the supplied *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 [EXECUTE](#), restoring the previous mode on return from the executed command.

### See also:

[CSV.MODE](#), [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.

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, 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 **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. 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 [\\$MODE](#) compiler directive is enabled. In this case, the default value will be applied if the argument variable passed from the calling program is unassigned.

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

```
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,

```
FUNCTION INVOICE(P, (Q = 7)) VAR.ARGS
```

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 matrix passed as an argument cannot be redimensioned in the function.

Programs that use the function must include a [DEFFUN](#) statement to define the function template.

### Example

```
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

```
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 11<sub>FM</sub>0<sub>VM</sub>14<sub>VM</sub>ABC<sub>FM2</sub>

B contains 12<sub>FM</sub>0<sub>VM</sub>14<sub>VM</sub>ACB<sub>FM2</sub>

C = GES(A, B)

C now contains 0<sub>FM</sub>1<sub>VM</sub>1<sub>VM</sub>1<sub>FM</sub>1

**See also:**

[ANDS\(\)](#), [EQS\(\)](#), [GTS\(\)](#), [IFS\(\)](#), [LES\(\)](#), [LTS\(\)](#), [NES\(\)](#), [NOTS\(\)](#), [ORS\(\)](#), [REUSE\(\)](#)

## GET(ARG.)

The **GET(ARG.)** statement fetches command line arguments.

### Format

```
GET(ARG. {, argno}) var {THEN statement(s)} {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 [!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 *argno* element or with *argno* less than one, returns the value of the next command line argument in *var*. Specifying a positive and non-zero value for *argno* retrieves the specified argument.

The **THEN** and **ELSE** clauses are both optional. If present, the **THEN** clause will be executed if the argument is retrieved successfully. The **ELSE** clause is executed if the argument was not present in the command line.

If the program is started using the [RUN](#) or [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

```
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. This function is not available on the PDA version of QM.

### 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>
```

## GETLIST

The **GETLIST** statement restores a select list from the \$SAVEDLISTS file.

### Format

```
GETLIST name { TO list.no }  
  { 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.

*statement(s)* are statements to be executed depending on the outcome of the operation.

The **GETLIST** statement restores 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.

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 exist or cannot be restored for any other reason, the **ELSE** clause is executed.

**See also:**

[SAVELIST](#)

## 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:

Parameter	Key	Meaning
1	NLS\$CURRENCY	Default currency symbol. Maximum 8 characters.
2	NLS\$THOUSANDS	Default thousands separator character.
3	NLS\$DECIMAL	Default decimal separator character.
4	NLS\$IMPLICIT.DECIMAL	Default decimal separator character for implicit conversions. See <a href="#">SETNLS</a> for details.

### See also:

[Conversion codes](#), [Format codes](#), [OCONV\(\)](#)

## GETPU()

The **GETPU()** function gets the characteristics of a print unit.

### Format

**GETPU**(*key, unit*)

where

<i>key</i>	identifies the parameter to retrieved. This may be:		
0	PU\$DEFINED	Is print unit defined?	
1	PU\$MODE	Print unit mode	
2	PU\$WIDTH	Characters per line	
3	PU\$LENGTH	Lines per page	
4	PU\$TOPMARGIN	Top margin size	
5	PU\$BOTMARGIN	Bottom margin size	
6	PU\$LEFTMARGIN	Left margin size	
7	PU\$SPOOLFLAGS	Various print unit flags	
9	PU\$FORM	Form name (not used by all spoolers)	
10	PU\$BANNER	Banner page text	
11	PU\$LOCATION	Printer / file name	
12	PU\$COPIES	Number of copies to print	
15	PU\$PAGENUMBER	Current page number	
1002	PU\$LINESLEFT	Lines left on page	
1003	PU\$HEADERLINES	Lines occupied by header	
1004	PU\$FOOTERLINES	Lines occupied by footer	
1005	PU\$DATA LINES	Lines between header and footer	
1006	PU\$OPTIONS	Options to be passed to the spooler	
1007	PU\$PREFIX	Pathname of file holding prefix data to be added to the start of the output	
1008	PU\$SPOOLER	Spooler to be used (ignored on Windows)	
1009	PU\$OVERLAY	Catalogued overlay subroutine name (see <a href="#">SETPTR</a> )	
1010	PU\$CPI	Characters per inch (may be non-integer value)	
1011	PU\$PAPER.SIZE	Paper size. See SYSCOM PCL.H	
1012	PU\$LPI	Lines per inch	
1013	PU\$WEIGHT	Font stroke weight. See SYSCOM PCL.H	
1014	PU\$SYMBOL.SET	Symbol set. See SYSCOM PCL.H	
1015	PU\$STYLE	Query processor style. See the Query processor <a href="#">STYLE</a> option for details.	
1016	PU\$NEWLINE	Newline string for this print unit	

1017	PU\$PRINTER.NAM E	Name of printer
1018	PU\$FILE.NAME	File name for output directed to a file
1019	PU\$SEQNO	Hold file sequence number. See also <a href="#">@SEQNO</a> .
1020	PU\$FORM.QUEUE	Last assigned form queue number with <a href="#">SP.ASSIGN</a> . 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
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

**GOSUB** *label*{:}

**GOSUB** *label*{:}(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 [LOCAL SUBROUTINE](#) statement.

The optional colon after the *label* has no effect on the action of the statement.

The program continues execution at the given *label*, saving the location of the **GOSUB** for a later [RETURN](#) which will resume execution at the statement following the **GOSUB**. See also the [RETURN TO](#) statement for details of alternate returns.

QMBasic defines two styles of internal subroutine. A conventional internal subroutine, as found in other multivalued database products, has no formal start or end. The *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 [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 local subroutine, is introduced by the [LOCAL](#) statement and is terminated by **END**. Local subroutines may have arguments and may have private local variables that are not visible outside the subroutine which are stacked if the subroutine is called recursively.

### Examples

```
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`.

```
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.

## 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.

## 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

**GTS**(*expr1*, *expr2*)

where

*expr1* 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 11<sub>FM</sub>0<sub>VM</sub>14<sub>VM</sub>ABC<sub>FM</sub>2

B contains 12<sub>FM</sub>0<sub>VM</sub>14<sub>VM</sub>ACB<sub>FM</sub>2

C = GTS(A, B)

C now contains 0<sub>FM</sub>0<sub>VM</sub>1<sub>VM</sub>1<sub>FM</sub>0

### 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

**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 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** Centres 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

**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*        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

**HUSH OFF** {**SETTING** *var*}

**HUSH ON** {**SETTING** *var*}

**HUSH** *expr* {**SETTING** *var*}

where

*expr* evaluates to a number.

*var* is a variable to receive the previous state of display output control.

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

```
HUSH ON
EXECUTE "SELECT STOCK.FILE WITH QTY > 50"
HUSH OFF
```

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 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.

### See also:

[Conversion codes](#), [OCONV\(\)](#)

## IDIV()

The **IDIV()** function divides one integer by another and returns an integer result.

### Format

**IDIV**(*dividend*, *divisor*)

where

*dividend* evaluates to the integer to be divided.

*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 *divisor* will cause a run time error.

Integer division can also be performed using the // operator.

### Example

```
X = 7
Y = 2
Z1 = X / Y
Z2 = IDIV(X, Y)
```

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:

[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 multivalued 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 dependant 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.

### Example

A contains 1 0 1 1 0

B contains 11 22 33 44 91 36 7

C contains 14 61 20 35 18 3

D = IFS(A, B, C)

D now contains 11 61 24 91 36 3

### See also:

[ANDS\(\)](#), [EQS\(\)](#), [GES\(\)](#), [GTS\(\)](#), [LES\(\)](#), [LTS\(\)](#), [NES\(\)](#), [NOTS\(\)](#), [ORS\(\)](#), [REUSE\(\)](#)

## IN

The **IN** statement reads a single byte 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 ASCII character number, not the character itself.

*timeout* is the timeout period in tenths of a second.

The **IN** statement reads a single byte from the terminal, returning the character value in *var*. Unless the character is a non-printing control code, it is echoed to the terminal.

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\(\)](#), [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.

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 [\\$NOCASE.STRINGS](#) compiler directive makes the comparison case insensitive.

### 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)
```

sets N to 5.

## INDICES()

The **INDICES()** function returns information about [alternate key indices](#).

### Format

**INDICES**(*file.var*)                      To retrieve a list of indices  
**INDICES**(*file.var, index.name*)      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.

Value 1	Index type (D, I, A, S or C)
Value 2	Set to 1 if the index needs to be built, otherwise null
Value 3	Set to 1 if the index is null-suppressed by use of the NO.NULLS option to <a href="#">CREATE.INDEX</a> , otherwise null
Value 4	Set to 1 if updates are enabled, otherwise null
Value 5	Internal AK numbers
Value 6	The key sort mode of record ids <u>within</u> each index entry (L or R), null for indices created prior to release 2.2-16 that were unsorted.
Value 7	Reserved for future use
Value 8	Set to 1 if the index is case insensitive, otherwise null
Value 9	Set to 1 if index is being built, otherwise null

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**

```
INDEX.NAMES = INDICES(FVAR)
NUM.INDICES = DCOUNT(INDEX.NAMES, @FM)
FOR I = 1 TO NUM.INDICES
  NAME = INDEX.NAMES<I>
  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

**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

**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

<b><u>DIMENSION</u></b>	0 if successful, 1 if insufficient memory.
<b><u>MATPARSE</u></b>	The number of elements assigned. Zero if overflows.
<b><u>MATREAD</u></b>	The number of elements assigned. Zero if overflows.
<b><u>MATREADL</u></b>	The number of elements assigned. Zero if overflows.
<b><u>MATREADU</u></b>	The number of elements assigned. Zero if overflows.
<b><u>OPEN</u></b>	The modulus of a dynamic file.

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. 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.

```
N = INMAT(A)
FOR I = 1 TO N
  A(I) += 1
NEXT I
```

This program fragment adds one to each element of matrix A. The **INMAT()** function is used because the matrix was dimension elsewhere and hence its size is not known.

## INPUT

The **INPUT** statement enables entry of data from the keyboard or from previously stored **DATA** statements.

### Format

```
INPUT var {, length} {_} {:} {TIMEOUT wait} {HIDDEN} {UPCASE}  
{THEN statement(s)}  
{ELSE statement(s)}
```

where

<i>var</i>	is the variable in which the data is to be stored.
<i>length</i>	is the maximum length of data to be allowed.
<b>HIDDEN</b>	echoes characters back to the screen as asterisks for password type fields.
<b>NO.ECHO</b>	Suppresses echo of input data.
<b>TIMEOUT</b> <i>wait</i>	Sets a timeout period in seconds. If input is not received in this time, the <b>INPUT</b> terminates, leaving <i>var</i> unchanged. The keywords <b>FOR</b> or <b>WAITING</b> can be used in place of <b>TIMEOUT</b> for compatibility with other environments.
<b>UPCASE</b>	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.

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 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

```
INPUT S, -1
```

will set S to 1 if there is data waiting, 0 if no data is waiting.

### 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

```
INPUT ACCOUNT.NO, 10
```

This statement reads data into ACCOUNT.NO with a maximum length of 10 characters.

```
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.

## INPUT @

The **INPUT @** statement enables entry of data from the keyboard at a specific screen position or from previously stored [DATA](#) statements.

### Format

```
INPUT @(x,y) {,} {::} var {, length} {__} {::} {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.

*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:

<b>APPEND</b>	Position the cursor at the end of the data. Use of this keyword also implies <b>EDIT</b> mode.
<b>EDIT</b>	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.
<b>HIDDEN</b>	echoes characters back to the screen as asterisks for password type fields.
<b>NO.ECHO</b>	Suppresses echo of input data.
<b>OVERLAY</b>	Starts in "overlay" mode where data entered by the user replaces the character under the cursor rather than being inserted.
<b>PANNING</b>	Allows entry of an unlimited number of characters in a field width of the given <i>length</i> by panning the data if it is longer than the display width. Use of this option requires <i>length</i> to be specified and implies the presence of the underscore.
<b>TIMEOUT</b> <i>wait</i>	Sets a timeout period in seconds. If input is not received in this time, the <b>INPUT</b> terminates, leaving <i>var</i> unchanged. The keywords <b>FOR</b> or <b>WAITING</b> can be used in place of <b>TIMEOUT</b> for compatibility with other environments.
<b>UPCASE</b>	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 of *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		Delete all characters after the cursor
	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. Unless the <b>OVERLAY</b> option is used, the input begins in insert mode.

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.

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](#), [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](#)

## INPUTFIELD

The **INPUTFIELD** statement enables entry of data from the keyboard at a specific screen position or from previously stored **DATA** statements. It differs from **INPUT @** in that it terminates 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

```
INPUTFIELD @(x, y) {,} {:] var, length [_] {:] {format} {modes}
{ THEN statement(s) }
{ ELSE statement(s) }
```

where

<i>x, y</i>	are the screen position (column and line) at which input is to occur.														
<i>var</i>	is the variable in which the data is to be stored.														
<i>length</i>	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.														
<i>format</i>	is the <a href="#">format specification</a> to be used for initial display of <i>var</i> and to redisplay the data on completion of input.														
<i>modes</i>	are any combination of the following keywords: <table> <tr> <td><b>APPEND</b></td> <td>Position the cursor at the end of the data. Use of this keyword also implies <b>EDIT</b> mode.</td> </tr> <tr> <td><b>EDIT</b></td> <td>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.</td> </tr> <tr> <td><b>HIDDEN</b></td> <td>echoes characters back to the screen as asterisks for password type fields.</td> </tr> <tr> <td><b>NO.ECHO</b></td> <td>Suppresses echo of input data.</td> </tr> <tr> <td><b>OVERLAY</b></td> <td>Starts in "overlay" mode where data entered by the user replaces the character under the cursor rather than being inserted.</td> </tr> <tr> <td><b>PANNING</b></td> <td>Allows entry of an unlimited number of characters in a field width of the given <i>length</i> by panning the data if it is longer than the display width. Use of this option requires <i>length</i> to be specified and implies the presence of the underscore.</td> </tr> <tr> <td><b>TIMEOUT</b> <i>wait</i></td> <td>Sets a timeout period in seconds. If input is not received in this time, the <b>INPUTFIELD</b> terminates, leaving <i>var</i> unchanged. The keywords <b>FOR</b> or <b>WAITING</b> can be used in place of <b>TIMEOUT</b> for compatibility with other</td> </tr> </table>	<b>APPEND</b>	Position the cursor at the end of the data. Use of this keyword also implies <b>EDIT</b> mode.	<b>EDIT</b>	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.	<b>HIDDEN</b>	echoes characters back to the screen as asterisks for password type fields.	<b>NO.ECHO</b>	Suppresses echo of input data.	<b>OVERLAY</b>	Starts in "overlay" mode where data entered by the user replaces the character under the cursor rather than being inserted.	<b>PANNING</b>	Allows entry of an unlimited number of characters in a field width of the given <i>length</i> by panning the data if it is longer than the display width. Use of this option requires <i>length</i> to be specified and implies the presence of the underscore.	<b>TIMEOUT</b> <i>wait</i>	Sets a timeout period in seconds. If input is not received in this time, the <b>INPUTFIELD</b> terminates, leaving <i>var</i> unchanged. The keywords <b>FOR</b> or <b>WAITING</b> can be used in place of <b>TIMEOUT</b> for compatibility with other
<b>APPEND</b>	Position the cursor at the end of the data. Use of this keyword also implies <b>EDIT</b> mode.														
<b>EDIT</b>	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.														
<b>HIDDEN</b>	echoes characters back to the screen as asterisks for password type fields.														
<b>NO.ECHO</b>	Suppresses echo of input data.														
<b>OVERLAY</b>	Starts in "overlay" mode where data entered by the user replaces the character under the cursor rather than being inserted.														
<b>PANNING</b>	Allows entry of an unlimited number of characters in a field width of the given <i>length</i> by panning the data if it is longer than the display width. Use of this option requires <i>length</i> to be specified and implies the presence of the underscore.														
<b>TIMEOUT</b> <i>wait</i>	Sets a timeout period in seconds. If input is not received in this time, the <b>INPUTFIELD</b> terminates, leaving <i>var</i> unchanged. The keywords <b>FOR</b> or <b>WAITING</b> can be used in place of <b>TIMEOUT</b> 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 for any other key.

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@](#), [KEYCODE\(\)](#), [KEYEDIT](#), [KEYEXIT](#), [KEYTRAP](#)

## INS

The **INS** statement and **INSERT()** function insert a field, value or subvalue into a dynamic array.

### Format

**INS** *string* **BEFORE** *dyn.array*<*field* {, *value* {, *subvalue*}}>

**INSERT**(*dyn.array*, *field* {, *value* {, *subvalue*}} , *string*)

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.

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,

```
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.

### Example

```
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.

**See also:**

[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.

## ITYPE()

The **ITYPE()** function executes a compiled I or C-type dictionary record or an A or S-type with a correlative.

### Format

**ITYPE**(*itype*)

where

*itype* 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 return 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.

## KEYCODE()

The **KEYCODE()** function reads a single keystroke from the keyboard.

### Format

**KEYCODE**({*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.

**KEYCODE()** does 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
- Backtab
- 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.

### Mouse Input on AccuTerm

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 and will almost certainly require device specific programming. The following code sample works for the AccuTerm emulator in its vt100 mode if the terminfo kmous key is defined as "\E[101~".

```
C = KEYCODE ( )
IF SEQ(C) = K$MOUSE THEN
  S = ' '
  LOOP
  C = KEYIN ( )
```

```

UNTIL C = 'R'
  S := C
REPEAT
  ROW = MATCHFIELD(S, '0X0N;0N', 2)
  COL = MATCHFIELD(S, '0X0N;0N', 4)
END

```

Note that AccuTerm numbers rows and columns from one.

Mouse click detection is disabled by default. It can be enabled by use of

```
DISPLAY CHAR(27) : CHAR(2) : '1' :
```

and subsequently disabled using

```
DISPLAY CHAR(27) : CHAR(2) : '0' :
```

### Stylus Input on a PDA

Stylus taps are enabled on a PDA using the @(IT\$STYLUS) function. When enabled, a stylus tap sends char(200) (K\$MOUSE) followed by the column and row coordinates, separated by a comma and terminated by a carriage return. The following function extends KEYCODE to recognise stylus taps, returning the mouse code to the caller and leaving the screen coordinates in a common block.

```

FUNCTION KEY()
  $CATALOGUE KEY

  $INCLUDE KEYS.H
  $INCLUDE KEYIN.H

  COMMON /STYLUS/STYLUS.ROW, STYLUS.COL

  DISPLAY @(IT$STYLUS,1):
  K = KEYCODE()
  DISPLAY @(IT$STYLUS,0):
  IF SEQ(K) = K$MOUSE THEN
    ECHO OFF
    INPUT S          ;* Get coordinate data
    HUSH OFF
    STYLUS.COL = MATCHFIELD(S, '0N","0N', 1)
    STYLUS.ROW = MATCHFIELD(S, '0N","0N', 3)
  END

  RETURN K
END

```

## 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

**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 <i>key</i> - 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 in 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

The printing characters and control characters are all represented by their normal ASCII characters. Other keystrokes such as the function keys, ALT sequences and special keys (Home, Delete, cursor moves, etc) are represented by characters with values of 128 and upwards. [Click here for a table of key code values.](#)

### See also:

[IN](#), [KEYCODE\(\)](#), [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

**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 <i>key</i> - 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 is 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](#)

## 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

**LEN**(*string*)

where

*string* is the string for which the length is to be returned.

The **LEN()** function returns the length of *string* including any trailing spaces.

### Example

```
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.

## 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 11<sub>FM</sub>0<sub>VM</sub>14<sub>VM</sub>ABC<sub>FM2</sub>

B contains 12<sub>FM</sub>0<sub>VM</sub>14<sub>VM</sub>ACB<sub>FM2</sub>

C = LES(A, B)

C now contains 1<sub>FM</sub>1<sub>VM</sub>0<sub>VM</sub>0<sub>FM1</sub>

### See also:

[ANDSQ](#), [EQSQ](#), [GESQ](#), [GTSQ](#), [IFSQ](#), [LTSQ](#), [NESQ](#), [NOTSQ](#), [ORSQ](#), [REUSEQ](#)

## 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

$\text{LN}(expr)$

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

$N = \text{LN}(X)$

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 redimension 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 {,value {,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

<i>string</i>	evaluates to the item to be located.
<i>dyn.array</i>	is the dynamic array in which searching is to occur.
<i>field</i>	is the field at which searching is to commence.
<i>value</i>	is the value at which searching is to commence. If omitted or zero, searching occurs at the field level.
<i>subvalue</i>	is the subvalue at which searching is to commence. If omitted or zero, searching occurs at the value level.
<i>order</i>	evaluates to the ordering string as described below. If omitted, no ordering is assumed.
<i>var</i>	is the variable to receive the position value.
<i>statement(s)</i>	are statements to be executed depending on the outcome of the <b>LOCATE</b> 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 multivalued 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 multivalued 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. The *order* must evaluate to a two character string, which is

- AL** Ascending, left justified. Items are considered to be sequenced in ascending collating sequence order.
- AR** Ascending, right justified. Items are considered to be sequenced in ascending collating sequence order. Where the item being examined is not of the same length as the *string* being located, the shorter of the two is right aligned within the length of the longer prior to comparison. Integer numeric data that can be represented as a 64 bit value is treated as a special case and is sorted into correct numeric sequence. In the Pick style syntax, fractional values are also allowed in numeric sorting.
- DL** Descending, left justified. Similar to **AL** except that the list is held in descending collating sequence.
- DR** Descending, right justified. Similar to **AR** except that the list is held in descending collating sequence.

The ordering string must be in upper case. 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.

Use of the [\\$NOCASE.STRINGS](#) compiler directive makes the comparison case insensitive. When used with a **BY** clause, the sorting is effectively as though both *string* and *dyn.array* are in uppercase.

## 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<I>
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 attempting to acquire each lock in turn and checking the value of the [STATUS\(\)](#) function. 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. This statement has no effect on the PDA version of QM.

### 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, 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
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:

<b>Original</b>	<b>Replacement</b>
Item mark	Field mark
Field mark	Value mark
Value mark	Subvalue mark
Subvalue mark	Text mark
Text mark	Text mark (unchanged)

### 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

**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 11<sub>FM</sub>0<sub>VM</sub>14<sub>VM</sub>ABC<sub>FM</sub>2

B contains 12<sub>FM</sub>0<sub>VM</sub>14<sub>VM</sub>ACB<sub>FM</sub>2

C = LTS(A, B)

C now contains 1<sub>FM</sub>0<sub>VM</sub>0<sub>VM</sub>0<sub>FM</sub>0

### See also:

[ANDSQ](#), [EQSQ](#), [GESQ](#), [GTSQ](#), [IFSQ](#), [LESQ](#), [NESQ](#), [NOTSQ](#), [ORSQ](#), [REUSEQ](#)

## 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

**MAT** *matrix* = *expr*

**MAT** *matrix2* = **MAT** *matrix1*

**IF** **MAT** *matrix1* = **MAT** *matrix2* **THEN** ...

where

*matrix* is the name of a previously dimensioned matrix.

*expr* evaluates to the value to be stored in each matrix element.

The first format of this statement copies the value of *expr* into all elements of *matrix*. The zero element is set to a null string.

The second format copies elements from *matrix1* to *matrix2* row by row. If the number of columns differs, the copy behaves as depicted below.

Source:			Target:			
	1	2		1	2	3
1	A	B	1	A	B	C
2	C	D	2	D	E	F
3	E	F				

The zero element of *src.matrix* is copied to the zero element of *matrix*.

If *src.matrix* has more elements than *matrix*, the excess elements are ignored. If *src.matrix* has fewer elements than *matrix*, the remaining elements of *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 *matrix1* is the same as the content of *matrix2*. Although most likely to be used as part of an IF statement as shown above, the

**MAT** *matrix1* = **MAT** *matrix2*

component of this statement may be used anywhere that a boolean value is appropriate.

### Examples

```
DIM A(25)
MAT 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.

## 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

<i>var</i>	is the variable to receive the dynamic array.
<i>mat</i>	is the matrix from which data is to be taken.
<i>start.expr</i>	evaluates to the index of the first matrix element to be used. If omitted or less than one, this defaults to one.
<i>end.expr</i>	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.
<i>delimiter</i>	evaluates to the delimiter to be used between elements of <i>mat</i> . This may be more than one character. If omitted, this defaults to the field mark.

The **MATBUILD** statement constructs a dynamic array by concatenating elements of *mat* from element *start.expr* to the last non-null element before element *end.expr*. The *delimiter* is inserted between each element. With the default style of matrix, if the zero element of *mat* is non-null, a *delimiter* followed by the content of the zero element is appended to the end of the resultant dynamic array. Pick style matrices do not have a zero element. See the [COMMON](#) and [DIMENSION](#) statements for more details.

### 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
<i>n</i> X	Exactly <i>n</i> characters of any type
<i>n-m</i> X	Between <i>n</i> and <i>m</i> characters of any type
0A	Zero or more alphabetic characters
<i>n</i> A	Exactly <i>n</i> alphabetic characters
<i>n-m</i> A	Between <i>n</i> and <i>m</i> alphabetic characters
0N	Zero or more numeric characters
<i>n</i> N	Exactly <i>n</i> numeric characters
<i>n-m</i> N	Between <i>n</i> and <i>m</i> numeric characters
" <i>string</i> "	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, *n*A, 0N, *n*N 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-m*X 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-m*A 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*.

### Example

```
VAR = "123":@VM:"ABC:@FM:"456"  
X = MATCHESS(VAR, "3N")
```

The variable X will be returned as

```
1VM0FM1
```

### See also:

[Pattern Matching](#)

## MATCHFIELD

The **MATCHFIELD()** function extracts a portion of a string that matches a pattern element.

### 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
<i>n</i> X	Exactly <i>n</i> characters of any type
<i>n</i> - <i>m</i> X	Between <i>n</i> and <i>m</i> characters of any type
0A	Zero or more alphabetic characters
<i>n</i> A	Exactly <i>n</i> alphabetic characters
<i>n</i> - <i>m</i> A	Between <i>n</i> and <i>m</i> alphabetic characters
0N	Zero or more numeric characters
<i>n</i> N	Exactly <i>n</i> numeric characters
<i>n</i> - <i>m</i> N	Between <i>n</i> and <i>m</i> numeric characters
" <i>string</i> "	A literal string which must match exactly. Either single or double quotation marks may be used. Unlike the <b>MATCHES</b> operator, <i>string</i> 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, *n*A, 0N, *n*N 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*-*m*X 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 **MATCHFIELD()** function tries each template in turn until one is a successful match against *string*.

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*.

### Example

```
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
```

### See also:

[Pattern Matching](#)

## 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}  
MATPARSE mat FROM string {USING delimiter}
```

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.

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.

In all cases, 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 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
```

This program fragment assigns successive fields of string S to elements of matrix A. If there are more than 30 fields, the remaining fields and delimiting mark characters are stored in A(0).

**See also:**

[MATBUILD](#)

## MATREAD

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 FROM file.var, record.id {ON ERROR statement(s)}
  {LOCKED statement(s)}
  {THEN statement(s)}
  {ELSE statement(s)}
```

where

<i>mat</i>	is the matrix into which fields are to be assigned. This matrix must already have been dimensioned.
<i>file.var</i>	is the file variable associated with the file.
<i>record.id</i>	evaluates to the key of the record to be read.
<i>statement(s)</i>	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.

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 [INMAT\(\)](#) function returns zero to indicate this condition. See the [COMMON](#) and [DIMENSION](#) statements for more details.

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)
```

**Example**

```
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 FROM file.var  
{ THEN statement(s) }  
{ ELSE statement(s) }
```

where

*matrix* is the dimensioned matrix 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.

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  
REPEAT
```

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](#)

## 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

```
MATWRITE mat TO file.var, record.id {ON ERROR statement(s)}
```

where

<i>mat</i>	is the matrix from which data is to be taken.
<i>file.var</i>	is the file variable associated with the file.
<i>record.id</i>	evaluates to the key of the record to be written.
<i>statement(s)</i>	are statements to be executed depending on the outcome of the operation.

The **MATWRITE** statement constructs a dynamic array by concatenating elements of *mat*, inserting a field mark between each element.

If the zero element of *mat* is a null string or unassigned, assembly of the dynamic array terminates after the last non-null element of *mat*. No trailing null fields will be written for later unassigned elements of *mat*.

If the zero element of *mat* contains data, all elements of *mat* are used and the zero element is concatenated as the final field of the record.

Pick style matrices do not have a zero element. See the [COMMON](#) and [DIMENSION](#) statements for more details.

If the **ON ERROR** clause is taken, the [STATUS\(\)](#) function can be used to determine the cause of the error. Otherwise, for the **MATWRITE** statement, the [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 [STATUS\(\)](#) function value is undefined after a successful **MATWRITEU** statement.

A **MATWRITE** statement is equivalent to a [MATBUILD](#) followed by a [WRITE](#).

```
MATBUILD REC FROM mat
WRITE REC TO file.var, record.id ON ERROR statement(s)
```

### Example

```
MATWRITE ITEMS TO ITEM.FILE, "ITEM.LIST" ON ERROR
  ABORT "Error " : STATUS() : " writing item list"
END
IF STATUS() = 0 THEN DISPLAY "Lock released"
```

This program fragment writes a record built from elements of matrix `ITEMS`. If it was locked prior to the `MATWRITE`, a message is displayed.

## MAX()

The **MAX()** function returns the greater of two values.

### Format

**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

```
A = 6  
B = 8  
C = MAX(A, B)
```

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 [CHECKSUM\(\)](#) function. The returned value is a 32 digit hexadecimal string.

### Example

```
DISPLAY MD5('The quick brown fox jumps over the lazy dog')
```

The above example would print '9e107d9d372bb6826bd81d3542a419d6'.

### See also:

[CHECKSUM\(\)](#)

**MIN()**

The **MIN()** function returns the lesser of two values.

**Format**

**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**

```
A = 6  
B = 8  
C = MIN(A, B)
```

C will be set to the lesser of the values in A and B, in this case 6.

**See also:**

[MAX\(\)](#)

## MINIMUM()

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

**MOD**(*dividend*, *divisor*)

where

*dividend* evaluates to a number or a numeric array.

*divisor* evaluates to a number or a numeric array.

The **MOD()** function returns the modulus value of dividing *dividend* by *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.

<b>Dividend</b>	<b>Divisor</b>	<b>MOD()</b>
530	100	30
-530	100	70
530	-100	-70
-530	-100	-30
0	100	0
0	-100	0
100	0	100
-100	0	-100

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 **QMB.REUSE** function is used.

### Example

N = MOD(T, 30)

---

This statement finds the modulus of dividing T by 30 and assigns this to N.

**See also:**

[REM\(\)](#), [ROUNDDOWN\(\)](#), [ROUNDUP\(\)](#)

## MVDATE()

The **MVDATE()** function returns the multivalued 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 multivalued 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 multivalued 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 multivalued 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](#)

## MVTIME()

The **MVTIME()** function returns the multivalued style time value (seconds since midnight) for a supplied epoch value using the currently selected time zone.

### Format

**MVTIME**(*epoch.value*)

where

*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

```
T = 1234567890
SET.TIMEZONE 'GMT'
DISPLAY MVTIME(T)
SET.TIMEZONE 'Pacific/Wake'
DISPLAY MVTIME(T)
```

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 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\(\)](#), [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 interruptable 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()

The **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

$N = \text{NEG}(A)$

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

**NES**(*expr1*, *expr2*)

where

*expr1* 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 11<sub>FM</sub>0<sub>VM</sub>14<sub>VM</sub>ABC<sub>FM</sub>2

B contains 12<sub>FM</sub>0<sub>VM</sub>14<sub>VM</sub>ACB<sub>FM</sub>2

C = NES(A, B)

C now contains 1<sub>FM</sub>0<sub>VM</sub>0<sub>VM</sub>1<sub>FM</sub>0

### 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](#), [WRITESEQ](#) and [WRITESEQF](#) 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:

[CLOSESEQ](#), [CREATE](#), [OPENSEQ](#), [READBLK](#), [READCSV](#), [READSEQ](#), [SEEK](#), [WEOFSEQ](#), [WRITEBLK](#), [WRITECSV](#), [WRITESEQ](#), [WRITESEQF](#)

## 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 numeric value

The **NOT()** function returns the logical inverse of *expr*. If *expr* is zero (or a null string which equates to zero), the **NOT()** function returns true (1). If *expr* is non-zero, the **NOT()** function returns false (0).

### 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 (1) if the *string* can be converted to a number. The function returns false (0) 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.

### 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.

## OBJECT()

The **OBJECT()** function instantiates an object for [object oriented programming](#).

### Format

**OBJECT**(*cat.name*, {*args...*})

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.

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.

```
OBJ = OBJECT ( "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 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.

### 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 | OI\$ISOBJ     | Returns true (1) if <i>var</i> is an object, false (0) if not.      |
| 1 | OI\$CLAS<br>S | Returns the class module catalogue name associated with the object. |

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

**OCONV**(*expr*, *conv.spec*)

**OCONVS**(*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 **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

- 0 Successful conversion.
- 1 Data to convert was invalid for the conversion specification.
- 2 The conversion code was invalid.

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

```
DT = 14992
DISPLAY OCONV(DT, 'D2')
```

The above example converts a date from internal format as a value 14992 to its external form as 16 JAN 09.

```
DT = 14992VM14000
S = OCONVS(DT, 'D2')
```

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<sub>VM</sub>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

```
ON expr 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

```
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

**ON** *expr* **GOTO** *label1*{:}, *label2*{:}, *label3*{:}

**ON** *expr* **GO** {**TO**} *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 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

```
ON ACTION GOTO DISPLAY.REPORT,  
              PRINT.REPORT,  
              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

<i>dict.expr</i>	evaluates to <b>DICT</b> 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.								
<i>filename.expr</i>	evaluates to the VOC name of the file to be opened.								
<i>options</i>	control the manner in which the file is opened and may be any combination of: <table> <tr> <td><b>NON.TRANSACTIONAL</b></td> <td>Ignore transaction boundaries</td> </tr> <tr> <td><b>NO.MAP</b></td> <td>Suppress id character translations in directory files</td> </tr> <tr> <td><b>READONLY</b></td> <td>Disable file updates</td> </tr> <tr> <td><b>SYNC</b></td> <td>Force write all file updates</td> </tr> </table>	<b>NON.TRANSACTIONAL</b>	Ignore transaction boundaries	<b>NO.MAP</b>	Suppress id character translations in directory files	<b>READONLY</b>	Disable file updates	<b>SYNC</b>	Force write all file updates
<b>NON.TRANSACTIONAL</b>	Ignore transaction boundaries								
<b>NO.MAP</b>	Suppress id character translations in directory files								
<b>READONLY</b>	Disable file updates								
<b>SYNC</b>	Force write all file updates								
<i>file.var</i>	is the name of the variable to hold the file reference for use in later operations on this file.								
<i>statement(s)</i>	are statements to be executed depending on the outcome of the <b>OPEN</b> 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 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 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 discussed [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 **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 **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 STOCK.FILE. If the open fails, the program aborts with an error message.

### See also:

[OPENPATH](#), [OPENSEQ](#)

## 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

<i>pathname</i>	evaluates to the VOC name of the file to be opened.								
<i>options</i>	control the manner in which the file is opened and may be any combination of: <table> <tr> <td><b>NON.TRANSACTIONAL</b></td> <td>Ignore transaction boundaries</td> </tr> <tr> <td><b>NO.MAP</b></td> <td>Suppress id character translations</td> </tr> <tr> <td><b>READONLY</b></td> <td>Disable file updates</td> </tr> <tr> <td><b>SYNC</b></td> <td>Force write all file updates.</td> </tr> </table>	<b>NON.TRANSACTIONAL</b>	Ignore transaction boundaries	<b>NO.MAP</b>	Suppress id character translations	<b>READONLY</b>	Disable file updates	<b>SYNC</b>	Force write all file updates.
<b>NON.TRANSACTIONAL</b>	Ignore transaction boundaries								
<b>NO.MAP</b>	Suppress id character translations								
<b>READONLY</b>	Disable file updates								
<b>SYNC</b>	Force write all file updates.								
<i>file.var</i>	is the name of the variable to hold the file reference for use in later operations on this file.								
<i>statement(s)</i>	are statements to be executed depending on the outcome of the <b>OPENPATH</b> 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 multivalued products.

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 discussed [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 **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

```
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](#)

## OPENSEQ

The **OPENSEQ** statement opens a record of a directory file, a device or a pipe for sequential access.

### Format

```
OPENSEQ file.name, id {APPEND | NOBUF | NO.MAP | OVERWRITE | READONLY}
TO file.var
{ON ERROR statement(s)}
{LOCKED statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

or

```
OPENSEQ pathname {APPEND | NOBUF | NO.MAP | OVERWRITE | READONLY}
TO file.var
{ON ERROR statement(s)}
{LOCKED statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

where

<i>file.name</i>	evaluates to the VOC name of the directory file holding the record to be opened.
<i>id</i>	evaluates to the name of the record to be opened.
<i>pathname</i>	evaluates to the operating system pathname of the record to be opened. This may be a file, a device or a pipe.
<i>file.var</i>	is the name of a variable to be used in later statements accessing this record.
<i>statement(s)</i>	are statement(s) to be executed depending on the outcome of the <b>OPENSEQ</b> 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 **READONLY** clause opens the item for read only access. Any attempt to write will fail.

Use of **APPEND** 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.

If the record already exists, the **THEN** clause is executed. 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 **ELSE** clause is executed and the [STATUS\(\)](#) function

returns zero. The record will have been locked and use of [WRITESEQ](#), [WRITESEQF](#), [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.

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 is relevant only to directory files and suppresses the normal translation of restricted characters in record ids. See [directory files](#) for more information.

A record open for sequential access may be read and written using [READSEQ](#) and [WRITESEQ](#) 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. On PDAs, the Windows device name must be 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"
END
```

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](#)

## OPEN.SOCKET()

The **OPEN.SOCKET()** function opens a data socket for an outgoing connection.

### Format

**OPEN.SOCKET**(*addr*, *port*, *flags* {, *timeout*})

where

*addr* is the address of the system to which a connection is to be established. This may be an IP address or a host name.

*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\$DGRM

A datagram type 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.SOCKET()** function opens a connection to the server with the given address and port number.

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.

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

```
SKT = OPEN.SOCKET("193.118.13.14", 3000, SKT$BLOCKING)
IF STATUS() THEN STOP 'Cannot open socket'
N = WRITE.SOCKET(SKT, DATA, 0, 0)
CLOSE.SOCKET 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.SOCKET](#),  
[CREATE.SERVER.SOCKET\(\)](#), [READ.SOCKET\(\)](#), [SELECT.SOCKET\(\)](#),  
[SERVER.ADDR\(\)](#), [SET.SOCKET.MODE\(\)](#), [SOCKET.INFO\(\)](#), [WRITE.SOCKET\(\)](#)

## 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

**ORS**(*expr1*, *expr2*)

where

*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 *expr1* and *expr2* are true. Any value other than zero or a null string is treated as 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 1VM1SM0VM0VM1FM0VM1

B contains 1VM0SM1VM0VM1FM1VM0

C = ORS(A, B)

C now contains 1VM1SM1VM0VM1FM1VM1

### 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 Linux, FreeBSD, AIX and Mac users 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. This function is not available on the PDA version of QM.

### Format

```
OS.EXECUTE expr { CAPTURING var }
```

where

*expr* evaluates to the command to be executed (maximum 4096 characters).

*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.

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.

### Example

```
OS.EXECUTE "MKDIR TEMPDIR"
```

This statement uses the operating system MKDIR command to create a directory named TEMPDIR.

## OSDELETE

The **OSDELETE** deletes an operating system file by pathname.

### Format

**OSDELETE** *path*

where

*path* is the pathname of the file to be deleted.

The **OSDELETE** statement provides a simple way for a program to delete an operating system file 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](#)

## 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.

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](#)

## 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

**OUTERJOIN**({**DICT**} *file.name*, *field.name*, *value*)

where

- file.name* evaluates to the name of the file from which data is to be retrieved. The optional **DICT** prefix specifies that the dictionary portion of the file is to be used. Alternatively, the *file.name* expression may include the uppercase word **DICT** before the actual file name and separated from it by a single space.
- field.name* is the name of the field in *file.name* that determines the record ids to be returned. There must be an alternate key index on this field.
- value* is the value that must appear in the specified field.

The **OUTERJOIN()** function uses an alternate key index on *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 [SELECTINDEX](#) cannot be used.

### Examples

```
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.

```
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
```

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

**PAGE** {**ON** *print.unit*} {*page.no*}

where

*print.unit* evaluates to the print unit number on which the action is to occur. If omitted, print unit zero is used.

*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 *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](#)

## PAUSE

The **PAUSE** statement pauses execution until awoken by another process. This function is not available on the PDA version of QM.

### 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. 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.

## 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 = 666.66666
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 {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.

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](#)

## PRINTER

The **PRINTER ON** and **OFF** statements determine whether output from **PRINT** statements to the default print unit (unit 0) is directed to the display or to the printer.

### Format

```
PRINTER ON  
PRINTER OFF
```

The **PRINTER ON** statement causes subsequent output to print unit zero to be directed to the printer. A later **PRINTER OFF** statement resumes output to the display.

The [STATUS\(\)](#) function returns the previous state of the **PRINTER** 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 **PRINTER ON** at the start of the program followed by a **PRINTER CLOSE** on return to the command prompt.

### Example

```
PRINTER ON  
PRINT "This is sent to the printer"  
PRINTER OFF  
PRINT "This is sent to the display"
```

## PRINTER CLOSE

The **PRINTER CLOSE** statement closes one or all print units.

### Format

```
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](#) command has 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.

## PRINTER DISPLAY

The **PRINTER DISPLAY** statement directs output sent to a print unit to the display.

### Format

```
PRINTER 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

```
PRINTER 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

<i>print.unit</i>	evaluates to the print unit on which the action is to be performed. If omitted, the default print unit (unit 0) is used.
<i>file.name</i>	evaluates to the VOC name of an existing directory file.
<i>record.name</i>	evaluates to the name of the record within <i>file.name</i> to which output to <i>print.unit</i> is to be directed. If the record already exists, it will be overwritten.
<i>statement(s)</i>	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**.

## PRINTER NAME

The **PRINTER NAME** statement associates a named printer device with a print unit.

### Format

```
PRINTER 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

```
PRINTER 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

<i>print.unit</i>	evaluates to the print unit on which the action is to be performed. If omitted, the default print unit (unit 0) is used.
<i>param</i>	identifies the parameter to be changed.
<i>new.value</i>	is the value to be set. A <i>new.value</i> 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.

Key	Token	Default	Function
1	LPTR\$WIDTH	80	Page width
2	LPTR\$LINES	66 or 24	Lines per page for printer Lines per page for display
3	LPTR\$TOP.MARGIN	0	Top margin size (lines)
4	LPTR\$BOTTOM.MARGIN	0	Bottom margin size (lines)
5	LPTR\$LEFT.MARGIN	0	Left margin size (characters)
11	LPTR\$FLAGS	Printer mode flags	

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.

## PRINTER.SETTING()

The **PRINTER.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

**PRINTER.SETTING**(*print.unit*, *param*, *new.value*)

where

<i>print.unit</i>	evaluates to the print unit on which the action is to be performed.
<i>param</i>	identifies the parameter to be changed using the keys shown below.
<i>new.value</i>	is the value to be set. A <i>new.value</i> of -1 sets the parameter to its default value. A <i>new.value</i> of -2 returns the current value without changing it.

The **PRINTER.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.

Key	Token	Default	Function
1	LPTR\$WIDTH	80	Page width
2	LPTR\$LINES	66 or 24	Lines per page for printer Lines per page for display
3	LPTR\$TOP.MARGIN	0	Top margin size (lines)
4	LPTR\$BOTTOM.MARGIN	0	Bottom margin size (lines)
5	LPTR\$LEFT.MARGIN	0	Left margin size (characters)
6 *	LPTR\$DATA.LINES		Lines excluding page heading and footing
7 *	LPTR\$HEADING.LINES		Heading lines per page
8 *	LPTR\$FOOTING.LINES		Footing lines per page
9 *	LPTR\$MODE		Printer mode number
10 *	LPTR\$NAME		Printer or file name
11	LPTR\$FLAGS		Printer mode flags
12 *	LPTR\$LINE.NO		Current position on page within data area
13 *	LPTR\$PAGE.NO		Current page number
14 *	LPTR\$LINES.LEFT		Lines remaining on current page
15	LPTR\$COPIES	1	Number of copies to print

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**

```
WIDTH = PRINTER.SETTING(1, LPTR$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

**PRIVATE** *var, 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.

### See also:

[Object oriented programming](#), [CLASS](#), [DISINHERIT](#), [INHERIT](#), [OBJECT\(\)](#), [OBJINFO\(\)](#), [PUBLIC](#).

## 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.

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* specified 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	Interpret break character as break key? (boolean)
2	PT\$INVERT	Invert case of alphabetic characters on input? (boolean)
3	PT\$BRKCH	Sets break character to be char( <i>value</i> ). The <i>value</i> must be in the range 1 to 31. For any other value, the current setting is returned.
4	PT\$PAGE.PAUS E	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)

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.

**See also:**

[PTERM](#)

## PUBLIC

The **PUBLIC** statement defines public property variables, subroutines and functions in a class module.

### Format

```
PUBLIC var, 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.

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 **PUT** 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 [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 calls to the subroutine or function must be the same as in the declaration unless the **VAR.ARGS** option is used in which case 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.

```
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 **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. For example,

```
PUBLIC 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".

### Examples

```
PUBLIC FUNCTION CONNECT(SERVER, PORT)
  SKT = OPEN.SOCKET(SERVER, PORT, SKT$BLOCKING)
  RETURN STATUS() = 0
END
```

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.

```
PUBLIC FUNCTION CONNECT(SERVER, PORT) VAR.ARGS
  IF UNASSIGNED(PORT) THEN PORT = 4000
  SKT = OPEN.SOCKET(SERVER, PORT, SKT$BLOCKING)
  RETURN STATUS() = 0
END
```

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.

```
PUBLIC SUBROUTINE INSERT.ITEMS(ID, ...)
  READU REC FROM FVAR, ID ELSE NULL
  FOR I = 2 TO ARG.COUNT()
    VALUE = ARG(I)
    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](#)

## PWR()

The **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 [REUSE\(\)](#) function is used.

### Example

```
N = 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\(\)](#)

## RAISE()

The **RAISE()** function converts mark characters in a string to the next higher level mark.

### Format

**RAISE**(*string*)

where

*string* evaluates to the string in which mark characters are to be converted.

The **RAISE()** function replaces mark characters according to the following table:

<b>Original</b>	<b>Replacement</b>
Item mark	Item mark (unchanged)
Field mark	Item mark
Value mark	Field mark
Subvalue mark	Value mark
Text mark	Subvalue mark

### 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

Dividend	Divisor	Result
132	10	13
135	10	14
-135	10	-14

**See also:**

[IDIV\(\)](#)

## READ

The **READ** statement reads a record from a previously opened file.

### Format

```

READ var FROM file.var, record.id { ON ERROR statement(s) }
  { THEN statement(s) }
  { ELSE statement(s) }

```

where

<i>var</i>	is the name of a variable to receive the dynamic array read from the file.
<i>file.var</i>	is the file variable associated with the file.
<i>record.id</i>	evaluates to the id of the record to be read.
<i>statement(s)</i>	are statements to be executed depending on the outcome of the <b>READ</b> operation.

At least one of the **THEN** and **ELSE** clauses must be present.

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 **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. 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.SOCKET\(\)](#) (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 default mode of data transfer as blocking.
SKT\$NON.BLOCKING	Sets the default mode of data transfer as non-blocking.

If neither blocking flag is given, the blocking mode set when the socket was opened is used.

*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.SOCKET("", 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.SOCKET SKT
CLOSE.SOCKET 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.SOCKET](#), [CREATE.SERVER.SOCKET\(\)](#), [OPEN.SOCKET\(\)](#), [SELECT.SOCKET\(\)](#), [SERVER.ADDR\(\)](#), [SET.SOCKET.MODE\(\)](#), [SOCKET.INFO\(\)](#), [WRITE.SOCKET\(\)](#)

## 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 FROM file.var, bytes
  { ON ERROR statement(s) }
  { THEN statement(s) }
  { ELSE statement(s) }
```

where

<i>var</i>	is the name of a variable to receive the data read from the file.
<i>file.var</i>	is the file variable associated with the file.
<i>bytes</i>	evaluates to the number of bytes to be read.
<i>statement(s)</i>	are statements to be executed depending on the outcome of the <b>READBLK</b> operation.

At least one of the **THEN** and **ELSE** clauses must be present.

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 ER\$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.

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](#), [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 FROM file.var TO var1, var2,...
  { THEN statement(s) }
  { ELSE statement(s) }
```

where

<i>file.var</i>	is the file variable associated with the record by a previous <b>OPENSEQ</b> statement.
<i>var1, var2 ...</i>	are the variables to receive the data read from the file.
<i>statement(s)</i>	are statement(s) to be executed depending on the outcome of the <b>READSEQ</b> .

At least one of the **THEN** and **ELSE** clauses must be present.

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.

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](#), [CSVDO\(\)](#), [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 FROM file.var, record.id {ON ERROR statement(s)}
  {LOCKED statement(s)}
  {THEN statement(s)}
  {ELSE statement(s)}
```

where

<i>var</i>	is the name of a variable to receive the dynamic array read from the file.
<i>file.var</i>	is the file variable associated with the file.
<i>record.id</i>	evaluates to the id of the record to be read.
<i>statement(s)</i>	are statements to be executed depending on the outcome of the <b>READL</b> operation.

At least one of the **THEN** and **ELSE** clauses must be present.

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 **LOCKED** clause is executed if the file or record is exclusively locked by another process. The [STATUS\(\)](#) function will return the user id of a process holding a lock on this file or record. 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 **READL** is successful.

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. 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
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 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

<i>var</i>	is the variable to receive the select list.
<i>list.no</i>	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.
<i>statement(s)</i>	are statement(s) to be performed depending on the outcome of the <b>READLIST</b> 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.

### 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

```
READNEXT var {, val.pos {, subval.pos}} {FROM list.no}  
{ON ERROR statement(s)}  
{THEN statement(s)}  
{ELSE statement(s)}
```

where

<i>var</i>	is the variable to receive the select list item.
<i>val.pos</i>	is the variable to receive the value position with an exploded select list.
<i>subval.pos</i>	is the variable to receive the subvalue position with an exploded select list.
<i>list.no</i>	is the select list number. If omitted, the default select list is used. The <b>READNEXT</b> statement can also use select list variables returned by the <a href="#">SELECTV</a> statement or the RTNLIST option of the <a href="#">EXECUTE</a> statement.
<i>statement(s)</i>	are statement(s) to be executed depending on the outcome of the <b>READNEXT</b> 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 an 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.

The [STATUS\(\)](#) function will return zero unless the **ON ERROR** clause is executed.

### Migration Note

Some other multivalued products leave *var* unchanged if the **ELSE** clause is taken.

### 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 multivalued 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
  PRINT ID
REPEAT
```

This program fragment produces a list of the record keys present in **STOCK.FILE**.

## READSEQ

The **READSEQ** statement reads the next field (line of text) from a directory file record previously opened for sequential access.

### Format

```
READSEQ var FROM file.var { ON ERROR statement(s) }  
  { THEN statement(s) }  
  { ELSE statement(s) }
```

where

<i>var</i>	is the variable to receive the data read from the file.
<i>file.var</i>	is the file variable associated with the record by a previous <a href="#">OPENSEQ</a> statement.
<i>statement(s)</i>	are statement(s) to be executed depending on the outcome of the <b>READSEQ</b> .

At least one of the **THEN** and **ELSE** clauses must be present.

The next field (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 field 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](#), [WEOFSEQ](#), [WRITEBLK](#), [WRITECSV](#), [WRITESEQ](#), [WRITESEQF](#)

## READU

The **READU** statement reads a record from a previously opened file, setting an update lock.

### Format

```
READU var FROM file.var, record.id {ON ERROR statement(s)}
  {LOCKED statement(s)}
  {THEN statement(s)}
  {ELSE statement(s)}
```

where

<i>var</i>	is the name of a variable to receive the dynamic array read from the file.
<i>file.var</i>	is the file variable associated with the file.
<i>record.id</i>	evaluates to the id of the record to be read.
<i>statement(s)</i>	are statements to be executed depending on the outcome of the <b>READU</b> operation.

At least one of the **THEN** and **ELSE** clauses must be present.

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 on this file or record. 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 **READU** is successful.

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. 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 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.

## READV

The **READV** statement reads a specific field from a record of a previously opened file.

### Format

```
READV var FROM file.var, record.id, field.expr
{ ON ERROR statement(s) }
{ THEN statement(s) }
{ ELSE statement(s) }
```

where

<i>var</i>	is the name of a variable to receive the dynamic array read from the file.
<i>file.var</i>	is the file variable associated with the file.
<i>record.id</i>	evaluates to the id of the record to be read.
<i>field.expr</i>	evaluates to the number of the field to be read.
<i>statement(s)</i>	are statements to be executed depending on the outcome of the <b>READV</b> operation.

At least one of the **THEN** and **ELSE** clauses must be present.

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.

### Example

```
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

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 FROM file.var, record.id, field.expr
{ ON ERROR statement(s) }
{ LOCKED statement(s) }
{ THEN statement(s) }
{ ELSE statement(s) }
```

where

<i>var</i>	is the name of a variable to receive the dynamic array read from the file.
<i>file.var</i>	is the file variable associated with the file.
<i>record.id</i>	evaluates to the id of the record to be read.
<i>field.expr</i>	evaluates to the number of the field to be read.
<i>statement(s)</i>	are statements to be executed depending on the outcome of the operation.

At least one of the **THEN** and **ELSE** clauses must be present.

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 on this file or record. 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**

```
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

**RECORDLOCKED**(*file.var*, *record.id*)

where

*file.var* is the file variable associated with the file.

*record.id* evaluates to the key of the record to be tested.

The **RECORDLOCKED()** function returns a value indicating the state of any locks on record *record.id* of the file open as *file.var*. The tokens shown in the table below are defined in the KEYS.H record of the SYSCOM file.

Value	Token	Lock state
-3	LOCK\$OTHER.FILELOCK	Another user holds a file lock
-2	LOCK\$OTHER.READU	Another user holds an update lock
-1	LOCK\$OTHER.READL	Another user holds a read lock
0	LOCK\$NO.LOCK	The record is not locked
1	LOCK\$MY.READL	This user holds a read lock
2	LOCK\$MY.READU	This user holds an update lock
3	LOCK\$MY.FILELOCK	This user holds a file lock

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

```
IF RECORDLOCKED(STOCK, "ORDER.LIST") THEN
  DISPLAY "Record is locked by user " : STATUS()
END
```

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 that prevents further locking. The [STATUS\(\)](#) function will return the user id of a process holding a lock on this file or record. 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.

## 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.

## REM()

The **REM()** function returns the remainder when one value is divided by another.

### Format

**REM**(*dividend*, *divisor*)

where

*dividend* evaluates to a number or a numeric array.

*divisor* evaluates to a number or a numeric array.

The **REM()** function returns the remainder of dividing *dividend* by *divisor*. This is defined as

$$\text{REM}(x, y) = \text{SIGN}(x) * \text{MOD}(\text{ABS}(X), \text{ABS}(Y))$$

where the **SIGN()** function returns 1 for  $x > 0$ , -1 for  $x < 0$  and 0 for  $x = 0$ . (**SIGN()** is not part of QMBasic. It is used here only to explain the action of the **MOD()** function).

The **REM()** function differs from the **MOD()** function when one of its arguments is negative. The following table shows the result of the **REM()** function.

<b>Dividend</b>	<b>Divisor</b>	<b>REM()</b>
530	100	30
-530	100	-30
530	-100	30
-530	-100	-30
0	100	0
0	-100	0
100	0	100
-100	0	-100

If either *dividend* or *divisor* is a numeric array (a dynamic array where all elements are numeric), the **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 *dividend* and *divisor* arrays if they are identical. For arrays of differing structure, the structure of the result depends on whether the **REUSE()** function is used.

### Example

```
N = REM(T, 30)
```

This statement finds the remainder of dividing T by 30 and assigns this to N.

See also:

[MOD\(\)](#), [ROUNDDOWN\(\)](#), [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

The **REMOVE** statement and **REMOVE()** function extract characters from a dynamic array up to the next mark character.

### Format

**REMOVE** *string* **FROM** *dyn.array* **SETTING** *var*

**REMOVE**(*dyn.array*, *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

S = REMOVE (X, Y)

is equivalent to

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 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.

Note that the **REMOVE** operation performs a type conversion on *dyn.array* if it is not already a string. Thus the program

```
S = 99
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

```
LOOP
  REMOVE BOOK.NO FROM BOOK.LIST SETTING 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).

```
S = ""
LOOP
  REMOVE FLD FROM REC SETTING 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

```
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\(\)](#), [REMOVEF\(\)](#), [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](#)

## REMOVEF()

The **REMOVEF()** function extracts data from a delimited character string.

### Format

**REMOVEF**(*string*{, *delimiter*{, *count*})

where

- |                  |  |
|------------------|--|
| <i>string</i>    | is the string from which data is to be extracted.  |
| <i>delimiter</i> | is the delimiter character that separates elements of <i>string</i> . If omitted, a field mark is used. If <i>delimiter</i> is more than one character, only the first character is used. If <i>delimiter</i> is a null string, <i>count</i> characters are extracted. |
| <i>count</i>     | is the number of consecutive delimited elements of <i>string</i> 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

```
S = S
```

which will assign S to itself thus resetting the remove pointer.

### Examples

```
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:**

[GETREM\(\)](#), [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

<i>dyn.array</i>	evaluates to a string in which the replacement is to occur.
<i>field</i>	evaluates to the field position number. If zero, this argument defaults to one.
<i>value</i>	evaluates to the value position number. If omitted or zero, the entire field is replaced.
<i>subvalue</i>	evaluates to the subvalue position number. If omitted or zero, the entire value is replaced.
<i>string</i>	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

**RETURN** {*expr*}

**RETURN TO** *label*{:}

**RETURN FROM PROGRAM**

where

*label* is a label in the same program or subroutine as the **RETURN** statement.

*expr* is the value to be returned from a user written function.

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 multivalued database products is with a statement of the form

```
ERROR.LABEL: 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

```
RETURN FROM PROGRAM
```

### Examples

```
SUBROUTINE PRINT.REPORT(ID)
...statements...
RETURN
END
```

This skeleton subroutine performs its task and then returns to its caller.

```
FUNCTION MATMAX(MAT A)
  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.

**See also:**

[CALL](#), [GOSUB](#)

## REUSE()

The **REUSE()** function determines how arithmetic operators applied to numeric arrays handle unequal numbers of fields, values or subvalues.

### Format

**REUSE**(*num.array*)

where

*num.array* is a numeric 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.

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<sub>FM</sub>22<sub>FM</sub>3" and D to "11<sub>FM</sub>22<sub>FM</sub>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<sub>FM</sub>2<sub>FM</sub>3" and D to "11<sub>FM</sub>12<sub>FM</sub>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 "11<sub>FM</sub>22<sub>VM</sub>3<sub>VM</sub>4<sub>FM</sub>35<sub>VM</sub>36<sub>VM</sub>7<sub>FM</sub>8".

D is set to "11<sub>FM</sub>22<sub>VM</sub>23<sub>VM</sub>24<sub>FM</sub>35<sub>VM</sub>36<sub>VM</sub>37<sub>FM</sub>48".

**See also:**

[ANDSQ](#), [EQSQ](#), [GESQ](#), [GTSQ](#), [IFSQ](#), [LESQ](#), [LTSQ](#), [NESQ](#), [NOTSQ](#), [ORSQ](#)

## RND()

The **RND()** function returns a random number.

### Format

**RND**(*expr*)

where

*expr* 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

```
TWO.DICE = RND(6) + 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

$\text{IDIV}(\text{value}, \text{increment}) * \text{increment}$

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

$\text{IDIV}(\text{value} + \text{increment} - 1, \text{increment}) * \text{increment}$

or, for a negative value

$\text{IDIV}(\text{value} - \text{increment} + 1, \text{increment}) * \text{increment}$

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
9	9	9
10	9	12

**See also:**

[MOD\(\)](#), [REM\(\)](#)

## 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

**SAVE.SCREEN**(*col, line, width, height*)

where

*col* is the screen column (from zero) of the leftmost column to be saved.

*line* is the screen line (from zero) of the top line to be saved.

*width* is the width of the screen region to be saved.

*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

```
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.no }  
  { THEN statement(s) }  
  { ELSE statement(s) }
```

where

*name* is the name of the \$SAVEDLISTS entry to be written.

*list.no* is the select list number to be saved. If omitted, this defaults to zero.

*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 numbered 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](#)

## 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

<i>file.var</i>	is the file variable associated with the record by a previous <a href="#">OPENSEQ</a> statement.
<i>offset</i>	is the byte position relative to the point given by <i>relto</i> . If <i>offset</i> and <i>relto</i> are both omitted, the file position is set to the start of the file.
<i>relto</i>	indicates the point from which <i>offset</i> is calculated. Defaults to 0 if omitted. <ul style="list-style-type: none"> <li>0 Start of file (<i>offset</i> must be zero or positive).</li> <li>1 Current position (<i>offset</i> may be any value)</li> <li>2 End of file (<i>offset</i> must be zero or negative)</li> </ul>
<i>statement(s)</i>	are statement(s) to be executed depending on the outcome of the <b>SEEK</b> .

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.

### 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:

[CLOSESEQ](#), [CREATE](#), [NOBUF](#), [OPENSEQ](#), [READBLK](#), [READCSV](#), [READSEQ](#), [WEOFSEQ](#), [WRITEBLK](#), [WRITECSV](#), [WRITESEQ](#), [WRITESEQF](#)

## SELECT, SELECTN, SELECTV

The **SELECT** statement creates a select list containing all record keys from a file.

### Format

**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

<i>var</i>	is the file variable associated with an open file or a field mark delimited dynamic array of items to form the list.
<i>list.no</i>	is the select list number of the list to be created. If omitted, select list zero is used.
<i>list.var</i>	is the select list variable to receive the list. Select list variables can be processed by the <a href="#">READNEXT</a> statement as an alternative to using numbered select lists. If the <b>TO</b> clause is omitted, a default select list variable is used.
<i>statement(s)</i>	are <i>statement(s)</i> 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 way 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

```
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.

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. 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.

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.INFO**(*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\$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\$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<br>Notify availability of data that can be read with<br><a href="#">READ.SOCKET()</a><br>Notify closure of connection |
| 2 | SKT\$WRITE_EVENT     | Notify when data can be sent with <a href="#">WRITE.SOCKET()</a>   |
| 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$INFO.OPEN)) then
            skt(j) = accept.socket.connection(skt(1), 0)
            if not(set.socket.mode(skt(j),
SKT$INFO.EVENTS, SKT$READ.EVENT)) then
              stop 'Error from set.socket.mode()'
            end
            exit
          end
        next j
      end else
        str = read.socket(skt(i), 1000, 0, 0)
        if status() then /* Socket closed
          skt(i) = 0
        end else
          ...process input...
        end
      end
    end
  next i
repeat

```

**See also:**

[Using Socket Connections](#), [ACCEPT.SOCKET.CONNECTION](#), [CLOSE.SOCKET](#), [CREATE.SERVER.SOCKET\(\)](#), [OPEN.SOCKET\(\)](#), [READ.SOCKET\(\)](#), [SERVER.ADDR\(\)](#), [SET.SOCKET.MODE\(\)](#), [WRITE.SOCKET\(\)](#)

## 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

The **SELECTINDEX** statement creates a select list from an [alternate key index](#) entry.

### Format

```
SELECTINDEX index.name {, value} FROM file.var {TO list.no}
```

where

<i>index.name</i>	is the name of the alternate key index to be processed.
<i>value</i>	is the value to be located in the index.
<i>file.var</i>	is the file variable associated with an open file.
<i>list.no</i>	is the select list number of the list to be created. If omitted, select list zero is used.

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 a file keyed by customer number for which there is an index on a field that contains the dates of orders placed by the customer. The index will have an entry for each date and each of these entries will hold a list of customers placing orders on that date. The order of the indexed values (order dates, left column in the diagram below) will be sorted left or right aligned according to the format code in the dictionary entry for the date field. The customer numbers stored for each date (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.

Date	Customers
15184	981 995 1031 1068 1077
15185	216 944 998 1044
15187	724 899 975 1121
15188	816 948 982 1000
15189	662 741 963 1017
15190	921 993 1002 1052
15191	851 1013 1015
15193	967 993 1002
15194	947 1009
15195	922 1014 1025 1064

### 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 (1) if the select list is active, false (0) if it is not active.   |
| 3 | SL\$COUNT  | Returns the number of items remaining to be processed in the select list. If the list is not active, the <b>SELECTINFO()</b> 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..

## SELECTLEFT and SELECTRIGHT

The **SELECTLEFT** and **SELECTRIGHT** statements create a select list from the entry in an [alternate key index](#) to the left or right of the last entry processed.

### Format

```
SELECTLEFT index.name FROM file.var {SETTING key} {TO list.no}  
SELECTRIGHT index.name FROM file.var {SETTING key} {TO list.no}
```

where

<i>index.name</i>	is the name of the alternate key index to be processed.
<i>file.var</i>	is the file variable associated with an open file.
<i>key</i>	is the variable to be set to the key value associated with the returned list.
<i>list.no</i>	is the select list number of the list to be created. If omitted, select list zero is used.

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
```

```
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
```

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 also:**

[SELECTINDEX](#), [SETLEFT](#), [SETRIGHT](#)

## 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 ASCII character set position value of a character.

### Format

**SEQ**(*char*)

where

*char* evaluates to the character to be processed.

The **SEQ()** function returns the character value of *char*. It is the inverse of the [CHAR\(\)](#) function.

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.

### Example

```
N = SEQ(KEYIN( ))
```

This statement reads a single character from the keyboard and then uses **SEQ()** to find its ASCII character set value.

### See also:

[CHAR\(\)](#)

## 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.SOCKET\(\)](#), [OPEN.SOCKET\(\)](#), [READ.SOCKET\(\)](#), [SELECT.SOCKET\(\)](#), [SET.SOCKET.MODE\(\)](#), [SOCKET.INFO\(\)](#), [WRITE.SOCKET\(\)](#)

## 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\(\)](#)

## 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

**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.EXIT.STATUS

The **SET.EXIT.STATUS** statement sets the final exit status returned by QM to the operating system. This operation has no effect on the PDA version of QM.

### 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 and 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.

### Example

```
SETLEFT 'POSTCODE' FROM CLIENTS.FILE  
LOOP  
    SECTRIGHT '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](#)

## SET.PORT.PARAMS()

The **SET.PORT.PARAMS()** function sets the communications parameters for a serial port. This function is not available on the PDA version of QM.

### 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 (1) if successful, false (0) 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)

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'
```

## 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 blocking mode.
SKT\$INFO.NO.DELAY	Nagle algorithm disabled?
SKT\$INFO.KEEP.ALIVE	Send keep alives?
SKT\$INFO.TRACE	Activate socket tracing to the file pathname in <i>value</i> . A null pathname turns off tracing.
SKT\$INFO.EVENTS	Specify events to be monitored for use with <a href="#">SELECT.SOCKET()</a> .

*value* is the required value of the parameter.

The **SET.SOCKET.MODE()** function returns TRUE (1) if the action is successful, FALSE (0) 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<br>Notify availability of data that can be read with <a href="#">READ.SOCKET()</a><br>Notify closure of connection |
| 2 | SKT\$WRITE_EVENT     | Notify when data can be sent with <a href="#">WRITE.SOCKET()</a>  |
| 4 | 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.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:

Parameter	Key	Meaning
1	NLS\$CURRENCY	Default currency symbol. Maximum 8 characters.
2	NLS\$THOUSANDS	Default thousands separator character.
3	NLS\$DECIMAL	Default decimal separator character.
4	NLS\$IMPLICIT.DECIMAL	Default decimal separator character for implicit conversions. See below.

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 NLS\$DECIMAL 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 NLS\$IMPLICIT.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 NLS\$DECIMAL value also sets the NLS\$IMPLICIT.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\$IMPLICIT.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:**

[NLS](#)

## SETPU

The **SETPU** statement sets the characteristics of a print unit.

### Format

**SETPU** *key, unit, value*

where

*key* identifies the parameter to set. This may be:

1	PU\$MODE	Print unit mode
2	PU\$WIDTH	Characters per line
3	PU\$LENGTH	Lines per page
4	PU\$TOPMARGIN	Top margin size
5	PU\$BOTMARGIN	Bottom margin size
6	PU\$LEFTMARGIN	Left margin size
7	PU\$SPOOLFLAGS	Various print unit flags
9	PU\$FORM	Form name (not used by all spoolers)
10	PU\$BANNER	Banner page text
11	PU\$LOCATION	Printer / file name
12	PU\$COPIES	Number of copies to print
15	PU\$PAGENUMBER	Current page number (see below)
1006	PU\$OPTIONS	Options to be passed to the spooler
1007	PU\$PREFIX	Pathname of file holding prefix data to be added to the start of the output
1008	PU\$SPOOLER	Spooler to be used (ignored on Windows)
1009	PU\$OVERLAY	Catalogued overlay subroutine name (see <a href="#">SETPTR</a> )
1010	PU\$CPI	Characters per inch (may be non-integer value)
1011	PU\$PAPER.SIZE	Paper size. See SYSCOM PCL.H
1012	PU\$LPI	Lines per inch. Must be 1, 2, 3, 4, 6, 8, 12, 16, 24, 48
1013	PU\$WEIGHT	Font stroke weight. See SYSCOM PCL.H
1014	PU\$SYMBOL.SET	Symbol set. See SYSCOM PCL.H
1015	PU\$STYLE	Query processor style. See the Query processor <a href="#">STYLE</a> option for details.
1016	PU\$NEWLINE	Newline string for this print unit. Must be char(10), char(13) or char(13):char(10)
1017	PU\$PRINTER.NAME	Name of printer
1018	PU\$FILE.NAME	File name for output directed to a file
1021	PU\$FORM.FEED	Set form feed code for this print unit. Must

		be char(12) or char(13):char(12)
2000	PU\$LINENO	Current line number

*unit* evaluates to the print unit number.

*value* is the value to set for the given parameter.

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 offset of the remove pointer 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 negative or greater than the length of *string*, any existing remove pointer is not altered and the [STATUS\(\)](#) function will return error code ER\$LENGTH.

**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](#)

## SHIFT()

The **SHIFT()** function performs a logical bit-shift operation on an integer value.

### Format

**SHIFT**(*value*, *shift.len*)

where

*value* evaluates to the integer to be shifted.

*shift.len* indicates the number of bit positions by which *value* is to be shifted.

The **SHIFT()** function converts *value* to a thirty two bit 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

```
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\(\)](#)

## SIN()

The **SIN()** function returns the sine of a value.

### Format

**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

```
OPP = HYP * 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:

[ACOS\(\)](#), [ASIN\(\)](#), [ATAN\(\)](#), [COS\(\)](#), [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:

SKT\$INFO.OPEN	Is <i>skt</i> a socket variable? Returns true (1) or false (0).
SKT\$INFO.TYPE	Type or socket. Returns one of the following values according to which socket function was used to open the socket:
SKT\$INFO.TYPE.SERVER	<a href="#">CREATE.SERVER.SOCKET()</a>
SKT\$INFO.TYPE.INCOMING	<a href="#">ACCEPT.SOCKET.CONNECTION()</a>
SKT\$INFO.TYPE.OUTGOING	<a href="#">OPEN.SOCKET()</a>
SKT\$INFO.PORT	Port number.
SKT\$INFO.IP.ADDR	IP address.
SKT\$INFO.BLOCKING	Default blocking mode.
SKT\$INFO.NO.DELAY	Nagle algorithm disabled?
SKT\$INFO.KEEP.ALIVE	Send keep alives?
SKT\$INFO.EVENTS	Report events related to use of <a href="#">SELECT.SOCKET()</a> .

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<br>Data is available to be read with <a href="#">READ.SOCKET()</a><br>The connection has been closed |
| 2 | SKT\$WRITE_EVENT     | Data can be sent with <a href="#">WRITE.SOCKET()</a>  |
| 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.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.

The **SORT.COMPARE()** function compares *string1* and *string2* according to the sort rules specified by the *mode* value.

- Mode 0 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 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 A partitioned 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 The same as mode 1 except that the test for numeric items allows non-integer values.

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 *string1* is equal to *string2*
- 1 *string1* comes after *string2*

### **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

**See also:**

[COMPARE\(\)](#)

## 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.

0	A E H I O U W Y
1	B F P V
2	C G J K Q S X Z
3	D T
4	L
5	M N
6	R

Letters in group 0 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

```

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

**SPACE**(*count*)

where

*count* 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

```
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 "ABC123<sub>VM</sub>456<sub>FM</sub>DEF789"

## SQRT()

The **SQRT()** function returns the square root of a value.

### Format

**SQRT**(*expr*)

where

*expr* evaluates to a number or a numeric array.

The **SQRT()** function returns the square root of *expr*.

If *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 *expr*.

A negative value of *expr* will cause a run time error.

### Example

```
N = 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

**SQUOTE**(*expr*)

where

*expr* evaluates to the source string.

The **SQUOTE()** function returns *expr* enclosed in single quotation marks.

### Example

```
A = SQUOTE('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 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 left justified ascending order.

### Format

```
SSELECT file.var {TO list.no} {ON ERROR statement(s)}  
SSELECTN file.var {TO list.no} {ON ERROR statement(s)}  
SSELECTV file.var {TO list.var} {ON ERROR statement(s)}
```

where

<i>file.var</i>	is the file variable associated with an open file.
<i>list.no</i>	is the select list number of the list to be created. If omitted, select list zero is used.
<i>list.var</i>	is the select list variable to receive the list. Select list variables can be processed by the <a href="#">READNEXT</a> statement as an alternative to using numbered select lists. If the <b>TO</b> clause is omitted, a default select list variable is used.
<i>statement(s)</i>	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. Keys will be stored in left justified ascending order. The [@SELECTED](#) variable is set to the number of records selected.

[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.

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 **SSELECT** where the *file.var* is replaced by a dynamic array in which each field becomes an entry in the target select list.

### Example

```
SSELECT STAFF TO 7
```

This statement creates a sorted list of the records on the file with file variable 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. Not all fields are returned on the PDA version of QM.

### 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 timezone.
- 14 Date of last access. Pick style day number in user's local timezone.
- 15 Time of last modification. Seconds since midnight in user's local timezone.
- 16 Date of last modification. Pick style day number in user's local timezone.
- 17-19 Unused on QM
- 20 Operating system file pathname
- 21 File type (see [FILEINFO](#) for a list of values)
- 22-33 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 multivalued 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.ERRMSG
```

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 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 **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.ERRMSG 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

**STR**(*string*, *count*)

where

*string* evaluates to the string to be repeated.

*count* evaluates to the number of repeats of *string* that are required.

The **STR()** function returns *count* occurrences of *string*. If *count* is less than one, a null string is returned.

### Examples

```
PRINT STR( "*", 79)
```

This statement prints a line of 79 asterisks.

```
S = STRS('X':@VM:'Y':@SM:'Z', 2)
```

This statement sets S to be

```
XXVMYYSMZZ
```

**See also:**

[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:

```
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:

```
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.

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 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. 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". 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 [\\$MODE](#) compiler directive is enabled. In this case, the default value will be applied if the argument variable passed from the calling program is unassigned.

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
```

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 matrix passed as an argument cannot be redimensioned in the subroutine.

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

**SUBSTITUTE**(*dyn.array*, *old.list*, *new.list* {, *delimiter*})

where

*dyn.array* is the dynamic array to be processed.

*old.list* is list of items to replace.

*new.list* is list of replacement items.

*delimiter* is the single character delimiter separating items in *old.list* and *new.list*. If omitted, this defaults to a value mark.

The **SUBSTITUTE()** function processes each element of dynamic array *dyn.array* constructing an equivalently structured new dynamic array result. Where an element of *dyn.array* contains a value in the *old.list*, the result contains the corresponding item from *new.list*. Where there is no match with an item in *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 *dyn.array* is a simple single valued string.

### Example

A contains D<sub>FM</sub>D<sub>VM</sub>F<sub>VM</sub>P

B = SUBSTITUTE(A, 'D|P', 'Done|Pending', '|')

B will contain Done<sub>FM</sub>Done<sub>VM</sub>F<sub>VM</sub>Pending

## 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 ABCDE<sub>FM</sub>FGHIJ<sub>VM</sub>KLMNO<sub>VM</sub>PQRST

B = SUBSTRINGS(A, 2, 3)

B will contain BCD<sub>FM</sub>GHI<sub>VM</sub>LMN<sub>VM</sub>QRS

## SUM()

The **SUM()** function eliminates the lowest level of a dynamic array by adding the elements to form an item of the next highest level.

### Format

**SUM**(*expr*)

where

*expr* evaluates to a numeric array.

The **SUM()** function identifies the lowest level elements present in *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 **SUM()** function returns *expr*.

Non-numeric elements of *expr* are ignored.

### Example

```
TOTAL.PAID = SUM(PAYMENTS)
```

This statement sums a multivalued list of payments to form the total amount paid.

### See also:

[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 multivalued 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

**SWAPCASE**(*string*)

where

*string* evaluates to the string in which substitution is to occur.

The **SWAPCASE()** function returns the value of *string* with all uppercase letters converted to lower case and all lowercase letters converted to uppercase. If *string* is a variable rather than an expression, the value of the variable is not affected.

### Example

```
S = "ABCdef"  
PRINT SWAPCASE(S)
```

This program fragment prints the string "abcDEF".

### See also:

[DOWNCASE\(\)](#), [UPCASE\(\)](#)

## SYSTEM()

The **SYSTEM()** function returns information regarding the status of various aspects of the system. On the PDA version of QM, key values that are inappropriate return a zero value.

### 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.

Key	Function
1	Returns 1 if a PRINTER ON statement is in effect
2	Current page width of the default print unit
3	Current page length of the default print unit
4	Lines remaining on current page of the default print unit
5	Current page number of the default print unit
6	Current line number of the default print unit
7	Terminal type (same as <a href="#">@TERM.TYPE</a> )
9	Cumulative processor time used (mS) by this QM session
10	Input waiting in the DATA queue? (1 if so, 0 if not)
11	Select list 0 active? (1 if so, 0 if not)
12	Time in seconds since midnight (same as <a href="#">TIME()</a> )
18	User number (same as <a href="#">@USERNO</a> )
19	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 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.
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 or PDA)

- 28 Returns the operating system effective uid for the user's process. (Not Windows or PDA)
- 29 Returns the operating system gid for the user's process. (Not Windows or PDA)
- 30 Returns the operating system effective gid for the user's process. (Not Windows or PDA)
- 31 Licence number
- 32 Returns the system (QMSYS) directory pathname
- 38 Returns temporary directory pathname
- 42 Returns telnet connection IP address, null for a console user
- 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 (NT and later).
- 1007 Returns the current transaction number, zero if not in a transaction.
- 1008 Returns the current transaction 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.
- 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.

- 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 Returns a field mark delimited list of active replication subscribers.
- 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.
- 1045 Returns time at which QM was started as an epoch value.
- 1046 Returns the current CSV delimiter character set by [CSV.MODE](#).

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.

## 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()

The **TERMINFO()** function returns information from the terminfo database.

### Format

**TERMINFO()**  
**TERMINFO(*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 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.

See also:

[TERM](#), [@\(\)](#)

## TESTLOCK()

The **TESTLOCK()** function returns the state of a task lock.

### Format

**TESTLOCK**(*lock.num*)

where

*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](#)

## 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.

See the [SYSTEM\(\)](#) function for a way to return the time of day to millisecond precision.

### 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

<i>hh</i>	hours in 24 hour format, zero filled
<i>mm</i>	minutes, zero filled
<i>ss</i>	seconds, zero filled
<i>dd</i>	day of month, zero filled
<i>mmm</i>	first three letters of the month name, first letter uppercase
<i>yyyy</i>	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). 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.

## 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 total 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

**TRANS**({**DICT**} *file.name*, *record.id*, *field*, *action*)

**RTRANS**({**DICT**} *file.name*, *record.id*, *field*, *action*)

where

*file.name* evaluates to the name of the file from which data is to be retrieved. The optional **DICT** prefix specifies that the dictionary portion of the file is to be used. Alternatively, the *file.name* expression may include the uppercase word **DICT** before the actual file name and separated from it by a single space.

In a QMBasic program, *file.name* is evaluated in the same way as any other expression. In a dictionary I-type record, *file.name* may be specified as a quoted string or as the actual name of the file, optionally preceded by the **DICT** qualifier.

*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 (D/I-type or A/S-type with no correlative) 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.

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).

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 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.

```
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 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.

## TRIM()

The **TRIM()** function removes excess characters from a string.

### Format

**TRIM**(*string*)

**TRIM**(*string*, *character*{, *mode*})

where

*string* evaluates to the string to be trimmed.

*character* is the character to be removed

*mode* evaluates to a single character which determines the mode of trimming:

- A Remove all occurrences of *character*.
- B Remove all leading and trailing occurrences of *character*.
- C Replace multiple instances of *character* with a single character.
- D Remove all leading and trailing spaces, replacing multiple embedded spaces with a single space. The value of *character* is ignored.
- E Remove all trailing spaces. The value of *character* is ignored.
- F Remove all leading spaces. The value of *character* is ignored.
- L Remove all leading occurrences of *character*.
- R Remove all leading and trailing occurrences of *character*, replacing multiple embedded instances of *character* with a single character.
- T Remove all trailing occurrences of *character*.

The first format of the **TRIM()** function removes all leading and trailing spaces from *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 = " 1 2 3 "
```

```
Y = TRIM(X)
```

This program fragment removes excess spaces from string X setting Y to "1 2 3"

```
X = "ABRACADABRA"
```

```
Y = TRIM(X, 'A', 'A')
```

This program fragment removes all occurrence of the letter A from string X setting Y to "BRCD BR"

```
X = "ABRACADABRA"  
Y = TRIM(X, 'A', 'B')
```

This program fragment removes leading and trailing occurrences of the letter A from string X setting Y to "BARCADABR"

**See also:**

[TRIMB\(\)](#), [TRIMF\(\)](#), [TRIMS\(\)](#)

## 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

**TRIMB**(*string*)

where

*string* evaluates to the string to be trimmed.

The **TRIMB()** function removes all trailing spaces from *string*.

### Examples

```
A = " 1 2 3 "
```

```
B = TRIMB(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 = TRIMBS(A)
```

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 3<sub>FM</sub> 4 5 6"

### See also:

[TRIM\(\)](#), [TRIME\(\)](#), [TRIMS\(\)](#)

## 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 FM4 5 6 "

### See also:

[TRIM\(\)](#), [TRIMB\(\)](#), [TRIMS\(\)](#)

## 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.

*character* is the character to be removed

*mode* evaluates to a single character which determines the mode of trimming:

- A Remove all occurrences of *character*.
- B Remove all leading and trailing occurrences of *character*.
- C Replace multiple instances of *character* with a single character.
- D Remove all leading and trailing spaces, replacing multiple embedded spaces with a single space. The value of *character* is ignored.
- E Remove all trailing spaces. The value of *character* is ignored.
- F Remove all leading spaces. The value of *character* is ignored.
- L Remove all leading occurrences of *character*.
- R Remove all leading and trailing occurrences of *character*, replacing multiple embedded instances of *character* with a single character.
- T Remove all trailing occurrences of *character*.

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 3<sub>FM</sub>4 5 6"

### See also:

[TRIM\(\)](#), [TRIMB\(\)](#), [TRIME\(\)](#)

## 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.

<b>Field</b>	<b>Content</b>
1	Ctrl-C treated as the break key? (PTERM BREAK <i>mode</i> )
2	Case inversion on? (PTERM CASE <i>mode</i> )
3	Break character value (PTERM BREAK <i>n</i> )
4	Output newline sequence (PTERM NEWLINE)
5	Input return key code (PTERM RETURN)

### 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 (1) if it is unassigned or (0) 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](#), [READSEQ](#), [READU](#),  
[READV](#), [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

**UPCASE**(*string*)

where

*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.

### Example

```
NAME = "Thomas Smith"  
PRINT UPCASE(NAME)
```

This program fragment prints the string "THOMAS SMITH".

### See also:

[DOWNCASE\(\)](#)

## VOCPATH()

The **VOCPATH()** function returns the pathname for a file referenced via the VOC.

### Format

**VOCPATH**(*filename* {, *dict.flag*})

where

<i>filename</i>	is the file name for which the pathname is to be returned. The extended file name syntaxes are allowed.
<i>dict.flag</i>	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

```
DISPLAY VOPATH( 'SALES' )
```

This statement would display the pathname of the data part of the SALES file.

```
DISPLAY VOPATH( 'SALES' , @true )
```

Adding the *dict.flag* as true would display the pathname of the dictionary part of the SALES file.

```
DISPLAY VOPATH( 'DICT SALES' )
```

Use of the DICT prefix would also display the pathname of the dictionary part of the SALES file.

```
DISPLAY VOPATH( 'CUSTOMERS ,NORTH' )
```

Used with a multi-file reference, the pathname returned will be that of the NORTH subfile of the CUSTOMERS file.

---

```
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.

## VSLICE()

The **VSLICE()** function returns a string formed by extracting a given value or subvalue position from a dynamic array.

### Format

**VSLICE**(*string*, *vpos*)

**VSLICE**(*string*, *vpos*, *svpos*)

where

*string* is the string from which the data is to be extracted.

*vpos* evaluates to the value position to be extracted.

*svpos* evaluates to the subvalue position to be extracted.

The first form of the **VSLICE()** function processes *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 *string* to build a new dynamic array containing only the specified sub value position from each field. If *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 *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 *vpos* and *svpos* are both less than one, the **VSLICE()** function returns the source *string*.

### Examples

The following examples are all based on a *string* that contains

1<sub>FM</sub>2<sub>VM</sub>3<sub>SM</sub>4<sub>VM</sub>5<sub>FM</sub>6<sub>VM</sub>7<sub>SM</sub>8<sub>VM</sub>9

VSLICE(S, 0) 1<sub>FM</sub>2<sub>VM</sub>3<sub>SM</sub>4<sub>VM</sub>5<sub>FM</sub>6<sub>VM</sub>7<sub>SM</sub>8<sub>VM</sub>9

VSLICE(S, 1) 1<sub>FM</sub>2<sub>FM</sub>6

VSLICE(S, 2) <sub>FM</sub>3<sub>SM</sub>4<sub>FM</sub>7<sub>SM</sub>8

VSLICE(S, 3)

VSLICE(S, 0, 1) 1<sub>FM</sub>2<sub>VM</sub>3<sub>VM</sub>5<sub>FM</sub>6<sub>VM</sub>7<sub>VM</sub>9

VSLICE(S, 0, 2) <sub>FM</sub>4<sub>FM</sub>8

VSLICE(S, 0, 3)

VSLICE(S, 1, 1) 1<sub>FM</sub>2<sub>FM</sub>6

VSLICE(S, 1, 2)

VSLICE(S, 2, 1) <sub>FM</sub>3<sub>FM</sub>7

VSLICE(S, 2, 2) <sub>FM</sub>4<sub>FM</sub>8

VSLICE(S, 2, 3)

VSLICE(S, 3, 1) FM5FM9  
VSLICE(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

```
EVT = FILE.EVENT("C:\IMPORT", FE$FILE.NAME)
LOOP
  IF WAIT.FILE.EVENT(EVT, 10) > 0 THEN
    ...Processing...
```

```
        END ELSE
        ...Periodic actions...
    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](#). This function is not available on the PDA version of QM.

### 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](#),  
[READV](#), [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
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 TO file.var, record.id {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.

*statement(s)* are statements to be executed if the write fails.

The keyword **ON** may be used in place of **TO**.

The contents of *var* are written to the file. 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 **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. The [STATUS\(\)](#) function will return an error number. If no **ON ERROR** clause is present, an abort would occur.

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 [UMASK](#) configuration parameter or, if this parameter is not present, a default of 002. This behaviour can be modified by executing a [UMASK](#) command within the session.

### Example

```
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.SOCKET()

The **WRITE.SOCKET()** function writes data to a socket.

### Format

**WRITE.SOCKET**(*skt*, *data*, *flags*, *timeout*)

where

*skt* is the socket variable returned by [ACCEPT.SOCKET.CONNECTION\(\)](#) (stream connection), [CREATE.SERVER.SOCKET\(\)](#) (datagram connection) or [OPEN.SOCKET\(\)](#).

*data* is the data to be written.

*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 default mode of data transfer as blocking.
SKT\$NON.BLOCKING	Sets the default mode of data transfer as non-blocking.

If neither blocking flag is given, the blocking mode set when the socket was opened is used.

*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.SOCKET("193.118.13.14", 3000, SKT$BLOCKING)
IF STATUS() THEN STOP 'Cannot open socket'
N = WRITE.SOCKET(SKT, DATA, 0, 0)
CLOSE.SOCKET 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.SOCKET](#), [CREATE.SERVER.SOCKET\(\)](#), [OPEN.SOCKET\(\)](#), [READ.SOCKET\(\)](#).

---

[SELECT.SOCKET\(\)](#), [SERVER.ADDR\(\)](#), [SET.SOCKET.MODE\(\)](#), [SOCKET.INFO\(\)](#)

## WRITEBLK

The **WRITEBLK** statement writes data at the current file position in a record previously opened using **OPENSEQ**.

### Format

```
WRITEBLK var TO file.var  
{ ON ERROR statement(s) }  
{ THEN statement(s) }  
{ ELSE statement(s) }
```

where

<i>var</i>	is the name of a variable holding the data to be written.
<i>file.var</i>	is the file variable associated with the file.
<i>statement(s)</i>	are statements to be executed depending on the outcome of the <b>WRITEBLK</b> 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.

### 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, ... 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 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.

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.

### 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](#), [CSV DQ\(\)](#), [CSV.MODE](#), [NOBUF](#), [OPENSEQ](#), [PRINTCSV](#),  
[READBLK](#), [READCSV](#), [READSEQ](#), [WEOFSEQ](#), [WRITESEQ](#), [WRITEBLK](#),  
[WRITESEQF](#)

## WRITESEQ

The **WRITESEQ** statement writes a string array 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 TO file.var {ON ERROR statement(s)}  
  {THEN statement(s)}  
  {ELSE statement(s)}
```

where

<i>var</i>	is the variable containing the data to be written.
<i>file.var</i>	is the file variable associated with the record by a previous <b>OPENSEQ</b> statement.
<i>statement(s)</i>	are statement(s) to be executed depending on the outcome of the <b>WRITESEQ</b> .

The keyword **TO** may be replaced by **ON**. At least one of the **THEN** and **ELSE** clauses must be present.

The data in *var* 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 **ELSE** clause is executed if the **WRITESEQ** operation fails.

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.

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 TO file.var, record.id, field.expr  
{ ON ERROR statement(s)}
```

where

<i>var</i>	is the name of a variable containing the data to be written.
<i>file.var</i>	is the file variable associated with the file.
<i>record.id</i>	evaluates to the id of the record to be written.
<i>field.expr</i>	evaluates to the number of the field to be written.
<i>statement(s)</i>	are statements to be executed if the write fails.

The keyword **ON** may be used in place of **TO**.

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.

### 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:**  
[READY](#)

## **XTD()**

The **XTD()** function converts a string of hexadecimal characters to a number.

### **Format**

**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

The table below shows the keys that produce each character value on Windows systems using QMConsole, on all systems using QMTerm, or when using [KEYCODE\(\)](#) to decode key sequences.

	0	1	2	3	4	5	6	7	8	9
00x		Ctrl-A	Ctrl-B	Ctrl-C	Ctrl-D	Ctrl-E	Ctrl-F	Ctrl-G	Ctrl-H	Ctrl-I
									Bsp	Tab
01x	Ctrl-J	Ctrl-K	Ctrl-L	Ctrl-M	Ctrl-N	Ctrl-O	Ctrl-P	Ctrl-Q	Ctrl-R	Ctrl-S
	Ctrl-rtn			Return						
02x	Ctrl-T	Ctrl-U	Ctrl-V	Ctrl-W	Ctrl-X	Ctrl-Y	Ctrl-Z	Esc		Ctrl-}
03x	Ctrl-^	Ctrl-_	Space	!	"	#	\$	%	&	'
04x	(	)	*	+	,	-	.	/	0	1
05x	2	3	4	5	6	7	8	9	:	;
06x	<	=	>	?	@	A	B	C	D	E
07x	F	G	H	I	J	K	L	M	N	O
08x	P	Q	R	S	T	U	V	W	X	Y
09x	Z	[	\	]	^	_	`	a	b	c
10x	d	e	f	g	h	i	j	k	l	m
11x	n	o	p	q	r	s	t	u	v	w
12x	x	y	z			}	~	Ctrl-Bs p	F1	F2
13x	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12
14x	Ctrl-F1	Ctrl-F2	Ctrl-F3	Ctrl-F4	Ctrl-F5	Ctrl-F6	Ctrl-F7	Ctrl-F8	Ctrl-F9	Ctrl-F10
15x	Ctrl-F1 1	Ctrl-F1 2	Alt-F1	Alt-F2	Alt-F3	Alt-F4	Alt-F5	Alt-F6	Alt-F7	Alt-F8
16x	Alt-F9	Alt-F10	Alt-F11	Alt-F12	Sh-F1	Sh-F2	Sh-F3	Sh-F4	Sh-F5	Sh-F6
17x	Sh-F7	Sh-F8	Sh-F9	Sh-F10	Sh-F11	Sh-F12				
18x										
19x										
20x	Mouse			CsrLeft	CsrRgt	CsrUp	CsrDn	Pg Up	Pg Dn	Home
21x	End	Insert	Delete	Ctrl-Ta b	C-PgUp	C-PgDn	C-Home	C-End	User0	User1

22x	User2	User3	User4	User5	User6	User7	User8	User9		
23x										
24x										
25x										

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.

## 6.7 @-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.

### 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 0

@TRUE 1

### Variables

Except where indicated in the descriptions, these items are read-only

@ABORT.CODE	<p>A value indicating the cause of execution of the last abort. This variable is particularly useful within <a href="#">ON.ABORT</a> paragraphs or programs invoked from them. Values are:</p> <ol style="list-style-type: none"><li>0 No abort has occurred</li><li>1 A QMBasic ABORT statement or the ABORT command has been used</li><li>2 The Abort option has been selected after the break key was pressed</li><li>3 An internal error has been detected</li></ol> <p>The value of @ABORT.CODE is initially zero and is reset to zero only by the EXECUTE statement</p>
@ABORT.MESSAGE	Contains the text of any message associated with the most recent abort event.
@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.
@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 <a href="#">user defined conversion codes</a> .
@CRTHIGH	Contains the number of lines per page of the display.
@CRTWIDE	Contains the width of the display.
@DATA	Data from <a href="#">REFORMAT</a> 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 <a href="#">BREAK.SUP</a> prefix). For dictionary records that contain object code (C/I types and A/S types with correlatives), the object code is omitted. This variable can be of use in post-processing query processor output.
@DS	Contains the operating system specific directory delimiter character, \ on Windows, / on other platforms.
@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 <a href="#">BREAK.SUP</a> prefix). This variable can be of use in post-processing query processor output.
@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 <b>EXECUTE</b> level increments this by one, decrementing on return from that level.
@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.
@MONTH	The month in which the last command started execution as a two digit value. Changing @DATE will also change this value.
@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.
@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.
@POB	The PROC primary output buffer.
@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.
@SOB	The PROC secondary output buffer.
@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 <a href="#">STARTUP</a> 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.
@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

QMBasic includes a set of standard subroutines that may be called from user programs. These all have an exclamation mark prefix to the subroutine name.

<a href="#"><u>!ABSPATH()</u></a>	Form an absolute pathname from a directory and file path
<a href="#"><u>!ACCOUNT.RULES</u></a>	Determine list of accounts that a user may or may not enter
<a href="#"><u>!ATVAR()</u></a>	Return value of an @-variable
<a href="#"><u>!ERRTEXT()</u></a>	Return text description of an error number
<a href="#"><u>!GETPU()</u></a>	Get print unit characteristics
<a href="#"><u>!LISTU()</u></a>	Get raw data similar to LISTU command
<a href="#"><u>!PATHTKN()</u></a>	Process special tokens in a VOC or ACCOUNTS file pathname
<a href="#"><u>!PARSER()</u></a>	Command line parser
<a href="#"><u>!PCL()</u></a>	PCL control code functions
<a href="#"><u>!PHLOG()</u></a>	Return the log record name for a running phantom process
<a href="#"><u>!PICK()</u></a>	Display a pick list of options
<a href="#"><u>!PICKLIST()</u></a>	Display a pick list of options
<a href="#"><u>!QMCLIENT()</u></a>	QMClient interface from QMBasic
<a href="#"><u>!SCREEN()</u></a>	Screen driver
<a href="#"><u>!SETPU()</u></a>	Set print unit characteristics
<a href="#"><u>!SETVAR()</u></a>	Set the value of an @-variable
<a href="#"><u>!SORT()</u></a>	Sort a delimited list
<a href="#"><u>!USERNAME()</u></a>	Return user name for a given user number
<a href="#"><u>!USERNO()</u></a>	Return a list of user numbers for a given user name
<a href="#"><u>!VOCREC()</u></a>	Read a VOC record, following remote pointers

## !ABSPATH()

The **!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

Dir	File	Path
Any	@QMSYS\ACCOUNTS	C:\QMSYS\ACCOUNTS
Any	\SALES\CUSTOMERS	\SALES\CUSTOMERS
Any	C:\SALES\CUSTOMERS	C:\SALES\CUSTOMERS
C:\SALES	CUSTOMERS	C:\SALES\CUSTOMERS
C:\	CUSTOMERS	C:\CUSTOMERS

## !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 (1) if the list of accounts is those from which the user is barred, false (0) 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 username 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.

## !ERRTEXT()

The **!ERRTEXT()** subroutine returns a text description of an error number.

### Format

```
CALL !ERRTEXT(text, errno)
```

where

*text* is the returned descriptive text.

*errno* is the error number.

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 be called before any actions are performed that might lose this value (e.g. file operations).

If *errno* is not recognised, the subroutine returns *errno* as the *text* description.

### Examples

```
CALL !ERRTEXT(TEXT, STATUS())  
DISPLAY 'Error ' : STATUS() : ' ' : TEXT
```

or

```
DEFFUN ERRTEXT(ERRNO) CALLING "!ERRTEXT"  
DISPLAY 'Error ' : STATUS() : ' ' : ERRTEXT(STATUS())
```

## !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

```
CALL !LISTU(dyn.array)
```

where

*dyn.array* 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 *dyn.array* is a dynamic array where each field is multivalued with a value for each active QM process. The fields are:

- 1 User number. This will be in ascending order.
- 2 Operating system level process id. This can be negative on some Windows platforms.
- 3 Process type (I = interactive, P = phantom, C = QMClient, N = QMNet).
- 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

Further fields or process types may be added in future releases.

### Example

```
CALL !LISTU(USERS)
NUM.USERS = DCOUNT(USERS<1>, @VM)
FOR I = 1 TO NUM.USERS
  IF USERS<3,I> = 'I' THEN
    DISPLAY FMT(USERS<1,I>, '3R') : ' ' : USERS<6,I>
  END
NEXT I
```

The above program fragment displays a list of the user numbers and user login names of all interactive processes.

## !PARSER()

The **!PARSER()** subroutine parses a command line.

### Format

**CALL !PARSER(key, type, string, keyword {, voc.rec})**

where

*key* identifies the operation to be performed:

0	PARSER\$RESET	Prepares to parse the data in <i>string</i> .
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 <i>string</i> 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 <i>string</i> .
2	PARSER\$STRING	A quoted string. The quotes are removed in <i>string</i> .
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 1 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 1 and 3.

*voc.rec* is an optional argument, returned as the VOC record when *string* corresponds to a VOC key.

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(*inpath*, *outpath*)**

where

*inpath* is the pathname to be processed.

*outpath* is the returned processed pathname. This may be the same variable as *inpath*.

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 <a href="#">TEMPDIR</a> 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.

## !PCL()

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, COMPRESSED, 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: *n*PT (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 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_hhmmss`.

The **!PHLOG()** subroutine allows a process to determine the name of the log record 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, *name* is returned as a null string.

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

<i>item</i>	is the returned item. This will be returned as a null string if no item is selected
<i>top.line</i>	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.
<i>item.list</i>	is a field mark delimited list of items to display. Long items will be truncated to fit a single line.
<i>title</i>	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.
<i>pos</i>	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 <i>pos</i> , this variable will be updated to contain position information related to the list. A subsequent call to <b>!PICK()</b> 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 # "" THE
    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\(\)](#)

## !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

```
session = object('!qmclient')
```

The table below lists the QMClient API calls and their actions available with this object.

QMConnect	bool = session->Connect(host, port, username, password, account)
QMCall	session->Call(subr{, args})
QMClearSelect	session->ClearSelect(listno)
QMClose	session->Close(fno)
QMConnected	bool = session->Connected
QMDelete	session->Delete(fno, id)
QMDeleteu	session->Deleteu(fno, id)
QMDisconnect	session->Disconnect
QMEndCommand	session->EndCommand
QMExecute	str = session->Execute(cmd)
QMLogto	bool = session->Logto(acc)
QMMarkMapping	session->MarkMapping(fno, state)
QMOpen	fno = session->Open(name)
QMRead	str = session->Read(fno, id, err)
QMReadl	str = session->Readl(fnom id, wait, err)
QMReadList	str = session->ReadList(listno, err)
QMReadNext	str = session->ReadNext(listno, err)
QMReadu	str = session->Readu(fno, id, wait, err)
QMRecordLock	session->RecordLock(fno, id, update, wait)
QMRelease	session->Release(fno, id)
QMRespond	str = session->Respond(response, err)
QMSelect	session->Select(fno, listno)
QMSelectIndex	session->SelectIndex(fno, indexname, indexvalue, listno)
QMSelectLeft	str = session->SelectLeft(fno, indexname, listno)
QMSelectRight	str = session->SelectRight(fno, indexname, listno)
QMSetLeft	session->Setleft(fno, indexname)
QMSetRight	session->SetRight(fno, indexname)
QMWrite	session->Write(fno, id, data)
QMWriteu	session->Writeu(fno, id, data)
QMStatus	session->ServerStatus
QMError	session->Error

For a more detailed description, see [QMClient](#).

**!SCREEN()**

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.

<i>step</i>	Action
0	Clear the screen and paint text and data from all steps except those items with X in their mode value.
-1	Paint text and data from all steps except those items with X in their mode value without clearing the screen.
-2	Clear the screen without painting any data.
-3	Return a single keystroke value.

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.
<i>n</i>	Function key used. <i>n</i> is the key value as in KEYIN.H.

**Example**

```
READ SCRIN FROM SCR.F, 'MY.SCREEN' ELSE ABORT 'Cannot read
screen'
DATA = ''
CALL !SCREEN(SCRIN, 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.

## !SORT()

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:
- 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.

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>
  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.

## !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.

## !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 and subroutines 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. 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. 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.

When used on other terminals, the debugger output is mixed with the application output.

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.

<b>NO.SOURCE.W</b>	Suppresses display of the warning message if the source code for a program
<b>ARNING</b>	module cannot be found.
<b>WINDOW <i>n</i></b>	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.

The current position in a program is referenced by a line number and an element number. Most QMBasic 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.

<b>ABORT</b>	Quit the program, generating an abort.
<b>BRK</b> { <i>n</i> }	Set a breakpoint on line <i>n</i> . Shows breakpoints if <i>n</i> omitted.
<b>CLR</b>	Clear all breakpoints.
<b>CLR</b> <i>n</i>	Clear breakpoint on line <i>n</i> .
<b>DUMP</b> <i>var path</i>	Dumps a variable to an operating system level file.
<b>EP</b>	Exit program, returning to parent program or external subroutine.

<b>EXit</b>	Exit subroutine, returning to parent program, internal or external subroutine.
<b>Goto <i>n</i></b>	Continue execution at line <i>n</i> .
<b>HELP</b>	Display help page.
<b>PDUMP</b>	Generate a <a href="#">process dump file</a> .
<b>Quit</b>	Quit the program, generating an abort.
<b>Run</b>	Free run.
<b>Run <i>n</i></b>	Run to line <i>n</i> .
<b>SET <i>var=value</i></b>	Change content of a program variable (see below)
<b>STACK</b>	Display the call stack. The current program is shown first.
<b>Step <i>n</i></b>	Execute <i>n</i> lines.
<b>Step .<i>n</i></b>	Execute <i>n</i> elements.
<b>StepOver</b>	Step over subroutine call (internal or external)
<b>STOP</b>	Quit the program, generating a stop.
<b>UnWatch</b>	Cancel an active watch action.
<b>View</b>	Display user screen (normal program output)
<b>Watch {<i>var</i>}</b>	Watches the named variable (see below). Shows watched variable if <i>var</i> omitted.
<b>XEQ <i>command</i></b>	Execute <i>command</i> . Note that some commands may interfere with correct operation of the debugger.

The following commands apply only to full screen mode debugging:

<b>SRC</b>	Revert to default program source display
<b>SRC <i>name</i></b>	Show source of program <i>name</i> .
<b>SRC <i>n</i></b>	Display around line <i>n</i> of currently displayed program.
<b>SRC +<i>n</i></b>	Move display forward <i>n</i> lines in program.
<b>SRC -<i>n</i></b>	Move display backward <i>n</i> lines in program.
<b>WINDow <i>n</i></b>	Sets the number of lines (range 5 to 15) in the command area of the screen.

The following commands apply only to non-full screen mode debugging:

<b>SRC</b>	Display current source line
<b>SRC <i>n</i></b>	Display source line <i>n</i> . Entering a blank debugger command line after this command will display the next source line.
<b>SRC <i>n,m</i></b>	Display <i>m</i> lines starting at source line <i>n</i> . The value of <i>m</i> is limited to three lines less than the screen size. Entering a blank debugger command line after this command will display the next <i>m</i> source lines.

## 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. If a subroutine is executed recursively, it is only possible to view the current instance of the variables.

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.

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): "487FM912VM338"
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, pressing the return key shows the elements of the array. 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.

<code>SET var = value</code>	to set a numeric value
<code>SET var = "string"</code>	to set a string value. Double quotes, single quotes or backslashes may be used to enclose the string.
<code>SET var(row,col) = value</code>	to set a matrix element

### Watching Variables

The **WATCH** command causes the debugger to monitor the named variable. 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, redimensioning a common block, etc.

## 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 three 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.

Selection of the P option following use of the break key.

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

### 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 `\nn` where `nn` 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\$TERMINFO	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
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	Terminfo 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.STORD	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\$NSEQ	Not a sequential file
3014	ER\$NEOF	Not at end of file
3015	ER\$SQRD	Sequential file record read before creation
3016	ER\$SQNC	Sequential record not created due to error
3017	ER\$SQEX	Sequential record already exists (CREATE)
3018	ER\$RDONLY	Update to read only file
3019	ER\$AKNF	AK index not found
3020	ER\$INVAPATH	Invalid pathname

---

3021	ER\$EXCLUSIVE	Cannot gain exclusive access to file
3022	ER\$TRIGGER	Trigger function error
3023	ER\$NOLOCK	Attempt to write/delete record with no lock
3024	ER\$REMOTE	Open of remote file not allowed
3025	ER\$NOTNOW	Action cannot be performed now
3026	ER\$PORT	File is a port
3027	ER\$NPORT	File is not a port
3028	ER\$SQSEEK	Seek to invalid <i>offset</i> in sequential file
3029	ER\$SQREL	Invalid SEEK <i>relto</i> in sequential file
3030	ER\$EOF	End of file
3031	ER\$CNF	Multifile component not found
3032	ER\$MFC	Multifile reference with no component name
3033	ER\$PNF	Port not found
3034	ER\$BAD.DICT	Bad dictionary entry
3035	ER\$PERM	Permissions error
3036	ER\$SEEK.ERROR	Seek error
3037	ER\$WRITE.ERROR	Write error
3038	ER\$VFS.NAME	Bad class name in VFS entry
3039	ER\$VFS.CLASS	VFS class routine not found
3040	ER\$VFS.NGLBL	VFS class routine is not globally catalogued
3041	ER\$ENCRYPTED	Access denied to encrypted file
3042	ER\$NOT.A.FILE	Pathname or variable is not a file
3043	ER\$COMMIT.FAIL	Transaction commit failed
3044	ER\$ROLLBACK	Transaction rolled back
3045	ER\$SIZE	File is too large for this operation
3046	ER\$FILETYPE	Invalid file type for this operation
3047	ER\$PART	Invalid part number
3048	ER\$AK.ERR	Execution error in AK expression
3049	ER\$DNF	Device not found
3050	ER\$IDX.VERIFY	Index verification error
3051	ER\$QPTR.PATH	Q-pointer with account name barred
3052	ER\$ACC.BARRED	Access to this account is barred
3053	ER\$TRG.NOCAT	Trigger function not catalogued
4000	ER\$SRVMEM	Insufficient memory for packet buffer
4001	ER\$CONTEXT	Context error
5000	ER\$NO.DLL	DLL not found
5001	ER\$NO.API	API not found
5002	ER\$NO.TEMP	Cannot open temporary file
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\$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	DH\$FILE.NOT.OPEN	DH.FILE pointer is NULL
8002	DH\$NOT.A.FILE	DH.FILE does not point to a file descriptor
8003	DH\$ID.LEN.ERR	Invalid record id length
8004	DH\$SEEK.ERROR	Error seeking in DH file
8005	DH\$READ.ERROR	Error reading DH file
8006	DH\$WRITE.ERROR	Error writing DH file
8007	DH\$NAME.TOO.LONG	File name is too long
8008	DH\$SIZE	File exceeds maximum allowable size
8009	DH\$STAT.ERR	Error from stat()
8100	DH\$OPEN.NO.MEMORY	No memory for DH.FILE structure
8101	DH\$FILE.NOT.FOUND	Cannot open primary subfile
8102	DH\$OPEN1.ERR	Cannot open overflow subfile
8103	DH\$PSFH.FAULT	Primary subfile header format error

8104	DHE\$OSFH.FAULT	Overflow subfile header format error
8105	DHE\$NO.BUFFERS	Unable to allocate file buffers
8106	DHE\$INVA.FILE.NAME	Invalid file name
8107	DHE\$TOO.MANY.FILES	The limit on the number of open files has been reached. See the <a href="#">NUMFILES</a> configuration parameter.
8108	DHE\$NO.MEM	No memory to allocate group buffer
8109	DHE\$AK.NOT.FOUND	Cannot open AK subfile
8110	DHE\$AK.HDR.READ.ERROR	Error reading AK header
8111	DHE\$AK.HDR.FAULT	AK subfile header format error
8112	DHE\$AK.ITYPE.ERROR	Format error in AK I-type code
8113	DHE\$AK.NODE.ERROR	Unrecognised node type
8114	DHE\$NO.SUCH.AK	Reference to non-existent AK
8115	DHE\$AK.DELETE.ERROR	Error deleting AK subfile
8116	DHE\$EXCLUSIVE	File is open for exclusive access
8117	DHE\$TRUSTED	Requires trusted program to open
8118	DHE\$VERSION	Incompatible file version
8119	DHE\$ID.LEN	File may contain id longer than <a href="#">MAXIDLEN</a>
8120	DHE\$AK.CROSS.CHECK	Relocated index pathname cross-check failure
8121	DHE\$HASH.TYPE	Unsupported hash type
8122	DHE\$AK.PERM	Permissions error on AK subfile
8123	DHE\$ECS	File requires ECS
8201	DHE\$ILLEGAL.GROUP.SIZE	Group size out of range
8202	DHE\$ILLEGAL.MIN.MODULUS	Minimum modulus < 1
8203	DHE\$ILLEGAL.BIG.RECORD.SIZE	Big record size invalid
8204	DHE\$ILLEGAL.MERGE.LOAD	Merge load invalid
8205	DHE\$ILLEGAL.SPLIT.LOAD	Split load invalid
8206	DHE\$FILE.EXISTS	File exists on create
8207	DHE\$CREATE.DIRECTORY	Cannot create directory
8208	DHE\$CREATE0.ERR	Cannot create primary subfile
8209	DHE\$CREATE1.ERR	Cannot create overflow subfile

8210	DHE\$PSFH.WRITE.E RROR	Failure writing primary subfile header
8211	DHE\$INIT.DATA.ERR OR	Failure initialising data bucket
8212	DHE\$ILLEGAL.HASH	Invalid hashing algorithm
8213	DHE\$OSFH.WRITE.E RROR	Failure writing overflow subfile header
8301	DHE\$RECORD.NOT.F OUND	Record not in file
8302	DHE\$BIG.CHAIN.EN D	Found end of big record chain early
8303	DHE\$NOT.BIG.REC	Big record pointer does not point to big record block
8401	DHE\$NO.SELECT	No select is active
8402	DHE\$OPEN2.ERR	Cannot open select list
8403	DHE\$GSL.WRITE.ER R	Error from write()
8404	DHE\$GSL.TRUNCAT E.ERR	Error from chsize()
8501	DHE\$AK.NAME.LEN	Index name too long
8502	DHE\$AK.EXISTS	AK already exists
8503	DHE\$AK.TOO.MANY	Too many AKs to create a new one
8504	DHE\$AK.CREATE.ER R	Unable to create AK subfile
8505	DHE\$AK.HDR.WRITE .ERROR	Error writing AK subfile header
8506	DHE\$AK.WRITE.ERR OR	Error writing AK node
8601	DHE\$PSF.CHSIZE.ER R	Error compacting primary subfile
8701	DHE\$ALL.LOCKED	All buffers are locked
8702	DHE\$SPLIT.HASH.ER ROR	Record does not hash to either group in split
8703	DHE\$WRONG.BIG.RE C	Big record chain error
8704	DHE\$FREE.COUNT.Z ERO	Overflow free count zero in dh.new.overflow()
8705	DHE\$FDS.OPEN.ERR	Cannot reopen subfile
8706	DHE\$POINTER.ERRO R	Internal file pointer fault
8707	DHE\$NO.INDICES	File has no AKs
8708	DHE\$RECLLEN.ERRO R	Invalid record length
8900	DHE\$ECB.TYPE	ECB has incorrect type
8901	DHE\$VAULT.CLOSE	Key vault not open

## D

---

9000 DHE\$RPL.LIMIT	Too many replication targets
9001 DHE\$RPL.MKDIR	Error creating replication buffer area
9002 DHE\$RCB.TYPE	RCB type error

## 6.12 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 dealers can contact Ladybridge Systems 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_2-11-1.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>

## 6.13 Building a Web Server Application

There are advanced third party web based packages available for QM but for many applications 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.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <\qmsys\syscom\qmclilib.h>          [ Windows ]
#include </usr/qmsys/SYSCOM/qmclilib.h>     [ Linux / FreeBSD /
AIX / Mac ]

// 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

Historically, multivalue databases have used a character based user interface. QM includes a set of Windows OLE compatible functions that enable development of applications in, for example, Visual Basic. This section describes these functions and includes examples of how to use them to develop a Windows style front end to your application.

The same functions are also available in the qmclilib library (qmclilib.dll on Windows, libqmclient.a on other platforms) for use in C programs and as a QMBasic class module for use in QMBasic programs. All of these API sets are discussed here.

In addition, the QMClient.pb record in the SYSCOM file contains an interface layer for use with the PureBasic product from Fantaisie Software.

A QMClient session is an interactive use of the system and hence is counted in determining the licensed user limit.

### Overview

The API functions enable applications written in languages such as Visual Basic or C to access data stored in a QM database or allow connection to remote QM systems from within QMBasic application programs. 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 catalogued subroutines on the server, etc. Where both ends of the connection support it, the network traffic is encrypted.

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.

### Licensing

The QMClient DLL and corresponding libraries 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.

### Using the Visual Basic API

The QMClient Windows API consists of two components; a Visual Basic module (QMClient.bas, QMClient.vb or QMClient 2005.vb) containing the API function definitions, and a dynamic link library (QMClient.dll) containing the actual interface functions.

To use the API functions in a Visual Basic application, include the appropriate module from the QMSYS file dependant on the version of Visual Basic in use. The distinction is based on the

representation of numeric data:

	16 bit	32 bit	64 bit
QMClient.bas	Integer	Long	
QMClient.vb	Integer	Long	
QMClient2005.vb	Short	Integer	Long

All function declaration prototypes shown in this section are based on the QMClient.bas definitions. Developers should see the relevant for the correct data types for the version of Visual Basic being used.

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.

QMClient allows multiple connections from a single client process. This allows development of applications that transfer data between accounts or servers.

Functions that return boolean values, return 0 for False, -1 for True in the Visual Basic API.

### Using the C API

Use of the C programmers' API is different depending on the compiler in use. On Linux, FreeBSD, AIX and Mac, 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 two import libraries are provided to include when linking the application; qmcllibl.lib for Borland C users and qmcllibms.lib for Microsoft C users. 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.

QMClient allows multiple connections from a single client process. This allows development of applications that transfer data between accounts or servers.

Functions that return boolean values, return 0 for False, 1 for True in the C API library.

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.

### 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

```
session = object("!qmclient")
```

The session is then connected to a server using the CONNECT method

```
ok = session->connect(hostname, port, user, password, account)
```

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 interface to QMClient.

## 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, 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 occurred. 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.

## API Function Summary

### Session Management

<a href="#">QMConnect()</a>	Establishes a QMClient session via a network
<a href="#">QMConnected()</a>	Verifies whether a QMClient session is open
<a href="#">QMConnectionType</a>	Sets parameters that affect the behaviour of QMClient
<a href="#">QMConnectLocal()</a>	Establishes a QMClient session on the local system
<a href="#">QMDisconnect</a>	Terminates a QMClient session
<a href="#">QMDisconnectAll</a>	Terminates all QMClient sessions from this client
<a href="#">QMGetSession</a>	Retrieves currently select session number
<a href="#">QMLogto()</a>	Moves to an alternative account
<a href="#">QMRevision()</a>	Returns the revision level of the client and server components.
<a href="#">QMSetSession</a>	Selects the session to which subsequent function calls relate
<a href="#">QMTrapCallAbort</a>	Enables client trapping of aborts in QMCall

### File Handling

<a href="#">QMClearSelect</a>	Clears a select list
<a href="#">QMClose</a>	Closes a file
<a href="#">QMDelete</a>	Deletes a record
<a href="#">QMDeleteu</a>	Deletes a record, retaining the lock
<a href="#">QMMarkMapping()</a>	Enables/disables mark mapping for a directory file
<a href="#">QMNextPartial()</a>	Return next part of a select list
<a href="#">QMOpen()</a>	Opens a file
<a href="#">QMRead()</a>	Reads a record without locking
<a href="#">QMReadl()</a>	Reads a record with a shareable read lock
<a href="#">QMReadList()</a>	Reads a select list
<a href="#">QMReadNext()</a>	Retrieves a record id from a select list
<a href="#">QMReadu()</a>	Reads a record with an exclusive update lock
<a href="#">QMRecordlock()</a>	Locks a record

<a href="#"><u>QMRecordlocked()</u></a>	Query the lock state of a record.
<a href="#"><u>QMRelease</u></a>	Releases a record lock
<a href="#"><u>QMSelect()</u></a>	Generates a select list
<a href="#"><u>QMSelectIndex()</u></a>	Generates a select list from an alternate key index
<a href="#"><u>QMSelectLeft()</u></a>	Scan left in an alternate key index
<a href="#"><u>QMSelectPartial()</u></a>	Optimised select operation that returns multiple ids
<a href="#"><u>QMSelectRight()</u></a>	Scan right in an alternate key index
<a href="#"><u>QMSetLeft()</u></a>	Position at the left in an alternate key index
<a href="#"><u>QMSetRight()</u></a>	Position at the right in an alternate key index
<a href="#"><u>QMWrite</u></a>	Writes a record
<a href="#"><u>QMWriteu</u></a>	Writes a record, retaining the lock

### Dynamic Array Manipulation

<a href="#"><u>QMDel()</u></a>	Deletes a field, value or subvalue
<a href="#"><u>QMExtract()</u></a>	Extracts a field, value or subvalue
<a href="#"><u>QMIns()</u></a>	Inserts a field, value or subvalue
<a href="#"><u>QMLocate()</u></a>	Searches for a field, value or subvalue
<a href="#"><u>QMReplace()</u></a>	Replaces a field, value or subvalue

### String Manipulation

<a href="#"><u>QMChange()</u></a>	Change substrings
<a href="#"><u>QMChecksum()</u></a>	Calculate checksum value
<a href="#"><u>QMDcount()</u></a>	Count delimited items in a string
<a href="#"><u>QMDecrypt()</u></a>	Decrypt data
<a href="#"><u>QMEncrypt()</u></a>	Encrypt data
<a href="#"><u>QMField()</u></a>	Extract substring from a delimited string
<a href="#"><u>QMFree()</u></a>	Free dynamically allocated memory (C API only)
<a href="#"><u>QMMatch()</u></a>	Test pattern match
<a href="#"><u>QMMatchfield()</u></a>	Extract data based on pattern match

### Command Execution

<a href="#"><u>QMEndCommand</u></a>	Abort an executed command
<a href="#"><u>QMExecute()</u></a>	Execute a command on the server
<a href="#"><u>QMRespond()</u></a>	Respond to a request for input from an executed command

### Subroutine Execution

<a href="#"><u>QMCall</u></a>	Call a catalogued subroutine on the server
<a href="#"><u>QMCallx</u></a>	Call a catalogued subroutine on the server, returning updated arguments with QMGetArg(). (C API only)
<a href="#"><u>QMGetArg()</u></a>	Retrieve an argument value after user of QMCallx. (C API only)

### Error Handling

<a href="#"><u>QMError()</u></a>	Returns extended error message text
<a href="#"><u>QMStatus()</u></a>	Returns STATUS() value

### Miscellaneous

[QMGetVar\(\)](#)            Return value of @-variable on server

Many functions have an *Errno* argument passed by reference as an Integer variable. This will be set to one of the following values broadly corresponding to the various clauses applicable to the equivalent QMBasic statements.

- 0 SV\_OK            Action successful
- 1 SV\_ON\_ERR OR    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 Visual Basic module relevant to the language in use.

## 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 [QMReadNext\(\)](#) 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 QMCall

The **QMCall** function calls a catalogued subroutine on the server. It is analogous to the QMBasic [CALL](#) statement.

### Format

VB     **QMCall ByVal** *SubrName* **as String, ByVal** *ArgCount* **as Integer, Optional ByRef**  
          *Arg1* **as String, ...**

C       **QMCall(char\*** *SubrName*, **short int** *ArgCount*, *ArgList...*)

Obj     *Session*->**Call**(*SubrName*, *ArgList...*)

where

*SubrName*     is the name of the subroutine to be called.

*ArgCount*     is the number of arguments following (not present in the QMBasic class module API).

*ArgList*      is a list of arguments to be passed to the subroutine. In the C API, these arguments are char\*.

The **QMCall** function calls the named catalogued subroutine on the server system. This subroutine may take up to 20 arguments. QMClient does not provide a method to call subroutines with a greater number of arguments or that are defined to pass a QMBasic dimensioned matrix as an argument.

In the Visual Basic and C APIs, there may be at most 20 variables named as arguments and these must be declared as strings.

In the C API, 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.

If the subroutine modifies the values of any of its arguments, this will be reflected in the variables specified in *ArgList*. It is a good idea to ensure that arguments that are only used for values returned from the subroutine are set to empty strings before the call to minimise data unnecessarily sent across the network.

The called subroutine may make use of any of the standard QMBasic programming statements and functions, however, it may not perform terminal input or output as there is no terminal associated with a server process.

### Examples

```
VB       ErrMsg = ""
          QMCall "SET.CREDIT.LIMIT", 3, Client, NewLimit, ErrMsg
          If ErrMsg <> "" Then MsgBox ErrMsg, VBExclamation,
```

```
        "Failed"
C      ErrMsg[0] = '\0';
      QMCall("SET.CREDIT.LIMIT", 3, Client, NewLimit, ErrMsg);
      If (ErrMsg[0] != '\0') printf("Failed - %s\n", ErrMsg);

QMBasic ErrMsg = ""
      session->Call("SET.CREDIT.LIMIT", Client, NewLimit,
      ErrMsg)
      if ErrMsg # "" then display "Failed - " : ErrMsg
```

The above example 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.

**See also:**

[QMCallx](#), [QMTrapCallAbort](#)

## 7.4 QMCallx

The **QMCallx** function calls a catalogued subroutine on the server. It is analogous to the QMBasic [CALL](#) statement.

### Format

```
C      int QMCallx(char * SubrName, short int ArgCount, ArgList...)
```

where

*SubrName* is the name of the subroutine to be called.

*ArgCount* is the number of arguments following (not present in the QMBasic class module API).

*ArgList* is a list of arguments to be passed to the subroutine. In the C API, these arguments are char \*.

The **QMCallx** function, available only in the C API, calls the named catalogued subroutine on the server system. This subroutine may take up to 20 arguments. QMClient does not provide a method to call subroutines with a greater number of arguments or that are defined to pass a QMBasic dimensioned matrix as an argument. The return value from QMCallx is zero if successful or non-zero in the case of an error. The actual return value is one of the SV\_XXX codes as defined in the qmclilib.h include record.

Unlike [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 [QMGetArg\(\)](#) function, removing the need to know the size of the returned data prior to the call. The memory allocated by this function must subsequently be released using [QMFree\(\)](#).

It is a good idea to ensure that arguments that are only used for values returned from the subroutine are set to empty strings before the call to minimise data unnecessarily sent across the network.

The called subroutine may make use of any of the standard QMBasic programming statements and functions, however, it may not perform terminal input or output as there is no terminal associated with a server process.

### Example

```
C      char * ErrMsg
      QMCallx("SET.CREDIT.LIMIT", 3, Client, NewLimit, "");
      ErrMsg = QMGetArg(3);
      If (ErrMsg != NULL) printf("Failed - %s\n", ErrMsg);
```

The above example 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 third argument is passed into **QMCallx()** as a null string 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.

**See also:**

[QMCall](#), [QMGetArg](#), [QMTrapCallAbort](#)

## 7.5 QMChange()

The **QMChange()** function replaces occurrences of one substring with another in a string. It is analogous to the QMBasic [CHANGE\(\)](#) function.

### Format

**VB**    **QMChange(ByVal Src as String, ByVal OldStr as String, ByVal NewStr as String, Optional ByRef Occurrences as Long, Optional ByRef Start as Long) as String**

**C**     **char \* QMChange(char \* Src, char \* OldStr, char \* NewStr, int Occurrences, int Start)**

where

*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.

*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*.

The **QMChange()** function returns a new string with the specified substrings replaced.

One 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.

Note that in the C API library, a statement of the form

```
rec = QMChange(rec, old, new, 0, 0)
```

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.

### Examples

```
VB    X = QMChange("ABRACADABRA", "A", "a", 3, 2)
```

```
C     X = QMChange("ABRACADABRA", "A", "a", 3, 2);
```

The above example changes three uppercase letter "A" to lowercase "a" in the supplied string starting at the second occurrence, returning the result. Note that the C version will set X too point to a dynamically allocated memory areas that must be released later using [QMFree\(\)](#).

## 7.6 QMChecksum()

The **QMChecksum()** function returns a checksum value for supplied data. It is analogous to the QMBasic [CHECKSUM\(\)](#) function.

### Format

**VB**    **QMChecksum(ByVal *Data* as String) as Long**

**C**     **int QMchecksum(char\* *Data*)**

where

*Data*            is the string to be processed

The **QMChecksum()** function calculates a checksum value for the supplied *data* item. 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.

### Examples

**VB**    **X = QMChecksum( "ABCDE " )**

**C**     **X = QMChecksum( "ABCDE " ) ;**

## 7.7 QMClearselect

The **QMClearSelect** function clears a select list. It is analogous to the QMBasic [CLEARSELECT](#) statement.

### Format

**VB**    **QMClearSelectByVal** *ListNo* as Integer

**C**      **void** **QMClearSelect**(int *ListNo*)

**Obj**    *Session*->**ClearSelect**(*ListNo*)

where

*ListNo*        is a valid select list number (0 to 10)

The **QMClearSelect** function clears the specified select list. No error occurs if the list was not active.

Applications that use select list 0 (the default select list) and could leave unprocessed items in the list should always clear it to avoid unwanted effects on later server processing.

### Examples

```
VB        QMSelect fClients, 1
          Do
            Id = QMReadNext(1, ErrNo)
            If ErrNo <> 0 Then Exit Do
            Rec = QMRead(fClients, Id, ErrNo)
            If QMExtract(Rec, 1, 0, 0) = "" Then Exit Do
          Loop
          QMClearSelect 1
```

```
C        QMSelect(fClients, 1);
          do {
            Id = QMReadNext(1);
            if (Id == NULL) break;
            Rec = QMRead(fClients, Id, ErrNo);
            QMFree(Id);
            Fld = QMExtract(Rec, 1, 0, 0);
            if (Fld == NULL) break;
            QMFree(Rec);
            QMFree(Fld);
          } while(1);
          QMClearSelect(1);
```

```
QMBasic session->Select(fClients, 1)
          loop
            Id = session->ReadNext(1)
          until Id = ""
            Rec = session->Read(fClients, Id, ErrNo)
```

```
until Rec<1> = ""
repeat
session->ClearSelect(1)
```

The above program fragment 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 will leave a partially processed select list, QMClearSelect is used to clear the list.

Note that the C example leaves variables Rec and Fld pointing to dynamically allocated memory areas on exit from the loop. These must be released using [QMFree\(\)](#) when they are no longer needed.

## 7.8 QMClose

The **QMClose** function closes a file. It is analogous to the QMBasic [CLOSE](#) statement.

### Format

VB **QMClose ByVal** *FileNo* as Integer

C **void QMClose(int** *FileNo*)

Obj *Session*->**Close**(*FileNo*)

where

*FileNo* 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.

### Examples

```
VB    fClients = QMOpen("CLIENTS")
      Rec = QMRead(fClients, ClientNo, ErrNo)
      QMClose(fClients)

C     fClients = QMOpen("CLIENTS");
      Rec = QMRead(fClients, ClientNo, ErrNo);
      QMClose(fClients);

QMBasic fClients = session->Open("CLIENTS")
      Rec = session->Read(fClients, ClientNo, ErrNo)
      session->Close(fClients)
```

The above program fragment opens the CLIENTS file, reads the record identified by ClientNo, and then closes the file. A real program should test the ErrNo status from the read to determine if the action was successful.

Note that the C example leaves variables Rec pointing to a dynamically allocated memory area that must be released using [QMFree\(\)](#) when no longer needed.

## 7.9 QMConnect()

The **QMConnect()** function establishes a QMClient session.

### Format

VB    **QMConnect(ByVal Host as String, ByVal Port as Integer, ByVal UserName as String, ByVal Password as String, ByVal Account as String) as Boolean**

C    **int QMConnect(char \* Host, int Port, char \* UserName, char \* Password, char \* Account)**

Obj    *Bool = Session->Connect(Host, Port, UserName, Password, Account)*

where

*Host*            is the IP address or name of the server system.

*Port*            is the port number to which connection is to be made. Set this to -1 to use the QM default 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. 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.

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.

### Examples

```
VB    If Not QMConnect(Host, -1, UserName, Password, Account)
      Then
          MsgBox QMError(), VBExclamation, "Failed to connect"
```

```
End If
C    if (!QMConnect(Host, -1, UserName, Password, Account))
        {
            printf("Failed to connect - %s\n", QMError());
            exit;
        }
QMBasic session = object("!qmclient")
    if not(session->Connect(Host, -1, UserName, Password,
Account)) then
        stop "Failed to connect - " : session->error()
    end
```

The above program fragment attempts to connect to the server identified by Host, using login credentials in UserName, Password and Account. If unsuccessful, it displays an error message including the text returned by [QMError\(\)](#).

**See also:**

[QMConnectLocal\(\)](#), [QMDisconnect](#)

## 7.10 QMConnected()

The **QMConnected()** function confirms whether a QMClient session is open.

### Format

VB    **QMConnected() as Boolean**

C     **int QMConnected()**

Obj   *Bool = Session->Connected*

The **QMConnected()** function can be used by an application to determine whether a client session is open.

### Examples

```
VB    If Not QMConnected() Then  
      MsgBox "Not connected", VbExclamation  
      End If
```

```
C     if (!QMConnected())  
      {  
      printf("Not connected");  
      exit;  
      }
```

```
QMBasic if not(session->Connected) then  
          stop "Not connected"  
          end
```

The above program tests whether a QMClient connection is open and, if not, displays an error message.

## 7.11 QMConnectionType

The **QMConnectionType** function sets parameters that affect the behaviour of QMClient. This function does not apply to the QMClient class module in QMBasic.

### Format

VB **QMConnectionType** ByVal *Type* as Integer

C **void QMConnectionType**(int *Type*)

where

*Type* A bit significant value that controls the session parameters.

The **QMConnectionType** function sets parameters that affect the behaviour of QMClient.

*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.

Additional bit values may be used in future releases or for internal undocumented purposes.

## 7.12 QMConnectLocal()

The **QMConnectLocal()** function establishes a QMClient session on the local system.

### Format

**VB**    **QMConnectLocal(ByVal Account as String) as Boolean**

**C**     **int QMConnectLocal(char\* Account)**

where

*Account*        is the name of the QM account to be accessed.

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. 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.

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.

NOTE: The underlying operating system call needed by **QMConnectLocal()** is not implemented on Windows 98/ME. It will be necessary to use [QMConnect](#) for these systems.

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.

**QMConnectLocal()** is not supported by the QMBasic class module API.

### Examples

```
VB  If Not QMConnectLocal(Account) Then
      MsgBox QMError(), VBExclamation, "Failed to connect"
    End If

C   if (!QMConnectLocal(Account))
      {
        printf("Failed to connect - %s\n", QMError());
        exit;
      }
```

The above program fragment attempts to connect to the local system using the account name in

Account. If unsuccessful, it displays an error message including the text returned by [QMError\(\)](#).

**See also:**

[QMConnect\(\)](#), [QMDisconnect](#)

## 7.13 QMDcount()

The **QMDcount()** function counts delimited items in a string. It is analogous to the QMBasic [DCOUNT\(\)](#) function.

### Format

**VB**    **QMDcount(ByVal Src as String, ByVal Delim as String) as Long**

**C**     **int QMDcount(char \* Src, char \* Delim)**

where

*Src*            is the string to be processed

*Delim*         is the delimiter character. If *Delim* is more than one character long, only the first character is used.

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 some single character delimiter.

### Examples

**VB**    `X = QMDcount(Address, ",")`

**C**     `X = QMDcount(Address, ",");`

The above example returns the number of comma separated items in the Address variable.

## 7.14 QMDecrypt()

The **QMDecrypt()** function decrypts data that has been encrypted for secure storage or transmission. It is analogous to the QMBasic [DECRYPT\(\)](#) function.

### Format

**VB**    **QMDecrypt(ByVal Data as String, ByVal KeyString as String) as String**

**C**     **char \* QMDecrypt(char \* Data, char \* KeyString)**

where

*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 useable 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 decrypted data.

Note that in the C API library, a statement of the form

```
rec = QMDecrypt(data, key)
```

will return a pointer to a newly allocated memory area that must be released using [QMFree\(\)](#) when no longer needed.

### Examples

```
VB    X = QMDecrypt(VAR, "MyKey")
```

```
C     X = QMDecrypt(VAR, "MyKey");
```

The above example decrypts the data in variable VAR using an encryption key of "MyKey".

### See also:

[QMEncrypt\(\)](#)

## 7.15 QMDel()

The **QMDel()** function deletes a field, value or subvalue from a dynamic array. It is analogous to the QMBasic [DELETE\(\)](#) function.

### Format

**VB**    **QMDel(ByVal Src as String, ByVal Fno as Integer, ByVal Vno as Integer, ByVal Svno as Integer) as String**

**C**     **char \* QMDel(char \* Src, int Fno, int Vno, int Svno)**

where

*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.

Note that in the C API library, a statement of the form

```
rec = QMDel(rec, 2, 1, 0)
```

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.

### Examples

**VB**    `NewRec = QMDel(Rec, 1, Pos, 0)`

**C**     `NewRec = QMDel(Rec, 1, Pos, 0);`

The above example returns a copy of the dynamic array *Rec* with field 1, value *Pos* removed.

Note that variables *Rec* and *NewRec* in the C example point to dynamically allocated memory areas that must be released using [QMFree\(\)](#) when they are no longer needed.

## 7.16 QMDelete()

The **QMDelete** function deletes a record from a file. It is analogous to the QMBasic [DELETE](#) statement.

### Format

VB **QMDelete ByVal FileNo as Integer, ByVal Id as String**

C **void QMDelete(int FileNo, char \* Id)**

Obj *Session->Delete(FileNo, Id)*

where

*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.

### Examples

```
VB    fClients = QMOpen("CLIENTS")
      QMRecordlock(fClients, ClientNo, True, True)
      QMDelete(fClients, ClientNo)
      QMClose(fClients)

C     fClients = QMOpen("CLIENTS");
      QMRecordlock(fClients, ClientNo, TRUE, TRUE);
      QMDelete(fClients, ClientNo);
      QMClose(fClients);

QMBasic fClients = session->Open("CLIENTS")
        session->Recordlock(fClients, ClientNo, @true, @true)
        session->Delete(fClients, ClientNo)
        session->Close(fClients)
```

The above program fragment 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.

## 7.17 QMDeleteu()

The **QMDeleteu** function deletes a record from a file, retaining the record lock. It is analogous to the QMBasic [DELETEU](#) statement.

### Format

VB **QMDeleteu ByVal FileNo as Integer, ByVal Id as String**

C **void QMDeleteu(int FileNo, char \* Id)**

Obj *Session*->**Deleteu**(FileNo, Id)

where

*FileNo* is the file number returned by a previous [QMOpen\(\)](#) call.

*Id* is 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 retained on return from this function.

### Examples

```
VB    fClients = QMOpen("CLIENTS")
      QMRecordlock(fClients, ClientNo, True, True)
      QMDeleteu(fClients, ClientNo)

C    fClients = QMOpen("CLIENTS");
      QMRecordlock(fClients, ClientNo, TRUE, TRUE);
      QMDeleteu(fClients, ClientNo);

QMBasic fClients = session->Open("CLIENTS")
        session->Recordlock(fClients, ClientNo, @true, @true)
        session->Deleteu(fClients, ClientNo)
```

The above program fragment opens the CLIENTS file, locks and deletes the record identified by ClientNo, retaining the lock.

## 7.18 QMDisconnect

The **QMDisconnect** function terminates a QMClient session.

### Format

VB    **QMDisconnect**  
C    **void QMDisconnect(void)**  
Obj   *Session*->**Disconnect**

The **QMDisconnect** function terminates a QMClient session previously established using [QMConnect\(\)](#) or [QMConnectLocal\(\)](#).

### Examples

```
VB    QMDisconnect  
C    QMDisconnect();  
QMBa session->disconnect  
sic
```

The above statement closes the current QMClient connection.

### See also:

[QMConnect\(\)](#), [QMConnectLocal\(\)](#), [QMDisconnectAll](#)

## 7.19 QMDisconnectAll

The **QMDisconnectAll** function terminates all QMClient sessions from a client process.

### Format

```
VB    QMDisconnectAll
C     void QMDisconnectAll(void)
```

A single client may establish multiple QMClient connections. The **QMDisconnectAll** function terminates all such connections.

The **QMDisconnectAll** function has no equivalent in the QMBasic class module API as each session is a separate instantiation of the object.

### Examples

```
VB    QMDisconnectAll
C     QMDisconnectAll();
```

The above statement closes all QMClient connections open by the program.

## 7.20 QMEncrypt()

The **QMEncrypt()** function decrypts data for secure storage or transmission. It is analogous to the QMBasic [ENCRYPT\(\)](#) function.

### Format

**VB**     **QMEncrypt(ByVal Data as String, ByVal KeyString as String) as String**

**C**       **char \* QMEncrypt(char \* Data, char \* KeyString)**

where

*Data*            is the data to be encrypted.

*KeyString*       is the encryption key to be used.

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 useable 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.

Note that in the C API library, a statement of the form

```
rec = QMEncrypt(data, key)
```

will return a pointer to a newly allocated memory area that must be released using [QMFree\(\)](#) when no longer needed.

### Examples

```
VB    X = QMEncrypt("ABRACADABRA", "MyKey")
```

```
C     X = QMEncrypt("ABRACADABRA", "MyKey");
```

The above example encrypts the string "ABRACADABRA" using an encryption key of "MyKey".

### See also:

[QMDecrypt\(\)](#)

## 7.21 QMEndCommand

The **QMEndCommand** function aborts a command executed on the server that is requesting input.

### Format

VB **QMEndCommand**

C **void QMEndCommand(void)**

Obj *Session*->**EndCommand**

This function may only be used when an immediately preceding [QMExecute\(\)](#) or [QMRespond\(\)](#) function has returned a status of SV\_PROMPT.

The **QMEndCommand** function causes the server command to be aborted.

### See also:

[QMExecute\(\)](#), [QMRespond\(\)](#)

## 7.22 QMError()

The **QMError()** function returns extended error message text.

### Format

VB **QMError()** as String

C char \* **QMError(void)**

VB *Str = Session->error*

Some API functions set extended error text and return an error code of **SV\_ERROR** when an error condition occurs. The **QMError()** function can be used to retrieve this text.

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.

## 7.23 QMExecute()

The `QMExecute()` function executes a command on the server.

### Format

VB `QMExecute(ByVal Cmnd as String, ByRef Errno as Integer) as String`

C `char * QMExecute(char * Cmnd, int * Errno)`

Obj `Str = Session->Execute(Cmnd, Errno)`

where

*Cmnd* is the command to be executed.

*Errno* is an integer variable to receive status information.

The `QMExecute()` function executes the specified command on the server. The output from this command is returned as a text string.

If the command completes without requesting input, the *Errno* variable is set to `SV_OK`.

If the command requests input, any output up to that point is returned and the *Errno* 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 of an [INPUT](#) statement will be ignored if present.

Execution of a further command from within the executed command may not behave correctly.

### Examples

```
VB  S = QMExecute("RUN MYPROG", ErrNo)
    while ErrNo = SV_PROMPT
        ...Process returned value and get new Response...
        S = QMRespond(Response, ErrNo)
    wend
    ...Process final response...
```

```
C  S = QMExecute("RUN MYPROG", &ErrNo)
    while(ErrNo == SV_PROMPT)
```

```
    {  
        ...Process returned value and get new Response...  
        QMFree(S);  
        S = QMRespond(Response, &ErrNo);  
    }  
    ...Process final response...  
QMBasic S = session->QMExecute("RUN MYPROG", ErrNo)  
LOOP  
WHILE ErrNo = SV$PROMPT  
    ...Process returned value and get new Response...  
REPEAT  
    ...Process final response...
```

The above program fragment runs 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.

**See also:**

[QMEndCommand](#), [QMRespond\(\)](#)

## 7.24 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.

### Format

**VB**    **QMExtract(ByVal Src as String, ByVal Fno as Integer, ByVal Vno as Integer, ByVal Svno as Integer) as String**

**C**     **char \* QMExtract(char \* Src, int Fno, int Vno, int Svno)**

where

*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.

### Examples

**VB**    `Rec = QMRead(fClients, ClientNo, ErrNo)  
      CliAddr = QMExtract(Rec, 4, 0, 0)`

**C**     `Rec = QMRead(fClients, ClientNo, ErrNo);  
      CliAddr = QMExtract(Rec, 4, 0, 0);`

The above program fragment reads the record identified by `ClientNo` from the file open as `fClients` and then extracts field 4 from it. A real program should test the `ErrNo` status from the read to determine if the action was successful.

Note that the C example leaves variables `CliAddr` pointing to a dynamically allocated memory area that must be released using [QMFree\(\)](#) when no longer needed.

## 7.25 QMField()

The **QMField()** function extracts one or more components of a delimited string. It is analogous to the QMBasic [FIELD\(\)](#) function.

### Format

**VB**    **QMField(ByVal *Src* as String, ByVal *Delimiter* as String, ByVal *Start* as Long, Optional ByRef *Occurrences* as Long) as String**

**C**     **char \* QMField(char \* *Src*, char \* *Delimiter*, int *Start*, int *Occurrences*)**

where

*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 omitted (VB only) or less than one, one component is returned.

The **QMField()** function returns the specified substring components of *Src*.

### Examples

```
VB    FirstName = QMField(Name, " ", 1)
```

```
C     FirstName = QMField(Name, ' ', 1, 1);
```

The above example takes a variable *Name* holding a person's first and last names separated by a space and returns just their first name.

Note that the C example leaves variables *FirstName* pointing to a dynamically allocated memory area that must be released using [QMFree\(\)](#) when no longer needed.

## 7.26 QMFree()

The **QMFree()** function releases memory returned by other functions. It is only used with the C API library.

### Format

```
void QMFree(void * addr)
```

where

*addr* is the pointer to a dynamic memory area returned by another API function.

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

```
Rec = QMReadu(fClients, ClientNo, TRUE, ErrNo);  
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 `ErrNo` status from the read operations to determine if they were successful. The [QMFree\(\)](#) is used to release the dynamic memory allocated to store `Rec` and `Rec2`.

## 7.27 QMGetArg()

The **QMGetArg()** function retrieves the value of argument variables updated by a catalogued subroutine on the server called using [QMCallx](#).

### Format

```
C      char * QMGetArg(short int ArgNo)
```

where

*ArgNo* is the argument number (from 1) to be retrieved.

The **QMGetArg()** function, available only in the C API, retrieves the value of an argument updated by a subroutine called using [QMCallx](#). Use of [QMCallx](#) and **QMGetArg()** removes the need for the calling program to know the maximum size of the returned data as is required with [QMCall](#).

The return value from this function is a pointer to a dynamically allocated memory area that receives the value of the specified argument. If the argument number is invalid or the called subroutine returned a null string, the pointer will be NULL. The memory allocated by **QMGetArg()** must subsequently be released using [QMFree\(\)](#).

An argument may be retrieved multiple times, each use of **QMGetArg()** allocating a new memory area. Use of **QMGetArg()** to retrieve an argument that was not modified by the subroutine will return the corresponding argument value as passed into [QMCallx](#). Argument data remains accessible until the next use of [QMCallx](#).

In applications that open multiple simultaneous server connections, the argument data is maintained on a per-session basis, reflecting the most recent use of [QMCallx](#) in the currently selected session.

### Example

```
C      char * ErrMsg
      QMCallx("SET.CREDIT.LIMIT", 3, Client, NewLimit, "");
      ErrMsg = QMGetArg(3);
      If (ErrMsg != NULL) printf("Failed - %s\n", ErrMsg);
```

The above example 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 third argument is passed into [QMCallx\(\)](#) as a null string 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.

**See also:**  
[QMCallx](#)

## 7.28 QMGetSession()

The **QMGetSession()** function returns the internal session number associated with the currently selected QMClient session.

### Format

VB    **QMGetSession() as Integer**

C     **int QMGetSession(void)**

A single client may open multiple QMClient connections, each identified by a session number. The [QMConnect\(\)](#) and [QMConnectLocal\(\)](#) 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\(\)](#).

The **QMGetSession()** function has no equivalent in the QMBasic class module API as each session is managed as a separate instantiation of the object.

## 7.29 QMGetVar()

The **QMGetVar()** function returns the value of an [@-variable](#) from the server.

### Format

VB **QMGetVar(ByVal Name as String) as String**

C **char \* QMGetVar(char \* Name)**

Obj *Str = Session->GetVar(Name)*

where

*Name* is the name of the @-variable to be returned. The leading @ symbol may be omitted.

The **QMGetVar()** function returns the value of an @-variable as a string. If the named variable does not exist, a null string is returned.

In the C API library, the dynamic memory allocated for the returned string must subsequently be freed by the calling program.

### Examples

```
VB SelectedCount = QMGetVar("@SELECTED")
```

```
C SelectedCount = QMGetVar("@SELECTED");
```

```
QMBasic SelectedCount = Session->QMGetVar("@SELECTED")
```

The above program fragment returns the value of the @SELECTED variable.

Note that the C example leaves variable SelectedCount pointing to dynamically allocated memory areas that must be released using [QMFree\(\)](#) when no longer needed.

## 7.30 QMIns()

The **QMIns()** function inserts a field, value or subvalue in a dynamic array. It is analogous to the QMBasic [INSERT\(\)](#) function.

### Format

**VB**    **QMIns(ByVal Src as String, ByVal Fno as Integer, ByVal Vno as Integer, ByVal Svno as Integer, ByVal NewData as String) as String**

**C**     **char \* QMIns(char \* Src, int Fno, int Vno, int Svno, char \* NewData)**

where

*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.

Note that in the C API library, a statement of the form

```
rec = QMIns(rec, 2, 1, 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.

### Examples

```
VB  Rec = QMReadu(fClients, ClientNo, True, ErrNo)
    Rec = QMIns(Rec, 1, Pos, 0, NewData)
    QMWrite fClients, ClientNo, Rec

C   Rec = QMReadu(fClients, ClientNo, TRUE, ErrNo);
    Rec2 = QMIns(Rec, 1, Pos, 0, NewData);
    QMWrite(fClients, ClientNo, Rec2);
    QMFree(Rec);
    QMFree(Rec2);
```

The above program fragment reads a record with an update lock, uses **QMIns()** to modify it and then writes it back to the file. A real program should test the ErrNo status from the read operations

to determine if they were successful.

Note how the C example uses [QMFree\(\)](#) to release the dynamic memory allocated to store Rec and Rec2.

## 7.31 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.

### Format

**VB**    **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**

**C**     **int QMLocate(char \* *Item*, char \* *DynArray*, int *Fno*, int *Vno*, int *Svno*, int \* *Pos*, char \* *Order*)**

where

*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.

*Order*            identifies the sort method to be applied. This may be:

AL     Ascending, left aligned

AR     Ascending, right aligned

DL     Descending, left aligned

DR     Descending, right aligned

If omitted, 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 **QMLocate()** function returns True if the item is found, False if it is not found.

### Examples

```
VB Found = QMLocate(PartNo, Order, 4, 1, 0, Pos, "AL")
C Found = QMLocate(PartNo, Order, 4, 1, 0, &Pos, "AL");
```

The above example searches field 4 of dynamic array Order for a value that contains the same data as the PartNo variable. 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, the above example specifies a value position of 1 to search the entire field.

## 7.32 QMLogto()

The **QMLogto()** function moves to an alternative account. It is analogous to the [LOGTO](#) command.

### Format

VB **QMLogto(ByVal Account as String) as Boolean**

C **int QMLogto(char \* Account)**

Obj *Bool = Session->Logto(Account)*

where

*Account* is the name of the QM account to be accessed.

The **QMLogto()** function attempts to move to the named account. 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.

### Examples

```
VB    If Not QMLogto(Account) Then
        MsgBox QMError(), VBExclamation, "Cannot move to new
        account"
    End If
```

```
C    if (!QMLogto(Account))
        {
            printf("Cannot move to new account - %s\n", QMError());
            exit;
        }
```

```
QMBasic if not(sesxsion->Logto(Account)) then
    stop "Cannot move to new account - " : QMError()
end
```

The above program fragment attempts to move to the account identified Account. If unsuccessful, it displays an error message including the text returned by [QMError\(\)](#).

## 7.33 QMMarkMapping

The **QMMarkMapping** function enables or disables mark mapping for a directory file.

### Format

VB **QMMarkMappingByVal** *FileNo* as Integer, **ByVal** *State* as Integer

C **void QMMarkMapping**(int *FileNo*, int *State*)

Obj *Session*->**MarkMapping**(*FileNo*, *State*)

where

*FileNo* is the file number returned by a previous [QMOpen\(\)](#) call.

*State* is non-zero to enable mark mapping, zero to disable.

The **QMMarkMapping** function enables or disables mark character mapping on the file open as *FileNo*. See the QMBasic [MARK.MAPPING](#) statement for more details.

## 7.34 QMMatch()

The **QMMatch()** function matches a string against a pattern template. It is analogous to the QMBasic [MATCHES](#) operator.

### Format

**VB**     **QMMatch(ByVal Src as String, ByVal Pattern as String) as Boolean**

**C**        **int QMMatch(char \* Src, char \* Pattern)**

where

*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
<i>n</i> X	Exactly <i>n</i> characters of any type
<i>n-m</i> X	Between <i>n</i> and <i>m</i> characters of any type
0A	Zero or more alphabetic characters
<i>n</i> A	Exactly <i>n</i> alphabetic characters
<i>n-m</i> A	Between <i>n</i> and <i>m</i> alphabetic characters
0N	Zero or more numeric characters
<i>n</i> N	Exactly <i>n</i> numeric characters
<i>n-m</i> N	Between <i>n</i> and <i>m</i> 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, *n*A, 0N, *n*N 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-m*X 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-m*A 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.

**Examples**

```
VB OK = QMMatch(PartCode, "2-4A3N")
```

```
C OK = QMMatch(PartCode, "2-4A3N");
```

The above example tests whether the contents of PartCode is formed from between two and four letters followed by three digits, returning OK as a true/false value.

## 7.35 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.

### Format

**VB**    **QMMatchfield(ByVal Src as String, ByVal Pattern as String, ByVal Component as Integer) as String**

**C**     **char \* QMMatchfield(char \* Src, char \* Pattern, int Component)**

where

*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.

### Examples

**VB**    Prefix = QMMatchfield(PartCode, "2-4A3N", 1)

**C**     Prefix = QMMatchfield(PartCode, "2-4A3N", 1);

The above example returns the alphabetic prefix of a value stored in *PartCode* if it is formed from between two and four letters followed three digits.

Note that *Prefix* variable in the C example points to dynamically a allocated memory area that must be released using [QMFree\(\)](#) when no longer needed.

## 7.36 QMNextPartial()

The **QMNextPartial** function returns the next part of a select list built using [QMSelectPartial\(\)](#).

### Format

VB **QMNextPartial(ByVal ListNo) as String**

C **void QMNextPartial(int ListNo)**

Obj *Session*->**NextPartial(ListNo)**

where

*ListNo* is 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.

### Examples

```
VB 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)
  Wend

C 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);
    }
QMBasic list = session->SelectPartial(fClients, 1)
loop
while list # ""
    for each id in list
        ...processing...
    next id
    list = session->NextPartial(1)
repeat
```

The above program fragment builds select list 1 and uses it to process records from the file open as fClients.

Note that the C example uses [QMFree\(\)](#) to release dynamically allocated memory areas returned from QMClient API calls.

## 7.37 QMOpen()

The **QMOpen()** function opens a file. It is analogous to the QMBasic [OPEN](#) statement.

### Format

VB     **QMOpen(ByVal *FileName* as String) as Integer**

C       **int QMOpen(char \* *FileName*)**

Obj     *FileNo* = *Session*->**Open(*FileName*)**

where

*FileName*     is 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 **QMOpen()** function opens a QM database file. 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.

### Examples

```
VB     fClients = QMOpen("CLIENTS")
       Rec = QMRead(fClients, ClientNo, ErrNo)
       QMClose(fClients)

C       fClients = QMOpen("CLIENTS");
       Rec = QMRead(fClients, ClientNo, ErrNo);
       QMClose(fClients);

QMBasic fClients = session->Open("CLIENTS")
       Rec = session->Read(fClients, ClientNo, ErrNo)
       session->Close(fClients)
```

The above program fragment opens the CLIENTS file, reads the record identified by ClientNo, and then closes the file. A real program should test the ErrNo status from the read to determine if the action was successful.

Note that the C example leaves variables Rec pointing to a dynamically allocated memory area that must be released using [QMFree\(\)](#) when no longer needed.

## 7.38 QMRead()

The **QMRead()** function reads a record without locking. It is analogous to the QMBasic [READ](#) statement.

### Format

**VB**     **QMRead(ByVal FileNo as Integer, ByVal Id as String, ByRef Errno as Integer) as String**

**C**       **char \* QMRead(int FileNo, char \* Id, int \* Errno)**

**Obj**     *Str = Session->Read(FileNo, Id, Errno)*

where

*FileNo*        is the file number returned by a previous [QMOpen\(\)](#) call.

*Id*             is the id of the record to be read.

*Errno*         is an integer variable to receive status information.

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 *Errno* variable is set to SV\_OK.

If the record cannot be found, the function returns a null string and the *Errno* 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 *Errno* variable is set to SV\_ON\_ERROR. The [QMStatus\(\)](#) function can be used to retrieve the error number.

In the C API library, the dynamic memory allocated for the returned string must subsequently be freed by the calling program. Note that attempting to read a non-existent record returns a pointer to a null string, not a NULL pointer.

### Examples

```
VB      fClients = QMOpen("CLIENTS")
        Rec = QMRead(fClients, ClientNo, ErrNo)
        QMClose(fClients)

C       fClients = QMOpen("CLIENTS");
        Rec = QMRead(fClients, ClientNo, &ErrNo);
        QMClose(fClients);

QMBasic fClients = session->Open("CLIENTS")
        Rec = session->Read(fClients, ClientNo, ErrNo)
```

---

```
session->Close(fClients)
```

The above program fragment opens the CLIENTS file, reads the record identified by ClientNo, and then closes the file. A real program should test the ErrNo status from the read to determine if the action was successful.

Note that the C example leaves variables Rec pointing to a dynamically allocated memory area that must be released using [QMFree\(\)](#) when no longer needed.

## 7.39 QMReadl()

The **QMReadl()** function reads a record with a shareable read lock. It is analogous to the QMBasic [READL](#) statement.

### Format

**VB**    **QMReadl(ByVal FileNo as Integer, ByVal Id as String, ByVal Wait as Boolean, ByRef Errno as Integer) as String**

**C**     **char \* QMReadl(int FileNo, char \* Id, int Wait, int \* Errno)**

**Obj**    *Str = Session->Readl(FileNo, Id, Wait, Errno)*

where

*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

*Errno*         is an integer variable to receive status information.

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 *Errno* 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 *Errno* 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 *Errno* 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.

In the C API library, the dynamic memory allocated for the returned string must subsequently be freed by the calling program. Note that attempting to read a non-existent record returns a pointer to

a null string, not a NULL pointer.

### Examples

```
VB      fClients = QMOpen("CLIENTS")
        Rec1 = QMReadl(fClients, ClientNo, True, ErrNo)
        Rec2 = QMReadl(fClients, OtherClient, True, ErrNo)

C      fClients = QMOpen("CLIENTS");
        Rec1 = QMReadl(fClients, ClientNo, TRUE, &ErrNo);
        Rec2 = QMReadl(fClients, OtherClient, TRUE, &ErrNo);

QMBasic fClients = session->Open("CLIENTS")
        Rec1 = session->Readl(fClients, ClientNo, @true, ErrNo)
        Rec2 = session->Readl(fClients, OtherClient, @true,
                               ErrNo)
```

The above program fragment opens 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 ErrNo status from the read operations to determine if they were successful.

Note that the C example leaves variables Rec1 and Rec2 pointing to dynamically allocated memory areas that must be released using **QMFree()** when no longer needed.

## 7.40 QMReadList()

The **QMReadList()** function reads a select list into a dynamic array in the client application. It is analogous to the QMBasic [READLIST](#) statement.

### Format

VB **QMReadList(ByVal *ListNo* as Integer, ByRef *Errno* as Integer) as String**

C **char \*QMReadList(int *ListNo*)**

Obj *Str* = *Session*->**ReadList**(*ListNo*, *Errno*)

where

*ListNo* is the number of the select list to be read in the range 0 to 10.

*Errno* receives an error value indicating the outcome of the request.

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.

The Visual Basic and object APIs returns an empty string if there is no data to read. The C API returns NULL in this situation.

See [Select lists in QMClient sessions](#) for a description of the alternative ways to handle select list with QMClient.

### Examples

```
VB QMSelect fClients, 1
   List = QMReadList(1, ErrNo)
```

```
C QMSelect(fClients, 1);
  List = QMReadList(1);
```

```
QMBasic session->Select(fClients, 1)
        List = session->ReadList(1, Errno)
```

The above program fragment builds select list 1 as a list of all records in the file open as fClients and then transfers this to a dynamic array named List.

Note that variable List in the C example points to a dynamically allocated memory area that must be released using [QMFree\(\)](#) when no longer required.

## 7.41 QMReadNext()

The **QMReadNext()** function retrieves the next entry from a select list. It is analogous to the QMBasic [READNEXT](#) statement.

### Format

VB **QMReadNext(ByVal ListNo as Integer, ByRef Errno as Integer) as String**

C **char\* QMReadNext(int ListNo)**

Obj *Str = Session->ReadNext(ListNo, ErrNo)*

where

*ListNo* is the number of the select list to be processed in the range 0 to 10.

*Errno* receives an error value indicating the outcome of the request. The C API does not have this argument and returns NULL if an error occurs.

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, *Errno* is set to SV\_OK.

If the list is empty, the Visual Basic and QMBasic API functions return a null string and *Errno* is set to SV\_ELSE. In the C API implementation, the function returns NULL.

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\(\)](#).

Note that in the C API library, the returned string is dynamically allocated. A loop containing a call to this function must free the memory from each call separately.

### Examples

```
VB    QMSelect fClients, 1
      Do
        Id = QMReadNext(1, ErrNo)
        If ErrNo <> 0 Then Exit Do
        Rec = QMRead(fClients, Id, ErrNo)
        If ErrNo = 0 Then Go Sub ProcessRecord
      Loop

C    QMSelect(fClients, 1);
    while((Id = QMReadNext(1)) != NULL)
```

```
        {
            if (Id == NULL) break;
            Rec = QMRead(fClients, Id, ErrNo);
            if (ErrNo == 0)
            {
                ProcessRecord();
                QMFree(Rec);
            }
            QMFree(Id);
        }
QMBasic session->Select(fClients, 1)
loop
    Id = session->ReadNext(1)
until Id = ""
    Rec = session->Read(fClients, Id, ErrNo)
    If ErrNo = 0 then gosub ProcessRecord
repeat
```

The above program fragment builds select list 1 and uses it to process records from the file open as fClients.

Note that the C example uses [QMFree\(\)](#) to release dynamically allocated memory areas returned from QMClient API calls.

## 7.42 QMReadu()

The **QMReadu()** function reads a record with an exclusive update lock. It is analogous to the QMBasic [READU](#) statement.

### Format

**VB**    **QMReadu(ByVal FileNo as Integer, ByVal Id as String, ByVal Wait as Boolean, ByRef Errno as Integer) as String**

**C**     **char \* QMReadu(int FileNo, char \* Id, int Wait, int \* Errno)**

**Obj**    *Str = Session->Readu(FileNo, Id, Wait, Errno)*

where

*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

*Errno*         is an integer variable to receive status information.

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 *Errno* 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 *Errno* 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 *Errno* 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 *Errno* variable is set to SV\_ON\_ERROR. The [QMStatus\(\)](#) function can be used to retrieve the error number.

In the C API library, the dynamic memory allocated for the returned string must subsequently be freed by the calling program. Note that attempting to read a non-existent record returns a pointer to

a null string, not a NULL pointer.

### Examples

```
VB      Rec = QMReadu(fClients, ClientNo, True, ErrNo)
        Rec = QMReplace(Rec, 1, Pos, 0, NewData)
        QMWrite fClients, ClientNo, Rec

C       Rec = QMReadu(fClients, ClientNo, TRUE, &ErrNo);
        Rec2 = QMReplace(Rec, 1, Pos, 0, NewData);
        QMWrite(fClients, ClientNo, Rec2);
        QMFree(Rec);
        QMFree(Rec2);

QMBasic Rec = session->Readu(fClients, ClientNo, @true, ErrNo)
        Rec<1, Pos> = NewData
        session->Write(fClients, Id, Rec)
```

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 ErrNo status from the read operations to determine if they were successful.

Note how the C example uses [QMFree\(\)](#) to release the dynamic memory allocated to store Rec and Rec2.

## 7.43 QMRecordlock

The **QMRecordlock** function locks a record. It is analogous to the QMBasic [RECORDLOCKL](#) and [RECRODLOCKU](#) statements.

### Format

**VB**    **QMRecordlock ByVal FileNo as Integer, ByVal Id as String, ByVal Update as Integer, ByVal Wait as Integer**

**C**      **void QMRecordlock(int FileNo, char \* Id, int Update, int Wait)**

**Obj**    *Session->***Recordlock**(FileNo, Id, Update, Wait)

where

*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.

### Examples

**VB**      QMRecordlock(fClients, ClientNo, True, True)  
           QMWrite fClients, ClientNo, Rec

**C**        QMRecordlock(fClients, ClientNo, TRUE, TRUE);  
           QMWrite(fClients, ClientNo, Rec2);

**QMBasic** session->Recordlock(fClients, ClientNo, @true, @true)  
           session->Write(fClients, Id, Rec)

The above program fragment writes 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.

**See also:**

[QMRecordlocked\(\)](#)

## 7.44 QMRecordlocked()

The **QMRecordlocked** function queries the lock on a record. It is analogous to the QMBasic [RECORDLOCKED\(\)](#) function.

### Format

VB **QMRecordlocked**(ByVal *FileNo* as Integer, ByVal *Id* as String) As Integer

C int **QMRecordlocked**(int *FileNo*, char \* *Id*)

Obj *Session*->**Recordlocked**(*FileNo*, *Id*)

where

*FileNo* is the file number returned by a previous [QMOpen\(\)](#) call.

*Id* is the id of the record to be locked.

The **QMRecordlocked()** function can be used to query the lock state of a record. The returned value is

Value	Token	Lock state
-3	LOCK\$OTHER.FILELOCK	Another user holds a file lock
-2	LOCK\$OTHER.READU	Another user holds an update lock
-1	LOCK\$OTHER.READL	Another user holds a read lock
0	LOCK\$NO.LOCK	The record is not locked
1	LOCK\$MY.READL	This user holds a read lock
2	LOCK\$MY.READU	This user holds an update lock
3	LOCK\$MY.FILELOCK	This user holds a file lock

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.

### Examples

VB State = QMRecordlocked(fClients, ClientNo)

C State = QMRecordlocked(fClients, ClientNo);

QMBasic State = session->Recordlock(fClients, ClientNo)

See also:

[RECORDLOCKED\(\)](#), [QMRecordlock](#)

## 7.45 QMRelease

The **QMRelease** function releases a record lock. It is analogous to the QMBasic [RELEASE](#) statement.

### Format

**VB**    **QMRelease ByVal FileNo as Integer, ByVal Id as String**

**C**     **void QMRelease(int FileNo, char \* Id)**

**Obj**    *Session*->**Release(FileNo, Id)**

where

*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.

### Examples

```
VB     Rec = QMReadu(fClients, ClientNo, True, ErrNo)
       If ErrNo = SV_ELSE Then
           QMRelease(fClients, ClientNo)
           Exit Sub
       End If
```

```
C     Rec = QMReadu(fClients, ClientNo, TRUE, ErrNo);
       if (ErrNo == SV_ELSE)
           {
               QMRelease(fClients, ClientNo);
               QMFree(Rec);
               return;
           }
```

```
QMBasic Rec = session->Readu(fClients, ClientNo, @true, ErrNo)
       if ErrNo = SV_ELSE then
           session->Release(fClients, Id)
           return
       end
```

The above program fragment attempts to read a record with an update lock. If the record is not found, the lock is released prior to exit from the subroutine.

Note how the C example uses [QMFree\(\)](#) to release the dynamic memory allocated to store Rec.

## 7.46 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.

### Format

**VB**    **QMReplace(ByVal Src as String, ByVal Fno as Integer, ByVal Vno as Integer, ByVal Svno as Integer, ByVal NewData as String) as String**

**C**     **char \* QMReplace(char \* Src, int Fno, int Vno, int Svno, char \* NewData as String)**

where

*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.

Note that in the C API library, a statement of the form

```
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.

### Examples

```
VB    Rec = QMReadu(fClients, ClientNo, True, ErrNo)
      Rec = QMReplace(Rec, 1, Pos, 0, NewData)
      QMWrite fClients, ClientNo, Rec
```

```
C     Rec = QMReadu(fClients, ClientNo, TRUE, ErrNo);
      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, uses **QMReplace()** to modify it and then writes it back to the file. A real program should test the ErrNo status from the read operations to determine if they were successful.

Note how the C example uses [QMFree\(\)](#) to release the dynamic memory allocated to store Rec and Rec2.

## 7.47 QMRespond()

The **QMRespond()** function responds to a request for input from a command executed on the server.

### Format

VB **QMRespond(ByVal *Response* as String, ByRef *Errno* as Integer) as String**

C **char \* QMRespond(char \* *Response*, int \* *Errno*)**

Obj *Str = Session->***Respond**(*Response*, *Errno*)

where

*Response* is the response to be sent.

*Errno* is an integer variable to receive status information.

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 completes without requesting input, the *Errno* variable is set to SV\_OK.

If the command requests further input, any output up to that point is returned and the *Errno* variable is set to SV\_PROMPT. The client may respond to this using the **QMRespond()** function or abort the command using the [QMEndCommand\(\)](#) function.

## 7.48 QMRevision()

The **QMRevision()** function returns the revision level of the client and server components.

### Format

VB    **QMRevision()** as String  
C     char \* **QMRevision(void)**  
VB    *Str = Session->revision*

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.

As for most functions in the C API that return strings, the calling program should free this memory using [QMFree\(\)](#).

## 7.49 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.

### Format

VB **QMSelect(ByVal FileNo as Integer, ByVal ListNo) as Integer**

C **void QMSelect(int FileNo, int ListNo)**

Obj *Session*->**Select**(FileNo, ListNo)

where

*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.

### Examples

```
VB    QMSelect fClients, 1
      Do
        Id = QMReadNext(1, ErrNo)
        If ErrNo <> 0 Then Exit Do
        Rec = QMRead(fClients, Id, ErrNo)
        If ErrNo = 0 Then Go Sub ProcessRecord
      Loop
```

```
C    QMSelect(fClients, 1);
    while((Id = QMReadNext(1)) != NULL)
    {
        Rec = QMRead(fClients, Id, ErrNo);
        if (ErrNo == 0)
        {
            ProcessRecord();
            QMFree(Rec);
        }
    }
    QMFree(Id);
```

```
    }  
QMBasic session->Select(fClients, 1)  
    loop  
        Id = session->ReadNext(1)  
    until Id = ""  
        Rec = session->Read(fClients, Id, ErrNo)  
        If ErrNo = 0 then gosub ProcessRecord  
    repeat
```

The above program fragment builds select list 1 and uses it to process records from the file open as fClients.

Note that the C example uses [QMFree\(\)](#) to release dynamically allocated memory areas returned from QMClient API calls.

## 7.50 QMSelectIndex

The **QMSelectIndex** function generates a select list from an alternate key index.

### Format

VB **QMSelectIndex ByVal** *FileNo* as Integer, **ByVal** *IndexName* as String, **ByVal** *IndexValue* as String, **ByVal** *ListNo* as Integer

C **void QMSelectIndex(int** *FileNo*, **char \*** *IndexName*, **char \*** *IndexValue*, **int** *ListNo*)

Obj *Session*->**SelectIndex**(*FileNo*, *IndexName*, *IndexValue*, *ListNo*)

where

*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.

## 7.51 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.

### Format

VB **QMSelectLeft(ByVal FileNo as Integer, ByVal IndexName as String, ByVal ListNo as Integer) as String**

C **char \* QMSelectLeft(int FileNo, char \* IndexName, int ListNo)**

Obj *Str = Session->SelectLeft(FileNo, IndexName, ListNo)*

where

*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 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**.

## 7.52 QMSelectPartial()

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.

### Format

VB **QMSelectPartial**(ByVal *FileNo* as Integer, ByVal *ListNo*) as String

C **void QMSelectPartial**(int *FileNo*, int *ListNo*)

Obj *Session*->**SelectPartial**(*FileNo*, *ListNo*)

where

*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.

### Examples

```
VB    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)
      Wend

C    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);
}
QMBasic list = session->SelectPartial(fClients, 1)
loop
while list # ""
  for each id in list
    ...processing...
  next id
  list = session->NextPartial(1)
repeat
```

The above program fragment builds select list 1 and uses it to process records from the file open as fClients.

Note that the C example uses [QMFree\(\)](#) to release dynamically allocated memory areas returned from QMClient API calls.

## 7.53 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.

### Format

VB **QMSetLeft** ByVal *FileNo* as Integer, ByVal *IndexName* as String

C **void QMSetLeft**(int *FileNo*, char \* *IndexName*)

Obj *Session*->**SetLeft**(*FileNo*, *IndexName*)

where

*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.

## 7.54 QMSetSession()

The **QMSetSession()** function selects an active QMClient session to be referenced by subsequent function calls.

### Format

VB     **QMSetSession(*Session as Integer*) as Boolean**

C     **int QMSetSession(int *session*)**

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 API as each session is managed by a separate instantiation of the object.

## 7.55 QMStatus()

The **QMStatus()** function returns the value of the QMBasic [STATUS\(\)](#) function for the last server function executed.

### Format

VB    **QMStatus()** as Long

C     **int QMStatus(void)**

Obj   *Var = Session->***ServerStatus**

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.

## 7.56 QMTrapCallAbort

The **QMTrapCallAbort** function enables or disables client side trapping of aborts in [QMCall](#).

### Format

VB    **QMTrapCallAbortByVal** *Mode* as Boolean

C     void **QMTrapCallAbort**(int *Mode*)

Obj   *Str = Session->TrapCallAbort*(*Mode*)

where

*Mode*            evaluates 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. 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

**See also:**

[QMCall](#)

## 7.57 QMWrite

The **QMWrite** function writes a record. This is analogous to the QMBasic [WRITE](#) statement.

### Format

VB **QMWrite ByVal *FileNo* as Integer, ByVal *Id* as String, ByVal *Rec* as String**

C **void QMWrite(int *FileNo*, char \* *Id*, char \* *Rec*)**

Obj *Session*->**Write(*FileNo*, *Id*, *Rec*)**

where

*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.

### Examples

```
VB Rec = QMReadu(fClients, ClientNo, True, ErrNo)
   Rec = QMReplace(Rec, 1, Pos, 0, NewData)
   QMWrite fClients, ClientNo, Rec
```

```
C Rec = QMReadu(fClients, ClientNo, TRUE, ErrNo);
  Rec2 = QMReplace(Rec, 1, Pos, 0, NewData);
  QMWrite(fClients, ClientNo, Rec2);
  QMFree(Rec);
  QMFree(Rec2);
```

```
QMBasic Rec = session->Readu(fClients, ClientNo, @true, ErrNo)
         Rec<1, Pos> = NewData
         session->Write(fClients, Id, Rec)
```

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 ErrNo status from the read operations to determine if they were successful.

---

Note how the C example uses [QMFree\(\)](#) to release the dynamic memory allocated to store Rec and Rec2.

## 7.58 QMWriteu

The **QMWriteu** function writes a record, retaining the lock. This is analogous to the QMBasic [WRITEU](#) statement.

### Format

VB **QMWriteu ByVal FileNo as Integer, ByVal Id as String, ByVal Rec as String**

C **void QMWriteu(int FileNo, char \* Id, char \* Rec)**

Obj *Session*->**Writeu**(FileNo, Id, Rec)

where

*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*. 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. The lock is retained on return from this function.

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.

### Examples

```
VB Rec = QMReadu(fClients, ClientNo, True, ErrNo)
   Rec = QMReplace(Rec, 1, Pos, 0, NewData)
   QMWriteu fClients, ClientNo, Rec
```

```
C Rec = QMReadu(fClients, ClientNo, TRUE, ErrNo);
  Rec2 = QMReplace(Rec, 1, Pos, 0, NewData);
  QMWriteu(fClients, ClientNo, Rec2);
  QMFree(Rec);
  QMFree(Rec2);
```

```
QMBasic Rec = session->Readu(fClients, ClientNo, @true, ErrNo)
         Rec<1, Pos> = NewData
         session->Writeu(fClients, Id, Rec)
```

The above program fragment reads a record with an update lock, modifies it and then writes it back to the file, retaining the lock so that the record may be written again later. A real program should

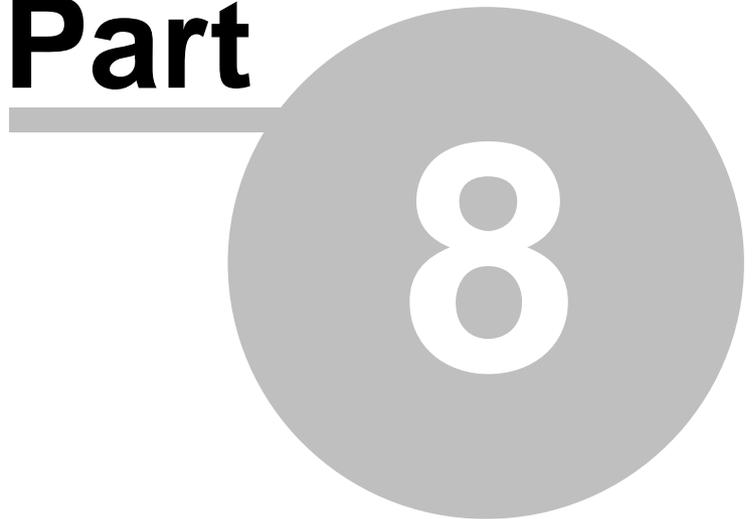
---

test the ErrNo status from the read operations to determine if they were successful.

Note how the C example uses [QMFree\(\)](#) to release the dynamic memory allocated to store Rec and Rec2.



**Part**



**System Administration**

## 8 System Administration

### System Configuration

<a href="#">Configuration Parameters</a>	Parameter descriptions
<a href="#">UPDATE.LICENCE</a>	Apply a new licence
<a href="#">The Termino Database</a>	Terminal configuration
<a href="#">The qmtic Utility</a>	Termino compiler

### Account Management

<a href="#">Accounts</a>	What is an account?
<a href="#">CREATE.ACCOUNT</a>	Create an account
<a href="#">DELETE.ACCOUNT</a>	Delete an account
<a href="#">The Login Process</a>	What happens when a user logs in

### System Security

<a href="#">ADMIN.USER</a>	User administration
<a href="#">CREATE.USER</a>	Create a user
<a href="#">DELETE.USER</a>	Delete a user
<a href="#">LIST.USERS</a>	List user name details
<a href="#">PASSWORD</a>	Password management (Windows 98/ME)
<a href="#">SECURITY</a>	Enable, disable or report security settings

### Process Management

<a href="#">LISTU</a>	Who is logged in?
<a href="#">PSTAT</a>	Monitoring processes
<a href="#">LOGOUT</a>	Killing a process

### Monitoring the File System

<a href="#">LIST.FILES</a>	Determining the files in use
<a href="#">LIST.LOCKS</a>	Monitoring task locks
<a href="#">LIST.READU</a>	Monitoring record and file locks
<a href="#">UNLOCK</a>	Releasing a lock
<a href="#">FSTAT</a>	Monitoring file activity
<a href="#">ANALYSE.FILE</a>	File analysis
<a href="#">FILE.STAT</a>	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 also contains 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.

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
```

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 <i>Lost Processes</i> in <a href="#">Monitoring the System</a> for more information.
CLIPORT	The port for incoming QMClient telnet connections. If not present or blank the default port 4243 is used.  Setting this port to zero disables incoming QMClient connections.  Take care not to confuse this parameter with the QMCLIENT parameter.
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 <a href="#">CONFIG</a> 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 the PDA version of QM. See <a href="#">Windows Code Pages</a> for more information.
DEADLOCK	If set to 1, QM aborts any program that attempts to wait for a lock that would result in a deadlock situation. The default value (0) allows deadlocks to occur. This parameter is not applicable to the PDA version of QM.

DHCACHE *	Determines the size of the dynamic file cache. See <a href="#">Dynamic Files</a> for more details
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.
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. This parameter is not applicable to the PDA version of QM.
EXCLREM *	If set to 1, remote files are omitted from <a href="#">ACCOUNT.SAVE</a> unless this exclusion is over-ridden by other mechanisms within the <a href="#">ACCOUNT.SAVE</a> command processing. This parameter is not applicable to the PDA version of QM.
FILERULE *	<p>Sets rules for special filename syntax usage. This value is formed by adding together the following options as required:</p> <ol style="list-style-type: none"> <li>1 Allow <i>account:file</i></li> <li>2 Allow <i>server:account:file</i></li> <li>4 Allow <i>PATH:pathname</i></li> </ol> <p>The <a href="#">CONFIG</a> 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.</p>
FIXUSERS	<p>Reserves a range of user numbers for exclusive use of users specifying a fixed user number when logging in using <code>qm -n</code> where <i>n</i> is the required user number.</p> <p>The format of this parameter is</p> <pre>FIXUSERS=u,n</pre> <p>where</p> <ul style="list-style-type: none"> <li>u is the lowest user number in the reserved range.</li> <li>n is the number of user numbers to be reserved.</li> </ul> <p>The highest available QM user number is normally 1023. Therefore, the value of <math>u + n</math> must not exceed 1024.</p> <p>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. This parameter is not applicable to the PDA version of QM.</p>
FLTDIFF *	<p>The FLTDIFF configuration parameter determines how floating point values are compared.</p> <p>Just as some numbers such as one third cannot be represented accurately</p>

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 (0.000000000291) 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.

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. This parameter is not applicable to the PDA version of QM.

#### GDI \*

Setting this parameter non-zero on Windows systems causes the [SETPTR](#) command to use GDI mode by default. This parameter is not applicable to the PDA version of QM.

#### 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 17.899999999999986.

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

```
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 [INTO](#) 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.

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: <ol style="list-style-type: none"> <li>1 Enable before imaging (allows roll back)</li> <li>2 Enable after imaging (allows roll forward)</li> </ol>
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.
LICENCE	Licence parameters. Use the <a href="#">UPDATE.LICENCE</a> command in the QMSYS account to apply new licence parameters.
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 QMBasic <a href="#">SYSTEM(1036)</a> function, allowing applications to monitor the average delay and determine whether a licence with an increased user count is required.
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 <a href="#">SETPTR</a> command or equivalent QMBasic print unit modification functions. This parameter is not applicable to the PDA version of QM.
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 <a href="#">SETPTR</a> command or equivalent QMBasic print unit modification functions. This parameter is not applicable to the PDA version of QM.
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. This parameter is not applicable to the PDA version of QM.
MAXIDLEN	Sets the maximum allowed length of 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 will correct the recorded longest id value if records have been deleted from the file.

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 ( <a href="#">READU</a> ) 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. This parameter is not applicable to the PDA version of QM.
NETFILES	<p>By default QM does not allow access to files on remote drives. This is because the locking system cannot detect that two systems are accessing the file simultaneously. Where it is certain that a file will <u>never</u> 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.</p> <p>Setting NETFILES to 2 enables incoming connections from other QM servers accessing files via the QMNet interface.</p> <p>These two mode settings are additive and can be used together. This parameter is not applicable to the PDA version of QM.</p>
NUMFILES	The maximum number of QM data files that may be opened at one time. This is a system wide limit. 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 <a href="#">LIST.FILES</a> 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 <a href="#">LIST.READU</a> command can be used to determine whether the value of this parameter is appropriate.
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 NT and later), its predecessor, QMSrvr (Windows 98/ME and USB installations) and qmlnxd on other platforms formed from:

- 1 Log client disconnection. There is a very small system overhead for this but it should be negligible. (QMSvc only)
- 2 No longer used.
- 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)

**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.

**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 QMSRVR PORT configuration parameter.

Note that this parameter appears in the QM section of the configuration file as it relates to both QM and QMSvc.

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 98/ME or on the PDA version of QM.

PWDELAY	The pause in seconds between successive attempts to enter the user name and password on network connections. This parameter defaults to 5.
QMCLIENT	<p>Provides additional security control for <a href="#">QMClient</a> sessions. This parameter has one of three values:</p> <ol style="list-style-type: none"><li>0 No restrictions.</li><li>1 Bans use of <a href="#">QMOpen()</a> and <a href="#">QMExecute()</a>, limiting clients to calling subroutines.</li><li>2 In addition to the level 1 restrictions, <a href="#">QMCall()</a> can only be used to call subroutines compiled with the <a href="#">\$QMCALL</a> compiler directive.</li></ol> <p>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. This parameter is not applicable to the PDA version of QM.</p>
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 <a href="#">TRANS()</a> 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.
REPLDIR	Pathname of directory to be used for <a href="#">replication</a> 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 <a href="#">replication</a> 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 <a href="#">replication</a> 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. This parameter is not applicable to the PDA version of QM.

SECURITY

Additive value controlling system wide security options. The only value currently supported is:

- 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.

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/bash -i" on Linux, AIX and Mac, and "/bin/sh -i" on FreeBSD. This parameter is not applicable to the PDA version of QM.

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/bash -c" on Linux, FreeBSD and Mac, and "/bin/sh -c" on FreeBSD. This parameter is not applicable to the PDA version of QM.

SORTMEM \*

The size in kilobytes at which a sort switches from memory based to disk based. This value is per user and defaults to 1024 (1Mb). 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 *	<p>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.</p>
SORTWORK *	<p>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. This parameter is not applicable to the PDA version of QM.</p>
SPOOLER *	<p>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.</p> <p>The qualifying data to this configuration parameter can include other options to be passed to the selected spooler. This parameter is not applicable to the PDA version of QM.</p> <p>See the description of the \$SPOOLER control record in the section on <a href="#">printing</a>.</p>
SRVRLOG	<p>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.</p>
STARTUP	<p>Sets a command to be executed when QM starts. This may not contain double quotes. This parameter is not applicable to the PDA version of QM.</p> <p>See <a href="#">Startup</a> for more details.</p>
TEMPDIR *	<p>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. This parameter is not applicable to the PDA version of QM.</p>
TERMINFO *	<p>The pathname of the directory holding the <a href="#">terminfo</a> database. This defaults to a subdirectory named terminfo under the QMSYS directory. This parameter is not applicable to the PDA version of QM.</p>
TIMEOUT	<p>The maximum wait period in seconds allowed during entry of a valid username and password on QMSvc connections. This parameter defaults to 30.</p>
TIMEZONE *	<p>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</p>

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.
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.
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 [TERMINFO](#) 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 a 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 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 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. For example, the first few lines of the vt100 definition are:

```
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,
```

String tokens may contain the following special character sequences:

<code>\b</code>	Backspace (char 8)
<code>\e</code>	Escape (char 27)
<code>\f</code>	Formfeed (char 12)
<code>\l</code>	Linefeed (char 10)
<code>\n</code>	Linefeed (char 10)
<code>\r</code>	Carriage return (char 13)
<code>\s</code>	Space (char 32)
<code>\t</code>	Tab (char 9)
<code>\^</code>	Caret (^)
<code>\\</code>	Backslash (\)
<code>^x</code>	Ctrl-x (chars 0 - 31)
<code>%x</code>	Parameter action as described below
<code>%%</code>	Percent sign (%)
<code>\$&lt;n&gt;</code>	Insert an <i>n</i> millisecond delay. This code is ignored by QM, removing it from the final string.

The `%x` 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:

<code>%c</code>	pop top stack item and print it as a character (like <code>%c</code> in <code>printf()</code> )
<code>%d</code>	pop top stack item and print it as an integer (like <code>%d</code> in <code>printf()</code> )
<code>%s</code>	pop top stack item and print it as a string (like <code>%s</code> in <code>printf()</code> )
<code>%[:]flags][width[.precision]][doxXs]</code>	as in <code>printf</code> , flags are <code>[-+#]</code> and space. The <code>'</code> is used to avoid making <code>%+</code> or <code>%-</code> patterns (see below).
<code>%p[1-9]</code>	push <i>ith</i> parm
<code>%P[a-z]</code>	set dynamic variable <code>[a-z]</code> from top stack item
<code>%g[a-z]</code>	get dynamic variable <code>[a-z]</code> and push it onto stack
<code>%P[A-Z]</code>	set static variable <code>[A-Z]</code> from top stack item
<code>%g[A-Z]</code>	get static variable <code>[A-Z]</code> 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
%i	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  $x \bmod y$ , 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
%i      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".

## 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.

u0 - u7	@(-100) to @(-107)	Undefined. Users may adopt these for any purpose.
u8	@(-108)	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.
u9	@(-109)	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.

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 standard 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.

## 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:

- 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 is:

```
qmtic {options} -i
```

The PDA version of QM does not support `qmtic` as there is no concept of a command line from which to run it. Instead, there is a QMBasic version of a simple `terminfo` compiler that is run when QM is installed to compile the `terminfo.src` and, if present, the `terminfo.mods` files. This can also be

run at any time by using the TIC command in the QMSYS account.

## 8.3 Windows Code Pages

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 – 255 are mapped to accented letters (such as é and ò), 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, 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 this 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:

- 850 — "Multilingual (Latin-1)" (Western European languages)
- 852 — "Slavic (Latin-2)" (Eastern European languages)
- 855 — Cyrillic
- 857 — Turkish
- 1250 — East European Latin
- 1252 — West European Latin
- 1253 — Greek
- 1257 — Baltic
- 1258 — Vietnamese

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:

```
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 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". We have successfully used a shareware product called FontCreator.
- Find the existing Lucida Console font, which is 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 systems running Windows 98/ME, the underlying operating system does not have adequate user authentication. If security is enabled, users entering QM over a direct network connection will be required to enter a valid username 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 later versions of Windows and on other platforms, users entering QM directly from a network will always be required to enter a valid username and password known to the operating system. Furthermore, if security is enabled, this username must also be in QM's own register of usernames. 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 username used to log into the operating system. If security is not enabled, any user with a valid operating system username 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 <a href="#">CREATE.ACCOUNT</a> , <a href="#">DELETE.ACCOUNT</a> and entering QM in a directory that is not already set up as an account.
Define private servers	Controls use of <a href="#">SET.PRIVATE.SERVER</a> .

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:

<a href="#">@IP.ADDR</a>	User's IP address for network connections. This is also available in QMBasic as <a href="#">SYSTEM(42)</a> .
<a href="#">@LOGNAME</a>	User's login name.
<a href="#">@TTY</a>	Terminal device name.
<a href="#">SYSTEM(1017)</a>	Port number for network connection.
<a href="#">SYSTEM(27 to 30)</a>	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 an 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.

		Developers	Others
\$FORMS	Form queue definitions created with SET.QUEUE for use with SP.ASSIGN.	Full	Full
\$HOLD	Hold file for QMSYS account	None	None
\$HOLD.DIC	Dictionary for \$HOLD	None	None
\$IPC	Inter-process communication file	Full	Full
\$LOGINS	User name database	Full	Full
\$MAP	Catalogue map	Full (note 1)	None
\$MAP.DIC	Dictionary for \$MAP	Read	Read
\$SCREENS	Screens database	Read	Read
\$SVLISTS	\$SAVEDLISTS file	None	None
\$VAULT	Encryption key vault	Read	Read
ACCOUNTS	Accounts database	Read (note 2)	Read (note 2)
ACCOUNTS.DIC	Dictionary for ACCOUNTS	Read	Read
audit.log	Encryption audit log	None	None
bin	Executable files	Read	Read

		Developers	Others
BP	Sample QMBasic items	Read	Read
cat	Private catalogue	None	None
DICTION.DIC	Dictionary for dictionaries	Read	Read
DIR_DICT	Dictionary for directory files	Read	Read
DOCS	Documentation (Windows only)	Read	Read
errlog	Optional error log file	Full (note 3)	Full (note 3)
ERRMSG	Pick style error message file	Read (note 4)	Read (note 4)
ERRMSG.DIC	Dictionary for ERRMSG	Read	Read
gcat	Global catalogue	Full	Read (note 5)
MESSAGES	Message database	Read	Read
NEWVOC	Template VOC file	Read	Read
qmconfig	Configuration parameter file	Read	Read
qmterm.ini	qmterm configuration parameters	Full	Full
QM.VOCLIB	VOC extension	Read	Read
stacks	Command stack repository	None	None
SYSCOM	System include records	Read	None
temp	Temporary directory (Windows only)	Full	Full
terminfo	Terminfo database	Read	Read
terminfo.src	Terminfo definitions	None	None
VOC	Vocabulary file	Read	Read
VOC.DIC	Dictionary for VOC	None	None
errlog	Error log	Full	Full
qm.chm	Help text (Windows only)	Read	Read
QMSvc.log	QMSvc log (Windows only)	None	None

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 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

Although it would be possible to use the [ACCOUNT.SAVE](#) command to create a backup of a QM account, it is recommended that operating system level tools are used for this purpose. The account transfer tools and [T.xxx](#) commands do, however, have the advantage that they can be used to merge data with existing files.

Because individual records in hashed files cannot be accessed from outside of QM, operating system level backup tools can only save and restore an entire file. The [T.DUMP](#) command can take a select list specifying the records to be saved.

### Live Backup

Sometimes there is no alternative to backing up a system while users are logged in. As mentioned above, simply copying the database files while they are being updated will almost certainly lead to inconsistent data, perhaps with structural problems that make restore impossible.

QM provides the ability to suspend database updates by use of `qm -suspend` from the operating

system command prompt. When this mode is in effect, applications will pause at any attempt to modify a file until updates are enabled using `qm -resume`.

It is also possible to prevent users logging in new sessions by use of the [\*\*INHIBIT.LOGIN\*\*](#) command. This does not logout other users who are already active. While login is inhibited, no new QM processes can start with the exception of phantoms started by the user that set the login inhibit.

Rather than pausing the database for the duration of a full backup, various mechanisms are available to minimise the time for which updates are suspended. If using mirrored disks, it would be possible to suspend updates, break the mirror, resume updates, backup the offline half of the mirrored data and then reconnect the mirror to catch up with changes during the backup.

Some environments provide snapshot backup systems where updates can be suspended, the snapshot initiated and updates resumed. The actual backup process performed by the snapshot will handle the complexities of database updates that occur during the backup.

Regardless of the technique used, beware that suspending updates will ensure structural integrity of the saved files but will not ensure business level data integrity as the suspension may occur part way through a series of related updates. Use of transaction programming techniques will help as the suspension will occur on committal of the transaction but it is still possible that a business level transaction is formed from multiple database transactions.

### **Recovery of Individual Records**

Where backup is performed by copying QM data files to an alternative location, a VOC pointer can be created for any file in the backup copy just as though it was part of the local data. This can then be used to recover specific records simply by copying them with the `COPY` command.

## 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, 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 cleanup mechanism will remove the user from the internal table, 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 cleanup 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 text message output in multiple languages. The standard message library installed with QM contains English messages texts. Additional language libraries may be available from Ladybridge Systems or from QM dealers. Users can also perform their own translations of the source text downloaded from the product web site.

The message text source file includes comment lines detailing the rules for successful translation. It is highly recommended that users should use the two letter international country code as the language identifier in the PREFIX line (e.g. FR for French).

Having obtained a suitable set of message texts by download of a pre-translated file or by translation of the master source, this is 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.

To select operation in a particular language, a QM process should execute

```
SET .LANGUAGE language.code
```

where *language.code* is the identifier used in the PREFIX line when the language was loaded. This would typically be included in the LOGIN or MASTER.LOGIN paragraph. If an application attempts to display a message for which there is no version in the selected language, the English version will be used.

## 8.9 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. Although it 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.

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 time zone of the user that started QM, 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.

## 8.10 QM Command Options

The QM executable stored in the bin subdirectory of the QMSYS account has a number of command line options. The command option letters shown below are all case insensitive. Note that to comply with Linux conventions, some of the options have a double hyphen prefix.

- <i>n</i>	Logs the user in as user <i>n</i> where the value of <i>n</i> must be in the range of user numbers reserved with the <a href="#">FIXUSERS</a> 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 <i>n</i> is not within the reserved range, the parameter is ignored.
-A	Causes QM to prompt for the account name on entry.
- <i>Aname</i>	Enters account <i>name</i> 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 <i>n</i>	Kill the QM process for user <i>n</i> .
-K PID <i>n</i>	Kill the QM process for operating system process id <i>n</i> .
-K <i>user</i>	Kill all QM processes for login id <i>user</i> .
-K all	Kill all QM processes.
-L	Apply a new licence.
-U	List current QM users.
-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.
-LOCKS	Show record locks.
-QUIET	Suppresses display of release information, etc on entry to QM. Useful in some scripted sessions.
-RESTART	Restart QM (not Windows).
-RESUME	Resume database updates.
-START	Start QM (not Windows).
-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.
- 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.
- HELP          Display usage help.
- 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. Specifies alternative default location for QMSYS account directory. Also used by [QMConnectLocal\(\)](#) if a non-default location is used.
- TMP             Specifies location of temporary directory if the TEMP configuration parameter is not set.

## 8.11 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 simply was not in the file.

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.

*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 *
```

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.**

Ladybridge Systems 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.

## 8.12 The qmidx Utility

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 one object 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:

<code>qmidx -d <i>data.path</i></code>	Deletes all indices for the named file.
<code>qmidx -m <i>data.path index.path</i></code>	Moves the indices for the named file to the new location specified by <i>index.path</i> . If the index path is omitted, the indices are returned to their default location in the directory identified by <i>data.path</i> .
<code>qmidx -p <i>data.path index.path</i></code>	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. <b>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.</b>
<code>qmidx -q <i>data.path</i></code>	Queries 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.**

## 8.13 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 or program 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.

In addition to use of qmconv, moving from Windows to Linux/FreeBSD/AIX/Mac or vice versa requires that pathnames in the VOC are converted to use the appropriate directory delimiter character. This can be achieved by typing

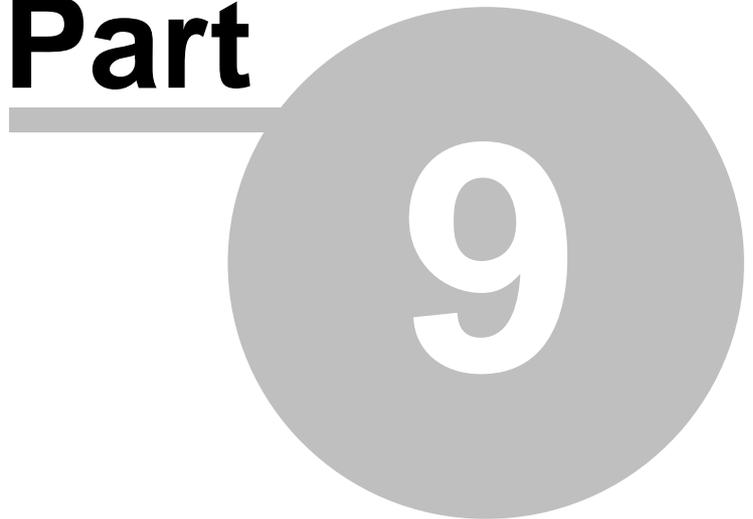
```
*fixpath
```

at the QM command prompt in the relevant account. Alternatively, an account name or pathname may be given

```
*fixpath acc.name
```



**Part**



**System Limits**

## 9 System Limits

Maximum number of users	Determined by the licence
Maximum number of phantoms	Determined by the licence
Maximum hashed file size	16384Gb
Maximum sequential file size	Limited only by the operating system
Maximum records in a file	Limited only by file size
Maximum record key size	63 bytes, configurable to 255 bytes ( <a href="#">MAXIDLEN</a> 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.
Maximum record size	A little under 2Gb or available memory space
Maximum indices per file	32
Maximum AK index key size	255 bytes
Maximum record locks	Set system wide by <a href="#">NUMLOCKS</a> parameter. The upper limit is determined only by available memory.
Maximum number of open files	Set system wide by <a href="#">NUMFILES</a> parameter. The upper limit is determined only by available memory. A single file opened by multiple users counts as one in this calculation.
Maximum program size	8Mb per program module
Maximum character string size	A little under 2Gb or available memory space
Maximum integer value	32 bits (2147483647). The QM run machine will switch to floating point representation for larger values
Maximum floating point value	IEEE 64 bit representation. Upper limit $2^{1024}$ .
Maximum number of public names in a CLASS module	32767
Maximum size of a program for debugging	65535 lines
Maximum pathname length	255 bytes

**Part**

---

**10**

**Glossary of Terms**

## 10 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, <i>menus</i> , <i>paragraphs</i> , etc active in the user's process are discarded and the user returns to the <i>command prompt</i> . The <a href="#">ON.ABORT</a> VOC item can be used to capture this event and take special action.
Account	A collection of database <i>files</i> , 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 <a href="#">ALIAS</a> to make <a href="#">COPY</a> run the <a href="#">COPYP</a> command.
Alternate Key Index	An index structure that allows applications to identify all records that have a particular value for a <i>secondary key</i> item. The index contains a list of <i>primary key</i> values for each secondary key value.
Association	The relationship between two or more <i>multivalued</i> data items where the values belong together. For example, an order processing system might have two associated <i>fields</i> , one holding a list of product codes, the other a list of quantities ordered.
Attribute	An alternative name for a <i>field</i> .
B-tree files	A file structure used to store <i>alternate key indices</i> . B-tree (balanced tree) files use an internal structure that stores data in sorted order and hence allow efficient access to ranges of <i>key</i> values.
Catalog(ue)	A repository for application programs. QM supports three modes of <a href="#">cataloguing</a> ; local, private and global. Local and private cataloguing normally restrict access to the program to the one <i>account</i> . 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 <a href="#">object oriented programming</a> .
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 <i>select list</i> 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. <a href="#">COMO</a> (command output) trapping is enabled/disabled with the <a href="#">COMO</a> 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>I-type</i> dictionary items supported by QM in <a href="#">A and S-type items</a> to simplify migration from other multivalued database products.
Database	A collection of data stored in a form that enables easy access to specific items.
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 <i>files</i> that contains a description of the format of <i>records</i> 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 <i>records</i> . 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 <i>dynamic files</i> as though they were a single file.
Dynamic array	A character <i>string</i> divided into <i>fields</i> , <i>values</i> and <i>subvalues</i> using the mark characters. A database <i>record</i> is stored in this way but dynamic arrays can be used by application developers for other lists of items.
Dynamic file	A type of <i>hashed file</i> that automatically changes its <i>modulus</i> value to react to changes in the volume of data stored.
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 <a href="#">LOGMSG</a> statement.
Field	A column from the tabular representation of a database <i>file</i> .
File	A database <i>table</i> . QM supports <i>hashed files</i> for high performance and <i>directory files</i> for data exchange.
Format code	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.
Group	A portion of a <i>hashed file</i> . The number of groups in a file varies automatically according to the volume of data stored in it.
Group size	The size of a <i>group</i> in multiples of 1024 bytes. QM allows group size values in the range 1 to 8.
Hashed files	High performance files in which the location of a record is calculated by applying the <i>hashing algorithm</i> .
Hashing algorithm	A mathematical calculation applied to the characters of a record <i>key</i> to deduce the <i>group</i> in which that record will be stored.
Hot Spot Monitor	A tool for identifying the number of times each module of an application is executed and the processing time spent in them.

ID	Another name for the <i>primary key</i> of a record.
I-descriptor	Another name for an <i>I-type</i> .
Information style	A reference to the command and programming language style found in the Prime Information multivalued database product and others that adopt this style.
Inline prompt	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.
I-type	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.
Key	A shortened term for the <i>primary key</i> of a record.
Keyword	A word or symbol affecting the behaviour of a command. Keywords correspond to <a href="#">K-type VOC entries</a> .
Laws of Normalisation	A set of rules that govern the construction of relational databases. Multivalued 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.
Link record	A dictionary <a href="#">L-type</a> record that defines the link between two related database files.
Locking	A mechanism used to ensure that two users cannot update the same data item simultaneously, a situation that would usually result in errors.
Mark characters	Characters used within the QM to separate <i>field</i> , <i>value</i> and <i>subvalues</i> within a database <i>record</i> or other <i>dynamic array</i> .
Menu	A <a href="#">M-type</a> VOC record that describes a numbered list of options to be displayed to the user. Each option would have an associated <i>sentence</i> to be executed and, optionally, some help text.
Modulus	The number of <i>groups</i> in a <i>hashed file</i> .
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.
Multivalued	Breaking a simple data item (typically a <i>field</i> ) into multiple instances of the same type of data.
Overflow	QM uses high performance <i>hashed files</i> in which the location of a record can be deduced from its <i>primary key</i> . If there is insufficient space to store the record at its calculated location, the file system extends the group by adding one or more <i>overflow</i> blocks.
Object	A run time instance of a <i>class module</i> .
Paragraph	A <i>VOC record</i> 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 <i>distributed file</i> .
Part number	The unique number used to identify a specific <i>part file</i> in a <i>distributed file</i> .

---

Partitioning algorithm	A dictionary I-type expression used to transform the record id to a <i>part number</i> in a <i>distributed file</i> .
Phantom	A QM process that runs in the background without a terminal. Phantom processes are started with the <a href="#">PHANTOM</a> 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 <i>sentence</i> , excluding the <i>verb</i> . <a href="#">Phrase entries</a> may appear in the <i>VOC</i> or in <i>dictionaries</i> . Phrases are used as short forms in commands, to set defaults for the query processor, and to link <i>fields</i> within an <i>association</i> .
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 <i>string</i> that uniquely identifies a <i>record</i> in a <i>file</i> . QM supports keys from 1 to 255 characters in length but lengths over 63 characters require the <a href="#">MAXIDLEN</a> configuration parameter to be amended.
Proc	The predecessor of <i>paragraphs</i> 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 <i>replication</i> .
QMBasic	The programming language used to develop QM applications.
QMClient	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 <i>files</i> after a system crash.
QMIdx	A tool for updating internal pointers if an <i>alternate key index</i> file is moved.
qmlnxd	A daemon process that runs on non-Windows system to perform background system monitoring tasks.
QMNet	An integrated component of QM that allows access to data stored on another QM server with full support for <i>locking</i> .
Record	An item stored in a <i>database file</i> . 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 <i>fields</i> which may in turn be broken down into <i>values</i> and <i>subvalues</i> .
Relational Database	A style of database that represents data in the form of <i>tables</i> (relations).
Replication	A mechanism whereby updates made to selected files on a <i>publisher</i> system are replicated on one or more <i>subscriber</i> systems.
Secondary key	A data item in a <i>record</i> , or a value calculated from data in the record, that is to be used to construct an <i>alternate key index</i> so that records can be selected based on this value.
Select list	A list of items, usually <i>primary keys</i> , 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 <i>command prompt</i> or it may be stored in the <i>VOC</i> for later execution simply by typing the name of the <i>VOC</i> record.
String	A sequence of characters stored as a data item.
Subvalue	A subdivision of a <i>value</i> 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 <i>association</i> .
Subscriber	The system that is receiving exported file updates in a system configured for data <i>replication</i> .
Table	Another name for a database <i>file</i> .
Terminfo	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 <i>database</i> that must either all happen or none must happen.
Trigger	A user written QMBasic subroutine that is executed whenever selected operations are performed against a <i>file</i> . Triggers get their name from their use to trigger other related updates but they can also be used for data validation.
Value	A subdivision of a <i>field</i> 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.
Verb	The command word in a <i>sentence</i> . This is always the first word and corresponds to a <a href="#">V-type</a> <i>VOC</i> entry.
VOC	A <i>file</i> found in all <i>accounts</i> 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 <i>VOC</i> , it is possible to extend or modify the QM command language.
Vocabulary	See <i>VOC</i> .

# Index

## - ! -

! 507  
 !ABSPATH() 1213  
 !ACCOUNT.RULES() 1214  
 !ATVAR() 1215  
 !ERRTEXT() 1216  
 !GETPU() 1217  
 !LISTU() 1218  
 !PARSER() 1219  
 !PATHTKN() 1221  
 !PCL() 1222  
 !PHLOG() 1225  
 !PICK() 1226  
 !PICKLIST() 1228  
 !QMCLIENT 1229  
 !SCREEN() 1230  
 !SETPU() 1232  
 !SETVAR() 1233  
 !SORT() 1234  
 !USERNAME() 1235  
 !USERNO() 1236  
 !VOCREC() 1237

## - # -

# 642

## - \$ -

\$BASIC.OPTIONS 227  
 \$CATALOG 726  
 \$CATALOGUE 726  
 \$COMMAND.STACK 42  
 \$DEBUG 728  
 \$DEBUG.OPTIONS record 1238  
 \$DEFINE 729  
 \$ECHO 206  
 \$ELSE 731  
 \$ENDIF 731  
 \$EXECUTE 730  
 \$IFDEF 731  
 \$IFNDEF 731  
 \$INCLUDE 733  
 \$INSERT 733  
 \$LIST 734

\$MODE 735  
 \$NO.CATALOG 738  
 \$NO.CATALOGUE 738  
 \$NO.XREF 739  
 \$NOCASE.STRINGS 740  
 \$PRIVATE.CATALOGUE 235, 284, 384  
 \$QMCALL 742  
 \$QUERY.DEFAULTS 546

## - % -

% 657

## - & -

& 574

## - \* -

\* 205

## - @ -

@(x,y) function 762  
 @ABORT.CODE 1207  
 @ABORT.MESSAGE 1207  
 @ACCOUNT 1207  
 @ANS 1207  
 @COMMAND 1208  
 @COMMAND.STACK 1208  
 @CONV 1208  
 @CRTHIGH 1208  
 @CRTWIDE 1208  
 @DATA 1208  
 @DATA.PENDING 1208  
 @DATE 1208  
 @DAY 1208  
 @DICTRECS 1208  
 @DS 1208  
 @FILE.NAME 1208  
 @FILENAME 1208  
 @FMT 1208  
 @GID 1208  
 @HOSTNAME 1208  
 @ID 1208  
 @IP.ADDR 1208  
 @ITYPE.MODE 1209  
 @LEVEL 1209  
 @LOGNAME 1209

@LPTRHIGH	1209	<=	632
@LPTRWIDE	1209	<>	642
@MONTH	1209		
@NB	1209		
@NI	1209	<b>- = -</b>	
@OPTION	1209	=	613
@PARASENTECE	1209	=<	632
@PATH	1209	=>	621
@PIB	1209		
@POB	1209		
@QM.GROUP	1209	<b>- &gt; -</b>	
@QMSYS	1209	>	624
@RECORD	1209	><	642
@SELECTED	1209	>=	621
@SENTENCE	1210		
@SEQNO	1210		
@SIB	1210	<b>- A -</b>	
@SOB	1210		
@STDFIL	711, 1210	ABORT	208, 769
@SYS.BELL	1210	ABORTE	769
@SYSTEM.RETURN.CODE	1210	ABORTM	769
@TERM.TYPE	1210	ABS()	771
@TIME	1210	ABSENT.IGNORE	571
@TRANSACTION.ID	1210	ABSENT.NULL	572
@TRANSACTION.LEVEL	1210	ABSS()	771
@TRIGGER.RETURN.CODE	1210	ACCEPT.SOCKET.CONNECTION()	772
@TTY	1210	ACCOUNT.RESTORE	209
@UID	1210	ACCOUNT.SAVE	211
@USER	1210	Accounts	28
@USER.NO	1211	A-correlative conversion (A)	134
@USER.RETURN.CODE	1211	A-Correlatives	118
@USER0	1211	ACOS()	773
@USER1	1211	ADD.DF	213
@USER2	1211	ADDS()	713
@USER3	1211	ADMIN	214
@USER4	1211	ADMIN.SERVER	216
@USERNO	1211	ADMIN.USER	219
@-Variables	1207	AFTER	624
@VOC	1211	ALIAS	221
@WHO	1211	ALL.MATCH	573
@YEAR	1211	ALPHA()	774
@YEAR4	1211	Alternate key indices	177
		ANALYSE.FILE	222
		ANALYZE.FILE	222
<b>- ~ -</b>		AND	574
~	661	ANDS()	775
		ARG ()	776
<b>- &lt; -</b>		ARG.COUNT()	777
<	636	AS	575
		ASCII()	779
		ASIN()	780

ASSIGNED() 781  
 Assignment statements (QMBasic) 697  
 ASSOC 576  
 ASSOC.WITH 577  
 Associations 130  
 ATAN() 782  
 AUTHENTICATE 224  
 AUTHKEY 224  
 AUTOLOGOUT 226  
 AVERAGE 578  
 AVG 578

## - B -

Backup and restore 1369  
 Base 64 conversion (B64) 136  
 BASIC 227  
 BEFORE 636  
 BEGIN TRANSACTION 783  
 BELL 231  
 BETWEEN 579  
 BINDKEY() 785  
 BITAND() 787  
 BITNOT() 788  
 BITOR() 789  
 BITRESET() 790  
 BITSET() 791  
 BITTEST() 792  
 BITXOR() 793  
 BLOCK.PRINT 232  
 BLOCK.TERM 232  
 Boolean conversion (B) 135  
 BOXED 580  
 BREAK 233, 794  
 BREAK.ON 581  
 BREAK.SUP 584  
 BUILD.INDEX 234  
 BY 587  
 BY.DSND 588  
 BY.EXP 589  
 BY.EXP.DSND 591  
 BY-EXP 589  
 BY-EXP-DSND 591

## - C -

CASE 799  
 CATALOG 235  
 CATALOGUE 235  
 CATALOGUED() 800  
 CATS() 801  
 CD 252  
 CHAIN 802  
 CHANGE() 803  
 CHAR() 804  
 Character conversion (MCx) 150  
 Character values 1205  
 CHECKSUM() 805  
 CHILD() 806  
 CLASS 807  
 CLEAN.ACCOUNT 238  
 CLEANUP configuration parameter 1345  
 CLEAR 809  
 CLEAR.COMMON 810  
 CLEAR.ABORT 239  
 CLEAR.DATA 240  
 CLEAR.FILE 241  
 CLEAR.INPUT 242  
 CLEAR.LOCKS 243  
 CLEAR.PROMPTS 244  
 CLEAR.SELECT 245  
 CLEAR.STACK 246  
 CLEARCOMMON 810  
 CLEARDATA 240, 811  
 CLEARFILE 812  
 CLEARINPUT 242, 813  
 CLEARPROMPTS 244  
 CLEARSELECT 245, 814  
 CLIPORT configuration parameter 1345  
 CLOSE 815  
 CLOSE.SOCKET 816  
 CLOSESEQ 817  
 CLR 247  
 CMDSTACK configuration parameter 1345  
 CNAME 248  
 Code pages 1361  
 CODEPAGE 1361  
 CODEPAGE configuration parameter 1345  
 COL.HDG 595  
 COL.HDG.ID 596  
 COL.HDR.SUPP 597  
 COL.SPACES 598  
 COL.SPCS 598  
 COL.SUP 601  
 COL1() 818  
 COL2() 819  
 COL-HDR-SUPP 597

COL-SUPP 601  
 Command editor 44  
 Command line options 1375  
 Command line parser 1219  
 Command Parsing 40  
 Command scripts 36  
 Command stack 42  
 Comment 205  
 COMMIT 783  
 COMMON 820  
 Common blocks (QMBasic) 690  
 COMO 250  
 COMPARE() 823  
 COMPARES() 823  
 COMPILE.DICT 252  
 Compiler directives (QMBasic) 725  
 Concatentation conversion (C) 137  
 CONFIG 253  
 CONFIG() 825  
 Configuration parameters 1345  
 CONFIGURE.FILE 254  
 CONNECT.PORT() 826  
 Constants (QMBasic) 687  
 CONTINUE 827  
 CONV 599  
 Conversion codes 133  
 CONVERT 828  
 CONVERT() 828  
 COPY 256  
 COPY.LIST 258  
 COPY.LISTP 260  
 COPYP 262  
 Correlatives 118  
 COS() 830  
 COUNT 562  
 COUNT() 831  
 COUNT.SUP 600  
 COUNTS() 831  
 CREATE 832  
 CREATE.ACCOUNT 264  
 CREATE.FILE 266, 833  
 CREATE.INDEX 270  
 CREATE.KEY 272  
 CREATE.SERVER.SOCKET 834  
 CREATE.USER 274  
 Creating and deleting files 99  
 Creating and modifying data 114  
 CROP() 836  
 CRT 856  
 CS 247  
 CSV 602

CSV.MODE 838  
 CSVDQ() 837  
 CT 275  
 CUMULATIVE 604  
 CURRENCY 395

## - D -

DATA 277, 840  
 Data Encryption 185  
 Data field definitions 51  
 Data types (QMBasic) 700  
 DATE 278  
 Date conversion (D) 138  
 DATE() 841  
 DATE.FORMAT 279  
 DBL.SPC 605  
 DBL-SPC 605  
 DCOUNT() 842  
 DEADLOCK configuration parameter 1345  
 DEBUG 280, 843  
 Debugger 1238  
 DECRYPT() 844  
 DEFAULT 606  
 Default file variable 711  
 DEFFUN 845  
 Deinstallation 27  
 DEL 847  
 DELETE 281, 848  
 DELETE() 847  
 DELETE.ACCOUNT 283  
 DELETE.CATALOG 284  
 DELETE.CATALOGUE 284  
 DELETE.COMMON 285  
 DELETE.DEMO 286  
 DELETE.FILE 287  
 DELETE.INDEX 289  
 DELETE.KEY 290  
 DELETE.LIST 291  
 DELETE.PRIVATE.SERVER 292  
 DELETE.SERVER 110, 292  
 DELETE.USER 293  
 DELETelist 849  
 DELETESEQ 850  
 DELETEU 848  
 DELIMITER 607  
 DET.SUP 609  
 Device Licensing 31  
 DFPART() 851  
 DHCACHE configuration parameter 1346

Dictionaries 115  
 Dictionary A and S-type records 116  
 Dictionary C-type records 124  
 Dictionary D-type records 125  
 Dictionary I-type records 126  
 Dictionary L-type records 127  
 Dictionary PH-type records 128  
 Dictionary X-type records 129  
 DIM 853  
 DIMENSION 853  
 DIR() 858  
 Directory files 101  
 DISABLE.INDEX 294  
 DISABLE.PUBLISHING 295  
 DISINHERIT 855  
 DISPLAY 296, 856  
 Display clause 551  
 DISPLAY.LIKE 611  
 DISPLAY.NAME 595  
 Distributed files 106  
 DIV() 859  
 DIVS() 713  
 DOWNCASE() 860  
 DPARSE 861  
 DPARSE.CSV 861  
 DQUOTE() 1058  
 DTX() 863  
 DUMP 297  
 DUMPPDIR configuration parameter 1346  
 Dynamic files 103

## - E -

EBCDIC() 864  
 ECHO 299, 865  
 ED 300  
 EDIT.CONFIG 311  
 EDIT.LIST 313  
 ENABLE.PUBLISHING 314  
 ENCRYPT() 866  
 ENCRYPT.FILE 315  
 END 867  
 ENTER 795  
 Entering QM 31  
 ENUM 612  
 ENUMERATE 612  
 ENV() 868  
 Environment variables 1375  
 Epoch conversion (E) 142  
 EPOCH() 869  
 EQ 613  
 EQS() 870  
 EQU 871  
 EQUAL 613  
 EQUATE 871  
 ERRLOG configuration parameter 1346  
 ERRMSG 874  
 Error Logging 1374  
 Error numbers 1245  
 EVAL 614  
 EVALUATE 614  
 EXCLREM configuration parameter 1346  
 EXECUTE 875  
 EXIT 877  
 EXP() 878  
 Expressions (QMBasic) 693  
 EXTRACT() 879

## - F -

FCONTROL() 880  
 F-correlative conversion (F) 146  
 F-Correlatives 120  
 Field extraction 160  
 FIELD() 881  
 FIELDS() 881  
 FIELDSTORE() 883  
 FILE 885  
 File definitions 52  
 File translation conversion 164  
 FILE.EVENT() 887  
 FILE.SAVE 316  
 FILE.STAT 318  
 FILEINFO() 889  
 FILELOCK 892  
 FILERULE configuration parameter 1346  
 Files 98  
 FILEUNLOCK 894  
 FIND 895  
 FIND.ACCOUNT 320  
 FIND.PROGRAM 321  
 FINDSTR 897  
 FIRST 662  
 FIXUSERS configuration parameter 1346  
 FLTDIFF configuration parameter 1346  
 FLUSH 899  
 FLUSH.DH.CACHE 900  
 FMT 615  
 FMT() 901  
 FMTS() 901

FOLD() 902  
 FOLDS() 902  
 FOOTER 616  
 FOOTING 616, 903  
 FOR 618, 905  
 FORCE 619  
 FORM.LIST 325  
 FORMAT 322  
 Format specifications 168  
 FORMCSV() 908  
 FORMLIST 909  
 F-pointers 52  
 FROM 620  
 FSTAT 326  
 FSYNC configuration parameter 1347  
 FUNCTION 910

## - G -

GDI configuration parameter 1347  
 GE 621  
 GENERATE 328  
 GES() 912  
 GET 1054  
 GET(ARG.) 913  
 GET.LIST 330  
 GET.MESSAGES() 915  
 GET.PORT.PARAMS() 916  
 GET.STACK 331  
 GETLIST 917  
 GETNLS() 918  
 GETPU() 919  
 GETREM() 921  
 Glossary of terms 1384  
 GO 332  
 GO TO 924  
 GOSUB 922  
 GOTO 924  
 GRAND.TOTAL 622  
 GRANT.KEY 333  
 GREATER 624  
 Group conversion (G) 147  
 GROUP.SIZE 254, 266  
 GRPSIZE configuration parameter 1347  
 GT 624  
 GTS() 925

## - H -

HDR.SUP 625

HDR-SUPP 625  
 HEADER 626  
 HEADING 626, 926  
 HELP 334  
 HSM 335  
 HUSH 336, 928

## - I -

ICONV() 929  
 ICONVS() 929  
 ID.ONLY 628  
 ID.SUP 629  
 IDIV() 930  
 ID-SUPP 629  
 IF 337, 931  
 IFS() 932  
 IN 630, 933  
 INCLUDE 733  
 INDEX() 934  
 INDEXS() 934  
 INDICES() 935  
 INHERIT 937  
 INHIBIT.LOGIN 339  
 Inline prompts 79  
 INMAT() 938  
 INPUT 939  
 INPUT @ 941  
 INPUTCLEAR 813  
 INPUTCSV 945  
 INPUTERR 1047  
 INPUTFIELD 946  
 INS 949  
 INSERT() 949  
 Installation 15  
 INT() 951  
 Integer conversion (IS, IL) 148  
 Interrupting commands 46  
 INTPREC configuration parameter 1347  
 Introduction 8  
 I-type expressions 131  
 ITYPE() 952

## - J -

JNLDIR configuration parameter 1348  
 JNLMODE configuration parameter 1348  
 JNLSize configuration parameter 1348

**- K -**

KEYCODE() 953  
KEYEDIT 955  
KEYEXIT 957  
KEYIN() 958  
KEYINC() 958  
KEYINR() 958  
KEYREADY() 959  
KEYTRAP 960  
Keywords 54  
Keywords (query processor) 567

**- L -**

LABEL 631  
Labels (QMBasic) 692  
LARGE.RECORD 254, 266  
LE 632  
LEN() 961  
Length conversion (L) 149  
LENS() 961  
LES() 962  
LESS 636  
LGNWAIT configuration parameter 1348  
LICENCE configuration parameter 1348  
Licensing 15  
LIKE 633  
Limits 1382  
Limits (QMBasic) 744  
LIST 556  
LIST.COMMON 340  
LIST.DF 341  
LIST.DIFF 342  
LIST.FILES 343  
LIST.INDEX 345  
LIST.INTER 347  
LIST.ITEM 558  
LIST.KEYS 348  
LIST.LABEL 559  
LIST.LOCKS 350  
LIST.PHANTOMS 351  
LIST.READU 352  
LIST.REPLICATED 354  
LIST.SERVERS 110, 355  
LIST.TRIGGERS 356  
LIST.UNION 357  
LIST.USERS 358  
LIST.VARS 359  
LISTDICT 360  
LISTF 361  
LISTFL 362  
LISTFR 363  
LISTINDEX() 963  
LISTK 364  
LISTM 365  
LISTPA 366  
LISTPH 367  
LISTPQ 368  
LISTQ 369  
LISTR 370  
LISTS 371  
LISTU 372  
LISTV 374  
LN() 964  
LOAD.LANGUAGE 1373  
LOCAL 965  
LOCATE 967  
LOCATE() 967  
LOCK 375, 971  
LOCKING 634  
Locks 172  
Logging in 31  
LOGIN paragraph 36  
Login process 34  
LOGIN VOC entry 34  
LOGIN.PORT 377  
LOGMSG 378, 973  
LOGOUT 379  
LOGTO 380  
LOOP 381, 974  
LOOP / REPEAT 381  
LOWER() 975  
LPTR 635  
LPTRHIGH configuration parameter 1348  
LPTRWIDE configuration parameter 1348  
LT 636  
LTS() 976

**- M -**

MAKE.INDEX 382  
MAP 384  
MARGIN 637  
MARK.MAPPING 977  
Masked decimal conversion (MD, MR) 151  
MASTER.LOGIN 34  
MASTER.LOGIN paragraph 36  
MAT 978

MATBUILD 980  
 MATCH (QMBasic) 693  
 MATCHES (QMBasic) 693  
 MATCHES (Query processor) 633  
 MATCHESS 981  
 MATCHFIELD 983  
 MATCHING (Query processor) 633  
 Matching templates 85  
 MATPARSE 985  
 MATREAD 987  
 MATREADCSV 989  
 MATREADL 987  
 MATREADU 987  
 Matrix file i/o 706  
 MATWRITE 991  
 MATWRITEU 991  
 MAX 638  
 MAX() 993  
 MAXCALL configuration parameter 1348  
 MAXIDLEN configuration parameter 1348  
 MAXIMUM() 994  
 MD5() 995  
 MED 385  
 Menu definitions 55  
 MERGE.LIST 388  
 MERGE.LOAD 254, 266  
 MESSAGE 390  
 MIN 639  
 MIN() 996  
 MINIMUM() 997  
 MINIMUM.MODULUS 254, 266  
 MOD() 998  
 MODIFY 391  
 MODS() 998  
 Monitoring the system 1371  
 Mouse input 953  
 MULS() 713  
 MULTI.VALUE 641  
 Multifiles 266  
 Multivalue functions 713  
 MULTIVALUED 641  
 MUSTLOCK configuration parameter 1349  
 Mutli-valued database 10  
 MVDATE() 1000  
 MVDATE.TIME() 1001  
 MVEPOCH() 1002  
 MVTIME() 1003

## - N -

NAP 1004  
 National language support 395  
 NE 642  
 NEG() 1005  
 NES() 1006  
 NETFILES configuration parameter 1349  
 NETWAIT 394  
 Network file access 110  
 NEW.PAGE 643  
 NEXT 905  
 NLS 395  
 NO 644  
 NO.CASE 645  
 NO.GRAND.TOTAL 646  
 NO.INDEX 647  
 NO.MATCH 648  
 NO.NULLS 649  
 NO.PAGE 650  
 NO.SPLIT 651  
 NOBUF 1007  
 Non-English messages 1373  
 NOPAGE 650  
 NOT 642  
 NOT() 1008  
 NOT.MATCHING 673  
 NSELECT 396  
 NULL 1009  
 NUM() 1010  
 NUMFILES configuration parameter 1349  
 NUMLOCKS configuration parameter 1349

## - O -

Object orientated programming 716  
 OBJECT() 1011  
 OBJETCS configuration parameter 1349  
 OBJINFO() 1012  
 OBJMEM configuration parameter 1349  
 OCONV() 1013  
 OCONVS() 1013  
 OFF 418  
 ON GOSUB 1015  
 ON GOTO 1016  
 ON.ABORT paragraph 36  
 ON.EXIT paragraph 36  
 ON.LOGTO paragraph 36  
 ONLY 628

OPEN 1017  
 OPEN.SOCKET() 1023  
 OPENPATH 1019  
 OPENSEQ 1021  
 Operators (QMBasic) 693  
 OPTION 397  
 OPTIONS configuration parameter 1349  
 OR 653  
 ORS() 1025  
 OS.ERROR() 1026  
 OS.EXECUTE 1027  
 OSDELETE 1028  
 OSREAD 1029  
 OSWRITE 1031  
 OUTERJOIN() 1032  
 OVERLAY 654

## - P -

PAGE 1033  
 PAGESEQ 655  
 Pagination 47  
 PAN 656  
 Paragraphs 57  
 PASSWORD 402  
 PATHNAME 254, 266  
 Pattern matching 85  
 Pattern matching conversion (P) 161  
 PAUSE 403, 1034  
 PCT 657  
 PDEBUG 404  
 PDUMP 405  
 PDUMP configuration parameter 1350  
 PERCENT 657  
 PERCENTAGE 657  
 PERFORM 875  
 Permissions 1366  
 PHANTOM 406  
 PHANTOMS configuration parameter 1350  
 Phrase definitions 59  
 PORT configuration parameter 1350  
 PORTMAP configuration parameter 1350  
 PRECISION 1035  
 PRINT 1036  
 PRINTCSV 1037  
 PRINTER 408  
 PRINTER CLOSE 1039  
 PRINTER DISPLAY 1040  
 PRINTER FILE 1041  
 PRINTER NAME 1042  
 PRINTER OFF 1038  
 PRINTER ON 1038  
 PRINTER RESET 1043  
 PRINTER SETTING 1044  
 PRINTER.SETTING() 1045  
 PRINTERR 1047  
 Printing 87  
 PRIVATE 1048  
 Private vocabulary 48  
 Process dump files 1243  
 PROCREAD 1049  
 PROCs 60  
 PROCWRITE 1050  
 PROGRAM 1051  
 PROMPT 1052  
 PSTAT 410  
 PTERM 412  
 PTERM() 1053  
 PUBLIC 1054  
 PUBLISH 414  
 PUBLISH.ACCOUNT 415  
 PUT 1054  
 PVOC 48  
 PWDELAY configuration parameter 1351  
 PWR() 1057

## - Q -

QM command options 1375  
 QM Commands 200  
 QMBasic 684  
 QMBasic debugger 1238  
 QMBasic functions 745, 754  
 QMBasic limits 744  
 QMBasic overview 685  
 QMBasic statements 745, 754  
 QMCall 1267  
 QMCallx 1269  
 QMChange() 1271  
 QMChecksum() 1272  
 QMClearselect 1273  
 QMClient 1260  
 QMCLIENT configuration parameter 1351  
 QMClient security 1266  
 QMClose 1275  
 QMConnect() 1276  
 QMConnected() 1278  
 QMConnectionType 1279  
 QMConnectLocal() 1280  
 qmconv 1379

QMDcount() 1282  
 QMDDecrypt() 1283  
 QMDel() 1284  
 QMDelete() 1285  
 QMDeleteu() 1286  
 QMDisconnect 1287  
 QMDisconnectAll 1288  
 QMEncrypt() 1289  
 QMEndCommand 1290  
 QMError() 1291  
 QMExecute() 1292  
 QMExtract() 1294  
 QMField() 1295  
 qmfix 1377  
 QMFree() 1296  
 QMGetSession() 1298  
 QMGetVar() 1299  
 qmidx 1378  
 QMIns() 1300  
 QMLocate() 1302  
 QMLogto() 1304  
 QMMarkMapping 1305  
 QMMatch() 1306  
 QMMatchfield() 1308  
 QMNet 110  
 QMOpen() 1311  
 QMRead() 1312  
 QMReadl() 1314  
 QMReadList() 1316  
 QMReadNext() 1317  
 QMReadu() 1319  
 QMRecordlock 1321  
 QMRecordlocked() 1322  
 QMRelease 1323  
 QMReplace() 1324  
 QMRespond() 1326  
 QMRevision() 1327  
 QMSelect 1328  
 QMSelectIndex 1330  
 QMSelectLeft() 1331  
 QMSelectPartial 1332  
 QMSelectRight() 1331  
 QMSetLeft 1334  
 QMSetRight 1334  
 QMSetSession() 1335  
 QMStatus() 1336  
 qmtic 1359  
 QMTrapCallAbort 1337  
 QMWrite 1338  
 QMWriteu 1340  
 Q-pointers 73

QSELECT 417  
 Query processing 546  
 QUIT 418  
 QUOTE() 1058

## - R -

Radix conversions (MB, MX) 158  
 Radix conversions (MCDX, MCXD) 159  
 RAISE() 1059  
 RANDOMIZE 1060  
 Range check conversion (R) 162  
 RDIV() 1061  
 READ 1062  
 READ.SOCKET() 1063  
 READBLK 1064  
 READCSV 1066  
 READL 1068  
 READLIST 1070  
 READNEXT 1071  
 READSEQ 1073  
 READU 1074  
 READV 1076  
 READVL 1078  
 READVU 1078  
 REBUILD.ALL.INDICES 419  
 RECCACHE configuration parameter 1351  
 RECORDLOCKED() 1080  
 RECORDLOCKL 1081  
 RECORDLOCKU 1081  
 REDO 420  
 REFORMAT 561  
 RELEASE 421, 1082  
 REM() 1083  
 REMARK 1085  
 Remote file pointers 73  
 Remote pointers 74  
 REMOVE 1086  
 REMOVE() 1086  
 REMOVE.BREAK.HANDLER 1089  
 REMOVE.DF 422  
 REMOVEF() 1090  
 RENAME 248  
 REPEAT 381, 974  
 REPEATING 660  
 REPLACE() 1092  
 REPLDIR configuration parameter 1351  
 Replication 189  
 REPLMAX configuration parameter 1351  
 REPLPORT configuration parameter 1351

- REPLSIZE configuration parameter 1351  
 REPLSRVR configuration parameter 1351  
 REPORT.SRC 423  
 REPORT.STYLE 424  
 REQUIRE.INDEX 658  
 REQUIRE.SELECT 659  
 RESET.MASTER.KEY 425  
 RESTORE.ACCOUNTS 426  
 RESTORE.SCREEN 1094  
 RETRIES configuration parameter 1351  
 RETURN 1095  
 RETURN FROM PROGRAM 1095  
 RETURN TO 1095  
 REUSE() 1097  
 REVOKE.KEY 427  
 RINGWAIT configuration parameter 1351  
 RND() 1099  
 ROLLBACK 783  
 ROUNDDOWN() 1100  
 ROUNDUP() 1100  
 RQM 1132  
 RTRANS() 1168  
 RUN 428
- S -**
- SAFEDIR configuration parameter 1352  
 SAID 661  
 SAMPLE 662  
 SAMPLED 663  
 SAVE.LIST 429  
 SAVE.SCREEN() 1102  
 SAVE.STACK 430  
 SAVELIST 1103  
 SAVING 664  
 SCRB 431  
 SCROLL 665  
 SEARCH 555  
 SECURITY 440  
 Security - application level 1363  
 Security - permissions 1366  
 SECURITY configuration parameter 1352  
 Security subroutines 78  
 SED 134, 441  
 SED extension programming 468  
 SEEK 1104  
 SEL.RESTORE 486  
 SELECT 553, 1105  
 Select lists 175  
 Select lists in QMClient sessions 1265  
 SELECT.SOCKET() 1107  
 SELECTE 1109  
 SELECTINDEX 487, 1110  
 SELECTINFO() 1112  
 Selection clause 549  
 SELECTLEFT 1113  
 SELECTN 1105  
 SELECTRIGHT 1113  
 SELECTV 1105  
 Self-installing applications 1252  
 SENTENCE() 1115  
 Sentences 75  
 SEQ() 1116  
 Sequential file i/o 708  
 SERIAL configuration parameter 1352  
 Serial port connection 31  
 SERVER.ADDR() 1117  
 SET 488  
 SET.ARG 1118  
 SET.BREAK.HANDLER 1119  
 SET.DEVICE 490  
 SET.ENCRYPTION.KEY.NAME 492  
 SET.EXIT.STATUS 493, 1120  
 SET.FILE 494  
 SET.LANGUAGE 1373  
 SET.PORT.PARAMS() 1122  
 SET.PRIVATE.SERVER 497  
 SET.QUEUE 495  
 SET.SERVER 110, 497  
 SET.SOCKET.MODE() 1123  
 SET.TRIGGER 499  
 SETLEFT 1121  
 SETNLS 1125  
 SETPORT 500  
 SETPTR 501  
 SETPU 1127  
 SETREM 1129  
 SETRIGHT 1121  
 SETUP.DEMO 506  
 SH 507  
 SH configuration parameter 1352  
 SH1 configuration parameter 1352  
 SHIFT() 1130  
 SHOW 564  
 SIN() 1131  
 SINGLE.VALUE 666  
 SINGLEVALUED 666  
 SLEEP 508, 1132  
 SOCKET.INFO() 1133  
 Sockets 722  
 SORT 556

Sort clause 550  
 SORT.COMPARE() 1134  
 SORT.ITEM 558  
 SORT.LABEL 559  
 SORT.LIST 509  
 SORTMEM configuration parameter 1352  
 SORTMRG configuration parameter 1353  
 SORTWORK configuration parameter 1353  
 SOUNDEX() 1136  
 SP.ASSIGN 510  
 SP.CLOSE 512  
 SP.OPEN 512  
 SP.VIEW 513  
 SPACE() 1137  
 SPACES() 1137  
 SPLICE() 1138  
 SPLIT.LOAD 254, 266  
 SPOKEN 661  
 SPOOL 515  
 SPOOLER configuration parameter 1353  
 SQRT() 1139  
 SQUOTE() 1140  
 SREFORMAT 561  
 SRVRLOG configuration parameter 1353  
 SSELECT 553, 1141  
 SSELECTN 1141  
 SSELECTV 1141  
 Standard subroutines 1212  
 Startup and shutdown 25  
 STARTUP configuration parameter 1353  
 STATUS 516, 1144  
 STATUS() 1143  
 STOP 517, 1145  
 STOPE 1145  
 STOPM 1145  
 STR() 1147  
 STRINGS 667  
 STRS() 1147  
 STYLE 668  
 Stylus input (PDA) 953  
 SUBR() 1148  
 SUBROUTINE 1150  
 SUBS() 713  
 SUBSCRIBE 518  
 SUBSTITUTE() 1152  
 Substitution conversion (S) 163  
 SUBSTRINGS() 1153  
 SUM 563  
 SUM() 1154  
 SUMMATION() 1155  
 SUPP 625

SWAP() 803  
 SWAPCASE() 1156  
 System Administration 1344  
 System limits 1382  
 SYSTEM() 1157

## - T -

T.ATT 490  
 T.DET 522  
 T.DUMP 519  
 T.EOD 522  
 T.FWD 522  
 T.LOAD 521  
 T.RDLBL 522  
 T.READ 522  
 T.REW 522  
 T.STAT 522  
 T.WEOF 522  
 TAN() 1160  
 TCLREAD 1161  
 TEMPDIR configuration parameter 1353  
 TERM 523  
 Terminal configuration 1355  
 Terminfo compiler 1359  
 TERMINFO configuration parameter 1353  
 Terminfo database 1355  
 TERMINFO() 1162  
 TESTLOCK() 1163  
 Text substring conversion 165  
 TIME 525  
 Time conversion (MT) 156  
 TIME() 1164  
 TIMEDATE() 1165  
 TIMEOUT 1166  
 TIMEOUT configuration parameter 1353  
 TIMEZONE configuration parameter 1353  
 TO (redirect delimited report) 602, 607  
 TO (REFORMAT) 671  
 TO (Selection verbs) 670  
 TOTAL 672  
 TOTAL() 1167  
 TRANS() 1168  
 TRANSACTION ABORT 1170  
 TRANSACTION COMMIT 1170  
 TRANSACTION START 1170  
 Transactions 188  
 Translation conversion 164  
 Triggers 181  
 TRIM() 1172

TRIMB() 1174  
 TRIMBS() 1174  
 TRIMF() 1175  
 TRIMFS() 1175  
 TRIMS() 1176  
 TTYGET() 1177  
 TTYSET 1178  
 TXCHAR configuration parameter 1354  
 Type conversion (QMBasic) 700

## - U -

UMASK 526  
 UMASK configuration parameter 1354  
 UNASSIGNED() 1179  
 UNIQUE 664  
 UNLIKE 673  
 UNLOCK 527, 1180  
 UNLOCK.KEY.VAULT 528  
 UNTIL 1181  
 UPCASE() 1183  
 UPDATE.ACCOUNT 529  
 UPDATE.LICENCE 530  
 UPDATE.RECORD 531  
 UPDATE.RECORD batch mode 534  
 UPDATE.RECORD visual mode 537  
 User defined conversions 166  
 User mode installation 22  
 USERPOOL configuration parameter 1354  
 USING 674  
 Using socket connections 722

## - V -

Variable names (QMBasic) 687  
 Variables (QMBasic) 689  
 VERSION 254, 266  
 VERT 675  
 VERTICALLY 675  
 VFS 112  
 Virtual File System 112  
 VOC D-type records 51  
 VOC file 40, 48  
 VOC F-type records 52  
 VOC K-type records 54  
 VOC M-type records 55  
 VOC PA-type records 57  
 VOC PH type records 59  
 VOC PQ type records 60  
 VOC Q-type records 73

VOC R-type records 74  
 VOC S-type records 75  
 VOC verb definitions 76  
 VOC V-type records 76  
 VOC X-type records 77  
 VOCPATH() 1184  
 VOID 1186  
 VSLICE() 1187

## - W -

WAKE 1191  
 Web server 1255  
 WEOFSEQ 1192  
 WHEN 677  
 WHERE 544  
 WHILE 1193  
 WHO 543  
 WITH 679  
 WITHOUT 682  
 WRITE 1195  
 WRITE.SOCKET() 1196  
 WRITEBLK 1198  
 WRITECSV 1199  
 WRITESEQ 1201  
 WRITESEQF 1201  
 WRITEU 1195  
 WRITEV 1203  
 WRITEVU 1203

## - X -

XLATE() 1168  
 XTD() 1204

## - Y -

YEARBASE configuration parameter 1354