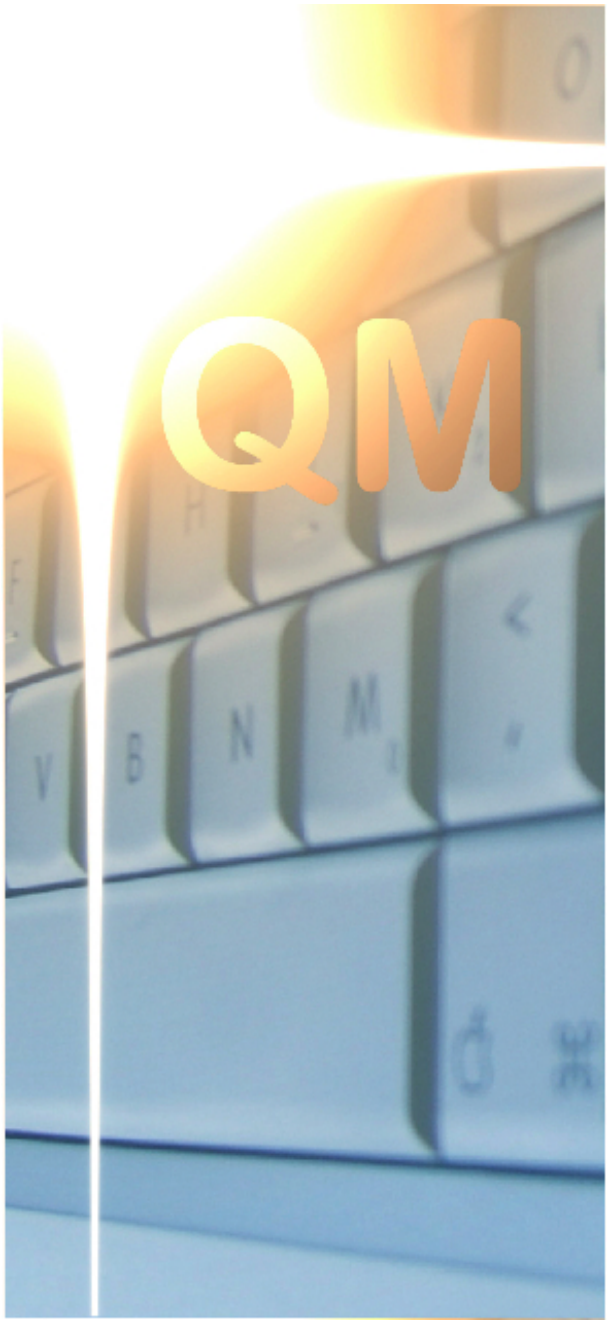


OpenQM

3.4-13

Teach Yourself OpenQM

Teach Yourself OpenQM

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

1 Introduction

This self-study course teaches you about the OpenQM database product. Although you may have experience of other database systems, this course assumes no prior knowledge. The material is designed to be equally useful to those involved with QM in their employment, leisure activities or in educational environments.

If you are a beginner, we recommend that you work through this course strictly in sequence as each stage may rely on knowledge acquired in earlier stages. If you are experienced, you may wish to skip some sections but try to take care not to skip things that you have not met before.

There are extensive exercises that cover the key areas of the QM environment including the command processor, database files and dictionaries, query processing and programming. All of these can be performed using the free Personal Version of QM, an evaluation licence, or a fully licensed commercial installation. We strongly recommend that you tackle all of the exercises and the examples included in the text. The more you experiment with the product, the more you will learn about it. In particular, note that there are some exercises that form a sequence through the sections that cover dictionaries and the query processor, each depending on the earlier ones. Similarly, the programming section builds an application step by step over the entire set of modules.

The material presented here has been assembled from modules that span ten days when delivered as a trainer led course. Do not expect to complete it in an afternoon! Instead, take your time and ensure that you fully understand each step before moving on.

As you work through this material, you may wish to consult the full *QM Reference Manual* or the help system (they contain the same text) for more detailed descriptions of specific syntax elements. In the longer term, you may find it useful to keep a printed copy of the *QM Quick Reference Guide* to hand. And, once you have mastered all that is covered in this course, there is much more to discover for yourself via the user documentation.

Document Conventions

All QM documentation uses a simple set of conventions in descriptions of commands or language elements. For example

DELETE.FILE {**DATA** | **DICT**} *file.name* {**FORCE**}

Items in bold type (**DELETE.FILE**) are words 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.

Unless explicitly stated, items enclosed in curly brackets (e.g. {**FORCE**}) are optional parts of a command or statement. The curly brackets are not part of the text and should not be typed.

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

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

The mark characters introduced in the description of the data model are represented by IM, FM, VM, SM and TM.

What is a Database?

There are many definitions but for our purposes let's say that a database is a collection of data organised in a manner that allows retrieval of specific items in an efficient manner. Ignoring computers for a while, imagine trying to find the telephone number of a friend in the phone book if it was not sorted into order. You would need to read through the directory until you found the entry you were looking for. This might be near the beginning; it might be near the end. On average you would need to read half of the directory. This would be totally useless for any realistic purpose.

Of course, the phone book is not constructed in this way. Instead, the entries are sorted into alphabetical order and each page has a heading to tell you what entries are on the page. You therefore only need to find the right page and then scan through the items on that page. The phone book perhaps fits our definition of a database as it is organised in a way that allows you to find items quickly.

It is interesting to look at what you do subconsciously when looking for a phone number. If you have a phone book to hand, read no further than the end of this sentence and then go to look up the first entry for "A Thomas".

What did you do? Well, hopefully, you did not start at page 1 and read through page by page until you found what you where looking for. You probably dived in part way through, looked to see what was on that page and then decided whether to work forwards or backwards. You probably also applied a little prior knowledge of the distribution of names, starting about two thirds the way through the directory and then moving forwards or backwards several pages at a time based on how far your were from your target entry.

This sort of searching process is typical of the way in which computers find things in large databases. The example above is closely related to a simple technique known as a "binary search" that can find an item in sorted lists with only minimal examination of the data. In fact, it is possible to find any entry in a list of a million items by examining at most just twenty of them. If you still have the phone book to hand, try the following method to find A Thomas again:

Open the phone book at (roughly) the centre page. If this is the page we want, well done, that was too easy! Assuming that things did not fall out that quickly, decide which half of the book contains the entry you want. Now turn to the page in the centre of that half of the book. Keep doing this, halving the size of the section of interest until you arrive on the right page.

The binary search described above is just one of a range of techniques used by database software to find information. We will meet others as we discuss the OpenQM product. The important thing at this stage is simply that it is an example of what we mean when we define a database as being organised in a manner that allows efficient data retrieval.

Relational Databases

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. (The term *relational* infers simply that the data in a table is related in some way).

Throughout this material, we will base our examples on a simple order processing system. This is a concept that is easily grasped whether or not you are involved in sales. The system that we construct will grow in complexity as we progress but for now we only 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	2
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.

Designing efficient databases requires some thought about what to store and, once we get to multiple tables, how to organise it. Much academic work has been done on the underlying theory. Relational databases are built following a set of rules known as the **Rules 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. We are not going to study these rules in detail here. If you are interested, there are many computing textbooks that cover this material.

The First Rule 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	2		
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 Rules of Normalisation do not allow this, mostly based on the way in which the data might be stored by the computer system. This is especially true for the programming languages that were most widely used for business applications when

the relational model was designed. If we are to observe the First Rule 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	2	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 Rules 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 data 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	2
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!

Multivalue Databases

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

Order no	Date	Customer	Product	Quantity

1001	12 Jan 05	1728	107	2
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 multivalue view of the world produces applications that have fewer tables compared to its fully normalised counterpart, often very many fewer. As a result of this, it is typically much quicker (and hence cheaper) to implement and faster to execute. It is also much easier to modify as new features are added to the data. Having said this, there will always be situations where this model is not ideal. In such cases, you can freely revert to using the fully normalised approach because 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, as we have seen, the multivalue model usually results in far fewer tables than in other data models. Tables are frequently referred to as **files**, the two terms are interchangeable. The rows of our table are known as **records** and the columns as **fields** (some systems 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.

The multivalue data model allows us to go one step further; values may be subdivided into **subvalues**. Perhaps every time that we sell an item we need to note its serial number. Thus each of the parts in our example above would have a list of serial numbers associated with it. Order 1002 in the example above might become

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

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.

It is useful to understand how the multi-valued data is represented inside QM. A database record in memory or within the file is stored as a sequence of characters known as a **dynamic array**. The boundaries between the fields are marked by a special character called a **field mark**. The boundaries between values within a multi-valued field are marked by **value marks** and the boundaries between subvalues are marked using **subvalue marks**. In our printed representation of a dynamic array, we use FM, VM and SM to represent the mark characters.

Thus the data for order number 1002 above would be represented as

13527FM3194FM318VM452FM2VM3FM21222SM21223VM41272SM41723SM41728

There are several important things to notice about this example. Firstly, the order number is not in the data. Every record must have a unique value by which it is identified in the table. This **record id** or **primary key** may be any sequence of characters and can be thought of as being the name of the record. Although it must be stored in the database internally, it is not considered to be part of the data. When an application asks the system to retrieve record 1002 from the orders table, the record id is used to locate the record but is not part of the returned data.

Secondly, note that the date (12 January 2005) has been stored as the number 13527. QM stores dates internally as a number of days from a reference date. We will discuss this in detail later.

Thirdly, notice that because we use mark characters to separate each field, value or subvalue, the actual data is of variable length. Although an application may display the customer number as four digits, the underlying database does not impose any restriction. We do not need to redesign the data structures when our ten thousandth customer walks through the door though the data entry screen may need a small change.

The way to interpret 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. Use of multi-values is very common; subvalues are much less often used so do not worry too much if you are having trouble grasping the concept - we are asking you to think in four dimensions!

There are actually five mark characters, the additional two being the **item mark** (sometimes called the **record mark** which is hardly used in QM) and the **text mark** which is used when splitting long text items into multiple lines. We will meet both of these again later.

If you are familiar with the way in which character data is stored as single byte values, you can see from the table below that the mark characters are stored internally as the final five characters of the 8-bit character set. This was no problem when the data model was originally devised as the upper half of the character set was not defined. This is no longer true and the upper half of the character set has various definitions largely corresponding to language specific characters such as the accented characters of European languages. Unfortunately, using the final five characters for our own purposes means that the characters that we have displaced cannot be stored. As an example, using the most common character set used in Europe, the subvalue mark replaces the German u-umlaut (ü) so we cannot store the German name Müller correctly. This is clearly unacceptable in

some countries and QM has an Extended Character Set (ECS) mode that supports the much larger set of characters that form the Unicode Basic Multilingual Plane. ECS is not discussed further in this course.

	ASCII	Symbol	@var
Item mark	255	IM	@IM
Field mark	254	FM	@FM
Value mark	253	VM	@VM
Subvalue mark	252	SM	@SM
Text mark	251	TM	@TM

The symbol column shows how these characters are represented in examples in QM documentation. The @var column gives symbolic names that can be used for these characters in some places within QM as we will discuss later.

The History of Multivalue Databases

The original multivalue database is usually attributed to Dick Pick (hence the frequently used term "Pick databases") back in 1968 though their origins can be tracked back further. The current D3 database 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 closely follows the Pick style of operation. The long defunct Prime Information database from Prime Computer retained the same data model and general principles but made some fairly significant changes to the command and programming languages.

In the mid-1980's the various companies with multivalue products hit a problem. The world was standardising on the Unix operating system but these products did not run on Unix. As a result of this, McDonnell Douglas developed an "open systems" version of Reality (Reality X) and Prime Computer developed the PI/open database. At the same time, two start up companies appeared each with their own Unix based multivalue implementation, VMark (UniVerse) and Unidata (Unidata). These companies set out to capture users from the existing products as well as taking on new users. The history is long and complex but to bring it up to date in one step, UniVerse, UniData and D3 are all now 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. Except where noted, the examples and exercises in this course are based on QM "straight out of the box" with no compatibility options enabled.

Developers who try to defend the fully normalised relational model usually start by pointing out that the multivalue model is around 50 years old and hence cannot be of relevance in the modern world. In saying this, they conveniently ignore that the relational and multivalue models were developed at the same time. A better way to look at this is that nothing in the computer industry lasts 50 years unless it has something good about it. Once you get started with the multivalue model you will wonder why anyone would ever use anything else.

[All trademarks referenced in this introduction are acknowledged.]

2 The Command Environment

In this section we will explore the QM command environment and learn to execute simple commands.

Accounts

The word "account" has many meanings in the computer world. In QM, it refers to a place to work, typically corresponding to a software application. A QM system may have many separate accounts representing different applications (sales, payroll, etc) or different versions of an application (development, test, production, etc).

An important concept in QM is that we try to avoid users needing to know about the underlying operating system (Windows, Linux, etc.) that is running on their system. If you do understand such things, it is worth knowing that from outside QM, an account is represented by an operating system directory that contains the data files, programs, etc that are used by the application. All QM systems also have a System Administrator account (QMSYS) under which all the QM system software and many control files can be found.

Although accounts represent separate working environments, they may share resources such as files and programs. For example, a system with two applications representing sales order processing and stock control is likely to need some degree of communication between the two. The QMSYS account is a good example of this as it contains a number of files that are visible from all other accounts.

Accounts can be created by the System Administrator using built-in administrative tools (more later) or simply by trying to run QM in a directory which is not already set up as an account.

The most important file in an account is the VOC (vocabulary) file which contains all the words and symbols that can be used within the account to form commands. If you are moving to QM from a Pick style system, the VOC is broadly similar to the MD file of your old system.

The VOC also contains references to all the files (database tables) accessible from the account and other user defined items. A newly created VOC has around 500 system supplied items in it and will usually grow considerably as an application is developed. There is a whole section on the VOC file later in this course.

Installation

This course assumes that QM is already installed on your system. If it is not and you are using a Windows system, a description of the installation process follows. For other operating systems, see the relevant section of the *QM Reference Manual* for guidance on how to install QM.

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

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_1.2-3.exe, where the numeric components identify the release. Execute this file by, for example, double clicking on it in Windows Explorer. 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.
QMAdmin	A Windows based system administration tool.
QMClient	The API for Windows developers who need to access QM from other languages.

The default action is to install all of these components and should only be changed for non-standard installation.

After the main installation has been performed, the install process displays a screen in which the authorisation data can be entered. If you are installing an upgrade, the previous licence data is displayed and can be retained by simply pressing the return key in each field. If you are installing a new commercial licence or an evaluation licence, enter the details from your licence. If you are planning to use the free Personal Version, simply enter the word Personal in the licence number field. Everything in this course with the exception of the data encryption material can be done on the Personal Version but it has some limitations.

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 then runs the QM Configuration Editor to allow changes to be made to configuration parameters. Leave everything at its default settings and click on *Close*

Finally, the installer offers to show the readme file.

Entering and Leaving QM

Although QM allows developers to construct applications that are accessed via a web browser or via graphical screens typical in Windows systems, we will do most of our work via the character mode "green screen" interface. We will discuss web and GUI (graphical user interface) concepts later.

A commercial QM licence includes free activation of the AccuTerm terminal emulator. Although we recommend this and AccuTerm includes some features specifically for QM users, QM should work with most other terminal emulators. If you are using the Personal Version of QM, AccuTerm is not included.

The time has come to get logged in. There are several ways to do this but you may need some help from your System Administrator as you get started:

- On a Windows system, navigate from the Start menu to Programs, QM and select QMConsole. This creates what is known as a **console session**.
- On a Linux, FreeBSD or Mac system, type "qm" at the operating system command prompt when in the account directory. This assumes that your System Administrator has added the QMSYS account bin subdirectory to the PATH environment variable that controls where your system looks for commands. If not, you will have to use the full pathname (probably /usr/qmsys/bin/qm).
- Directly over a network. Your System Administrator will need to set up a user name and password for access to the system. By default, QM uses port 4242 but this may have been changed on your system. Make sure that whatever port you are using is open in your firewall.

Depending on how your system has been set up, you may need to enter an account name as part of this process. Consult your System Administrator for advice. Do not use the QMSYS account for development or training purposes though you may need to start there and create a private account of your own. Once in QM, you can create your own account to run the exercises in this course by typing

```
CREATE.ACCOUNT name pathname
```

where name is the *name* you would like to give the account and *pathname* is the operating system pathname for the account directory. For example,

```
CREATE.ACCOUNT JOE C:\JOE
```

To exit from QM when you have finished your session, type QUIT, LOGOUT or OFF.

Case Inversion

As you start to use QM, you will probably notice something very strange happening. The characters that you type are "case inverted" - that is, lowercase letters are displayed in uppercase and vice versa. The original multivalue 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. We will discuss more about how to do this later. Your System Administrator can also disable case inversion in the QM configuration parameters.

QM Command Syntax

A complete QM command is known as a **sentence**. A sentence always begins with a **verb** which is the command name. This may be followed by qualifying information such as file names, record ids and **keywords**. Sentences may be executed directly from the command prompt or stored in the VOC for later execution.

The individual tokens that make up a sentence are separated by one or more spaces. Where a token contains spaces, it should be enclosed in quotes. In most cases, QM allows use of single quotes, double quotes or backslashes interchangeably.

Some simple commands with no qualifying information are shown below. Try each of these.

CS	Clears the terminal screen
DATE	Displays the date and time in 12 hour clock format
LISTU	Displays a list of active QM users
TIME	Displays the date and time in 24 hour clock format
WHO	Displays your QM user number and account name

More complex commands take qualifying information. Some useful ones are shown below but we are not yet in a state to try these.

DATE.FORMAT ON	In common with other multivalue products, QM defaults to American date format (month day year). This command switches to European date format. Used with OFF in place of ON, it switches back to American date format. You will not notice any difference to the output from DATE or TIME as these commands explicitly set their date display format. The DATE.FORMAT setting only affects situations where an application says "print a date" but doesn't specify how it should appear. Application software is usually written to output dates in specific formats.
LOGTO *account*	Switches to the named account. This allows a user to move between accounts subject to having sufficient access rights.
PHANTOM *command*	Executes the given command as a background process allowing the terminal to be used for other tasks. A phantom process has no terminal associated

with it, therefore it will fail if it asks for input. Any output that would normally have appeared on the terminal is written to the file system in a file named $COMO (command output). Phantoms are typically used for overnight processing and other lengthy tasks.

Many commands reference records in data files. These tend to share a common syntax that takes the file name and a record id (or, in some cases, multiple record ids). Some examples are:

CT *file id*	Copies the named record to your terminal display (CT is short for Copy to Terminal).
DELETE *file id*	Deletes the named database record. Careful! Once you have deleted a record, it has gone. There is no equivalent of the Windows recycle bin to undo your mistakes though it is possible to implement this using techniques that we will meet later.

Now is a good time to create the demonstration database that is used throughout this course. To do this type

```
SETUP.DEMO
```

in an account that you will use for the exercises in this course. This command creates several files related to two demonstration databases. For the purposes of this course, we will be interested in the files belonging to a very simple stationery shop sales application; STOCK, CUSTOMERS and SALES. You can take a quick look at the content of the STOCK file by typing

```
LIST STOCK
```

and similarly for the other files.

You can repeat the SETUP.DEMO command at any time to revert back to the original file structure. If you want to restore the original data but leave any new items that you have created in place, use

```
SETUP.DEMO UPDATING
```

Now try the CT command by typing

```
CT SALES 12001
```

What does this show? As we will discuss in more detail later, the SALES file has seven fields representing the order date, customer number, part number, quantity, price, payment date and payment value. Fields 3, 4 and 5 are associated multivalued fields because a customer may buy more than one thing. Similarly, we might want to allow a customer to pay for his order in stages so fields 6 and 7 form a separate set of associated multivalued fields.

The CT command shows this as

```
SALES 12001
1: 14400
2: 1000
3: 001y003
4: 2y1
5: 170y170
6: 14407
```

```
7: 510
```

The first line shows the file name and record id. The remaining lines are the fields from this record. Field 1 does not look like a date but it is. This is because QM stores dates internally as a number of days from a reference point in time. We will discuss this in detail later but the value 14400 corresponds to 4 June 2007. The multivalued fields may not appear on your terminal exactly as shown above because different terminal emulators show the value mark character in varying ways. You will find it useful to discover how your particular terminal emulator displays the mark characters. A well behaved application never displays marks as they represent the boundaries between pieces of data but there are times when we might need to use the CT command to look at how data is stored internally.

Notice that the data record contains nothing to show associations. This is the job of the **dictionary** that defines the data layout for the file. There is a whole section on database dictionaries later.

The CT command is useful for displaying simple text items (which we will do several times) and it is a good tool for detecting errors in the way in which data is stored. For example, a common mistake in one of the programming exercises later in this course results in an extra value mark on the end of the list of part numbers. This would be clearly visible in the output from the CT command but might be quite difficult to spot using other tools.

The Command Stack

QM keeps a record of the recent commands that you executed. The command processor includes mechanisms by which you can look back at these commands, modify them and repeat them. For this to work correctly, your QM session must be set up to match the terminal type that you are using. Normally, this happens automatically but you can check the terminal type that QM is using by typing

```
TERM
```

and, if it is incorrect, change it with, for example,

```
TERM vt100-at
```

to select the vt100 terminal emulation. The -at suffix in the example above is for use with AccuTerm. Other emulations of a vt100 terminal should not use this suffix. A QMConsole session uses a terminal type of "qmterm".

To walk back through the command history, press the cursor up key. If you go too far, you can walk back down again with the cursor down key. The currently displayed command can be repeated simply by pressing the return key and you can edit a command using the horizontal cursor keys, backspace, etc. The full list of edit operations is

Ctrl-A or HOME	Move cursor to start of command.
Ctrl-B or Cursor Left	Move cursor left one character.
Ctrl-D or DELETE	Delete character under cursor.
Ctrl-E or END	Move cursor to end of command.
Ctrl-F or Cursor Right	Move cursor right one character.
Ctrl-G	Exit from the command stack and return to a clear command line.
Ctrl-K	Delete all characters to the right of the cursor.
Ctrl-N or Cursor Down	Display "next" command from command stack.
Ctrl-O or Insert	Toggle insert/overlay mode. When in overlay mode, characters entered at the keyboard overwrite any existing data at the cursor position. In insert mode, new characters are inserted into the existing text.
Ctrl-P	Display "previous" command from command stack.
Ctrl-R	Search back up the command stack for a given string.
Ctrl-T	Interchange characters before cursor.
Ctrl-U	Convert command to uppercase.
Ctrl-Z or Cursor Up	Display "previous" command from command stack.
Backspace	Backspace one character.

Although the control key codes above may be a little difficult to remember at first, you will find that they are shared by many components of QM and you will learn them quite quickly.

QM also supports a second command editing system based on commands prefixed by a period (.) for compatibility with other multivalue products. For example typing **.L** at the

command prompt will show you a list of the most recent 20 commands. For more details, see the *QM Reference Manual* .

Pagination

As you continue using QM you will notice that commands that produce more than a single page of output display a continuation prompt at the end of each page. For the query processor, the options available may vary. For other commands, the options are:

A Abort processing and return to the command prompt, possibly running the ON.ABORT paragraph described later.

Q Quit the command and continue processing.

S Suppress pagination, continuing with no more pagination prompts. This is useful when capturing the output locally with the terminal emulator.

Any other key continues processing.

Aborting Commands

Sometimes we need to terminate processing because something is not working as expected. We can do this with the break key which is usually Ctrl-C but may be moved. Pressing this key displays a list of options, some of which do things that we have yet to discuss:

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 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 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.

Note that applications often run with the break key disabled in live environments for additional security.

Compatibility Options

If you are moving to QM from another multivalue environment, you may find that although QM is essentially similar to the environment you have come from, it has some significant differences. QM broadly follows the "Information style" model but has options to enable closer compatibility with other systems. You can find out more about these by looking at the OPTION command and the $MODE compiler directive in the *QM Reference Manual* . Except where specifically stated, this course uses QM in its default settings.

3 The QM File System

A QM application stores its data in **files** (some users call these **tables**) each of which usually has a corresponding **dictionary** describing the data file. For the purposes of this section, a dictionary is just a file with a special purpose. (You may have come from a database environment that stores a **schema** that provides a view of the entire database. In a multivalue database, each file has its own separate dictionary that describes the content of the file and its relationship with other files).

QM has two types of file which offer different characteristics and are appropriate to different usage in the application software. **Directory files** do not give high performance but allow data to be viewed or modified from outside of the QM environment. They are therefore frequently used for data interchange with other software. **Dynamic files** offer very high performance but cannot be accessed from outside QM. They are typically used for the bulk of the data stored by an application.

In most cases, application software doesn't care what file type is used.

Creating and Deleting Files

Files are created using the CREATE.FILE command. In its simplest form, this is

```
CREATE.FILE name
```

for a dynamic file or

```
CREATE.FILE name DIRECTORY
```

for a directory file. There are additional options discussed in the *QM Reference Manual* that allow the file to be created in a non-default location or modify the configuration details for a dynamic file. In most cases, these can be omitted.

The *name* component is the name to be used to reference the file within QM. This must not already exist as a record in the VOC file. File names used by an application should be chosen to be meaningful but not so long that they become a nuisance to type. The name may be formed from any printable characters but must not contain spaces. An operating system directory will be created to represent the file and this will have the same name as the QM file unless this is not a valid operating system file name. In this case QM manufactures a valid operating system file name based an automatic translation of the name used in the command. From inside QM, you do not need to know about this translation.

A file normally has two parts; a **data part** that holds the application data and a **dictionary part** that holds a definition of the structure of the data records. The dictionary also sets default actions for the query processor.

The CREATE.FILE command normally creates both the data and dictionary parts of the file. The operating system name for the dictionary is the same as the data portion name but with a suffix of .DIC added. A default description of the record id named @ID is automatically added to the dictionary and may be modified but must not be removed. An application developer will normally add dictionary entries to describe each field of the data record.

It is possible to create a data file that has no dictionary by prefixing the name with DATA. This might be used if, for example, the file holds simple unstructured text rather than being a database table with defined fields. For example,

```
CREATE.FILE DATA PROGRAMS
```

Similarly, the DICT keyword can be used to create just a dictionary. A dictionary that has no data file is probably meaningless but this form of the CREATE.FILE command can be used to add a dictionary to a file previously created without one. For example, a dictionary could later be added the PROGRAMS file created above using

```
CREATE.FILE DICT PROGRAMS
```

If you are migrating to QM from another multivalue database product, you may be familiar with **multi-files**. These can be considered as a set of identically structured data files that share a common dictionary. A typical use of these might be to divide a client database into business regions. QM has full support for multi-files but we will not discuss them in detail in this course.

The DELETE.FILE command is used to delete a file. The syntax of this command is

```
DELETE.FILE name
```

Again, the name may be prefixed by DATA or DICT to delete just one part of the file. Beware that once a file has been deleted, the only way to get it back is to restore it from a backup.

The DELETE.FILE command will prompt for confirmation if the operating system pathname of the file as recorded in the VOC is not the default that CREATE.FILE would use. This helps to avoid accidental deletion of files in other accounts.

Directory Files

These are very simple. What the QM user sees as a database file is represented by an operating system directory. The records within that file are represented by text files in the directory.

QM	Operating System
File	Directory
Record	Text file

This structure allows access to the records stored in the QM file from outside of QM but will not give high performance. Directory files are also widely used to store very large records (perhaps over 100kb for a single record) as the operating system file structures are optimised for very large sizes whereas the high performance hashed files are optimised for smaller sizes. Storing very small records in directory files can be inefficient as some operating systems allocate space in large chunks, perhaps as large as 32kb, resulting in a record that is just a few bytes long requiring a large amount of disk space.

The text file created to represent each record will have the same name as the record id unless this contains characters that are not valid in operating system file names. In this case, QM applies an automatic translation to yield a valid name. Note that on Windows systems, file names are case insensitive and hence it is not possible to store two records in a directory file with ids that differ only in casing.

Because directory files are frequently used to exchange data with other software that may not understand the multivalue data model, data written to a directory file has field marks replaced by newlines. The effect of this is that when the record is viewed from outside QM with an operating system file editor, etc, each field appears as a separate line of text. Lower level mark characters (value marks, subvalue marks and text marks) are not affected because exporting multivalued data implies that the software that will read it must understand multi-values. The reverse transformation occurs when reading from a directory file, replacing newlines by field marks.

The potential problem with this data transformation is that records containing binary data such as bit-mapped images (scanned documents, digital photographs, etc) may be corrupted if they are stored in directory files. Such data may contain any pattern of bytes. When the data is written out, all field marks are converted to newlines. When read back in again, all newlines, including those that were in the original data, are converted to field marks. Programmers can use the MARK.MAPPING statement to suppress this mapping as described in the QMBasic programming sections of this course.

Directory files are sometimes used to import or export very large records. These might, perhaps, consist of many thousand of lines of text. Although a programmer can treat the file as a database record and access the whole item in a single operation, the QMBasic programming language allows the application to read or write the item line by line. This is not possible with dynamic files and hence programs written to use this feature will not work with dynamic files. There is more about this in the section that discusses QMBasic sequential file processing.

Although they are very simple and do not offer high performance, all applications include use of directory files. Some of the standard elements of QM use directory files.

Dynamic Files

QM's dynamic files use a mathematical technique called **hashing** to optimise performance. Although you do not need to understand this to use QM, a brief description may be useful. You can skip this if it is not of interest.

To understand hashing, let's forget computers for a while and consider a simple analogy.

Imagine that you walk into a library to find a book written by Fred Smith and that the library (which hasn't quite caught up with technology) has a big wooden card index cabinet. What do you do?

What you certainly don't do is to open the drawer labelled A, scan all the cards, move to the drawer labelled B and so on. Clearly, you would go straight to the correct drawer. In our computer model, we need a similar way to go directly to the small part of the database where we expect the record to be stored.

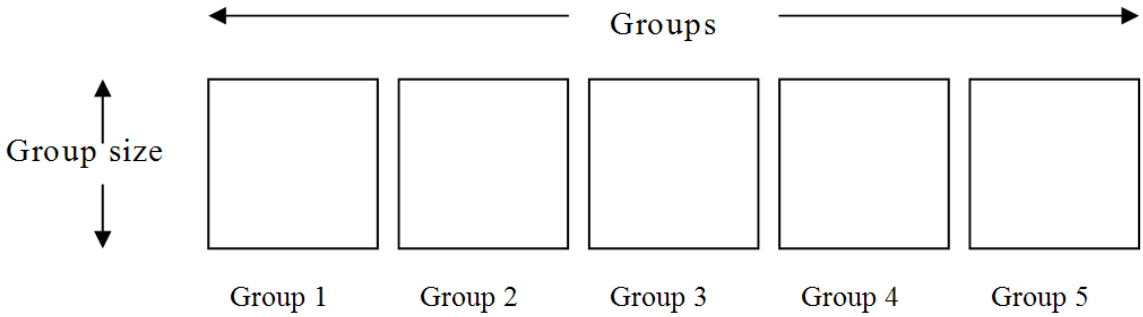
The second problem is that, if you were to look in all the drawers, some would hold many cards whilst others may be nearly empty. There aren't many authors with names beginning with X, for example. This means that the card index cabinet contains wasted space. In our computer model, we would like to minimise the wasted space by ensuring that the distribution of records is as even as possible. It is probably not acceptable to be told that the library has reduced wasted space by sorting the author catalogue on the third letter of the author's name but we can do whatever is convenient to us.

Finally, as the library grows, there will come a time when they purchase a new book, type up the card, and find that it will not fit in the drawer. Just as a library is not going to say that they cannot stock a book because the card won't fit in the drawer, so in our computer system we need a way to store a record somewhere else if it won't go where it should.

So, how does all this work in the computer model?

A hashed file is made up of a series of identically sized areas called **groups**, analogous to the drawers of the card index cabinet. The number of groups in a file is referred to as the **modulus** (or modulo). In QM, a group may be from 1 to 8kb in size and there may be up to 2147483647 groups in a file (that's a maximum of 16 terabytes - far more than enough for even the most complex application).

A real data file would have many groups, perhaps thousands or even hundreds of thousands. The purpose of dividing the file into groups is to allow the position at which any particular record exists to be determined mathematically and hence enabling the record to be read with the minimum of searching. For the purposes of this explanation, consider a file that has a modulus of just 5.



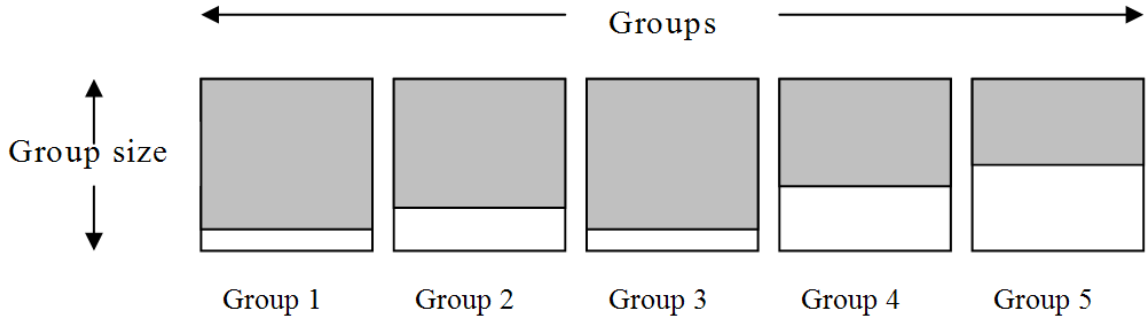
We wish to insert a new record with id 102. We need a system by which the record id can be transformed into a group number and that will give a reasonably even distribution of records across the file. Probably the simplest system would be to divide the record id (102) by the modulus (5) and use the remainder from this division plus one as the group number. Thus record 102 should be placed into group 3.

Some time later, we come to read record 102. Applying the same transformation to the record id, we know that this record belongs in group 3. We can read this one group from the disk and search through it. If we find record 102, the job is done; if we don't find the record, it is not in the file and we do not need to read any other groups.

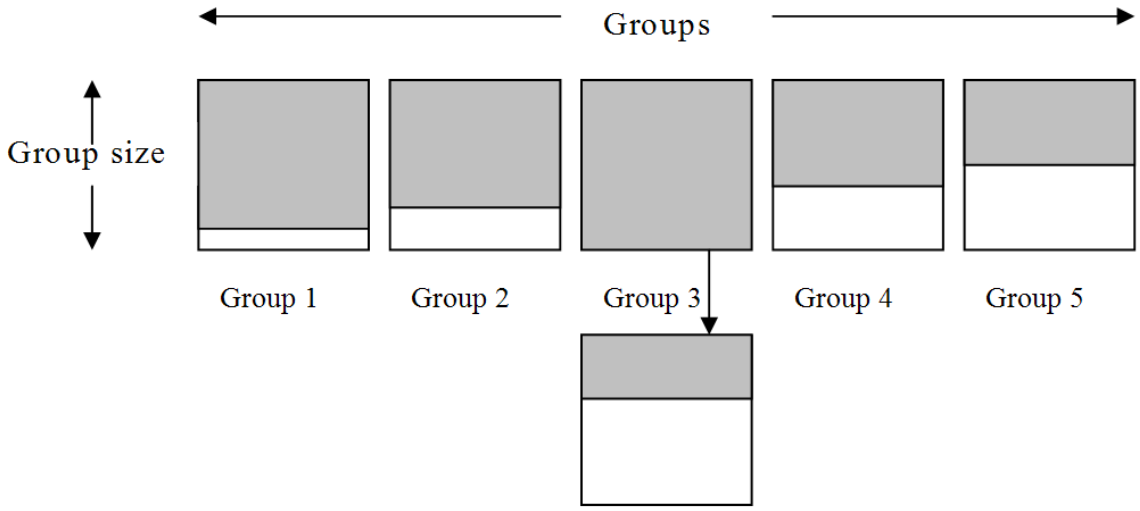
If the group size can be made sufficiently small, we can read any record with a single disk transfer. Clearly, it is not possible to get better performance than this. We have the world's fastest file system. Unfortunately, in the real world things don't quite work out this well but, as we will see, we should be able to get close to this performance.

A real data file is likely to have non-numeric record ids so that the above transformation is an over simplification of what QM actually does. Internally, QM performs a calculation based on the ASCII values of the characters in the record id to form a large number and then follows the procedure described above of dividing by the modulus and using the remainder plus one as the group number.

If the hashing process is working well, the records should be distributed reasonably evenly across the file resulting in a file something like that below where the grey portion of each group represents the used space, filling the group from the top down.



There will always be some unevenness in the packing. Group 5 above is only about half full while group 3 is nearly completely full. What happens if we try to add more data to group 3 and it won't fit? It is not acceptable to say that we cannot store a record because it won't fit where it should go. Instead, we add a new disk block to the end of the full group.



Group 3 in the diagram above is said to be **overflowed**. It is possible for the overflow block itself to become overflowed. In such instances, the affected group is said to be **badly overflowed**. This could progress to the point where a group has many overflow blocks attached to it though this is unlikely.

Overflow should be minimised as it affects performance and also increases the risk of file damage in the event of a system failure while writing a group back to disk. Dynamic files are so called because there is a built in algorithm that adjusts the modulus value to respond to changes in the volume of data stored in the file. The effect of this is to keep overflow within an acceptable range without the need for any system administration tools.

A newly created dynamic file normally has just one group. During the operation of the file, QM tracks the file **load value** (the volume of data as a percentage of the space allocated for groups, averaged over the whole file). When this value exceeds the **split load** percentage, a new group is added. As a result of this, some data in the file may need to be moved to the new group. Adding a group will have the effect of decreasing the load value because the file size has increased.

Similarly, if the load value falls below the **merge load** percentage, the final group of the file is removed, moving its data records into one of the remaining groups.

The effect of splitting and merging is to adjust the modulus in response to changes in the volume of data such that the file remains reasonably tightly packed and hence performs well.

A realistic dynamic file will still have overflow though the splitting and merging process tends to minimise this. To allow for overflow in a file that must also be able to adjust its modulus value, a dynamic file is represented by an operating system directory containing two files that store the primary groups (%0) and overflow blocks (%1) separately. The files are interlinked by pointers between the group buffers and must always be backed up or restored together. There may be additional elements (%2, %3, etc) if the file uses QM's alternate key index system.

When used to create a dynamic file, the CREATE.FILE command has several optional parameters. It is usually sufficient to omit all of these, leaving everything at its default value.

GROUP.SIZE *n*	Specifies the group size in multiples of 1024 bytes. Although this parameter may have any value between 1 and 8, best performance is obtained with 1, 2, 4 or 8.
MINIMUM.MODULUS *n*	A dynamic file created with this parameter will never shrink below the given number of groups. This parameter can be useful to avoid excessive splits and merges of a small file if records are added and deleted frequently or to pre-allocate disk space.
SPLIT.LOAD *n*	The split load percentage determines the load value at which a new group is added to the file. The default value of 80 works well for most files and should be changed only with great care.
MERGE.LOAD *n*	The merge load percentage determines the load value at which a group is removed from the file. The default value of 50 works well for most files and should be changed only with great care.
LARGE.RECORD *n*	Records of more than this size (in bytes) are treated as a special case and allocated their own private set of overflow blocks. This parameter can improve performance of files which contain predominately small records but also have some larger records. It defaults to 80% of the group size.

If you are interested in more detail of how dynamic files work, there is a detailed discussion in the *Dynamic Files* technical note on the OpenQM web site.

Case Sensitivity

With the exception of directory files on Windows, record ids in QM files are normally case sensitive. The NO.CASE option to CREATE.FILE creates a file with case insensitive record ids. One good example of use of case insensitive record ids is for files that use email

addresses as the record key. Most mail service providers treat these as case insensitive and so we would need to do the same.

4 Editing Data

As we progress through the exercises in this course, you will need to edit records in various files, sometimes to add new items, sometimes to modify what is already there. QM includes several editing tools that can be used to create and modify data. In this module we will look briefly at the more important features of the QM line editor, ED, and the full screen editor SED. There is much more to learn about both of these from the *QM Reference Manual* , especially SED which has many features specifically aimed at application developers.

We need to create some test data that you can use while exploring the editors. To do this, type

```
SORT NEWVOC CSV TO TEST FIRST 100
```

Do not worry about what this really does. Just accept that you now have a record named TEST in a file named $ACC (which is actually your account directory viewed as a directory file). You can use this record to try out the editor commands described in this section. It does not matter how much you destroy this data because we will not use it again elsewhere. You can always repeat the above command to recreate the data later but it will ask for confirmation before overwriting the old version.

The ED Line Editor

In a line editor, the record being edited is processed in a line by line manner where each line corresponds to a field in the data record. Although this is a fairly antiquated style of editor, it turns out to be very appropriate for some of the things that you will do with QM.

The early multivalue databases had only a line editor. ED is closely compatible with its equivalents in some other products (and annoyingly not quite the same as some others!). Programmers usually prefer a full screen editor such as SED.

The format of the ED command is

```
ED {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.

It is also possible to provide ED with a list of records to edit via the **default select list**, a concept that we will discuss as part of the query processor. 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 * on the command line will cause ED to select all records of the file and edit each in turn.

QM includes a locking mechanism to prevent unwanted interactions between users. The editor maintains an **update lock** on the record that is being edited so that any other user attempting to modify the same record will either wait for the lock to be released or report an error. Locking is discussed in detail as part of the programming sections of this course.

If you are going to try out the editor commands as you read this section, type

```
ED $ACC TEST
```

The editor responds by showing you how many lines (fields) there are in the record and then shows a command prompt.

Editor Commands

Each line entered from the keyboard contains one editor command. The command names are case insensitive but qualifying information may be case sensitive depending on the command and the current modes of the editor. Commands that take qualifying information specifying their exact function can usually be repeated by entering just the command name, carrying forward the qualifiers from the previous use of the same command within the same editor session. We will see examples of this as we begin to explore the editor command set.

The editor maintains the concept of a current position (line number) within the data. On entry to the editor, this is above the first line. Commands are provided to move the current position to a specific line or by searching. Many of the editor commands will move the current position as a result of their action.

The editor operates in two modes; edit and input. Edit mode accepts commands to move around within the text of the record, to make changes to the data or various other actions. 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. (If the record being edited has more than 9999 lines, the line number and prompt widths are adjusted to fit).

If a non-printing character is to be entered, it can be typed as ^*nnn* where *nnn* is the decimal value of the character within the ASCII character set. For example, although there may be a key combination on your keyboard to send a value mark character, this can always be entered as ^253. The only character that cannot be entered in this way is the field mark (character 254) as the editor treats each field as a separate line of text.

Positioning Commands

The commands listed below alter the position of the current line.

Entering a blank line at the command prompt advances the current position by one line, displaying the newly selected line. If you are trying out the editor as you read this, press the return key a few times and notice how this moves you down through the data, showing each line as it goes. Work your way to the last line (the editor will tell you when you get there) and notice that if you press the return key again, you come back up to the top. Then try each of the commands below:

T
: The T (top) command moves to before line 1. There is no current line after this action.

B
: The B (bottom) command moves to the last line of the record.

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.

L {*string*}
: The L (locate) command moves forward to the next line containing *string* which must be preceded by a single space. Any additional spaces will be treated as part of the string to be located. The L command is case sensitive by default. Typing

```
CASE OFF
```

: makes searches case insensitive for the remainder of the editing session. See the description of ED in the *QM Reference Manual* to discover how to make the editor case insensitive by default.

: 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.

Displaying Text

There are two main commands to display text:

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.

PP{*n*}
: The PP (print position) command displays *n* lines surrounding the current line position. The value of *n* defaults to 21 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.

Deleting Lines

D{*n*}

The D *n* (delete) command deletes *n* lines starting at the current line position. If *n* is not specified, only the current line is deleted. Note carefully that the optional numeric qualifier is the number of lines to delete, not the line to be deleted.

Editing 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.

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

The C (change) command changes the first occurrence of *old.string* to *new.string* in the current line. The delimiter around the strings may be any character other than a letter, a digit, a space or a question mark.

The optional *n* component specifies that *n* lines starting at the current line are to be changed.

G causes all occurrences of *old.string* in the line(s) being processed to be replaced. Without G only the first occurrence on the line is changed.

Entering C with no strings repeats the last substitution.

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. Entering R with no further characters repeats the last replacement operation.

Miscellaneous Commands

^

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. A value mark, for example, would appear as ^253.

HELP *topic*

The HELP command displays a short description associated with the command identified by *topic*.

OOPS

The OOPS command undoes the most recent function that modified the record in the current editing session. It is only possible to go back one step in this way.

XEQ *command*

The XEQ command executes the specified *command* which may be any valid QM command.

File Handling Commands and Leaving the Editor

FILE {{*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 original location.

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 record update lock is released and 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.

QUIT

The QUIT command (which may be abbreviated to Q) releases the record update lock and leaves the editor without saving any changes made to the record. A confirmation prompt is issued if there are unsaved changes.

The QUIT command terminates editing of the current record. 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.

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.

The editor contains many other commands that are not described in this short overview. See the *QM Reference Manual* or the editor help text for details.

The SED Full Screen Editor

A full screen editor displays a page of data from the item being edited and allows the user to move through it with the cursor keys or other control codes. The SED editor is based on the industry standard EMACS editor but includes many extensions tightly linked to the QM environment. This overview will introduce just a small subset of the editor's features, sufficient to make a start using QM.

The format of the SED command is

```
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.

SED behaves in exactly the same way as ED with regard to use of select lists or prompting for record ids. The record(s) being processed are locked to prevent unwanted interactions with other users.

To explore SED, edit the test record that you created at the start of this section:

```
SED $ACC TEST
```

The screen displayed by this command shows the start of the item being edited. There is a two line status area at the bottom of the screen that identifies the record being edited, its size, the current line and column positions and various editor status flags.

To move around in the data, use the cursor keys. You can modify the data simply by typing new characters to be inserted or using the delete or backspace keys to remove data.

The editor has many special functions. Some are control key combinations entered by holding the Ctrl key down while typing a letter (represented in the documentation as, for example, Ctrl-G). Others are prefixed by use of the Escape key without holding it down (represented as, for example, Esc-X) and some involve more complex sequences such as Ctrl-X followed by C (represented as Ctrl-X C).

The only functions that you need to know to make a start with this editor are

Ctrl-X S	Save the current data back to the file
Ctrl-X C	Close (exit from) the editor

Rather than trying to learn all the SED functions and their key bindings in one go, try to learn a few at a time and get used to using them. Other useful ones to learn soon are

Ctrl-A	Move the the start of the line
Ctrl-E	Move to the end of the line

Ctrl-G	Cancel a partially completed function key entry
Ctrl-V	Move down one screen
Esc-V	Move up one screen
Esc-<	Move to the top of the data
Esc->	Move to the bottom of the data

There are many powerful features to learn about later by reading the relevant section of the *QM Reference Manual* . SED has some really good cut and paste features for rearranging data and can work on multiple records in a single editing session.

5 The VOC File

The VOC file is the vocabulary of words and symbols that can be used in a QM command. Every account has a VOC file, indeed it is the presence of a VOC file in a directory that makes the directory into a QM account. If you are migrating to QM from a Pick style multivalue environment, the VOC is broadly similar in purpose to the MD (master dictionary) file though its content is very different. The name MD can be used as a synonym for VOC in QM.

The VOC of a newly created account contains around 500 items. This will expand rapidly during the development of an application.

Records in the VOC file serve many purposes. The function of each record is determined by a type code in the first field. The valid VOC record types are:

Type	Function
D	Defines a data field. More usually found in dictionaries, the VOC may contain definitions for fields common to many files.
F	Defines the operating system pathnames of a QM file.
K	Defines a keyword which controls some action of a command.
M	A menu definition. Although these can appear in the VOC, they are usually located elsewhere and accessed via an R-type VOC item.
PA	A paragraph. This is a sequence of stored commands as a script that can be executed by entering its name at the command prompt.
PH	A phrase. This is part of a query processor command.
PQ	A Proc program. These are supported for compatibility with other database systems and their use is discouraged in QM.
Q	A pointer to a file in another account.
R	A remote item. This points to a VOC style record stored in some other file. The addressed record must be of an executable type.
S	A sentence. This is a single command stored to allow its execution by typing just its name.
V	A verb. This is a QM command. Users can add verbs of their own to the command set in various ways.
X	A miscellaneous storage item. X-type records can be used to store any data.

The type code in field 1 of a VOC entry (line 1 when viewed with the editor) may contain comment text after the type code. This is useful to explain what the item does or to record who owns a sentence or paragraph entry. There does not need to be a space between the type code and the comment text.

This section looks only at the VOC record types that users are likely to create or amend; F, PA, PH, Q, R, S and X. D-type items are discussed in the section on dictionaries. For other record types, see the *QM Reference Manual* .

Because users are likely to generate VOC items that are specific to their own activities, particularly PA and S type records, QM provides the concept of a personal VOC which will be examined if the item cannot be found in the main vocabulary. The personal VOC is

normally private to an individual user within the account but can be shared between users or accounts as desired. Use of the personal VOC avoids the need to have potentially complex naming conventions for personal items in large systems. The personal VOC is fully described in the *QM Reference Manual* but not mentioned further here.

File Entries, Types F and Q

All files accessed from your account must be defined in the VOC file. There are two ways to do this.

The F-type VOC entry maps a QM file name to its underlying operating system pathnames. The basic format of an F-type entry is:

1 F

2 Data file pathname

3 Dictionary pathname

It is very useful to add a description of the file's role in the application to the end of the first field of this VOC entry.

The F-type VOC entry is created automatically by the CREATE.FILE command.

A file may be created with only a data portion or, more rarely, only a dictionary portion. In this case, the pathname for the absent portion is left blank.

The pathnames may be relative to the account directory or given as a full pathname.

Using the default naming for the operating system files, creating a QM file named SALES would result in a VOC entry as below:

```
SALES       F
            SALES
            SALES.DIC
```

Because all application software and commands use the VOC entry to locate the file, should we choose to move the file to a new location on the disk, we need only change pathname stored in the VOC entry. Nothing within the application itself is affected.

The pathnames in an F-type VOC entry may be prefixed by one of a set of special constructs to provide flexibility. Each of these will be replaced as shown below:

@QMSYS	The pathname of the QMSYS account
@TMP	The pathname of the QM temporary directory
@HOME	The user's home directory pathname
@DRIVE	The QMSYS account drive letter (Windows only)

For compatibility with other multivalue environments, QM supports the concept of **multifiles** which may be considered as a file that has subfiles, each containing data of the same type and sharing a common dictionary. A typical use of a multifile might be to divide a customers file into parts to correspond to business regions. The name used to reference each part is formed from the file name and subfile name separated by a comma, for example

```
CUSTOMERS,NORTH
```

The VOC entry for a multifile is extended to become

1 F
2 Data file pathnames (multivalued)
3 Dictionary pathname
4 Subfile names (multivalued)

There should only be one F-type entry for any given file. If we wish to access a file from another account we use a Q type VOC entry to do this. A Q-type VOC entry, often called a Q pointer, points to an F-type entry in another VOC file.

Consider a system with a sales processing account and a stock control account. The STOCK file might be required from both applications. Although this could be done with an F-type VOC entry in each account, this is the wrong way to do it.

Instead, one account uses a Q-type VOC entry to access the file via the other account's VOC. This ensures that if we later need to move the file there is only one VOC entry to modify and also gives a sense of ownership of the file. The account with the F-type entry owns the file. The account with the Q-type entry is borrowing the file.

The format of a Q-type VOC entry is

1 Q
2 Target account name or pathname
3 VOC entry name of file to access

Field 2 may either contain the actual pathname of the target account or the name of this account as defined in the central account register.

Although there is a performance penalty in this indirect access to the file, it is usually insignificant. It is even possible to have a Q pointer that points to a further Q pointer and ultimately leads to an F-type VOC entry though this is not recommended.

QM also includes a mechanism by which it is possible to access files on another server running its own QM system. Although this could be done simply by referencing a file pathname that referred to the remote system, this would not provide any concurrency control to ensure that conflicts from simultaneous updates could not occur. Instead, remote files on other QM servers should be accessed using QMNet, a built-in component of QM that provides full concurrency control over a network.

To create a reference to a remote file, the remote server is defined using the SET.SERVER command and the Q-type VOC entry is extended to include the server name in field 4. QMNet is not discussed further in this course but is fully described in the *QM Reference Manual* .

QM provides three commands to list the F-type VOC entries:

LISTF	Lists all F-type entries
LISTFL	Lists only local files (those in our account directory)
LISTFR	Lists only remote files (those not in our account directory)

Extended syntax

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 and may have been disabled by your system administrator as they have a potential impact on security.

The three extended syntaxes are:

Implicit Q-pointer	*account* : *file*
Implicit QMNet pointer	*server* : *account:file*
Pathname	PATH: *pathname*

Note that in the final form, depending on context, Windows users may need to use forward slash characters (/) as directory delimiters or enclose the pathname in quotes because the backslash (\) is reserved as a string quote.

Sentence Entries, Type S

A **sentence** is a complete QM command. Sentences may be stored in the VOC file so that they can be executed by typing just the sentence name.

The format of a sentence VOC entry is

1 S

2 Sentence

Long sentences may be split into multiple lines for ease of maintenance by ending each line other than the last with an underscore character. When executed, the lines are merged together, replacing the underscore with a single space. Don't worry yet about what this command actually does but an example of use of continuation lines would be:

```
S
LIST SALES_
BY CUST_
BREAK.ON "'VB'" CUST_
TOTAL SALE.VALUE_
HEADING "Orders for customer 'B'"
```

For many of the exercises and examples in this course we will use the demonstration database STOCK file. To view the content of this file we could type

```
LIST STOCK
```

If we are doing this continually, it might be worth setting up a VOC sentence with a short name to do the listing for us. This might appear as

```
LS      S
        LIST STOCK
```

Although you could create this VOC item using the ED or SED editors, the command stack editor provides an easy way to add a sentence to your VOC. First execute the command that you want to save and check that it works correctly.

```
LIST STOCK
```

Then type

```
.S LS
```

(note the leading period) to save the most recent command (LIST STOCK) as a sentence named LS.

Check that this has work by typing LS at the command prompt. You should see exactly the same listing of the STOCK file as when you typed the full LIST command.

When we execute a sentence in this way we can add further items to the end of the sentence by typing them after the sentence name. Try typing

```
LS WITH PRICE > 5
```

We can use the LISTS command to list the sentences in our VOC.

Paragraph Entries, Type PA

A **paragraph** is a stored sequence of commands that can be executed by typing just the paragraph name. Paragraphs are very useful for automating tasks that you perform frequently.

The format of a paragraph VOC entry is

1 PA
2 First sentence
3 Second sentence
4 etc....

Long sentences within a paragraph may be split into multiple lines for ease of maintenance by ending each line other than the last with an underscore character. When executed, the lines are merged together, replacing the underscore with a single space.

A paragraph starts at the first sentence and executes each sentence in turn until it reaches the end of the paragraph. There are constructs discussed in a later module to add conditional execution, loops, prompting for input, etc to a paragraph.

There are four reserved paragraph names that have specific purposes in QM. All of these are optional.

LOGIN	When you first enter QM or when you use the LOGTO command to move from one account to another, QM looks for a paragraph (or other executable VOC entry type) named LOGIN. If this is present, it is executed. The LOGIN paragraph is typically used to initialise the user's environment, performing security checks, setting up terminal types, printers, etc. It usually also takes the user into the application so that he never sees the QM command prompt. The break key is disabled until completion of this paragraph unless enabled by use of the BREAK command. The LOGIN paragraph is run for all QM sessions, including phantoms and QMClient connections.
ON.EXIT	This paragraph is executed, if present, when you leave QM. It could be used to tidy up work files used by an application or for logging.
ON.LOGTO	This paragraph is executed on use of the LOGTO command before leaving the previous account.
ON.ABORT	An **abort** is an event that occurs when things go wrong in your application. Aborts can be generated by QM itself, by statements within the application software to handle catastrophic disaster or by the user selecting the abort action from the break key options. When an abort occurs, QM discards all programs, sentences, paragraphs, menus, etc that are active in your session and returns to the command prompt. Just before it displays the prompt, it checks for a paragraph named ON.ABORT and, if present, runs it. The purpose of this paragraph is to ensure that the user does not end up at a QM command prompt if the application fails. Very often this paragraph simply logs the event and terminates the session.

There is a further reserved paragraph name in the QMSYS account. Sometimes, there may be other initialisation tasks that need to be run on initial entry to QM, regardless of which account we are going into. This can be achieved by creating a paragraph named MASTER.LOGIN in the VOC of the QMSYS account. This paragraph, if present, runs before the LOGIN paragraph but is run only once, not on use of LOGTO. The MASTER.LOGIN paragraph is run for all QM sessions except QMClient.

We can use the LISTPA command to list the paragraphs in our VOC.

Phrase Entries, Type PH

A **phrase** is part of a query processor sentence. They are used as short forms or to provide synonyms.

The format of a phrase VOC entry is

1 PH

2 Expansion

The phrase expansion may be split into multiple lines for ease of maintenance by ending each line other than the last with an underscore character.

As a simple example of use of PH records to provide synonyms, the VOC includes a phrase named WITHOUT. Use the CT command to look at this:

```
CT VOC WITHOUT
```

You should see that this phrase expands to

```
WITH NO
```

allowing the user to use the single word WITHOUT as a synonym for WITH NO.

We can use the LISTPH command to list the phrases in our VOC.

Remote Entries, Type R

Remote VOC entries are pointers to items that would normally be found in the VOC file but have been placed elsewhere. There are several reasons to use remote VOC entries:

- To move a large paragraph out of the VOC file to avoid very large VOC records.
- To access a common item from several accounts without duplicating it in each VOC.
- To enable use of security features.

The general form of an R type VOC entry is

1 R

2 Name of file holding remote item

3 Record name of remote item

4 Security subroutine (optional)

Some standard QM commands are implemented using remote pointers to a file named QM.VOCLIB which is in the System Administrator's account. For example, the LISTF, LISTS, LISTPH (etc) commands that we have met in this section work in this way

The optional security subroutine named in field 4 of an R type VOC entry enables verification that the user is to be allowed to execute the command. When a user attempts to execute a command that is accessed via an R type VOC entry, this subroutine is called. It can perform whatever processing the application designer wishes and returns a yes/no response. If the subroutine indicates that the command is valid, QM will execute the remote item. If it indicates that the command is to be disallowed, QM displays an error message.

Although V-type VOC records (verbs) are not discussed in this course because they are rarely created directly by a user, QM also allows security subroutines to be attached directly to V-type entries, removing the need for an intermediate R-type item as is required in other multivalue environments.

Full details of how to write a security subroutine can be found in the *QM Reference Manual* .

We can use the LISTR command to list the remote items in our VOC.

Miscellaneous Entries, Type X

X type VOC entries can be used to store whatever the application designer wishes. They might, for example, hold the version number of the application or control information that the application uses to determine its behaviour.

The general form of an X type VOC entry is

1	X
2+	Anything

6 Dictionaries

Every QM file normally has an associated dictionary which describes the structure of the data records stored in the file and the default way in which the query processor should present the data in a report.

The dictionary contains records of various types, each identified by a code in the first field of the entry. The dictionary record types are:

Type	Function
D	A D-type entry defines a data field present in the file and specifies its location and how it is to be displayed in a report.
I	An I-type entry (often referred to as a **virtual attribute**) defines a value that can be calculated from the data in the file and specifies how it is to be displayed in a report. The general concept of I-type items is included in this section. There is a whole module on virtual attributes later.
L	An L-type record defines a link to another related file. Use of link records can significantly reduce the number of I-type records needed in a dictionary but they can also weaken security.
A / S	A and S-type entries are provided for compatibility with Pick and Reality systems and provide many of the same features as D and I-type records. These are discussed in a later section.
PH	A PH-type entry is a phrase which can be substituted into a query processing sentence. There are some reserved phrases which control the default actions of the query processor.
X	An X-type entry is a miscellaneous storage item and may be used for any purpose.

D-Type Dictionary Records

A dictionary normally contains a D-type record to describe each field of the database records. A single field may be described by multiple dictionary records to provide alternative ways to view the data. Which record is used by the query processor depends on how the query is phrased.

Consider our SALES file. Each field of the data record has a corresponding dictionary record to describe it. There is also a dictionary record to describe the record id. The name of the dictionary record is the name by which the query processor will refer to the field.

D-type dictionary records normally consist of 7 fields. The table below shows the dictionary definitions of the first few fields of the SALES record. The first column shows the conventional name of the dictionary field, the second column is the dictionary field number and the remaining columns show what the dictionary entry would contain to describe the record id and the first five data fields.

	@ID	DATE	CUST	ITEM	QTY	PRICE

Type	1	D	D	D	D	D	D
Loc	2	0	1	2	3	4	5
Conv	3		D2DMYL[,A3]				MD2
Name	4		Date	Cust	Item	'RX'Qty	'RX'Price
Format	5	5R	9R	4R	3R	4R	6R
SM	6	S	S	S	M	M	M
Assoc	7				LINE	LINE	LINE

The role of each dictionary field is described below:

Type — The type field contains the dictionary entry type, D for a description of a field within the database record. The type code may optionally be followed by descriptive text.

Loc — The location field contains the position of the field within the database record. The record id is shown as field zero in dictionaries.

Conv — Data is sometimes stored in an internal format. The conversion code describes the conversion to be performed before the data is displayed in a report. Conversion codes are discussed in detail later. In the SALES file, the DATE field has a conversion code of D2DMYL[,A3] which tells the system that this is a date and the form in which it is to be displayed. We will look at how this code works in the next section.

Name — The name field contains the default column heading to be used in reports. The headings for the QTY and PRICE fields include some special control tokens. These must appear at the start of the text and are enclosed in single quotes. The R token says that the heading text is to be right justified in the column. When the text is narrower than the column, periods (.) are normally output in the unused positions. The X control token causes the unused positions to be space filled. A multi-line column heading can be created by making the name text multivalued. Each value appears as a separate line.

Format — The format field specifies the number of columns to be used to show this data and how the data is to be aligned within the given width. Format codes are discussed in detail later. In all the examples above, the numeric part is the number of columns to be used and the R says that the data is to be right justified within this width.

SM — This flag indicates whether the field is always single valued (S) or may be multi-valued (M). In our SALES file the date and customer number fields are always single valued but a customer can order multiple items. You may wonder why we need this flag as a single valued field is just like a multivalued field with only one value. The SM flag has an impact on how conversion codes and format codes are applied. Very little goes wrong if it is set incorrectly but there are occasional situations where it is important.

Assoc	Used only with multivalued fields, this field shows the relationship between associated multivalued fields. Any fields that have the same word in this dictionary field are associated together. There is a further step in defining an association described below. In our SALES file, the association name LINE has been used to link the fields that form a line of the order.

Use the command

```
LIST DICT SALES
```

to display the dictionary of the SALES file and see how this relates to the chart above. There are actually more D-type items than appear in the table above. We will meet these fields in later sections. There are also some additional items of other types that we will discuss later.

Notice how there are two entries describing the record id. The @ID entry is inserted by the CREATE.FILE command and may be modified but must not be removed. In this dictionary we also have an entry named SALE as an alternative view of the record id with a more useful name. This technique is very common.

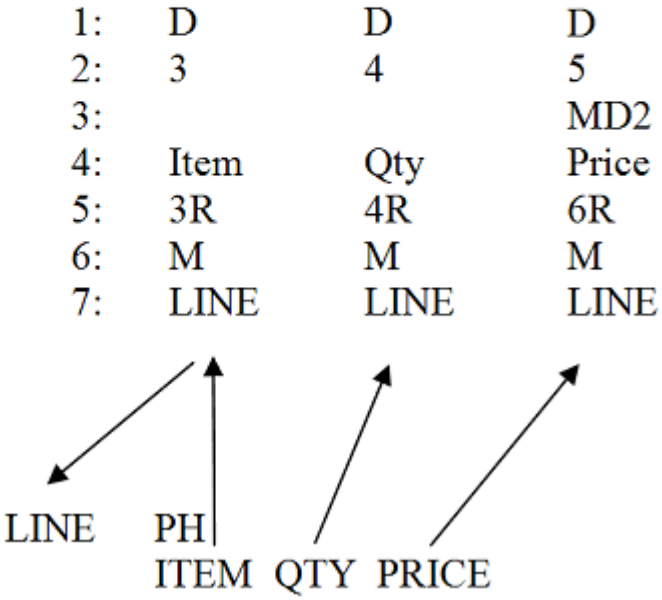
There are also two items for field 1, DATE and MONTH. These have different conversion codes that would result in the date being displayed in a different form.

Associations

An association is a set of two or more multivalued fields that are related such that the values are inter-dependent. For example, our SALES file contains a multivalued list of products, a corresponding multivalued list of quantities and a further corresponding multivalued list of prices. A realistic data file may contain several associated sets of fields. Out SALES file actually has two.

The query processor and the MODIFY data entry utility need 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. Applications migrated to QM from some other multivalue systems may also have a phrase record with this name which contains a list of the component fields though this is not required in QM. In our SALES dictionary, we have a phrase record named LINE that references the three fields in this association:

DATE	CUST	ITEM	QTY	PRICE
04 Jun 07	1000	001	2	1.70
		003	1	1.70



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

I-Type Dictionary Records

An I-type dictionary record defines a calculation based on data on 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 as for a D-type entry.

As an example, examine the dictionary of the STOCK file by typing

```
LIST DICT STOCK
```

The PRICE field holds the selling price for the item. The QTY field holds the number of the item we have in stock. If we wanted to calculate the value of the stock of each item type, we could construct an I-type dictionary record that multiplies these two values together. This dictionary record would be

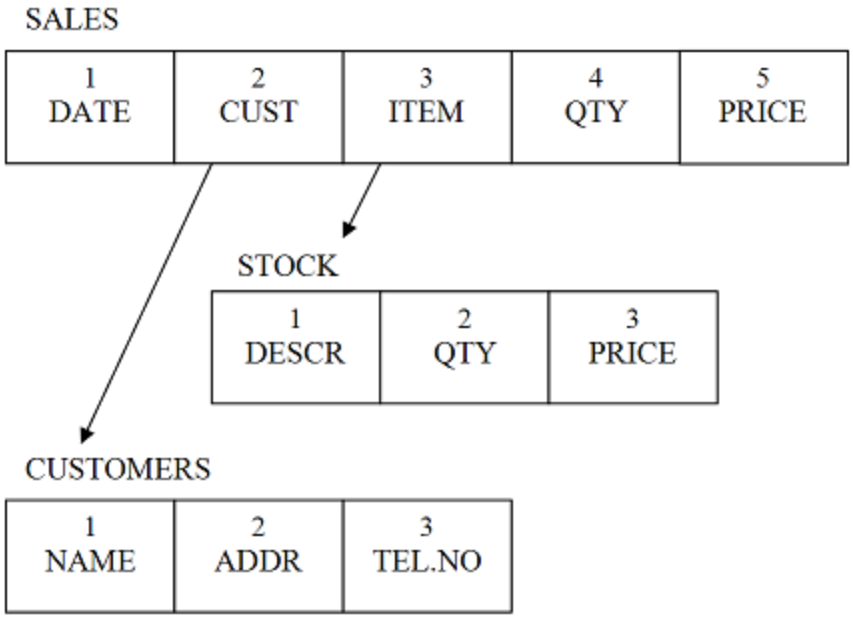
		VALUE
Type	1	I
Loc	2	PRICE * QTY
Conv	3	MD2
Name	4	Stock Value
Format	5	6R
SM	6	S
Assoc	7	

We will create this I-type record and many others in a later module where we discuss I-type records in detail.

L-Type Dictionary Records

An L-type record defines the relationship between two files.

Consider our SALES file. The CUST field contains the customer number. This value is also the record id to the CUSTOMERS file record that holds the details of the customer. Therefore, we could read a SALES record and use the customer number stored in it to find the related CUSTOMERS record. A similar relationship applies to the ITEM field and the STOCK file except that a single order may reference multiple stock items.



We might want to use the query processor to produce a report that shows the orders with their corresponding customer numbers. We could do this by adding an I-type record that uses a special function called TRANS() that we will meet in the I-types section. This function follows the link to another related file and returns a specific field from it. If we also wanted to be able to show the customer address or telephone number in a report of the SALES file, we would need a further I-type record for each of these.

QM provides a simpler way to do this. Instead of having a separate I-type for each item we want to be able to report from the second file, we have a single link record that defines the relationship between the files and then use a special syntax in our queries to refer to the data in that second file.

		CUSTOMERS
Type	1	L
Link	2	ITEM
Filename	3	CUSTOMERS

In this example, we have created a link record named CUSTOMERS to define the relationship with the CUSTOMERS file. It makes sense to use the name of the linked file as the id of the link record unless it causes a clash with other dictionary items.

Field 2 of the link record contains an expression that evaluates to the appropriate record id in the linked file. This is constructed in exactly the same way as an I-type expression. In this simple example, it is simply the name of the field (ITEM) that contains the record id of the linked file.

Field 3 holds the name of the file referenced by this link.

Having created the link record, we can now reference any field in the linked file in a query by using a special syntax of

linkname %field

where *linkname* is the name of the link record and *field* is the name of the field to be retrieved. For example, we could use

```
CUSTOMERS%NAME
```

to get the customer name. Although this mechanism can result in considerable simplification of dictionaries (we could access any of the customer fields without needing a separate I-type for each), it potentially weakens system security. Without link records, a user who can enter ad hoc queries can only access fields in the second file for which we have provided an I-type. With link records, all fields can be referenced. The data encryption mechanisms of QM could be used to hide restricted data.

Before we move on, you may be wondering why we have chosen to store the price in the SALES record when we could get it by following the link to the corresponding STOCK record. If we did this, any future change of price would also apply to past orders. By copying the price into the sales record when the order is placed, we fix the price at time of order.

PH-Type Dictionary Records

A phrase record defines a short form for some part of a query sentence. Where the query processor finds the name of a phrase in the query sentence it replaces the phrase name with the content of field 2 of the phrase entry. Long phrases may be split over several lines to ease maintenance by ending each line except the last with an underscore.

There are several optional phrases that are used by the query processor to control its behaviour. These will be discussed in a later section.

X-Type Dictionary Records

X-type dictionary records are for storage of miscellaneous data in any way that the application designer might find useful.

One common use is to store the id of the next record to be created in a file with sequentially allocated numeric record ids.

7 Conversion and Formatting

Conversion Codes

Data is not always stored on the database files in the same form that we would use to display it to a user. QM provides a wide range of conversion codes to transform data from its internal form to the external form when generating a report. The reverse transformation would be performed by data entry programs and is discussed in the programming sections of this course.

In this section we will examine the three most commonly used conversion codes; dates, times and decimal values. The other conversion codes can be found in the user documentation. It is also possible to add your own conversion codes to QM.

Date Conversion

Dates are stored internally as a number of days from 31 December 1967, that date being day zero. All later dates are positive numbers; all earlier dates are negative numbers.

By storing dates in this form, date manipulation becomes very easy. To work out the due date of an invoice, for example, we simply add the payment period to the issue date without any need to worry about month ends, etc. This date format also means that QM was unaffected by the "millennium bug" (1 January 2000 was day 11689). We had our own date crisis on 18 May 1995 when the internal representation of the date became five digits. Many application designers had assumed that dates were always four digits.

Of course, the customer receiving the invoice does not want to see the date as a day number. We need some system to convert the internal date to the conventional written form of the date.

The simplest date conversion code is just D. This applies a default conversion which can be altered using the DATE.FORMAT command. Converting dates in this simple way is not recommended because any change to the default settings of the system will affect the application output. Instead, most dictionaries use an extended form of the date conversion code to specify precisely what format they wish to use.

D {*n*} {*s*} {*fmt*} {E} {L}

where

n is the number of digits to appear in the year. If omitted, four digit year numbers are used.

s is a non-numeric separator to be used between the day, month and year.

fmt Determines the components to be included in the converted date and the order in which they are to appear. If both *s* and *fmt* are omitted, the date is converted in the form 19 JUL 2000.

The *fmt* specification may include the following codes:

D	Day of month
DO	Ordinal day of month (1st, 2nd, 3rd, etc)
J	Day number in year (Julian date, 1 to 366)
M	Month number
MA	Month name
Q	Quarter in year (1 to 4)
W	Day of week as a number (Monday = 1, Sunday = 7)
WA	Day of week as a name
WI	ISO week number
Y	Year number
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.

The *fmt* may be followed by qualifying information enclosed in square brackets with one comma separated qualifier for each component in the *fmt*. These qualifiers may contain:

{Z}*n*	Number of digits for a numeric component, number of characters for a non-numeric component. If Z is present, leading zeros are suppressed.
A{*n*}	Requests alphabetic form of the item. *n* specifies the field width.

E Toggles the date format between day/month/year and month/day/year.

L Retains lowercase characters is day and month names. If not present, the name is converted to uppercase.

Examples

All of the following are conversions of internal date 11879. For cases where there is any difference, the result is shown for both settings of the QM DATE.FORMAT option.

	DATE.FORMAT	
Code	Off	On
D	09 JUL 2000	
D2	09 JUL 00	
D4	09 JUL 2000	
D/	07/09/2000	09/07/2000
D2/	07/09/00	09/07/00
D4/E	09/07/2000	07/09/2000
D4/DMY	09/7/2000	
DWA	SUNDAY	
D/WADMYL	Sunday 09/7/2000	
DJY	191 2000	
DQ	3	

DDMYL[,A3]	09 Jul 2000
DDMYL[Z,A]	9 July 2000
DYA	DRAGON

An extended form of the DATE command can be used to translate a date between internal and external form from the command prompt. For example,

`DATE 21 Apr 08`	translates the supplied date to internal form
`DATE 14408`	translates the supplied date to external form
`DATE INTERNAL`	displays the current date in internal form

Time Conversion

Times are stored internally as a number of seconds since midnight. The time conversion code converts a time from its internal representation to hours, minutes and (optionally) seconds.

The full format of this conversion code is

`MT{H}{S}{`*c*`}`

where

- 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.
- S specifies that the converted time is to include the seconds component.
- *c* is the character to separate the hours, minutes and seconds fields. If omitted, a colon is used.

Decimal Value Conversion

Decimal values such as currencies and other weights and measures are normally stored internally scaled by some number of decimal places to remove the decimal point. For example, our STOCK file stores the prices in cents rather than in dollars (or pence instead of pounds, etc).

We scale numbers in this way for three reasons:

Firstly, removal of the decimal point reduces the storage space required. When this style of database was first created, disks were extremely expensive and this was an important point. With current storage device prices, this reason has largely gone away.

Secondly, by scaling monetary values such as dollars to cents often results in values which are whole numbers. The computer hardware is considerably faster performing whole number (integer) arithmetic than decimal (floating point) arithmetic.

Thirdly, just as there are numbers that we cannot write accurately in decimal notation (e.g. one third), so there are numbers that cannot be represented accurately in the binary form

used inside computers. Scaling the number may remove the rounding error and preserve accuracy through calculations.

The masked decimal conversion code, MD, converts numbers by applying scaling and also specifies the use of currency symbols, negative value representation, etc.

The format of the major features of this conversion code is

MD*n*{*f*}{,}{$}{*modifier*}{*s*}{Z}{T}{*x*{*c*}}

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 commas are to be inserted as the thousands delimiter.

$ specifies that a dollar sign should be used as a prefix to the converted data on output conversion and may be present on input conversion. Other currency symbols are handled by use of the *modifier* element below.

modifier consists of up to four comma separated components enclosed in square brackets which allow specification of a prefix, alternative thousands separator, alternative decimal separator and suffix.

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 angle brackets. A positive value has a space placed to its right.
- < encloses negative values in round 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.
- D places the letters DB to the right of negative values or two spaces to the right of positive values.

Input conversion accepts any of these representations of a negative value regardless of the actual conversion code used.

T specifies that the value is to be truncated rather than rounded to the required number of decimal places.

Z specifies that a zero value should be represented by a null string on output conversion.

x{*c*} specifies that the result of an output conversion is to be a field of *x* characters. The optional *c* component is the character to be used to fill unused positions and defaults to a space if omitted. The value of *x* may be one or two digits.

Examples

The table below shows various conversion codes used with the value 123456789.

Conversion code	Result
MD2	1234567.89
MD25	1234.57
MD25T	1234.56
MD2['DM', '.', ',']	DM1.234.567,89
MD2[,,,'kg']	1234567.89kg

Format Codes

Format codes determine the format in which the converted data is output by the query processor.

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 QMBasic. Within the query processor, data formatted with this code that is wider than *field.width* is not wrapped over multiple lines but extends into the space to its right, possibly overwriting whitespace in later columns.

n specifies the number of decimal places to appear in the result when formatting numeric data. The value is rounded in the normal manner. If *n* is zero, the value is rounded to an integer.

m specifies the scaling factor to be applied. The value being formatted is scaled by moving the decimal point *m* - *p* places to the left, where *p* is the current precision value.

conv is any meaningful combination of the following codes:

$ specifies that the national currency symbol should be used as a prefix to the converted data on output conversion and may be present on input conversion. The default currency symbol is a dollar sign but this may be changed by use of the NLS command or the SETNLS QMBasic statement.

, indicates that the national language convention thousands delimiter is to be inserted every third digit to the left of the decimal point when converting numeric data. This delimiter defaults to a comma.

B appends db to negative numbers, two spaces to positive numbers. Use the CRDB.UPCASE keyword of the OPTION command to change this to DB.

C appends cr to negative numbers, two spaces to positive numbers. Use the CRDB.UPCASE keyword of the OPTION command to change this to CR.

D appends db to positive numbers, two spaces to negative numbers. Use the CRDB.UPCASE keyword of the OPTION command to change this to DB.

E encloses negative number in angle brackets (<…>). Positive numbers are followed by a single space.

M appends a minus sign to negative numbers.

N suppresses any sign indicator.

Z indicates that a value of zero should be represented by a null string.

mask specifies a mask to be used to format the data. If omitted, *field.width* must be specified. Both can be used together.

The mask consists of a character string containing #, * or % characters and other characters. Each #, * or % is substituted by one character from the source data. Other characters are copied directly to the result string. Multiple #, * or % characters may be represented by a single #, * or % followed by a number indicating the number of characters to be inserted. Characters having special meaning within the format string may be prefixed by a backslash (\) to indicate that they are to be treated as text.

The value 1234567 with a format specification of 9L#2-#3-#2 would return 12-345-67.

Where the *mask* specifies more characters than in the data being converted, positions corresponding to # characters in the mask are replaced by the *fill.char* , positions corresponding to * characters in the mask are replaced by asterisks and positions corresponding to % characters in the mask are replaced by zeros. If the data is left aligned, the padding is inserted in the rightmost positions. If the data is right aligned, the padding is inserted in the leftmost positions.

If the *mask* specifies fewer characters than in the data being converted, part of the source data will be lost. A left aligned format will truncate the source data and a right aligned format will lose data from the start of the source.

Data formatting attempts to handle the data as a number if the decimal places, currency symbol, comma insertion or null zero features are included in the format specification. If these features are all absent, or if the data cannot be converted to a number, it is handled as a string. The difference in handling is relevant when processing data such as a string with leading zeros.

Format Code Examples

The following table shows some uses of format codes.

Value	Format code	Result

'ABCDE'	'8L'	'ABCDE '
'ABCDE'	'8R'	' ABCDE'
'ABCDE'	'8'*'L'	'ABCDE***'
'0012345'	'8R'	' 0012345'
'0012345'	'8RZ'	' 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'
'43'	'"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[TM]LINE '
'A LONG LINE'	'7T'	'A LONG[TM]LINE '
'A LONG LINE'	'8T'	'A LONG[TM]LINE '
'A LONG LINE'	'8R'	'A LONG L[TM]INE'
'BANANAS'	'3T'	'BAN[TM]ANA[TM]S '

8 Virtual Attributes

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 also 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 as for a D-type entry.

We saw a simple example in the dictionaries module where we calculated the value of each stock item type by multiplying the price by the quantity in stock:

		VALUE
Type	1	I
Loc	2	PRICE * QTY
Conv	3	MD2
Name	4	Stock Value
Format	5	6R
SM	6	S
Assoc	7	

Field 2 holds the expression to calculate the value. Fields 3 to 7 of the dictionary item are exactly as for a D-type item. In this example, because we are multiplying a monetary value by a simple number, the result is also a monetary value and hence requires the same conversion code.

A virtual attribute is a little QMBasic program. Such programs have to be compiled into a form that can be executed by the QMBasic run machine. This will be done automatically when the virtual attribute is first used in a query. Alternatively, compilation can be forced by using the COMPILE.DICT (CD) command.

The compiled version of the expression and related control data appears in the dictionary in fields 15 onwards. If you edit an I-type record using ED or SED, the compiled data is hidden. Other editors may not do this and you need to beware that this can contain absolutely any character sequence. Displaying these fields with the editor might have unfortunate effects on your terminal state. It is best to avoid displaying these fields.

A typical dictionary might contain many I-type items. The calculations defined by them are only performed when they are to be used in a query and hence there is no processing overhead involved in having many seldom used I-types in a dictionary.

Exercise

Use ED or SED to add the VALUE I-type item shown above to the dictionary of your STOCK file. To check that it works, you will need to type

```
LIST STOCK @ VALUE
```

The @ in this query sentence tells the query processor that we want to see all of the default items as well as the VALUE item named later in the sentence. We will look more closely at how this works when we discuss the query processor.

Notice how the query processor has expanded the width of the VALUE column from the 6 characters specified in the format code so that the column heading fits. As an alternative, we could make the column heading into two lines by replacing the space between the words with a value mark. To do this with ED, position on line 4 of the dictionary item and type

```
C/ /^253
```

or, with SED, delete the space and type

```
Esc-Q v
```

then try your report again (Don't forget that you can use the cursor up key to walk back through previously typed commands rather than retyping them).

Note how the column heading has now appeared left aligned with periods used to fill unused spaces. This might be better with the 'RX' control code as used in the headings for PRICE and QTY. Modify your VALUE dictionary item again to add this and repeat the query to display the modified report.

Solution

		VALUE
Type	1	I
Loc	2	PRICE * QTY
Conv	3	MD2
Name	4	'RX'Stock Value
Format	5	6R
SM	6	S
Assoc	7	

The only change from the original I-type shown in the main text is that we have added the 'RX' control code to the display name. The R shows the heading right justified. The X suppresses the default action of filling unused columns with periods.

The Virtual Attribute Expression

A virtual attribute expression consists of other data items defined in the dictionary, constants, operators which manipulate the data, and a wide range of functions taken from the QMBasic programming language. The value of the virtual attribute is the result of evaluating this expression.

Complex virtual attributes are sometimes written as **compound expressions** in which the calculation is performed in stages. A compound expression consists of a series of expressions separated by semicolons. The result of each expression is stored in variables named @1, @2, @3, etc so that they are available for use in later expressions. The value of the compound virtual attribute is the result of the final expression.

Data Items

The data items referenced by a virtual attribute may be real data fields defined by D-type entries in the same dictionary or other virtual attributes. Where one virtual attribute uses the result of another, they are combined during the compilation phase rather than during execution to improve efficiency. If you amend the expression of an virtual attribute that is used by some other virtual attribute, you must also recompile the other item. It is best to use the COMPILE.DICT (CD) command to compile all of the virtual attributes if you are uncertain whether there are inter-dependant items.

A later section will introduce Pick style A and S-type dictionary items which are supported by QM to ease migration. Once you have learned about these, A/S types with a correlative expression are essentially the same as an I-type as far as use in virtual attributes is concerned. A/S types without a correlative expression are essentially the same as D-types.

Virtual attribute expressions often need to use constants. These may be numbers, which are written without quotes, or character strings, which are enclosed in either single or double quotes or in backslashes. Examples of constants are

```
14
-17.5
"Red"
'Part number'
\ABC\
```

There are also many useful data items available via @-variables. Some of the commonly used ones are shown below.

@FM	Field mark
@VM	Value mark
@SM	Subvalue mark
@TM	Text mark
@DATE	The internal form of the date at which the query started execution
@FILENAME	The name of the file being processed
@ID	The name of the record being processed

@TIME	The internal form of the time at which the query started execution
@RECORD	The database record being processed

Operators

The operators available to manipulate the data items are shown in the table below. The operations are performed in the sequence shown. Where a single section of the table contains more than one operator, they are of equal priority and are evaluated left to right. Brackets may be used to modify the evaluation sequence in the same way as in other mathematical formulae.

** or ^	Exponentiation. A**B is A raised to the power B
* /	Multiply Divide
+ -	Addition Subtraction
[start, length]	Substring extraction. "ABCDEF"[2,3] is "BCD"
[length]	Trailing substring extraction. "ABCDEF"[2] is "EF"
:	String concatenation. "ABC":"DEF" is "ABCDEF".

There is also a conditional expression of the form

```
IF condition THEN expr.1 ELSE expr.2
```

The value of this expression is that of *expr.1* if the *condition* is true, *expr.2* if the *condition* is false. Any value other than zero or an empty character string is treated as true.

Exercise

Add a virtual attribute named TAX to the dictionary of your STOCK file to calculate the tax due on selling an item. Test it in the same way as your VALUE item. Assume that the tax should be 17.5% of the selling price.

If you have got this right, the tax on item 001 should be 0.30.

Solution

		TAX
Type	1	I
Loc	2	PRICE * .175
Conv	3	MD2
Name	4	'RX'Tax
Format	5	5R
SM	6	S
Assoc	7	

You may have chosen to write the expression as

```
PRICE * 17.5 / 100
```

which gives the same answer but it is a good idea to resolve the constant part of this expression as in our example.

Relational Operators

Relational operators compare two data items, producing a true/false result. The true value is represented by 1 and false by 0.

=	EQ		Equal to
#	NE	<>	Not equal to
>	GT		Greater than
<	LT		Less than
>=	GE	=>	Greater than or equal to
<=	LE	=<	Less than or equal to

Complex conditions can be constructed using the AND and OR logical operators. The AND operator returns true if both expressions are true. The OR operator returns true if either (or both) expressions are true.

Example

```
PRICE >= 500 AND QTY > 50
```

This expression returns true if the item has a selling price of at least 5 (note that the comparison is done on internal form data) and a stock level of over 50.

The MATCHES Operator

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
*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 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 "".

Example

```
DESCR MATCHES "Red..."
```

This expression returns true for items where the description begins with "Red".

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.

Group Extraction

We frequently find character sequences that are constructed from multiple concatenated items. This is particularly common in record ids. A transaction file might, for example, be keyed by an id of the form client-date-sequence where sequence is a simple sequential number appended to the client-date to ensure that multiple orders from the same client on one day have distinct ids.

If we need to extract the components from this composite item, we cannot use substring extraction because each of the components is of variable length and hence we do not know where it starts and ends. Group extraction lets us pick out one or more component parts from a composite string where there is a separator character between each part.

There are two ways to write this operation. The first uses the FIELD() function.

```
FIELD(string, delimiter, occurrence)
```

The FIELD() function takes a string divided into components by a single character delimiter and extracts the given occurrence. In our client-date-sequence record id example for a record id of 1744-11984-2, the three components could be extracted using

```
FIELD(@ID, '-', 1) for the client (1744)
FIELD(@ID, '-', 2) for the date (11984)
FIELD(@ID, '-', 3) for the sequence number (2)
```

Perhaps we want the date and the sequence number. The FIELD() function can be extended to include the number of consecutive components to be extracted.

```
FIELD(string, delimiter, occurrence, count)
```

Thus

```
FIELD(@ID, '-', 2, 2)
```

would return 11984-2. Note that the embedded delimiter is included in the returned data.

The group extraction operation may be written in a different form as

```
string[delimiter, occurrence, count]
```

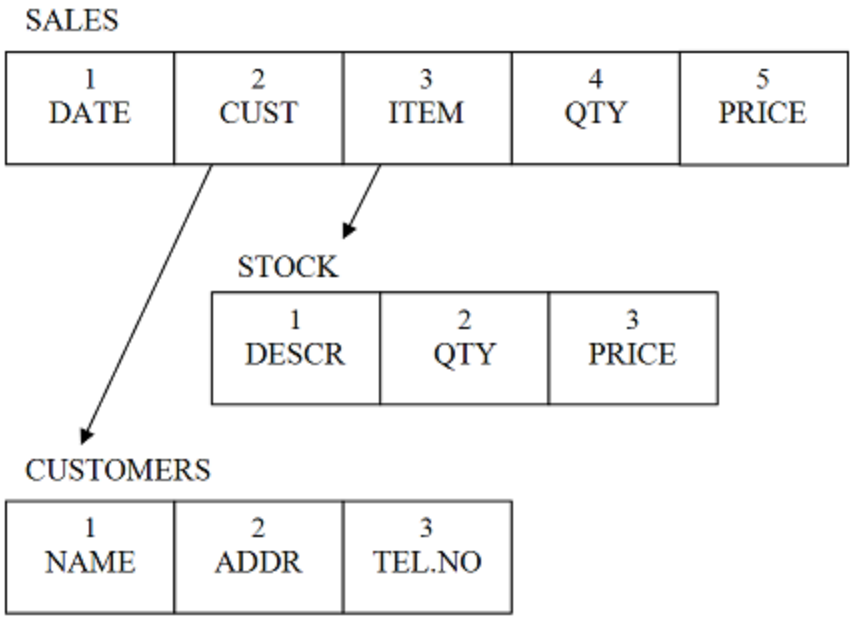
In this form, the count must be given, even if it is 1. The previous examples become

```
@ID['-', 1, 1]    for the client (1744)
```

```
@ID['-', 2, 1]     for the date (11984)
@ID['-', 3, 1]     for the sequence number (2)
@ID['-', 2, 2]     for the date and sequence number (11984-2)
```

The TRANS() Function

Our three demonstration database files do not operate in isolation. The CUST field of the SALES file contains the id of the CUSTOMERS file record for the customer placing the order. Similarly, the ITEM field contains the id of the STOCK file record for the product ordered.



When generating a report based on the orders file, we might want to fetch an associated item from one of the other files. We can do this using the TRANS() function.

Perhaps, even though our SALES record contains a copy of the price at the time the order was taken, we wish to fetch the current selling price for the item ordered. We could add a new virtual attribute named CURRENT.PRICE that contains

```
TRANS(STOCK, ITEM, PRICE, 'V')
```

The TRANS() function requires four items inside the brackets. These are:

The file name	In this example we are going to fetch data from the STOCK file.
The record id	The name given here must be a D or I-type item defined in the same dictionary as the virtual attribute and is the field holding the id of the record to be read from the target file (PROD.NO).
The field name	This is the name of the field to be read from the target file (SELL). This name must be defined in the dictionary of the target file. In QM, this may be a simple D-type item or a virtual attribute.

The error code — This determines what happens if we fail to find the data we are looking for. The error code must be quoted. Possible values are:

- C Return the id of the record we were trying to read.
- V Return a null (blank) value and display an error message.
- X Return a null value.

An order can contain many product numbers. If the second parameter to the TRANS() function is multivalued, it reads each of the specified records and returns a multivalued result. The dictionary entry for the virtual attribute needs to define it as multivalued if either the second parameter to the TRANS() function or the field being returned is multivalued.

Add the CURRENT.PRICE virtual attribute to your SALES dictionary as an I-type item. It should appear as

		CURRENT.PRICE
Type	1	I
Loc	2	TRANS(STOCK, ITEM, PRICE, 'V')
Conv	3	MD2
Name	4	Current$_{VM}$Price
Format	5	6R
SM	6	M
Assoc	7	LINE

Note that field 6 defines this as a multivalued item because although a single item only has one price, the ITEM field may be multivalued, causing TRANS() to return multiple prices in one operation. Also note that we have defined this I-type as being part of the LINE association. Don't forget that you will also need to modify the LINE phrase to add the new field name.

Now run a query

```
LIST SALES @ CURRENT.PRICE
```

Note how item 013 was priced at 0.28 when the order was placed but has a current price of 0.30.

The next step in building a useful set of I-types for this dictionary is to add one that calculates the total value of each line of the order. This needs to multiply corresponding values in the PRICE and QTY fields to produce a similarly structured multivalued list of line totals. Fortunately, the system manages the hard part of this for us and all we need to do is calculate

```
PRICE * QTY
```

All of the arithmetic operators work in this way when given multivalued data, operating on corresponding pairs of values. The LINE.VALUE I-type becomes

	LINE.VALUE

Type	1	I
Loc	2	PRICE * QTY
Conv	3	MD2
Name	4	Line Total
Format	5	6R
SM	6	M
Assoc	7	LINE

Notice how the expression is identical to the one we used in the STOCK dictionary to calculate the value of the stock for each item. The only things that change because we are now working with multivalued data are that field 6 contains M and we need to add this item to the LINE association.

We can test this new I-type with

```
LIST SALES @ LINE.VALUE
```

The final step in building this dictionary is to insert a further I-type to calculate the total value of the order. To do this, we need to use the SUM() function which adds up the elements of a multivalued data item. The expression is simply

```
SUM(LINE.VALUE)
```

Add this as an I-type named SALE.VALUE that contains

		SALE.VALUE
Type	1	I
Loc	2	SUM(LINE.VALUE)
Conv	3	MD2
Name	4	'RX'Total
Format	5	7R
SM	6	S
Assoc	7	

Test this final I-type with

```
LIST SALES @ LINE.VALUE SALE.VALUE
```

It would be nice to arrange that the LINE.VALUE and SALE.VALUE items appear automatically when we list the SALES file. We will discuss the detail of this in the query processor module but we can do it by adding the two field names to the end of line 2 in the item named @ in the SALES dictionary. If you have got this right, typing just

```
LIST SALES
```

now produces the complete report.

Think about what the SALE.VALUE I-type does. It evaluates

```
SUM(LINE.VALUE)
```

where LINE.VALUE is itself an I-type defined as

```
PRICE * QTY
```

This is a good example of one I-type that uses the result of another. Remember that, as described at the start of this section, the process of compiling the I-type expression performs a substitution so that the actual expression evaluated by SALE.VALUE is

```
SUM(PRICE * QTY)
```

If we were subsequently to modify the LINE.VALUE expression in some way, we need to force recompilation of the SALE.VALUE expression too. This can be done with

```
COMPILE.DICT SALES
```

and it is a good idea to do this whenever there is a possibility that a change might affect some other I-type.

Exercises

Add a virtual attribute called NAME to the SALES dictionary to fetch the customer name from the CUSTOMERS file.

Add a virtual attribute called DESCR to the SALES dictionary to fetch the part description from the STOCK file, displaying this in a 20 character wide field.

Solutions

		NAME
Type	1	I
Loc	2	TRANS(CUSTOMERS, CUST, NAME, 'C')
Conv	3	
Name	4	Customer Name
Format	5	20T
SM	6	S
Assoc	7	

We have used the C error code so that, if we are unable to find the customer record, we will display the customer number in place of his name. Of course, failing to find the customer record implies that our data is faulty but it is a good idea to allow for such problems.

		DESCR
Type	1	I
Loc	2	TRANS(STOCK, ITEM, DESCR, 'C')
Conv	3	
Name	4	Description
Format	5	20T
SM	6	M
Assoc	7	LINE

Notice how the result of this I-type is defined as being multivalued because the ITEM field may contain a multivalued list of item ids.

Note also that we have added this to the LINE association. The LINE phrase must also be modified to add DESCR to the list of fields within the association.

Accessing BASIC Subroutines from I-Types

Sometimes we want to develop a virtual attribute that cannot be evaluated by a simple (or even complex!) expression. A virtual attribute may make use of subroutines written using the full QMBasic programming language. Such I-types give almost limitless possibilities but it is easy to lose sight of what the expression is doing.

To call a QMBasic subroutine from an I-type, we use the SUBR() function. In its simplest form this is written as

```
SUBR('subroutine.name')
```

The named subroutine must be in the system catalogue, a concept discussed later in the programming sections of this course. The programmer developing this subroutine must arrange that it returns its result via a single subroutine argument (or use the FUNCTION equivalent).

In many cases, although the subroutine can see the data record being processed via the @RECORD variable and its id in @ID, it needs additional data passed in. The SUBR() function can be extended to include further subroutine arguments.

```
SUBR('subroutine.name', arg1, arg2, ...)
```

In all cases, the result of the subroutine must be returned via its first argument.

Working with Multivalues

Virtual attribute expressions can do some amazingly complex things with very simple expressions. This is because the language has a large number of functions that perform operations on multivalued data on a value by value basis. The following paragraphs give a brief overview of these. You can only really discover their power by seeing how they can be used in real applications.

The REUSE() Function

We have seen that the arithmetic operators (*, /, +, -) all work on corresponding pairs of values when the items on either side of the operator are numeric arrays. For example,

A contains 12VM9VM 6VM18
B contains 4VM3VM2VM12

A + B evaluates to 16VM12VM8VM30
A / B evaluates to 3VM3VM3VM1.5

What if there are fewer items in B than in A?

A contains 12VM9VM 6VM18
B contains 4VM3VM2

A + B evaluates to 16VM12VM8VM18
A / B evaluates to 3VM3VM3VM18

The addition has assumed that the "missing" element of B is zero. This is also true for subtraction and multiplication. For division, the missing item is assumed to be 1 if it is the divisor to avoid a divide by zero error.

Sometimes, we want to reuse the last value in place of the missing item. The REUSE() function allows us to do this.

A contains 12VM9VM 6VM18
B contains 4VM3VM2

A + REUSE(B) evaluates to 16VM12VM8VM20

This function is most often used when its argument is a single value, perhaps even a constant. For example, to add 17.5% tax to a list of prices, we could write

```
EX.TAX.PRICE * REUSE(1.175)
```

The REUSE() function is frequently used in dictionary I-type items, often in conjunction with the other multivalued functions described below.

Other Multivalue Functions

Imagine we have two fields such that

A contains ABC$_{VM}$DEF$_{VM}$GHI
and
B contains 123$_{VM}$456$_{VM}$789

we can join these two fields using the concatenation operator.

A : B results in ABC$_{VM}$DEF$_{VM}$GHI123$_{VM}$456$_{VM}$789

The concatenation operator has done exactly what we asked it to do. It has joined the two items end to end. It is more likely that what we really wanted to do was to concatenate corresponding elements of each value. This can be done using the CATS() function.

CATS(A, B) results in ABC123$_{VM}$DEF456$_{VM}$GHI789

There are many other functions which similarly work element by element through multivalued items. In general, the multivalued function name is formed by adding a S to the equivalent single valued function or operator to pluralise the name.

The multivalued string functions are:

CATS()	Multi-valued concatenation
COUNTS()	Multi-valued variant of COUNT()
FIELDS()	Multi-valued variant of FIELD()
FMTS()	Multi-valued format
ICONVS()	Multi-valued input conversion
INDEXS()	Multi-valued equivalent of INDEX()
NUMS()	Multi-valued variant of NUM()
OCONVS()	Multi-valued output conversion
SPACES()	Multi-valued variant of SPACE()
STRS()	Multi-valued variant of STR()
SUBSTRINGS()	Multi-valued substring extraction
TRIMBS()	Multi-valued variant of TRIMB()
TRIMFS()	Multi-valued variant of TRIMF()
TRIMS()	Multi-valued variant of TRIM()

There are also a number of multivalued logical functions. These provide equivalents to the relational operators and other functions that return boolean (true/false) values.

For example, the GTS(*a*, *b*) function takes two multivalued items and returns a new multivalued list of true / false values indicating whether the corresponding elements of *a* are greater than those of *b*.

Thus, if A contains 11$_{VM}$0$_{VM}$17$_{VM}$PQR$_{VM}$2
and B contains 12$_{VM}$0$_{VM}$14$_{VM}$ACB$_{VM}$2

```
GTS(A, B)
```

returns 0$_{VM}$0$_{VM}$1$_{VM}$1$_{VM}$0

The multivalued logical functions are:

ANDS()	Multi-valued logical AND
EQS()	Multi-valued equality test
GES()	Multi-valued greater than or equal to test
GTS()	Multi-valued greater than test
LES()	Multi-valued less than test
LTS()	Multi-valued less than or equal to test
NES()	Multi-valued inequality test
NOTS()	Multi-valued logical NOT
ORS()	Multi-valued logical OR

The IFS() function returns a multivalued list constructed from elements chosen from two other multivalued lists depending on the content of a third list.

```
IFS(control.list, true.list, false.list)
```

where

control.list	is a list of true / false values.
true.list	holds values to be returned where the corresponding element of *control.list* is true.
false.list	holds values to be returned where the corresponding element of *control.list* is false.

The IFS() function examines successive elements of control.list and constructs a result array where elements are selected from the corresponding elements of either true.list or false.list depending on the control.list value.

Example

A contains 1VM0VM0VM1VM1VM1VM0
B contains 11VM22VM3VM4VM91VM36VM7
C contains 14VM61VM2VM0VM35VM18VM3

IFS(A, B, C) returns 11VM61VM2VM4VM91VM36VM3

The DCOUNT() Function

The DCOUNT() function can be used to count the number of items in a list separated by some delimiting character.

DCOUNT(*list*, *delimiter*)

where

list is the list to be counted

delimiter is the single character separating list items.

Although DCOUNT() works for any delimiter character, it is most frequently used with the mark characters.

Example

```
DCOUNT(ITEM, @VM)
```

Used in our SALES file, this would return the number of lines in the order.

9 A and S-type Dictionary Records

Unless you are migrating an application from a Pick style system to QM, you can safely skip this section.

A and S-type dictionary records originated in the Pick and Reality database products and are supported by QM for compatibility. If your application was migrated from these systems, you are likely to have A and S-type dictionary records. It is strongly recommended that new developments use D and I-type records.

In QM, there is no difference between A and S-type fields. In Pick systems, A types were used to define attributes (fields) and S types to define synonyms as alternative ways to view the data.

Field	Content	Description
1	A {*description* } S {*description* }	Type code and optional descriptive text
2	Field number	Location of the data within the data records. Where field 8 contains a correlative, the field number is ignored.
3	Column heading	Text to appear as a column heading in a query processor report. A value mark character in the heading text inserts a newline thus forming a multiline heading. If blank, the field name is used.
4	C;*number* {;*number* } D;*number*	A code used to link associated multivalue fields.
5		Not used in QM
6		Not used in QM
7	{Conversion code}	A conversion code applied to convert the data to external form
8	{Correlative code}	A formula applied to the calculate the value of the item
9	L R T U	Left justified Right justified Text wrapping, breaking on word boundaries Left justified, overwriting whitespace
10	Width	Column width in query processor report

Associations in A and S-type dictionary items are defined using the code in field 4. In Pick style systems, one of the elements of an association is said to control the others. In the context of our sales application, the customer did not come into the shop and say "I think I will buy three of something, what shall I buy?" but he came in to buy a specific product and then decided how many he needed. In this example, the product number is considered to be the controlling item and the quantity and price are dependent on it.

One of the associated fields must include the C code in field 4 of its dictionary entry to define it as the controlling field. This is followed by a semicolon delimited list of field

numbers of the dependent items in the association. The other associated fields must use the D code to point back to the field that contains the C code.

The correlative code in field 8 is like an I-type in that it defines a calculation to be applied to evaluate the item but it is handled very differently.

Also, note that correlatives are applied as soon as the data is read from the file, before selection or sorting whereas conversions are applied as the final step before display of the data.

Correlatives

There are two types of correlative expression. An A (algebraic) correlative is an expression written in much the same way as one might write an I-type expression but is far more limited in what it can do. The expression is prefixed by the letter A optionally followed by a semicolon. An F (function) correlative starts with an F and an optional semicolon and is written in reverse Polish notation, separating each element with a semicolon. F-correlatives are difficult to read and hence difficult to maintain.

On Pick systems, A-correlatives are transformed into F-correlatives at the start of a query and then executed interpretively. Some developers write F-correlatives directly to skip this initial translation. On QM, both types of correlative are compiled in much the same way as I-types and hence there is never a justification to write F-correlatives which are much harder to understand. If you really want to know about F-correlatives, see the *QM Reference Manual* . Better still, when migrating an application, replace correlatives with I-types as soon as possible.

There are several key differences between A-correlatives and I-type expressions:

- Correlatives normally access data by field number, not field name
- Correlatives perform only integer (whole number) arithmetic
- Correlatives are limited to a few simple operations

The data items in a correlative expression may be:

number	A field number
N(*field name*)	To access a field by name.
"*string* "	A literal string enclosed in single quotes, double quotes or backslashes. Note that numeric constants must be written as strings or they appear to be field numbers.
D	The system date in internal form
T	The system time in internal form
@NI	The number of items processed
@ND	The number of detail lines since the last breakpoint
@NV	The value counter for a tabular report
@NS	The subvalue counter for a tabular report
@NB	The breakpoint level number

Thus the value of the stock for each item of our STOCK file could be calculated by a correlative expression

```
A;2*3
```

or, more meaningfully,

```
A;N(QTY)*N(PRICE)
```

Either of these is equivalent to an I-type expression

```
QTY * PRICE
```

The operators available within correlative expressions are

*	Multiplication
/	Integer division '5' / '2' is 2, not 2.5
+	Addition
-	Subtraction
:	String concatenation

The standard six relational operators are available as =, # or <>, <, >, <= and >=.

The AND and OR operators are available for logical relationships.

Expressions may include brackets to modify the order of evaluation.

Correlatives may also include some special functions:

R(*expr1* , *expr2*)	Returns the remainder after integer division of *expr1* by *expr2* . Thus R('5','2') would return 1.
S(*expr*)	Sums all the values in *expr* .
expr1 [*expr2* , *expr3*]	Substring extraction. Returns *expr3* characters of *expr1* , starting at character position *expr2* .
IF *expr1* THEN *expr2* ELSE *expr3*	Returns the value of *expr2* if *expr1* is true; otherwise returns the value of *expr3* . The expressions may be enclosed in brackets.
(*expr*)	Applies *expr* as a conversion code. It is often used with the T code to provide an equivalent of the I-type TRANS() function. For example: `A;N(ITEM)(TSTOCK;X;1;1)` in the dictionary of the SALES file could be used to fetch the description of each item.

10 The Query Processor

The QM query processor can be used to produce reports based on the data in one or more files. It can also be used to construct lists of records to be processed by other operations. In this module, we will learn about the various query processor verbs, the components of a query sentence and how we can piece the components together. There are a large number of examples for you to try as well as some longer exercises.

Query Processor Verbs

There are many query processor verbs. The most important is LIST which produces reports either on the terminal screen or on a printer.

All query processor verbs share a common general format though not all elements are applicable to all verbs. This format is

verb filename selection.clause sort.clause display.clause options

where

verb	is the query processor verb to be used.
filename	specifies the file to be processed.
selection.clause	specifies which records are to be included in the report.
sort.clause	specifies the order in which the reported data is to be output.
display.clause	specifies the fields to be displayed in the report and how they are to be reported.
options	are various additional options to control the page layout, output destination, etc.

The clauses that follow the file name are all optional and may appear in any order. The only order specific features are that fields will be shown left to right across the report in the order in which they are specified in the query sentence and multiple selection or sort clauses will be applied in the order in which they appear.

During processing of a query sentence, each word and symbol on the command line is looked up first in the dictionary of the file being processed and then, if not found there, in the VOC file. Quoted items are always treated as literal values.

The major query processor verbs are summarised in the table below.

COUNT	Count records meeting given criteria
LIST	Produce a report on the terminal or printer
LIST.ITEM	List the raw data from a file
LIST.LABEL	Print address labels, etc.
REFORMAT	Restructure data to produce a new file
SEARCH	Search for a text string in a file
SELECT	Build a list of records meeting given criteria
SORT	Produce a report on the terminal or printer, sorted by record id
SORT.ITEM	List the raw data from a file, sorted by record id
SORT.LABEL	Print address labels, etc., sorted by record id
SSELECT	Build a list of records meeting given criteria, sorted by record id
SUM	Sum the values of given data fields

The Filename

The filename in a query sentence specifies the file to be processed. This must correspond to a file defined in the VOC via an F or Q-type record.

A query processor sentence may only reference a single file though the dictionary of that file may include virtual attributes to fetch data from other files. In an earlier exercise, we added an entry to the SALES file dictionary named CURRENT.PRICE to reference the current selling price stored in the STOCK file.

The filename may be prefixed by DICT to process the dictionary part of the file.

Examples

LIST STOCK	will list the data part of the STOCK file.
LIST DICT STOCK	will list the dictionary of the STOCK file.

Use of Quotes and Casing

QM is very relaxed regarding use of quotes. Pick style systems usually require that record ids referenced in query processor commands are enclosed in single quotes and that all other constants are enclosed in double quotes. All examples in this module omit quotes where they are not needed in QM.

Also, the query processor is largely case insensitive for keyword and field names. The examples in this section are shown in uppercase simply because this is historically how most users work.

The Selection Clause

The selection clause specifies which records are to be included in the report. QM offers several methods to select records. If no selection criteria are given, all records are reported.

The simplest selection clause consists of one or more record ids. These should be quoted if there is any risk that they might also appear as dictionary or VOC items.

For example

```
LIST STOCK 001 013
```

Record selection performed in this way is very fast because the record(s) can be found by applying the file's hashing algorithm to go directly to the part of the file holding the data. The same performance benefits apply to use of a select list, a topic that we will discuss later.

The WITH Keyword

The most frequently used form of selection clause is the WITH keyword. This is used to select only those records that meet certain criteria.

Example

```
LIST STOCK WITH PRICE > 5
```

This query shows only those items with a price of over 5.00.

The WITH clause has many variations and many short forms. The example above uses a simple relational operator to compare the PRICE field with the value 5. Note that the comparison is written using the external form of the data even though the cost price is stored in the data file scaled up to a lower unit (cents, pence, etc).

Where a field that has a conversion code is compared with a constant value, the constant is converted into internal format and the comparison is done using the internal form data. In the above example, this results in a performance benefit by converting the constant value once rather than converting each record's price to external form. More importantly, it allows some flexibility in how the constant is written. Consider the examples below.

```
LIST SALES WITH DATE > "5 JUN 07"
LIST SALES WITH DATE > 6/5/07
LIST SALES WITH DATE > "JUN 5 2007"
```

Note the need for quotes around the dates in two of these examples as they contain spaces.

With American date formatting enabled (the default), all three queries produce the same result because the comparison is done using the internal representation of 5 June 2007 (14401). If your system is running with European date format enabled, the date in the second query would be taken as 6 May 2007.

The relational operators available are

=	EQ	EQUAL		Equal to
#	NE	<>	NO	Not equal to
>	GT	GREATER	AFTER	Greater than
<	LT	LESS	BEFORE	Less than
>=	GE	=>		Greater than or equal to
<=	LE	=<		Less than or equal to

The AFTER and BEFORE keywords make many queries using date comparisons more readable.

```
LIST SALES WITH DATE AFTER "5 JUN 07"
```

The dictionary of our SALES file includes an item named MONTH that shows the month in which an order was placed. This is simply a reference to the DATE field with a conversion code of DMA (date, month alphabetic). Try this with a query such as

```
LIST SALES MONTH
```

Suppose we want to see only orders placed in July. It seems that we could do this with

```
LIST SALES WITH MONTH = "JULY"
```

Why doesn't this work? (Hint: Think your way through the actions of the query processor as described above).

This query takes the external form of the date and converts it to an internal day number to do the comparison. The external form is simply JULY. What day of July, What year?

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 the above query is actually asking for orders placed on the first of July in the current year.

How could we modify the dictionary to make this work? One solution would be to replace the MONTH item with an I-type expression that evaluated to the month name but had no conversion code. This requires us to introduce a new function, OCONV(), that performs output conversion within the expression being evaluated.

		MONTH2
Type	1	I
Loc	2	OCONV(DATE, 'DMA')
Conv	3	
Name	4	Month
Format	5	9L
SM	6	S
Assoc	7	

Add this to the SALES dictionary as MONTH2. Then try the query again with the new dictionary item name

```
LIST SALES WITH MONTH2 = "JULY"
```

This now works but the month name must be entered in uppercase. The OPTION command of QM (not discussed in detail in this course) has a mode that makes query processor operators case insensitive by default. Alternatively, we can ask that just this one use of the equals operator is treated as case insensitive by modifying the query to become

```
LIST SALES WITH MONTH2 = NO.CASE "July"
```

The LIKE Operator

The LIKE operator (synonyms MATCHES and MATCHING) matches a string against a pattern consisting of one or more concatenated items from the following list (which should look familiar as we met it before in the section on virtual attributes).

...	Zero or more characters of any 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 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 "".

Examples

```
LIST STOCK WITH DESCR LIKE "'Pencil'..."
```

Note that the above example needs the full quoting or one literal string inside another as omitting the outer quotes would cause the command parser to treat 'Pencil' and ... as two separate items. It is, however, possible in this example to omit all of the quotes

```
LIST STOCK WITH DESCR LIKE Pencil...
```

Note also that the query we have executed here is looking for items where the description begins with "Pencil". There are two problems with this.

Firstly, the LIKE operator used in this way is case sensitive so we need to know exactly how the data is stored. We can overcome this problem by use of the case insensitivity mode of the OPTION command mentioned earlier or by the NO.CASE qualifier to LIKE

```
LIST STOCK WITH DESCR LIKE NO.CASE PENCIL...
```

Secondly, the LIKE operator in this query is not looking for the word Pencil but simply for descriptions that start with those letters. Try the query again but this time look for pens

```
LIST STOCK WITH DESCR LIKE Pen...
```

This time our report contains both pens and pencils. We could extend our query to include the space after the word Pen

```
LIST STOCK WITH DESCR LIKE "Pen ..."
```

but notice how item 005 has disappeared because it's description is simply Pen with no following space. What we really need is a way to look for a whole word within the description. The query processor does not provide a way to do this directly but it is very easy with a simple I-type as we will see later.

The UNLIKE Operator

The UNLIKE operator selects records that do not match the given pattern.

```
LIST STOCK WITH DESCR UNLIKE Pencil...
```

This query will show us everything except the pencils. Although this example query is unlikely to be very useful, there are many times with realistic data where the UNLIKE operator is of great value.

The SPOKEN Operator

The SPOKEN operator (synonyms SAID and ~) selects records in which the specified field "sounds like" some given value.

This data filtering method, known as Soundex, pre-dates QM and was originally developed for use in systems where users were taking telephone calls and might not be able to spell the caller's name correctly. A Soundex match could be used to offer a list of names from which the correct caller could be selected.

Try this query:

```
LIST CUSTOMERS WITH NAME SPOKEN ARKWRIGHT
```

Note how the query processor has found the customer even though we have spelt the named incorrectly. The SPOKEN operator is always case insensitive so we could have used Arkwright in place of ARKWRIGHT.

How Does it Work?

The Soundex system converts the data items to be compared into a four character sound code.

The letters of the alphabet are divided into seven groups:

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							

The first letter of the word is retained (You must get that right!).
The remaining letters are replaced by their group number except that
All letters in group zero are ignored
A letter that is in the same group as the immediately previous one is ignored
Non-alphabetic characters are ignored
The sound code is limited to four characters, padded with trailing zeros if necessary

Although Soundex matching is a useful technique, it is not perfect and is best when combined with other selection methods.

Case Insensitivity

All of the relational operators and the LIKE and UNLIKE operators may be followed by NO.CASE to make the comparison case insensitive. For example,

```
LIST CUSTOMERS WITH NAME LIKE NO.CASE COOPER...
```

There is also a QM option setting to make the query processor case insensitive by default.

Complex Selection Criteria

A selection clause may contain several of the above criteria linked together with the AND and OR keywords to form compound conditions. A condition using the AND keyword is true if both of the component conditions is true. A condition using the OR keyword is true if either (or both) of the component conditions is true.

Examples

```
LIST STOCK WITH DESCR LIKE Pencil... AND PRICE > 5

LIST STOCK WITH DESCR LIKE ...red OR DESCR LIKE ...green
```

The elements of the condition are evaluated left to right with the AND and OR operators having equal priority. Thus a single condition containing both AND and OR may not do what the developer intended. Brackets can be used to force evaluation into a specific order. Brackets should always be included in conditions mixing AND and OR to aid readability, even if they make no difference to the evaluation.

Example

```
LIST STOCK WITH DESCR LIKE Pencil... AND DESCR LIKE ...red OR
DESCR LIKE ...blue
```

This query might not do what you would expect on reading it quickly. It shows records 003, 012 and 013, two of which are pencils but one is a pen. Adding some brackets fixes this problem

```
LIST STOCK WITH DESCR LIKE Pencil... AND (DESCR LIKE ...red OR
DESCR LIKE ...blue)
```

and we now only see the red and blue pencils.

QM has an option to treat AND as higher priority than OR for compatibility with some other systems.

Selection Criteria Short Forms

These last few queries have bgun to get a little verbose. The query processor has several short forms to make queries more readable and to reduce the typing required.

1. Omitting the field name

A relational, pattern match or sound match operator that is not preceded by a field name uses the same field as in the previous condition.

```
LIST STOCK WITH PRICE >= 1 AND PRICE <= 5
```

can be reduced to

```
LIST STOCK WITH PRICE >= 1 AND <= 5
```

or even

```
LIST STOCK WITH PRICE BETWEEN 1 5
```

Our combination AND/OR query above,

```
LIST STOCK WITH DESCR LIKE Pencil... AND (DESCR LIKE ...red OR
DESCR LIKE ...blue)
```

is reduced to

```
LIST STOCK WITH DESCR LIKE Pencil... AND (LIKE ...red OR LIKE
...blue)
```

If the field name is omitted from the first condition in the sentence, the condition is assumed to relate to the record id.

```
LIST STOCK WITH @ID >= 100
```

can be reduced to

```
LIST STOCK >= 100
```

2. Implied OR

A relational operator followed by a list of values assumes an OR relationship.

```
LIST STOCK WITH DESCR LIKE ...red OR DESCR LIKE ...blue
```

can be reduced to

```
LIST STOCK WITH DESCR LIKE ...red ...blue
```

```
LIST SALES WITH ITEM = "001" OR ITEM = "004"
```

can be reduced to

```
LIST SALES WITH ITEM = "001" "004"
```

The implied OR short form is frequently used for mechanically produced queries where an application takes the root query and then appends a list of possible values.

Combining the first two short form methods allows us to further reduce our combined AND/OR query from

```
LIST STOCK WITH DESCR LIKE Pencil... AND (DESCR LIKE ...red OR
DESCR LIKE ...blue)
```

to

```
LIST STOCK WITH DESCR LIKE Pencil... AND (LIKE ...red ...blue)
```

3. Testing for empty fields

A field name in a selection clause that is not followed by an operator is assumed to be a test for a non-empty value. Our SALES file includes a pair of associated fields that we have not yet discussed. These are PAYMENT.DATE and PAYMENT. These hold the date on which the payment was made and the amount paid. By making these multivalued, we can allow customers to pay their bills in stages.

To see the payment data, type

```
LIST SALES @ PAID
```

Just as in earlier exercises, the @ symbol asks the query processor to show all the columns it would normally show by default plus the item identified by PAID. But what is PAID? Take a look at the dictionary.

PAID is the phrase that links the PAYMENT.DATE and PAYMENT fields as an association. By using this phrase name within the query, it gets expanded and we end up with a report that contains all the elements of the association. You can use any suitable phrase in this way.

Perhaps we want to list all the orders for which payments have been received. We can do this with a query of the form

```
LIST SALES WITH PAYMENT # ""
```

but this can be reduced to a much more natural language form of

```
LIST SALES WITH PAYMENT
```

Similarly, if we want to find the orders for which no payments have been made, we can use

```
LIST SALES WITH PAYMENT = ""
```

which can be reduced to

```
LIST SALES WITH NO PAYMENT
```

or

```
LIST SALES WITHOUT PAYMENT
```

Selection Against Multivalued Fields

If the field named in a WITH clause is multivalued, at least one of the values must match the test for the record to be included in the report. For example,

```
LIST SALES WITH ITEM = 001
```

The above query shows two records. One has only product 001, the other is for two products one of which is product 001.

Alternatively, we might want to select the records that include item 001 but show only the information for the selected item. We can do this with the WHEN keyword. For example,

```
LIST SALES WHEN ITEM = 001
```

This report filters out just the values within the associated fields for which the condition is met.

As a third alternative, we might want to select records where all of the values in a multivalued field satisfy the selection criteria. This can be done using the WITH EVERY construct. For example,

```
LIST SALES WITH EVERY ITEM = 001
```

This query restricts the report to orders where the only product number is 001.

Multiple WITH Clauses

Multiple conditions can also be included by using more than one WITH clause.

```
LIST STOCK WITH PRICE < 10 WITH QTY < 10
```

This query shows the items costing less that 10.00 for which we have fewer than 10 in stock. There is an implied AND between the two WITH clauses.

BEWARE: Multiple WITH clauses are handled differently in different multivalue products. Information style systems such as QM have an implied AND between the clauses but Pick style systems have an implied OR. QM has an OPTION setting to use an implied OR for Pick compatibility but the safest approach is always to include the AND or OR keyword.

Sampled Reports

Sometimes, particularly when testing reports against very large files, we want to process just a small proportion of the total data. Two methods exist to do this.

The FIRST keyword limits the report to a specific number of records that meet the selection criteria. If the number is omitted, ten records are reported.

Example

```
LIST STOCK FIRST 3 WITH DESCR LIKE Pencil...
```

This query produces a report of the pencils in stock but limits it to show at most three items. Note that, despite the FIRST clause coming before the other selection criteria, the record count limit imposed by FIRST is applied after other selection. This query finds the pencils and stops when it has found three. It does not look at the first three records in the file and return only the pencils.

The FIRST keyword has a (confusing) synonym, SAMPLE. Why is it confusing?....

The SAMPLED Keyword

The SAMPLED keyword limits the report by processing only every n'th record that meets the selection criteria. If the number is omitted, every tenth record is reported.

Example

```
LIST STOCK WITH QTY > 10 SAMPLED 3
```

This query shows every third record that has a quantity of greater than 10.

The filtering of both FIRST and SAMPLED is applied after selection but before sorting.

The Sort Clause

You may have noticed that the records in our reports so far do not appear to be in any particular order. In the absence of other instructions, the query processor simply works through the file group by group for best performance. The hashing process will mean that this is effectively a random sequence of records.

We can use a sort clause to specify the order in which the reported records should appear. In its most commonly used form it is introduced by the BY keyword.

```
LIST STOCK BY DESCR
```

This query produces a report sorted into alphabetical order of description. Similarly, we could sort on a numeric field such as PRICE

```
LIST STOCK BY PRICE
```

We can reverse the order by use of the BY.DSND keyword to perform a descending sort

```
LIST STOCK BY.DSND PRICE
```

The BY and BY.DSND clause use the format specification in the dictionary to determine how the comparison is performed. If the format specification defines the field as either left or text justified (L or T), the comparison process used in the sort examines characters from the left end of each item to determine their correct sequence. If the format specification defines the field as right justified (R) and the two items being compared are both numbers, a strict numeric comparison is performed. For a right aligned item where one or both values are not numbers, the comparison process adds leading spaces to the shorter item and then compares character by character from the left.

This distinction is very important and frequently leads to errors. The conversion applied to display dates usually results in a fixed number of characters so that it appears to make no difference whether the field is defined as left or right justified. However, a date field must be defined as right justified for the sort to work correctly. If a date field is defined as left justified, dates before 18 May 1995 (day 10000) are shown after dates later than that point. This is because a character by character comparison would treat 9999 (17 May 1995) as being after 10000.

A single query may contain multiple sort clauses. They are applied left to right, later sorts being applied where two or more records have the same values in the fields for earlier sorts.

```
LIST STOCK BY PRICE BY DESCR
```

This query shows the stock items in ascending price order and, within each group of items at the same price, sorts them alphabetically.

There is a common misconception that a query that uses BY or BY.DSND must use the SORT verb. Clearly this is not true because the examples above work correctly with LIST. The SORT verb is equivalent to LIST with a final BY @ID component. Thus the following two queries produce identical output

```
SORT STOCK BY PRICE
LIST STOCK BY PRICE BY @ID
```

Do not use SORT unless you actually want to sort by record id as it involves extra processing.

Sorting Multivalued Fields

The BY clause does not take any special action for fields containing value marks or subvalue marks. For example, listing the SALES file with a query such as

```
LIST SALES BY ITEM
```

produces a report in which the items are not correctly sorted because any record with multivalued data in the ITEM field treats this as a simple character string without giving any special attention to the value marks.

To resolve this problem we can use the BY.EXP keyword in place of BY to "explode" the multivalues, effectively applying the first law of normalisation to yield a fully normalised view of the data. A query such as

```
LIST SALES BY.EXP ITEM
```

produces a report with the product numbers correctly sorted.

Note how the query processor has made use of the associations as defined in the dictionary so that related fields are correctly paired up.

The BY.EXP.DSND keyword can be used for a descending exploded sort.

Great care should be taken to ensure that the data reported makes sense. The final column of this report shows the order total value, even though each order has been broken into multiple lines. Adding up the figures in this column gives a very healthy view of the order book!

A better field to report would be to omit SALE.VALUE and show only LINE.VALUE. We will discuss how we modify the displayed fields later.

The Display Clause

The display clause determines which fields will appear in the report and how they will be displayed.

In its simplest form, the display clause consists of a list of field names. These will appear in the report left to right in the order that they occur in the query sentence. The default view of the record id, defined by the @ID dictionary record, always appears as the first column of the report unless it is suppressed using the ID.SUP keyword.

Examples

```
LIST STOCK QTY DESCR
LIST CUSTOMERS NAME TELNO ID.SUP
```

The names used for displayed fields must be defined in the dictionary as D, I, A or S-type items.

The EVAL Keyword

Sometimes we wish to include a calculated value in a report for which there is not already a suitable dictionary I-type entry and which will probably not be needed again. We could create a new I-type item, produce the report and then remove the I-type item. Alternatively, we can use the EVAL keyword to introduce a temporary virtual attribute.

Example

```
LIST STOCK EVAL "PRICE * QTY"
```

When we use EVAL in this way, the evaluated expression takes its conversion code, format, single/multi-value flag and association from the dictionary entry for the first data item referenced. In this case, the PRICE item yields appropriate values for these. The expression itself is used as the column heading.

As another example, consider a task management system where we store the start and end dates of each task. If we wanted to know how long a task took, we might use a query such as

```
LIST TASKS EVAL "END - START"
```

Because END would be defined in the dictionary to have a date conversion, a task that took five days, for example, would show the result of this evaluated expression as 5 January 1968. This is hardly useful! We will discover later how we can supply alternatives for the conversion and other properties used to display the results.

The EVAL keyword can be used anywhere in a query sentence where a field name is required, not just in the display clause. For example,

```
LIST STOCK BY EVAL "PRICE * QTY"
```

We might even want to use the evaluated expression more than once:

```
LIST STOCK BY EVAL "PRICE * QTY" EVAL "PRICE * QTY"
```

This is clearly becoming unreadable. Instead, we can use the AS qualifier to EVAL to give a name to the calculated value and use this name later in the query to reference the same value for a second time.

```
LIST STOCK BY EVAL "PRICE * QTY" AS VAL VAL
```

When AS is used, the default column heading becomes the name assigned to the evaluated expression.

The Default Listing Phrase

You have probably been wondering how the query processor decides which fields (columns) to show in the report. 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.

Take a look at the dictionary of the SALES file and see how the @ phrase contains the names of the fields we want by default and also the ID.SUP keyword. We modified this phrase when we created the LINE.VALUE and SALE.VALUE I-types but we did not explore how this worked.

Where a report is directed to a printer by using the LPTR keyword discussed later, 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 us 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. Our demonstration file dictionaries do not have an @LPTR phrase.

Qualified Display Clauses

Pick style systems extend the display clause to allow some elements of record selection. This feature is disabled in QM by default but can be enabled with the OPTION command. When enabled, the name of a field to be reported may be followed by a relational, pattern match or sound match operator and a value.

```
LIST STOCK DESCR LIKE Pencil...
```

is equivalent to

```
LIST STOCK WITH DESCR LIKE Pencil... DESCR
```

where the WITH DESCR LIKE Pencil... is the selection clause and DESCR is the display clause

For multivalued fields, the qualified display clause is handled in the same way as the WHEN keyword. Thus

```
LIST SALES CUST ITEM = 001 QTY
```

is equivalent to

```
LIST SALES WHEN ITEM = 001 CUST ITEM QTY
```

The value part of a qualified display clause may contain special wildcard characters. A left square bracket at the start of the value in an equality comparison substitutes for any number of leading characters. Thus

```
LIST STOCK DESCR = [small
```

is equivalent to

```
LIST STOCK WITH DESCR LIKE ...small DESCR
```

A right square bracket at the end of the value substitutes for any number of trailing characters. Thus

```
LIST STOCK DESCR = Pencil]
```

is equivalent to

```
LIST STOCK WITH DESCR LIKE Pencil... DESCR
```

Both of the above can be used together. Thus

```
LIST STOCK DESCR = [80g]
```

is equivalent to

```
LIST STOCK WITH DESCR LIKE ...80g... DESCR
```

The caret (^) can be used anywhere in the value to substitute a single character. For example

```
LIST STOCK ITEM = 1^1 DESCR
```

shows parts 101, 111, 121, etc

Field Qualifiers

Any field in a display clause may be followed by one or more field qualifiers. These override the dictionary definition of how the data is to be displayed. Used with the EVAL keyword, these qualifiers can be used to replace the defaults taken from the first field in the expression.

CONV "*code*"	Applies an alternative conversion code. `LIST SALES DATE CONV "D2/"`
FMT "*spec*"	Applies an alternative format specification. `LIST STOCK DESCR FMT "20T"`
COL.HDG "*text*"	Specifies an alternative column heading. A multi-line heading may be produced by inserting 'L' (including the quotes) in the text. E.g. COL.HDG "Selling'L'Price". `LIST SALES DATE COL.HDG "Order'L'Date"`
SINGLE.VALUE SINGLEVALUED	Treat the field as single valued. This is frequently used with EVAL.
MULTI.VALUE MULTIVALUED	Treat the field as multi-valued. This is frequently used with EVAL.
ASSOC "*name*"	Treat the field as part of the named association. The quotes must be included otherwise the phrase defining the association will be substituted into the query.
ASSOC.WITH *name*	Treat the field as associated with the named field.
DISPLAY.LIKE *name*	Take on the display characteristics of the named field. This can be used with other field modifiers.

Arithmetic Operations

Any field in a display clause may be preceded by one of the keywords below to calculate values to be displayed at the end of the report.

TOTAL	Calculates the total of the reported values. `LIST SALES TOTAL SALE.VALUE`
AVERAGE, AVG	Calculates the average of the reported values. `LIST SALES AVG SALE.VALUE`

PERCENT, PCT, %	Calculates the percentage of the total that each value represents. By default, 2 decimal places are shown. This can be changed by following the keyword with a value in the range 0 to 5.
	`LIST SALES TOTAL SALE.VALUE PCT SALE.VALUE`
	`LIST SALES PCT 5 SALE.VALUE`

MAX	Reports the maximum value in the report column. Although normally used with numeric data, applied to non-numeric data the MAX keyword reports the "alphabetically last" value.
	`LIST SALES MAX SALE.VALUE`
	`LIST STOCK MAX DESCR`

MIN	Reports the minimum value in the report column. Although normally used with numeric data, applied to non-numeric data the MIN keyword reports the "alphabetically first" value.
	`LIST SALES MIN SALE.VALUE`
	`LIST STOCK MIN DESCR`

ENUM	Counts the values in the column. `LIST SALES ENUM ITEM`
CUMULATIVE	Maintains a running total value for the named field. `LIST SALES TOTAL SALE.VALUE CUMULATIVE SALE.VALUE`

The AVERAGE and MIN calculations normally include all the data values in the result. We might want to ignore empty (not zero) fields. This can be achieved by following the field name with the NO.NULLS keyword. For example,

```
LIST SALES MIN PAYMENT NO.NULLS
```

produces a blank as the minimum because there is an unpaid sale.

```
LIST SALES MIN PAYMENT NO.NULLS
```

shows the minimum value of the orders hat have been paid.

The GRAND.TOTAL keyword can be used to insert text at the left of the total line of the report. For example,

```
LIST SALES CUST ITEM QTY LINE.VALUE TOTAL SALE.VALUE
GRAND.TOTAL "Total"
```

This query simply inserts the text "Total" at the bottom left.

The grand total can be moved to a page of its own by including the P control token in single quotes inside the quoted grand total text.

```
LIST SALES CUST ITEM QTY LINE.VALUE TOTAL SALE.VALUE
GRAND.TOTAL "Total'P'"
```

Reporting Options

There are many options to modify the format of a report. These include setting page headings and footings, specifying margins and column spacing, and much more.

Suppression Keywords

These keywords allow us to suppress parts of a report. Many of them have synonyms for compatibility with other products. Only the preferred name is shown here.

HDR.SUP	By default the query processor displays the query as the page heading though this can be modified. This keyword suppresses display of the page heading completely.
COL.SUP	Suppress display of column headings.
COL.HDR.SUPP	Suppress both page and column headings.
COUNT.SUP	Suppress the count of records processed at the end of the report.
ID.SUP	The query processor automatically includes the default view of the record id (@ID) as the first column in the report. This keyword suppresses this column. It may be used because we do not want to see the record id, because we have an alternative definition of the record id, or because we want the id but not as the first column.
ID.ONLY	Suppresses use of the default listing phrase resulting in a display of only the record id.
DET.SUP	Suppress the details lines of the report, leaving just the totals.

Page Format Keywords

COL.SPCS *n*	Specifies the number of spaces to appear between each column of the report. By default, the query processor tries to insert four spaces but will reduce this if the report is too wide.
MARGIN *n*	Specifies the width of a left margin. This might be used to leave space for ring binding holes, etc.
DBL.SPC	Inserts a blank line between each record in the report.
NO.SPLIT	Starts a new page wherever a record will not entirely fit on the current page.
VERTICALLY	Produces a vertical format report with one field per line. Associated fields are still shown side by side. The query processor automatically uses vertical mode if the report is wider than the screen or printer. (Synonym VERT is easier to type!)
NO.PAGE	Suppresses pagination on reports to the screen.
NEW.PAGE	Each record starts on a new page.

Panned Reports

As mentioned above, the query processor will switch into vertical report mode if the fields listed in the display clause will not fit within the available line width. When sending the report to the terminal, the query processor allows selection of panning mode to retain the tabular report style. The user can move across the columns using the left and right cursor keys or the letters L and R.

In the I-types module, we added NAME and DESCR to the SALES dictionary. If you skipped that exercise, go back and do it now.

Ensure that you are using an 80 character wide display and then try this query:

```
LIST SALES SALE DATE CUST NAME ITEM DESCR QTY PRICE LINE.VALUE
SALE.VALUE PAYMENT.DATE PAYMENT ID.SUP
```

Note how the query processor has switched into vertical mode. The report is very hard to read.

Now add the PAN keyword to the end of the query. Remember to use the command editor rather than retyping the whole command.

```
LIST SALES SALE DATE CUST NAME ITEM DESCR QTY PRICE LINE.VALUE
SALE.VALUE PAYMENT.DATE PAYMENT ID.SUP PAN
```

The report now shows only fields up to the description. Use the right cursor key or R to move across the columns. You can then move back again using the left cursor key or L.

This report is panning the entire output. Sometime we might want the order number and date to remain fixed and just pan later columns. We can do this by moving the PAN keyword be be immediately after DATE.

```
LIST SALES SALE DATE CUST NAME ITEM DESCR QTY PRICE LINE.VALUE
SALE.VALUE PAYMENT.DATE PAYMENT ID.SUP PAN
```

If PAN appears either before or after all display clause elements, the entire report is panned. If it appears in the middle of the display clause, all fields named before it are fixed.

Scrolling

Sometimes we page through a report rapidly and the item we need disappears off the top of the screen before we realise. It would be nice to be able to page back through the report. We can do this by using the SCROLL keyword in our query. Try this with

```
LIST VOC SCROLL
```

You can page forwards through this report in the usual way but you will also find that the up and down cursor keys work. Alternatively, you can use N for the next page and P for the previous page. You can also go directly back to a page that has already been displayed by entering the page number and pressing the return key.

When the SCROLL keyword is used, the query processor maintains a copy of the report as it is generated so that you can page back through it. Looking at a previous page does not regenerate the output, it merely recalls the original displayed data.

Headings and Footings

Each page of a report can have a user defined heading and footing applied. These are specified using the HEADING and FOOTING keywords, each of which is followed by the quoted text to be used.

Example

```
LIST STOCK DESCR QTY HEADING "Stock Report"
```

The text supplied for a page heading or footing can include control tokens to insert data such as page number or to control the layout of the text. These control tokens must be enclosed in single quotes within the heading text.

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*.

Example

```
LIST STOCK DESCR QTY HEADING "'C'Stock Report'L'" FOOTING
"'LDG'Page 'S'"
```

Directing a Report to a Printer

If the query sentence includes the LPTR keyword, the report is directed to a printer instead of the terminal. Which printer is used depends on the way in which your system is set up. There is a detailed section on printing later. For now, all we need to know is that every QM session has 256 numbered print units (0 to 255) which represent different destinations for printed output. The application usually has no knowledge of where its printed output will go, only that it prints to a specific print unit. Each print unit might represent a physically different printer, different paper types in the same printer, different printer setting such as portrait or landscape mode.

Using numbered print units instead of specifying the actual destination in our application gives us greater flexibility. An application that prints sales orders, for example, might always send them to printer 6 but this could be a different printer for each user, a feature that is especially useful with auxiliary port printing where the output is sent to a printer connected to the user's own PC. The association between a print unit number and where the output goes is set using the SETPTR command, often from the LOGIN paragraph or other initialisation script.

Used alone, the LPTR keyword causes the query processor to send its output to print unit 0, the default print unit. Alternatively, the LPTR keyword may be followed by a print unit number.

Report Styles

The query processor treats a report as being constructed from seven types of line: 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 turns 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.

Exercise

Create a VOC record named MYSTYLE that contains

```
1: X
2: H=bright blue
3: T=bright red
```

then execute a query

```
LIST SALES CUST ITEM DESCR QTY PRICE LINE.VALUE TOTAL
SALE.VALUE STYLE MYSTYLE
```

Sequence Numbering

Sometimes we might want to produce a monthly report that can be filed as a continuously numbered set of pages that will aggregate to form an annual report. Normally, a query report starts with page 1. We can create a control record that maintains contiguous numbering over successive runs of the report. Note that this feature only operates in conjunction with the LPTR keyword to direct the report to a printer or file. It does not affect reports sent to the display.

For the purpose of this exercise, we will put this in the $ACC file that refers to our account directory as a directory file.

First, ensure that the control record we are going to use does not exist by typing

```
DELETE $ACC SQ
```

This command may report that the record is not found.

Now execute a query such as

```
LIST SALES PAGESEQ $ACC SQ LPTR 9
```

Then take a look at the results using SED

```
SED $HOLD P9
```

The report should begin at page 1. The PAGESEQ keyword has instructed the query processor to maintain a record of page numbering in the named item, in this case the $ACC file, record SQ. This will be created because it does not already exist.

Now repeat the query and look again at the file

```
LIST SALES PAGESEQ $ACC SQ LPTR 9
```

Then take a look at the results using SED

```
SED $HOLD P9
```

If all has gone to plan, this report starts at page 2. (Remember: To exit SED when you have done this, type Ctrl-X followed by C).

Delimited Reports

The query processor can produce reports in various special forms. The CSV keyword produces "comma separated variable" format output that can be read direectly into some other software such as Excel.

To see this in action, try

```
LIST STOCK CSV COUNT.SUP
```

Note how items that contain commas ave automatically been enclosed in quotes. The CSV keyword has an optional number following it to determine the rules to be applied. The query above is identical to

```
LIST STOCK CSV 1 COUNT.SUP
```

which applies the rules set out in the industry standard CSV specification. Alternatively,

```
LIST STOCK CSV 2 COUNT.SUP
```

quotes all non-null items except for numeric items that do not contain a comma.

For greater flexibility, the DELIMITER keyword allows alternative separators to be used

```
LIST STOCK DELIMITER "|" COUNT.SUP
```

Breakpoints

A breakpoint occurs when the value of a specified field changes. Breakpoints are used to trigger actions such as sub-totals, new page headings, etc. A breakpoint only makes sense if the field used to trigger a breakpoint is also used to sort the data in the report. The sort is not applied automatically because, as we have seen, there are four different modes of sorting in QM (BY, BY.DSND, BY.EXP, BY.EXP.DSND). At least one of these will always be present in a query using breakpoints.

A breakpoint is defined by including the BREAK.ON keyword before the field name. For example

```
LIST SALES BY CUST BREAK.ON CUST SALE.VALUE
```

Whenever a breakpoint occurs, a line is printed with asterisks in the column for the breakpoint field. This is particularly useful when a report uses multiple breakpoints.

Where the report includes arithmetic operations on fields (TOTAL, AVERAGE, etc), the value is calculated at each breakpoint to give sub-totals , etc as well as the grand total at the end of the report.

```
LIST SALES BY CUST BREAK.ON CUST TOTAL SALE.VALUE
```

(As you try these queries, don't forget that the command stack editor allows you to walk back to the previous command with the cursor up key and then modify this command. You do not need to retype the whole command).

The BREAK.ON keyword can also take qualifying text which will be printed instead of the default asterisks. The text will be truncated if it is wider than the field in which it is to appear.

```
LIST SALES BY CUST BREAK.ON "Total" CUST TOTAL SALE.VALUE
```

There is a simple but critical syntax difference between the default behaviour of QM (and other Information style multivalue products) and Pick style systems where the qualifying text goes after the field name:
flavours of QM regarding where this text appears.

```
LIST SALES BY CUST BREAK.ON CUST "Total" TOTAL SALE.VALUE
```

To ease migration, QM provides a mode of the OPTION command, PICK.BREAKPOINT, to support this syntax. All examples in this module use the default syntax.

Breakpoint Control Tokens

The breakpoint text may include control tokens in the form of uppercase letters enclosed in single quotes within the text string. Where more than one token is used, they can be enclosed in a single set of quotes or separately quoted.

As you try the example queries shown below, you only need to change the breakpoint control tokens and associated text. Although it will result in a slightly odd page heading on the queries that don't have a heading below, you can simplify the editing needed to get from

one command to the next and the feature of each control token will still be adequately demonstrated.

B{*n*} Start a new page, retaining the value of the breakpoint field for inclusion in the page heading/footing by use of the B heading text option. The optional single digit qualifier, *n*, allows collection of values from multiple breakpoints for inclusion in a composite heading. If omitted, the value of *n* defaults to zero. Thus use of B alone is equivalent to use of B0.

```
LIST SALES BY CUST BREAK.ON "Total'B'" CUST TOTAL
SALE.VALUE HEADING "Customer 'B'"
```

D Omit the subtotal line if there is only one line of detail for this breakpoint.

```
LIST SALES BY CUST BREAK.ON "Total'BD'" CUST TOTAL
SALE.VALUE HEADING "Customer 'B'"
```

L Emit a blank line in place of the breakpoint. Any text in the *options* string will be ignored.

```
LIST SALES BY CUST BREAK.ON "'L'" CUST TOTAL
SALE.VALUE
```

O Only show the value of the breakpoint field on the first detail line within the breakpoint.

```
LIST SALES BY CUST BREAK.ON "Total'LO'" CUST TOTAL
SALE.VALUE
```

P Start a new page. This differs from B in that it does not capture the value of the break point field for insertion into the page heading.

```
LIST SALES BY CUST BREAK.ON "Total'P'" CUST TOTAL
SALE.VALUE
```

U Pick style systems do not show the line of hyphens above subtotals. For ease of migration, the PICK.BREAKPOINT.U mode of the OPTION command can be used make QM behave like Pick systems. For reports where the underline is required to improve readability, the U breakpoint option can then be used to reinstate it. If the U option is used when the PICK.BREAKPOINT.U option is not active, it causes the underline to be suppressed.

```
LIST SALES BY CUST BREAK.ON "Total'U'" CUST TOTAL
SALE.VALUE
```

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.

```
LIST SALES BY CUST BREAK.ON "Total 'V'" CUST TOTAL
SALE.VALUE
```

(Note the space after the Total in the breakpoint control text this time)

The BREAK.SUP Keyword

Some of the queries that we have just done have the customer number in the page heading. It would be nice to perform the breakpoint actions on this field but not include this as a column in the report. We can do this by using BREAK.SUP in place of BREAK.ON in our queries. For example

```
LIST SALES BY CUST BREAK.SUP "'B'" CUST TOTAL SALE.VALUE
HEADING "Customer 'B'"
```

Suppressing the Grand Total Line

The grand total can be suppressed from a report by including NO.GRAND.TOTAL in our query.

LIST SALES BY CUST BREAK.ON "Total'B'" CUST TOTAL SALE.VALUE HEADING "Customer 'B'" NO.GRAND.TOTAL

Examination of the VOC shows that NO.GRAND.TOTAL is actually a phrase that expands to

```
GRAND.TOTAL "'L'"
```

This rather odd construct is what is needed on other multivalue products and is consistent with how control codes work. The additional VOC item on QM makes the query command more readable.

Summary of Display Clause Components

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

Each display clause element consists of an item from the middle column, optionally preceded by at most one item from the left column and followed by any number of items from the right column.

Breakpoint Exercise

See how close you can get to the report below.

```
Order Details for Customer 1002   Ross, Alan

Sale.  Product  Description.............   Price   Qty  Line Total
12002      013  Pencil, blue                0.28     2        0.56
           012  Pencil, red                 0.28     3        0.84

13013      001  Pen, black                  1.70     6       10.20

                                                      ----------
                                                           11.60




13 MAR 2009                                                Page 3
```

Each customer should start on a new page with the customer number in the page heading and a blank line under the heading.

The example above has the customer's name too. Can you work out how to do this?

The orders for each customer should appear in ascending sequence of order number.

For each order, include the order number, product numbers and descriptions, unit selling price, quantity and calculated line total value. Note that some column headings are not the defaults.

There is a blank line between each order.

Print a total for all orders placed by each customer.

The page footer includes the date and page number.

The grand total line should be on a page of its own.

Do not print the count of records processed at the end of the report.

You might need to add some virtual attributes to the SALES file dictionary if they have not been created in earlier exercises. Alternatively, you could use the EVAL keyword within your query sentence.

You can develop this query on the command line using the command stack editor or you can create a VOC sentence with each step on a separate line. We recommend that you build it up in easy stages rather than trying to do the whole query in one attempt.

Solution

This is one possible solution as it might appear as a VOC sentence:

```
0001: S
0002: LIST SALES BY CUST BY SALE_
0003: BREAK.SUP "'B'" EVAL "CUST:'
':TRANS(CUSTOMERS,CUST,NAME,'X')" _
0004: SALE ITEM COL.HDG "Product" _
0005: EVAL "TRANS(STOCK,ITEM,DESCR,'C')" FMT "25T" COL.HDG
"Description" _
0006: PRICE QTY TOTAL LINE.VALUE_
0007: GRAND.TOTAL "'P'" _
0008: HEADER "Order Details for Customer 'BL'" _
0009: FOOTER "'DG'Page 'S'" _
0010: DBL.SPC ID.SUP COUNT.SUP
```

Line 2: We are reporting the SALES file and must sort by customer number if we are going to break on this field.

Line 3: We want to break on change of customer number but, because we wish to cut both the customer number and his name into the heading, we actually perform the break on an evaluated expression which joins together the customer number, three spaces and his name (which we get from the CUSTOMERS file using TRANS).

Line 4: Display the order number and the product number (with a non-default column heading).

Line 5: Display the product description in a suitable field width and with a non-default heading.

Line 6: Display the selling price, quantity and calculated line value (using an I-type that we created earlier.

Line 7: Move the grand total to a page of its own.

Line 8: Set up a page heading using our breakpoint value.

Line 9: Set up a page footing.

Line 10: Insert a blank line between each record, suppress the default display of the order number (because we want to use SALE which has a better column heading) and suppress the record count at the end of the report.

Select Lists

Every QM session has eleven numbered **select lists** in which lists of record ids can be built up for subsequent processing. The lists are numbered from 0 to 10. List 0 is known as the default select list.

The query processor includes a SELECT verb which performs the selection phase of a query but saves the generated list of record ids without producing a report. The SELECT verb takes the same selection and sort clauses as LIST.

Many QM verbs, including all of the query processor verbs, check for an active default select list and, if found, use this to provide a list of items to be processed. Because of the dangers of unwanted effects if a list is left active by accident, the command prompt changes to :: when the default select list is active.

The SELECT verb constructs list 0 by default. An alternative list can be built using the TO clause to specify the target list number. Similarly, all query processor verbs have an optional FROM clause to specify the source list.

Examples

```
SELECT STOCK WITH PRICE > 5
LIST STOCK
```

This pair of commands is equivalent to

```
LIST STOCK WITH PRICE > 5
```

There are times when we might perform a SELECT against one file to generate a list of record ids to process in some other file.

```
SELECT STOCK WITH DESCR LIKE Pencil...
DELETE STOCK
```

This pair of commands deletes all the red items from our STOCK file. Don't do it!

The SSELECT verb constructs a select list in which the entries are sorted by record id after any other sorting is performed. Thus

```
SSELECT STOCK BY PRICE
```

is equivalent to

```
SELECT STOCK BY PRICE BY @ID
```

The SAVING Clause

We might want to build a select list by extracting data from a field other than the record id. For example, our SALES file contains the customer number for each order. This could be used to print out the customers' details.

We can do this using the SAVING clause of the SELECT verb:

```
SELECT SALES SAVING CUST
LIST CUSTOMERS
```

Try the above pair of commands. Is there anything wrong with the results?

Customer 1002 appears twice because there are two orders for this customer. We can eliminate duplicate entries in the select list by using the UNIQUE keyword in the SAVING clause:

```
SELECT SALES SAVING UNIQUE CUST
LIST CUSTOMERS
```

Sometimes we may build a list from a field that might be empty in some records. The null entries can be omitted from the select list by using the NO.NULLS keyword. Although it makes no sense with our demonstration data, we could do

```
SELECT SALES SAVING UNIQUE CUST NO.NULLS
```

Handling Empty Select Lists

Try the following pair of commands to print a list of customers who have placed orders for over 10.00

```
SELECT SALES WITH LINE.VALUE > "10" SAVING UNIQUE CUST
LIST CUSTOMERS
```

The report should show two customers. Try it again for orders of over 100.00, being sure to do both commands even if you see what is going wrong.

All of our customers are reported. Why did this happen?

The SELECT found no records meeting the specified criteria and hence did not build a list. The following LIST command, finding no active list, showed all the customers.

We can avoid this by using the REQUIRE.SELECT keyword in the LIST command. To see what this does, you need only type the LIST command:
LIST CUSTOMERS REQUIRE.SELECT

This might be very useful in automated reporting procedures.

Building Select Lists from Multivalued Fields

For historic reasons related to compatibility with other multivalue database systems, the SAVING clause does not normally treat multi-valued fields in any special way. The value marks are just part of the data. Hence a pair of commands such as

```
SELECT SALES SAVING UNIQUE ITEM
LIST STOCK
```

does not work. Only the first product in each order is reported.

The SAVING clause has a further optional element to resolve this problem

```
SELECT SALES SAVING UNIQUE MULTIVALUED ITEM
LIST STOCK
```

Removing Entries from a Select List

Sometimes we construct a select list and then want to remove from it all entries that correspond to record ids in some other file. We can do this with NSELECT.

Our demonstration database contains no suitable data to try this, however, we can use it to find the records in your VOC file that have been added since your account was created from the template VOC file, NEWVOC.

```
SELECT VOC
NSELECT NEWVOC
LIST VOC
```

Clearing Select Lists

Sometimes we construct a select list and then want to throw it away. Perhaps we realise that we used the wrong selection criteria.

The CLEARSELECT command can be used to discard any or all select lists.

`CLEARSELECT`	discards the default list, list 0.
`CLEARSELECT` *n*	discards list *n*
`CLEARSELECT ALL`	discards all active select lists

Saving Select Lists

The numbered select lists that we have been using so far are transient in nature. They are automatically discarded when they are used and also if we leave QM.

We might want to save a select list for later use, perhaps just a few moments later or maybe after many days.

Select lists can be saved in the $SAVEDLISTS file and later restored using the SAVE.LIST and GET.LIST commands.

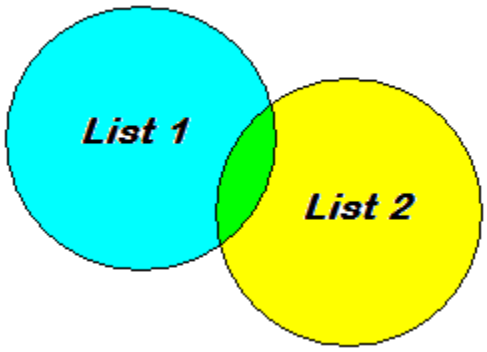
```
SAVE.LIST name {FROM n}
GET.LIST name {TO n}
```

Saving a list in this way destroys the numbered list from which it was saved. A named select list remains in the $SAVEDLISTS file until it is explicitly deleted. Although the QM DELETE command could be used to do this, there is also a DELETE.LIST command:

```
DELETE.LIST name
```

Merging Select Lists

Consider two separate select lists. There may be some overlap such that some records appear in both lists. This can be represented diagrammatically as:



We can construct a new list from these two in three different ways:

- The **union** contains all items in either list but without duplicating the items that are in both lists.
- The **intersection** contains those items that are in both lists.
- The **difference** contains those items in one list but not in the other. This could be those in list 1 but not in list 2 or vice versa.

QM allows us to merge either numbered or named lists.

For numbered select lists, we use the MERGE.LIST command:

```
MERGE.LIST list1 UNION list2 { TO list3 }
MERGE.LIST list1 INTERSECTION list2 { TO list3 }
MERGE.LIST list1 DIFFERENCE list2 { TO list3 }
```

For named lists, we use the LIST.UNION, LIST.INTER or LIST.DIFF commands:

```
LIST.UNION list1 { list2 { list3 }}
LIST.INTER list1 { list2 { list3 }}
LIST.DIFF list1 { list2 { list3 }}
```

For compatibility with other products, the second source list (*list2*) and the target list (*list3*) can be omitted from the command line in which case a prompt will appear.

Example

```
LIST.INTER REGION1.CLIENTS OVERDUE.ACCOUNTS OVERDUE.REGION1
```

The above command might take a list of clients in one business region and merge it with a list of overdue accounts to create a list of overdue accounts in that region.

The SEARCH Command

The SEARCH command allows us to build a select list by searching for records that contain (or do not contain) a specific character string.

The SEARCH command takes all the same selection and sorting clauses as a SELECT command but then prompts for one or more search strings.

By default, SEARCH builds a list of all records that contain any of the entered strings. The command is case sensitive by default but supports a NO.CASE option for case insensitivity.

The ALL.MATCH keyword can be used to select records that contain all of the search strings.

The NO.MATCH keyword can be used to select records that do not contain any of the search strings.

The SEARCH command takes no account of the field position at which the string is found. It is usually used for searching files holding text records. One particularly common use is searching program source text for references to specific items.

Example

```
SEARCH STOCK
STRING: Pencil
```

This would build a list of the STOCK records containing the word "Pencil".

Other Query Processor Verbs

COUNT

The COUNT verb processes a file to produce a count of records meeting the given selection criteria.

Examples

```
COUNT STOCK
COUNT VOC WITH TYPE = "V"
```

LIST.ITEM and SORT.ITEM

The LIST.ITEM verb shows the content of selected records as stored in the data file. It can be useful for tracking down problems where data has been incorrectly stored. SORT.ITEM is identical but sorts by record id after any other sort clauses.

Example

```
LIST.ITEM SALES '12003'
12003
001: 14402
002: 1001
003: 012y013
004: 5y20
005: 28y28
```

Note how the date field (field 1) has no conversion applied to it and how the multivalued fields are displayed with their embedded value marks.

LIST.LABEL and SORT.LABEL

The LIST.LABEL and SORT.LABEL commands allow easy generation of address labels, etc. They take all the same clauses as LIST.

These commands prompt for entry of the label page shape:

Labels per line - The number of labels across the page
Labels per column - The number of lines per label
Label width - The number of characters across a single label
Label height - The number of lines of text per label
Indentation of first column
Horizontal space between labels
Vertical space between labels
Omit blank lines? (Y/N)

The full description of these commands in the *QM Reference Manual* explains how the responses to these questions can be stored in the VOC for repeated use.

REFORMAT

The REFORMAT command takes data from a specified file (and perhaps from other files via virtual attributes) and creates a new file with fields in a specified order.

The name of the new file can be given on the command line with the TO option or, if absent, provided in response to a prompt.

The fields named on the command line are used to construct the records for the new file. The first field becomes the record id, the remaining fields are written as field 1 onwards in the order in which they are referenced.

Beware that the REFORMAT command writes the external format of the data as defined by the dictionary of the source file. A date field, for example, would have any conversion code specified in the dictionary applied to the transferred data. It would probably be necessary to include alternative dictionary definitions without conversion or formatting. Alternatively, field qualifiers such as CONV and FMT could be used.

Example

Our demonstration database gives little scope for a good example, however, the following shows the principles of REFORMAT:

```
REFORMAT SALES SALE EVAL "TRANS(CUSTOMERS, CUST, NAME, 'X')"
SALE.VALUE TO CUST.ORDERS
```

This example populates a new file named CUST.ORDERS (which must have been created previously) with the order number as the record id and two data fields, one holding the customer's name, the other holding the order value. Note that the order value will be in external format unless we use CONV to suppress the conversion.

SUM

The SUM verb takes a file name and one or more field names. It produces a report of the total of the specified field(s). A selection clause may be included.

```
SUM STOCK EVAL "PRICE * QTY"
```

11 Alternate Key Indices

The QM file system identifies records within a file by a unique record key. A record is retrieved by performing a mathematical transformation of the record key to the location in the file (the group number) where the record should be stored. This process is known as hashing and enables QM to retrieve a record with minimal searching.

The effect of hashing is to distribute records across the file in an even manner. Records written one after the other may be stored in widely separated parts of the file. There is also no grouping together of records that hold similar data. Although retrieval of a specific record using its record key will be very fast, it becomes difficult to access data based on some other field value.

Consider our SALES file but scaled up to a realistic size with, perhaps, 100,000 orders where 10,000 customers have each placed 10 orders. Because of the hashing process, if we want to find all the orders placed by a particular customer, we would need to read all 100,000 records, rejecting those that are for other customers. Clearly, as the file gets larger, this process becomes even more costly.

What we need is an index of orders for each customer. Using this, we can read one index record and then go directly to the SALES records of interest. The performance improvement using such an index is likely to be very large, often factors of tens of thousands or greater.

QM allows us to create an alternate key index on one or more fields defined in the dictionary. Once created and built, these indices are maintained completely automatically and used by the query processor completely automatically. The application requires no changes.

When a new record is added to the file, the appropriate index entries are added. When a record is deleted from a file, the index entries are deleted. When a record is modified, QM compares the new and old data to see if the indices are affected and makes only the necessary changes.

There is a small but significant cost in maintaining an alternate key index. The more indices we have, the greater the cost will be. However, the index can give us a substantial improvement in performance when retrieving records which usually far outweighs the costs.

We can create an index on any field defined in the dictionary. An index constructed on a multivalued field will have one entry for each value and hence may be very large.

Alternate Key Indices and Virtual Attributes

An alternate key index may be constructed on a calculated value from a I-type dictionary item. Because the index is only updated when the data record is written, there is an important rule to remember:

> **The virtual attribute expression must always return the same value when evaluated for the same input data**.

This will not be true if the expression uses the date (e.g. calculating an age from a date of birth) or if it uses the TRANS() function to fetch data from another file.

Creating and Building Alternate Key Indices

Historically, construction of an alternate key index within a multivalue database has always been a two stage process. First, the index must be created:

```
CREATE.INDEX filename field(s) { NO.NULLS }
```

The CREATE.INDEX command sets up the file structures to hold the index. The optional NO.NULLS keyword causes the index to omit entries for records in which the index field in empty. Creation of the index is very fast.

Once the index has been created, it can be built:

```
BUILD.INDEX filename field(s)
```

The keyword ALL can be used in place of field names to build all indices for the named file. The BUILD.INDEX command reads the entire data file and constructs the index entries. Because the data must not change during the process, BUILD.INDEX takes a lock on the file to hold off updates. For a large file, this may take some time and is best performed overnight or at other quiet times. When the build is complete, QM will use and maintain the index completely automatically.

QM extends the two stage create/build process found in other products by adding a single command, MAKE.INDEX, that combines the two operations into a single command.

New indices can be added to a file at any time. Indices can also be removed using the DELETE.INDEX command:

```
DELETE.INDEX filename field(s)
```

Again, the ALL keyword can be used to delete all indices.

Sometimes is it useful to add indices prior to major activities such as end of year processing and then remove them to avoid the cost of index updates for the rest of the year.

To be effective, an alternate key index should have a large number of indexed values, each of which leads to a small number of records. An index constructed on a simple yes/no field or other item with a very small number of possible values is a complete waste of time, partly because it is likely that reading the records referenced by either index entry will require every group to be read, and partly because the index record itself will be very large and costly to update.

We can check how well an index works using the LIST.INDEX command:

```
LIST.INDEX filename field(s) mode
```

Again, the ALL keyword is available. The mode may be:

omitted	displays a simple summary of indices in the file.
STATS	display statistics showing the number of indexed values and the minimum, maximum and average number of records referenced by each entry.
DETAIL	displays the above statistics together with details of each indexed value.

Where a field is described by multiple dictionary entries, perhaps with different conversion codes and format specifications, the alternate key index system recognises that they are the same data and will automatically share any index.

Alternate Key Indices and the Query Processor

The query processor uses alternate key indices automatically. This includes query processor commands, particularly SELECTs, executed from within application software.

There are, however, some important rules to be aware of. Consider a query with various combinations of conditions on indexed and non-indexed fields:

No selection clause	An index has no benefits for a query that has no selection clause and hence will not be used.
WITH *indexed*	A single selection condition on an indexed field will take advantage of the index to go directly to the relevant records.
WITH *indexed* OR *non-indexed*	This cannot use the index as the OR relationship requires the entire file to be processed for the second condition.
WITH *indexed* AND *non-indexed*	The index will be used to derive a list of records meeting the first condition. These records will be read, rejecting those that do not meet the second condition.
WITH *non-indexed* AND *indexed*	The query processor applies selection clauses left to right so this query would not use the index and hence would be very much slower than the previous example even though it might produce the same report.
WITH *indexed* AND *indexed*	A query only ever uses at most one index. The index for the first condition will be used and the second condition applied as records are processed. A query of this form will give best performance if the first condition

	leads to fewer records than the second condition.
NO.INDEX keyword present	No attempt will be made to use alternate key indices.
REQUIRE.INDEX keyword present	The query will be aborted with an error message if it cannot make use of an alternate key index.

Indices created using the NO.NULLS keyword have an extra complication. Consider a condition such as:

```
WITH field > 4
```

where *field* has an index which excludes NULL values. This query will use the index to identify the records meeting the given condition.

A condition such as

```
WITH field < 4
```

however, will not use the index and hence may be very slow.

Why doesn't this query use the index?

The condition (*field* < 4) is satisfied by a null field. Records with the field null are excluded from the index and hence would not be found if we used the index.

If we want to report the records with *field* < 4 but not null, the condition

```
WITH field # "" AND field < 4
```

would use the index as it is not satisfied by a null field.

Exercise

Create a virtual attribute named WORD in the dictionary of the STOCK file. This item will transform multi-word product descriptions such that each word is a separate value stored in uppercase. The dictionary item should read:

```
1: I
2: CONVERT(" ", @VM, UPCASE(DESCR))
3:
4: Word
5: 12L
6: M
```

Use the query

```
LIST STOCK WORD
```

to check that this virtual attribute works. Each word of the description should appear in uppercase on a separate line.

Use MAKE.INDEX to construct an index on this virtual attribute.

Use LIST.INDEX in all three modes to examine this index.

Try a query such as

```
LIST STOCK WITH WORD = "PEN"
```

to show all items where the description contains a particular word.

12 Paragraphs

We met the concept of sentences and paragraphs in the section of this course that discusses the VOC file. In this module, we will look at how we can use these to automate aspects of our application.

Sentences

A VOC sentence entry (S-type record) holds a single QM command which we can then execute simply by typing its name. A sentence has S as the first character of field 1 and the sentence itself in field 2.

Query commands are often very long. We can avoid continuously retyping queries that we use frequently by setting up our own sentences. With very long queries, it can simplify editing if the query sentence is broken over several lines (fields) in the VOC by ending each line other than the last with an underscore. When the sentence is executed, the lines are joined together, replacing each underscore with a single space.

Example

```
CUST.ORDERS
1: S
2: LIST SALES_
3: BY CUST_
4: BREAK.ON "'VB'" CUST_
5: TOTAL SALE.VALUE_
6: HEADING "Orders for customer 'B'"
```

In this example, each clause making up the query and each element of the display clause has been placed on a separate line. We can execute this query simply by typing its name (CUST.ORDERS).

Anything we type after the sentence name when we execute it is joined onto the end of the sentence text. This allows us to save partial sentences where the final components are added from the command line. Look at the EDIT.LIST sentence as an example.

We could add a sentence such as that below to list dictionaries

```
LD
1: S
2: LIST DICT
```

Paragraphs

Whereas a sentence is a single stored command, a paragraph is a series of commands. The whole series is executed simply by typing the paragraph name.

A paragraph has PA as the first two characters of field 1. The remaining lines (fields) are the sentences making up the paragraph. As with sentence entries, any individual sentence in the paragraph can be extended over multiple lines by ending each line except the last with an underscore.

Example

We might frequently need to produce a report of customers who have placed orders with us. This could be done using a simple paragraph:

```
CUST.RPT
1: PA
2: SELECT SALES SAVING UNIQUE CUST
3: LIST CUSTOMERS NAME
```

The sentences in a paragraph may include anything that could be executed from the command prompt, including references to VOC sentences and other paragraphs. There are also several extra constructs that we can use in paragraphs that we will meet in this module.

Comments

Paragraphs may be made more readable by inserting blank lines to separate sections of processing and by indenting commands that are part of the loop constructs described later. Even quite short paragraphs may be a total mystery to other users (or even the developer after a few weeks!). We can add comment lines to paragraphs to explain the processing and each maintenance.

A comment line starts with an asterisk as the first non-space character. This must be followed by at least one space. Any further text on the line is totally ignored (Well, almost! We will come to that later).

Our earlier customer report paragraph could have a helpful comment added:

```
CUST.RPT
1: PA
2: * List the names of customers who have placed orders
3: SELECT SALES SAVING UNIQUE CUST
4: LIST CUSTOMERS NAME
```

There is actually a slightly better way to do this example. The first line can have comment text following the PA type code without the need for an asterisk:

```
CUST.RPT
1: PA List the names of customers who have placed orders
3: SELECT SALES SAVING UNIQUE CUST
4: LIST CUSTOMERS NAME
```

Any lengthy paragraph should be liberally scattered with comments. An effective comment should explain what the subsequent sentences do and any necessary pre-conditions. Comments that just restate the sentence in English are virtually useless.

Inline Prompts

Paragraphs frequently need to obtain information from the user or elsewhere to form part of the executed commands. This is achieved using inline prompts.

The general form of an inline prompt is

```
<<control,text,option>>
```

where

control	determines when, where and how the prompt appears.
text	is the prompt text to be used.
option	specifies validation to be applied to the data.

There must be no spaces within the inline prompt structure unless they form part of the *control*, *text* or *option* components.

All three components are optional though either a *control* or *text* will always be present. There may be multiple *control* components. The inline prompt is processed left to right. All elements that can be interpreted as controls are use in that way. The next element is taken to be the text and any remaining item is the option.

The simplest inline prompt has only a text element. Edit your VOC to add the paragraph below:

```
MYPARA
1: PA
2: LIST <<File>>
```

Execute this paragraph by typing its name. It will prompt for a file name. Enter the name of one of your files (STOCK, SALES, CUSTOMERS, VOC, etc).

Notice how the data you entered in response to the prompt has been substituted for the inline prompt construct.

Modify your paragraph to read:

```
MYPARA
1: PA
2: SELECT <<File>>
3: LIST <<File>>
```

(OK, the SELECT serves no useful purpose but we are trying to show a feature of inline prompts).

Execute it again. Does it behave quite as you expected?

Wherever an inline prompt uses exactly the same prompt text as one that has already been executed in the paragraph, the prompt is not repeated. The previous value is simply substituted again.

We can make use of this feature to avoid repeated prompts from paragraphs that need to use the data several times.

Example

```
CUST.RPT
1: PA
2: SELECT SALES SAVING UNIQUE CUST_
3:   WITH DATE AFTER <<Start date>>
3: LIST CUSTOMERS NAME_
4:   HEADING "Clients ordering after <<Start date>>"
```

Notice here that the prompt text can contain spaces. Notice also that the inline prompt substitution occurs even though it is part of a quoted string.

Inline prompts are handled before any other command processing. This leads to a rather strange effect. A comment line that contains an inline prompt will perform the prompt and then examine the line, find it to be a comment and ignore it.

This turns out to be a useful feature because it allows us to use inline prompts in comments to collect all the prompt responses at the start of the paragraph before any processing.

We might, for example, have a paragraph that produces overnight reports:

```
OVN.REPORTS
1: PA
2: * <<Target file>>
3: RUN OVN.PROCESS
4: OVN.RPT <<Target file>>
```

The first program executed by this paragraph might run for several hours. We do not have to stay at our terminal waiting for the prompt on line 4 to appear.

We could also use this feature to ask for the prompt responses in a more logical order to that in which they are used within the paragraph.

Inline Prompt Controls

The control element of an inline prompt determines when, where and how the prompt appears. It may consist of multiple elements selected from those described below, separated by commas. There are restrictions about which elements may be used together but they are obvious from the functions of the controls.

Controlling the Terminal

The inline prompt may include any combination of the following control elements:

@(CLR) Clears the screen.

@(*col*,*line*) Moves the cursor to the given screen positions. The top left of the screen is 0,0.

@(BELL) Sounds the terminal "bell".

Example

<<@(CLR),@(5,2),Enter filename>>

This prompt clears the screen and positions the prompt text at column 5 of line 2.

Repeated Prompts

The R control element causes the prompt to be repeated until a blank response is entered. The inline prompt is substituted with a space separated list of the responses entered.

Example

```
LIST STOCK <<R,Part number>>
```

This will ask for a list of part numbers. When a blank response is entered, the LIST command is executed with the list of part numbers in place of the prompt construct.

Strictly, the record ids in the above example should be quoted in case they match the name of a VOC or dictionary item. We can achieve this using an extended form of the R control element. This has the character sequence to appear between each list entry in round brackets following the R. Thus we need to use a control element of R(' ').

This will insert the quotes between the list items but we also need quotes at either end. These are simply put around the inline prompt:

```
LIST STOCK '<<R(' '),Part number>>'
```

This may look like a very odd way to do things but it is a simple and very flexible mechanism. We can produce quoted lists, comma separated lists, etc.

Taking Data from the Command Line

Consider our first inline prompt example:

```
MYPARA
1: PA
2: LIST <<File>>
```

It would be nice if we could supply the filename on the command line by typing, for example:

```
MYPARA ORDERS
```

Although not really a prompt, the inline prompt system allows us to do this using the C control element. The command line is considered to be made up of a series of elements separated by spaces. The filename in this case is the second element on the command line and we can access this by changing our paragraph to use an inline prompt such as:

```
MYPARA
```

```
1: PA
2: LIST <<C2,File>>
```

Modify your paragraph to work this way and check that it works.

The prompt text can be omitted in this case, leaving just <<C2>>, however, it does aid readability of the paragraph and we may want to reuse this prompt later by name.

Of course, now if we type the paragraph name with no following filename we get an error because the second element if the command line is a null string leaving the command on line 2 of the paragraph without a filename.

Change your paragraph to read

```
MYPARA
1: PA
2: LIST <<I2,File>>
```

Now try it both with and without a filename to find out what the I control does.

There is a third variation as shown in the summary below:

C*n*	Take the *n*'th space separated element from the command that started the paragraph or sentence containing the inline prompt.
I*n*	As C*n* but if the result is null, prompt for the value.
S*n*	Take the *n*'th space separated element from the command line typed by the user. This is rarely of use.

QM extends the C*n* control code from its definition in other systems to allow

C*m*-*n*	Returns tokens *m* to *n*.
C*n*+	Returns tokens *n* onwards.
C#	Returns the number of tokens in the command line.

All formats of the C control code may include a default value. For example,

```
<<C4:SALES>>
```

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

Obtaining Values from a File

The inline prompt mechanism can also be used to fetch data from a record stored in a QM file. It can substitute the whole record, a field, a value or a subvalue. The corresponding control element formats are shown below.

F(*filename* ,*id*)	Substitute entire content of specified record
F(*filename* ,*id*,*fld*)	Substitute content field *fld* of specified record
F(*filename* ,*id*,*fld*,*val*)	Substitute content field *fld*, value *val* of specified record
F(*filename* ,*id*,*fld*,*val*,*subval*)	Substitute content field *fld*, value *val*, subvalue *subval* of specified record

Example

```
LIST STOCK @ EVAL "PRICE * <<F(VOC,TAX.RATE,2)>>"
```

where the VOC record is

```
TAX.RATE
1: X
2: .175
```

This example adds a tax column to the STOCK file report. The tax rate is taken from an X-type VOC record.

Use of Select Lists

An inline prompt can be used in a paragraph to process elements from a select list by use of the L control code.

L*n*	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.

Re-prompting for Known Text

There are times when we want to repeat a prompt even though we have already prompted for the information. The most common need for this is prompts that appear inside the loop constructs we will discuss later.

The A control element causes an inline prompt always to prompt, even if the text matches that of a previous prompt.

For example,

```
LIST <<A,File name>>
```

would always prompt, even if we had executed this prompt or another with the same text earlier.

Special Control Codes

SUBR(*name*)	Execute catalogued QMBasic function *name*, returning the result of this function as the value of the inline prompt.
SUBR(*name*, *arg1*, *arg2*)	Execute catalogued QMBasic function *name*, passing in the given arguments and returning the result of this function as the value of the inline prompt. Up to 254 arguments may be specified. These may be enclosed in quotes if necessary to avoid any syntactic ambiguity.
SYSTEM(*n*)	Returns the value of the QMBasic SYSTEM(*n*) function.
@*var*	The name of an @-variable, including user defined names (see the SET command), may be used to retrieve the value of the given variable. A default value may be applied by use of a prompt of the form: <<@*name*:*value*>> The default value will be applied if the prompt would otherwise return a null string.
$*var*	The name of an operating system environment variable may be used to retrieve the value of the given variable. A default value may be applied by use of a prompt of the form: <<$*name*:*value*>> The default value will be applied if the prompt would otherwise return a null string.

Examples

To save a select list with a name based on the user's QM user number:

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

To display the QM licence number:

```
DISPLAY Licence number <<SYSTEM(31)>>
```

Clearing Prompts

The CLEARPROMPTS command causes QM to discard all inline stored prompt responses.

Inline Prompt Options

The *option* element of an inline prompt allows some simple validation of the response entered by the user. If the response fails to pass the validation, the prompt is repeated. The error message produced is not very useful to an inexperienced user and cannot be modified.

Validation by Conversion

The *option* may be specified as a conversion code as used in dictionaries. This must be enclosed in round brackets.

For example,

```
LIST SALES WITH DATE AFTER <<Start date,(D)>>
```

QM attempts to convert the response entered at the prompt to internal format using the given conversion code, a date conversion in this example. If the conversion is successful, the original response entered (not the converted value) is substituted for the prompt construct.

Validation by Pattern Matching

The *option* may be specified as a pattern match expression as found in the query processor LIKE operator, the editor M search or the MATCHES programming language operator. The pattern expression is simply used as the *option* .

Example

```
LIST STOCK <<Part no,3N>>
```

Multiple alternative patterns may be given, separated by the word OR.

Example

```
LIST SALES <<LPTR for printer,'LPTR' OR ''>>
```

The above example accepts only LPTR or a null response to the prompt.

Aborting an Inline Prompt

Sometimes we want to abort a paragraph or sentence when it is asking for a reply to an inline prompt. We can do this by entering the word QUIT in uppercase.

The DATA Statement

Inline prompts allow us to get information to substitute into command lines in stored sentences and paragraphs. The DATA statement lets us pass data into commands that would normally prompt for user input. The two can be used together as we will see.

We might have a paragraph that runs ED to perform a simple scripted edit that modifies a edits a file to replace an embedded $$DATE$$ marker with the current internal form date:

```
1: PA
2: ED TFILE SCRIPT
3: DATA L $$DATE$$
4: DATA C/$$DATE$$/<<@DATE>>/
5: DATA FI
```

You may have as many DATA statements as you like following a command in a paragraph. Note how the second DATA statement in this example uses an inline prompt to get the internal form date.

The DATA statement is useful in paragraphs that run commands that would normally prompt for input and in phantom processes (which have no terminal for data entry).

You might think that the data should be queued up before the command to which it applies. So that the paragraph more closely represents the order in which things will happen, the command processor looks ahead for DATA statements as its executes each command in the paragraph, storing the results in a queue for use in subsequent input actions.

If you are a programmer, this queue is the same one that the QMBasic DATA statement can access.

As we have seen above, a DATA statement can include an inline prompt. The converse of this is that data queued using the DATA statement cannot be used to satisfy an inline prompt.

The CLEARDATA Command

The data from DATA statements remains in the queue until it is processed by an input request from some program or we return to the command prompt.

Consider a paragraph that runs a program to perform end of year processing. After the main processing is complete, the program asks if the content of the live data files should be copied to a set of archive files. Because in this use of the program the developer always wants to copy the data, he has included the response to the question in the paragraph.

```
1: PA
2: RUN EOY.PROCESSING
3: DATA Y
```

The paragraph will then go on to delete the certain records from the live data files. To make this step optional, the developer has let the DELETE command ask whether it should continue. So the complete paragraph becomes

```
1: PA
2: RUN EOY.PROCESSING
3: DATA Y
```

```
4: SELECT TRANSACTIONS WITH CODE = "7"
5: DELETE TRANSACTIONS
```

This looks like it should do what we want. But, what happens if the EOY.PROCESSING program fails before it has asked about archiving the data? The "Y" response to this question is still in the queue. The user is ready to enter N in response to the confirmation prompt for the DELETE, but it never appears as the queued "Y" is used instead. We have lost our data!

We need a way to prevent this sort of disaster from happening. The CLEARDATA command is one solution. This discards all queued data and should be included after any use of the DATA statement where the program might terminate without processing all of the queued data.

Our paragraph now becomes:

```
1: PA
2: RUN EOY.PROCESSING
3: DATA Y
4: CLEARDATA
5: SELECT TRANSACTIONS WITH CODE = "7"
6: DELETE TRANSACTIONS
```

Conditional Processing in Paragraphs

So far, our paragraphs have all started at the top and executed each command in turn until they arrive at the bottom. There are times when we need to include conditions around execution of one or more commands. We can do this with the IF statement.

```
IF condition THEN sentence
```

Unlike its programming language equivalent, the IF statement does not have an ELSE clause and can only condition a single sentence. We can work our way around these limitations very easily as we shall see.

The condition part of the IF statement uses a relational operator to compare two values. The values may be the results of inline prompts, constants or system supplied items known as @-variables.

Example

```
IF <<Print report?>> = "Y" THEN LIST ORDERS LPTR
```

Although all @-variables can be used in an IF statement, the most commonly used ones are:

@ABORT.CODE	Used in the ON.ABORT paragraph, this identifies why the process was aborted: 1 = application triggered abort, 2 = user requested abort, 3 = internal error detected.
@DATE	The current date in internal form.
@DAY	The day of the month (1 - 31).
@LOGNAME	The user's login name.
@MONTH	The current month (1 - 12).
@SYSTEM.RETURN.CODE	Most QM commands leave status information in this variable to allow paragraphs to check if the action was successful. The query processor verbs set this to the number of records processed.
@TIME	The current time in internal form.
@TERM.TYPE	The current terminal type (e.g. vt100).
@TTY	The terminal device name. There are several special values that may appear here. For example, in a phantom process this contains "phantom". See the *QM Reference Manual* for a complete list.
@USER0 to @USER4	User defined. Initially zero, these may be used by user applications in any way the designer wishes.
@USER.RETURN.CODE	User defined. Intended as a user application equivalent to @SYSTEM.RETURN.CODE, this variable is initially zero and is never changed by QM.

@USERNO	QM user number.
@WHO	The name of the current QM account.
@YEAR	The last two digits of the current year.

The relational operators are:

=	EQ	EQUAL		Equal to
#	NE	<>	NO	Not equal to
>	GT	GREATER	AFTER	Greater than
<	LT	LESS	BEFORE	Less than
>=	GE	=>		Greater than or equal to
<=	LE	=<		Less than or equal to

Example

Consider a paragraph to list all customers who have ordered a specific part:

```
1: PA
2: SELECT SALES WITH ITEM = <<Part>> SAVING UNIQUE CUST
3: LIST CUSTOMERS
```

What will happen if there are no records meeting the selection criteria? No list is constructed so the LIST command lists all of the customers.

One way around this would be to condition the LIST:

```
1: PA
2: SELECT SALES WITH ITEM = <<Part>> SAVING UNIQUE CUST
3: IF @SYSTEM.RETURN.CODE > 0 THEN LIST CUSTOMERS
```

Note carefully that inline prompts are expanded as the first step in processing a command. Thus a command such as

```
IF <<Produce report?>> = "Y" THEN LIST ORDERS <<LPTR for
printer, 'LPTR' OR ''>>
```

will ask both of the inline prompt questions before determining whether the repor is actually required.

Branching in Paragraphs

The GO statement allows us to jump around one or more sentences in a paragraph.

Consider a paragraph to tidy up the $HOLD file by deleting items over 30 days old. The DTM item in this example would be an I-type dictionary item that returned the date of the last modification to the item.

```
TIDY.UP
1: PA
2: SELECT $HOLD WITH DTM < EVAL "DATE() - 30"
3: DELETE $HOLD
4: DATA Y
```

Again, we have the possibility that no records are selected. We need to condition both the DELETE and the DATA statement. Although there are perhaps better ways to do this, we can use a conditional jump:

```
TIDY.UP
1: PA
2: SELECT $HOLD WITH DTM < EVAL "DATE() - 30"
3: IF @SYSTEM.RETURN.CODE < 1 THEN GO SKIP
4: DELETE $HOLD
5: DATA Y
6: SKIP:
```

The GO statement skips forward through the paragraph until it finds a line commencing with the specified label. The label must start in the first column of the line and has two possible formats. It may be entirely made of digits or, as is more usual, start with a letter followed by further letters, digits, dots and dollar signs. The label name must be followed by a colon.

If there is a sentence on the same line as the label, there must be at least one space after the colon.

Note that the action of GO is to skip forward to the label. The GO statement cannot be used to jump backwards.

By using IF and GO together we can construct the effects of an ELSE clause:

```
IF condition THEN GO LABEL1
   sentence(s)
   GO LABEL2
LABEL1:
   sentence(s)
LABEL2:
```

Loops

Paragraphs frequently contain sequences of sentences that we wish to execute repeatedly until some terminating condition occurs. The LOOP / REPEAT construct allows us to do this.

```
1: PA
2: LOOP
3:    IF <<A,Customer>> = "" THEN GO END
4:    LIST ORDERS WITH CUST = <<Customer>>
5: REPEAT
6: END:
```

This paragraph will ask repeatedly for customer numbers, displaying the orders for the given customer, until a blank response is entered.

Note the need to use the A control element in the first inline prompt. Without this, the paragraph would loop endlessly reporting the same customer.

The use of indentation as in the example above can aid readability of loops.

A single paragraph may contain many separate LOOP / REPEAT constructs and one loop may contain another.

Loop constructs can be used with the inline prompt L control code to process elements from a select list:

```
LOOP
IF <<L,ID>> = "" THEN STOP
MYPROG <<ID>>
REPEAT
```

Note how the L option always reads the next list entry. The A control code is not needed in this example.

Aborting Paragraphs

A paragraph ends when it arrives at the bottom. Jumping to the END label in the earlier example is a common way to terminate from earlier lines though QM supports the STOP command to terminate the paragraph in which it is executed.

When the paragraph terminates, control returns to whatever started the paragraph. This might be back to the command prompt or return to another paragraph, a menu, a program, etc.

The ABORT statement allows us to terminate all processing for the user, returning to the command prompt. It should only be used when error conditions are detected as it will also execute any ON.ABORT paragraph which might also terminate the QM session.

Example

```
1: PA
2: RUN OVERNIGHT.PROCESS
3: IF @USER.RETURN.CODE # 0 THEN ABORT
4: etc....
```

User Defined @-Variables

QM allows users to define their own @-variables that may allow a task that would require a program to be written in other multivalue systems to be achieved by a paragraph in QM. User defined @-variables can also be accessed by QMBasic programs via the !ATVAR() and !SETVAR() subroutines that are described in the *QM Reference Manual* .

A user defined @-variable is created using the SET command:

```
SET variable value
```

where *value* is the value to be stored. Quotes should not be used unless they are part of what is to be stored. Leading and trailing spaces in *value* are removed; embedded spaces are retained
For example,

```
SET COUNT 10
SET CLIENT George Thomas
```

The variable name may optionally have a leading @ character. The value can then be accessed using the inline prompt constructs referenced earlier in this section.

The SET command can also evaluate simple arithmetic expressions by use of EVAL.

```
SET COUNT EVAL <<@COUNT>> - 1
```

Combining this with other techniques we have seen in this module, we can construct a simple loop such as

```
SET COUNT 10
LOOP
   ...do something...
   SET COUNT EVAL <<@COUNT>> - 1
   IF @COUNT = 0 THEN STOP
REPEAT
```

Note that the IF command does not need the inline prompt structure as, unlike all other commands, it automatically substitutes @-variables.

13 Menus

A menu is a numbered list of options, each of which will execute some associated command.

QM includes a simple editor (MED) that allows very rapid construction of menus. To see this in operation, we will construct a menu that allows us to list the three files in our demonstration database. You may choose to extend this menu later to add a route into the programs constructed in later sections of this course.

To start the process of building a menu type

```
MED
```

You will be asked which file you would like to keep the menu in. We will put this in the VOC so enter VOC in response to the prompt or simply press the return key as this is the default file. Menus that are stored elsewhere would require an R-type VOC entry to link to them.

You now need to enter the menu name. We will use MYMENU. Answer Y in response to the prompt confirming that you wish to create a new menu.

The screen changes to show five lines that affect the entire menu:

Title	This is the title line to appear on the menu screen.
Subr	is the name of an optional subroutine that makes run time decisions about which options are to be displayed.
Prompt	is the prompt text to appear on the menu.
Exits	are codes that can be entered at the option prompt to return to a parent menu, paragraph, etc.
Stops	are codes that can be entered at the option prompt to abort the menu processor completely.

As you work through the steps below, notice how the short help message at the bottom of the screen changes from line to line. You can also press F1 for extended help.

Enter "Sales System Main Menu" on the title line and press the return key. The cursor moves down to the next item. If you make a mistake, you can always use the cursor keys to move back up a line and correct it.

We don't yet know how to write QMBasic subroutines so we will leave the Subr line empty. All users will see all options that we define on the menu. You can find out about access control subroutines later from the *QM Reference Manual* .

We will use the default prompt, exit and stop settings so these can also be left blank.

You should now be on the line of hyphens. Press return to create the first menu option.

Text is the text to appear on the menu. The option number is generated automatically. Enter "List the stock file" and press the return key.

Action is the command to be executed when this option is selected. Enter "LIST STOCK;" and press the return key. Note the semicolon after the command. This causes the menu

processor to display a continuation prompt before returning to the menu. We need this because we want the user to read the displayed report. Menu items that simply run some action that does not display data would usually not have the semicolon so that the user is returned to the menu with no intervening confirmation prompt.

Help is the optional one line help message to be displayed if the user enters an option number followed by a question mark. Put something appropriate in here and press return.

Because we are not using an access control subroutine, leave the Access and Hide fields blank.

Now go on to create options to list the CUSTOMERS and SALES files in the same way.

Once you have done this and you are on the line of hyphens after the final item, examine how your menu would look by pressing F4. Then press any key to return to the menu edit screen.

The key bindings of MED are based on those of SED. If you cannot remember them, the F1 help has a list. Enter Ctrl-X S to save the menu and Ctrl-X C to exit from MED. If you accidentally hit return too many times and have partially created option 4, this will be removed when the menu is filed.

Now that you are back at the command prompt, test your menu by typing MYMENU.

14 Printing

QM applications do not drive printing devices directly. Instead they reference numbered print units with no knowledge of where the output will actually go. This leads to a very flexible printing system where the output can be sent to a printer, a file or the user's screen. QM uses the underlying Print Manager on Windows or the operating system spooler on other platforms to perform output to printer devices.

Each QM session has its own pool of print units, numbered from 0 to 255. In most cases, if a print unit is not specified in a command, printer 0, the default printer, is used. Application developers are free to use these print units in any way that meets their needs. They might correspond to different printers, different paper types on the one printer, selection of portrait or landscape mode, etc. Although it is unlikely, all 256 print units can be used simultaneously.

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 may be executed from the MASTER.LOGIN paragraph in the QMSYS account (to affect all users), from the LOGIN paragraph of a specific account (to affect only users of that account), or from within the application.

The SETPTR command defines the shape of the printed page (width, depth, margins), the destination and various options relating to the treatment of the output.

```
SETPTR unit {, width, depth, top.margin, bottom.margin, mode
{, options }}
```

A print unit can operate in several modes:

Mode 1 directs the output to the underlying operating system print processor, usually to send it to a physical printer.

Mode 3 formats the data ready for printing but directs it to a record in the $HOLD file from where it may subsequently be viewed on the screen with a suitable editor or sent on to a printer when required. The hold file is most commonly used to defer printing until a process has completed, to gather diagnostic output, or for testing. The name of the file used by mode 3 can be set using the AS option of the SETPTR command which includes the ability to add a rotating sequence number for generate a different name for each output job. This sequence number can be determined using the QMBasic GETPU() function or the @SEQNO variable.

Mode 4 directs the output to the stdout (standard output) file unit.

Mode 5 buffers the data in the $HOLD file and then sends it to the terminal when the printer is closed, prefixing it with the control code to enable the terminal auxiliary 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 detailed information on printing, see the *QM Reference Manual* .

15 Introduction to QMBasic Programming

The QMBasic programming language used by the QM database system is easy to learn and allows very rapid development of application software. Although derived from the original BASIC language of the 1960's, it has many powerful features that fit in with today's database technology.

This course introduces the major features of the language. There are many more components that we do not cover. Programmers will be able to find these for themselves by examination of the *QM Reference Manual* .

The precise syntax and semantics of the Basic language used by the different multivalue database products varies. These training notes assume that you are using QM in its default modes. To find out more about the programming language compatibility options within QM, see the $MODE compiler directive in the *QM Reference Manual* .

Uses

QM provides a wide range of built-in tools for creating and processing data. There are times, however, when the standard tools will not meet our requirements and we need to develop our own programs. These might include:

- Application front end interfaces
- Batch processing activities
- Complex reports

Because QMBasic programs can be mixed with other QM concepts such as paragraphs and menus with no user visible difference in operation, complex applications can be built by choosing the most appropriate way to implement each area. Much of QM is itself written in QMBasic.

The QMBasic language provides a totally device independent method of handling terminals so that an application developed using one type of terminal device can work with other types of terminals with no change to the software.

Development Process

In this section we outline the process of developing a QMBasic program.

1. Entering the Source Program

The source form of a QMBasic program must be stored as a record in a directory type file. These files are represented by operating system directories and the records in them are represented by operating system files. Because we use this type of file, the program source may be created and maintained using any convenient text editor.

QM provides a full screen editor, SED, which is used by many programmers and has several features specifically for program development. The ED line editor can also be used for program development. The AccuTerm terminal emulator bundled with QM includes a

GUI editor, WED. Programmers can also use any underlying operating system editor such as vi on Linux or Notepad on Windows.

Whichever editor we choose to use, the application is built as one or more separate program modules. A very simple program may consist of only a single module. A typical database application will have several hundred or, perhaps, thousands of modules. Unlike most other programming languages, these modules are not brought together to form a single file holding the executable version of the application. Instead, this linking process takes place when the program is executed. This gives us a very flexible system, reduces memory requirements by only loading the parts of the application that are in use, and allows some degree of maintenance activity to an application whilst it is in use.

By convention, the source version of an application is often stored in a file named BP (Basic Programs) and this is the name used in all examples in this course. Any other name may be used but some commands assume BP by default.

2. Compiling the Program

Each module of the application must be compiled to convert it to its executable format. This is done using the BASIC command.

BASIC *file.name record.name* { *options* }

where

file.name	is the name of the QM file holding the program.
record.name	is the name of the program source record to be compiled.
options	are compilation options described in the *QM Reference Manual* .

The compiled form of the program is placed in a record of the same name as the source but in a separate file named by adding a .OUT suffix to the source file name.

3. Running the Program

A compiled QMBasic program is run by typing

RUN *file.name record.name*

where

file.name	is the name of the source file. The .OUT suffix is added automatically to access the compiled version of the program. If the file name is omitted, the BP file is used by default.
record.name	is the name of the program to be run.

Alternatively, a V-type (verb) VOC entry may be created to access the program as a QM command without the need to use the RUN command. This VOC item may be created using an editor or by use of the CATALOG verb discussed later.

Overview of Program Structure

All QMBasic programs have the same structure:

```
PROGRAM name
   program statements
END
```

The only lines allowed before the PROGRAM line or after the END are comments and blank lines. Typically, a program might have some descriptive text before the PROGRAM line.

The name given in the PROGRAM line must follow the rules of QMBasic variable names but is otherwise ignored. For ease of maintenance, it makes sense for this name to be the same as the name of the source program record.

The PROGRAM line is optional. It may be omitted or, for situations that we will meet later, may be replaced by a SUBROUTINE, FUNCTION or CLASS line.

Each QMBasic program statement normally appears on a separate line. Some statements have a multi-line syntax and any statement that has a comma in its syntax may start a new line immediately after the comma. A statement can also be split across lines at some arbitrary point by ending the line with a tilde (~) character.

Multiple statements can be written on a single line by separating them with semicolons.

Comments

A well written program will include many comments explaining to the reader how the program works.

A comment line may be written in one of three forms:

```
* comment text
! comment text
REM comment text
```

In each case, the comment text is totally ignored.

A comment may appear on the same line as a statement by use of the semicolon separator. For example,

```
TOTAL = QTY * PRICE   ;* Calculate total value
```

This is a good example of a totally pointless comment. It does not tell the reader anything that he could not see for himself.

Modularity

To ease maintenance, applications may be broken into separate program modules, each performing some part of the overall functionality. At run time, a program may use the CALL statement to call another program as an **external subroutine**.

Individual program modules can also contain **internal subroutines** to break the task of the program into smaller steps. These are entered using the GOSUB statement.

Both types of subroutines are discussed in detail in a later module of this course.

Very often, the same sequence of QMBasic statements is required in several modules. These might perhaps be statements that define the structure of a database file. Rather than repeat these statements in each module, which would increase the cost of maintenance if they had to be changed, we can use the concept of an **include record**. This is a program fragment that can be imported into the source of programs that need it during the compilation process by use of a line of the form

```
$INCLUDE {file.name} record.name
```

where

file.name	is the name of the file holding the record to be imported. If omitted, this defaults to the file holding the program being compiled and, if the item is not found, a second search is performed in the SYSCOM file that holds standard include records.
record.name	is the name of the record to be imported.

The program is compiled just as though the source lines contained in the include record had appeared in place of the $INCLUDE line. The include record itself is not compiled. For this reason, include records frequently have a recognisable suffix of .H added. The compiler is aware of this convention and a command sequence such as

```
SELECT BP
BASIC BP
```

would compile all records in the BP file except those ending .H (or .SCR which is used for screen definitions as discussed elsewhere).

One of the most frequent uses of include records is with **equated tokens**. These give names to constants such as field positions or limiting values.

```
EQUATE name TO value
```

Careful use of equated tokens can significantly reduce maintenance costs by requiring only a single line to be amended if the value is to be changed. It is also very easy to find references to a token name using the SEARCH verb.

15.1 QMBasic Language Elements

In this section, we will look at the building blocks from which a QMBasic program is constructed. We will return to examine each component in more detail in later modules.

Variables and Constants

A QMBasic program may contain many **variables** holding the data used by the program. Variable names must commence with a letter and may contain further letters, digits, periods (full stops), percentage signs and dollar signs. There is no practical limit to the length of a variable name in QMBasic. Names should be chosen to be meaningful. In general, names are case insensitive though case sensitivity can be selected via a compiler option.

Although there are few restrictions on the choice of names, it is advisable to avoid using names which correspond to QMBasic statements, functions and keywords.

Variables used in QMBasic programs do not have to be declared at the start of the program. The compiler determines what variables are required as it processes the program.

QMBasic variables are of variant type, that is, the type of data that they hold may change during execution of the program. There are many different data types, the type of a variable being determined by the value stored in it. The types include:

Unassigned	All local variables used by a program start as this type. Any attempt to use the value of the variable in an expression will give an error message.
Integer	Stores a whole number value.
Floating point	Stores a fractional value or a value that is too large to be represented as an integer.
String	Stores a character string (a series of characters). There is no practical limitation on the length of the string. The special case of a zero length string is known as a **null string**.
File	Holds control information for reference to a QM file. File variables are used in all file handling statements.

Type Conversion

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. This allows programmers largely to ignore the way in which data is stored internally.

If this arithmetic calculation is performed many times in a loop, it may be worth forcing a type conversion to prevent repeated temporary conversions. For this reason, programs often contain apparently redundant looking statements of the form

```
B = A + 0
```

to force a value to be converted to a number. Do not overuse this construct. It only provides a useful performance benefit if the value will be used many thousands of times.

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. They should not be enclosed in quotes.

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 three styles of string. Three styles of quote are provided because we may need to enclose one quoted string inside another, sometimes even enclosing this inside some third string.

The mark characters are available as @IM, @FM, @VM, @SM and @TM.

Scalars, Matrices and Dynamic Arrays

QMBasic provides support for both scalar and matrix variables. A **scalar variable** is a simple value referred to by its name alone. It may contain data of any type. A **matrix** is a one or two dimensional array of data items. Each element may be of a different type. Matrices are discussed in detail in a later module.

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 to operate on dynamic arrays. These include sorted and unsorted searching, insertion, deletion, replacement and extraction.

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.

Scope of Variables

By default, variables are local to the program module in which they appear but are shared by all internal subroutines. QM extends the Basic language found in other multivalue products to add the concept of a local subroutine having its own variables that have scope only within the subroutine. This concept is discussed later.

QMBasic also provides **common blocks** for data which is to be shared between two or more programs. These are discussed in a later module.

Expressions and Operators

An expression consists of one or more data items (constants or variables) linked by operators.

constant	Use constant value (string or numeric)
var	Use value of named variable
var[*s*,*n*]	Use *n* character substring starting at character *s* of the named variable
var[*n*]	Use last *n* characters of a string

var<*f*>	Use field *f* of a dynamic array variable
var<*f*, *v*>	Use field *f*, value *v* of a dynamic array variable
var<*f*, *v*, *s*>	Use field *f*, value *v*, subvalue *s* of a dynamic array variable
func(*args*)	Use the value of a function which may take arguments

In all cases above, *var* may be a matrix reference, for example

```
var(r, c)[s, n]
```

where r and c are expressions which evaluate to the desired matrix index values.

The substring extraction operation x[s, n] extracts n characters starting a character s of the string x. Character positions are numbered from one. Thus

```
A = "abcdefghijkl"
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 = "abcdefghijkl"
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 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

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* . Because this expression must return a value, the THEN and ELSE elements must both be present.

The boolean (true/false) values are such that values other than zero or a null string is treated as true. An expression returning a boolean value returns the integer value 1 for true, 0 for false. The boolean values are available as @TRUE and @FALSE for use in programs. A statement such as

```
OVERDUE = 1
```

may not be clear to someone reading the program that the variable is intended to hold a boolean value rather than just a number whereas

```
OVERDUE = @TRUE
```

makes the intention immediately clear.

The QMBasic operators 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.

< >	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 FMT() 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

The following alternative logical and relational operator formats may be used

<	LT		
>	GT		
=	EQ		
#	NE	<>	><
<=	LE	=<	#>
>=	GE	=>	#<

MATCHES	MATCH
AND	&
OR	!

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 expression

```
A<B> + 0
```

could be extracting field B of dynamic array A and forcing it to be stored as number by adding zero, or it could be testing whether A is less than B and the result is greater than 1. 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 relational operators (=, #, <, >, <= and >=) perform a numeric comparison if both items to which the operator is applied are numbers or can be converted to numbers. If either item cannot be treated as a number, a string comparison is performed, comparing characters from the left of the strings until their correct sequence can be determined.

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
*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. 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, *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 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.

Assignment Statements

Variables may be assigned values by statements of the following forms

var = *expr*	Assign *expr* to *var*
var[*s*, *n*] = *expr*	Assign *expr* to substring of *var*
var<*f*> = *expr*	Assign *expr* to field f of *var*
var<*f*, *v*> = *expr*	Assign *expr* to field *f*, value *v* of *var*
var<*f*, *v*, *s*> = *expr*	Assign *expr* to field *f*, value *v*, subvalue *s* of *var*

In all cases, *var* may be a dimensioned matrix element.

Except for the substring assignment, the operator shown as = in the above examples may be any of the following operators:

=	Simple assignment
+=	Add *expr* to original value
-=	Subtract *expr* from original value
:=	Concatenate *expr* as a string to original value

Substring assignment can only use the simple = operator and 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.

Z = "ABCDEFGHIJ"

Z[3,4] = "1234"	results in "AB1234GHIJ"
Z[3,2] = "1234"	results in "AB12EFGHIJ"
Z[3,4] = "12"	results in "AB12 GHIJ"

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.

A new field, value or subvalue may be appended to a dynamic array without knowing the existing number of items by an assignment of the form

Z<-1> = *expr*

Z<5,-1> = *expr*

Z<5,3,-1> = *expr*

@-Variables and Constants

The QMBasic language provides a number of special variables and constants with names prefixed by the @ character. Some @-variables can be updated by programs though most are read-only.

The @-variables are also available for use in I-type definitions or within paragraphs.

Compile-time Constants

These constants are available in 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

There are many of these. The most useful are shown below. Except where indicated, these items are read-only.

@COMMAND	The last command entered at the command prompt or initiated using the BASIC EXECUTE statement.
@CRTHIGH	Contains the number of lines per page of the display.
@CRTWIDE	Contains the width of the display.
@DATE	The internal format date value (days since 31 December 1967) at which the current command started execution.
@DAY	The day of the month at which the current command started execution as a two digit value.
@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 BASIC program.
@LOGNAME	User's login name.
@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 current command started execution as a two digit value.
@PATH	The pathname of the current account.
@RECORD	The data of the record being processed by an I-type function. This variable may be updated by a QMBasic program.
@SENTENCE	The currently active sentence. This is different from @COMMAND if the command runs a paragraph, sentence or menu.
@SYSTEM.RETURN.CODE	A status value returned from most commands.
@TERM.TYPE	Terminal type.
@TIME	The internal format time value (seconds since midnight) at which the current command started execution.
@TTY	Terminal device name.
@USERNO	User number.
@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.
@WHO	User's account name.
@YEAR	The last two digits of the year in which the current command started execution.
@YEAR4	The year in which the current command started execution.

Statements

The QMBasic language has many statements to perform such actions as terminal input/output, file handling, searching in dynamic arrays, etc.

A statement consists of a statement name followed by any appropriate qualifying information for that statement. There is a space between the statement name and the qualifying information.

Functions

A function returns a value which can be used in an expression. Most functions require one or more data items used to calculate their returned value. These appear in round brackets after the function name. Functions that do not require any input data must have a pair of empty brackets after the function name.

QMBasic has a very large collection of functions, covering mathematical operations, character string processing, etc. Users can also define their own functions as we shall see in a later module.

15.2 Terminal Handling

The QMBasic language provides a terminal input / output system which is independent of terminal device type. A program written using one terminal type can be run on any mixture of different terminal types. To support this system, QM has an underlying terminal definition library from which it can find the control sequences appropriate to any particular terminal type.

The DISPLAY Statement

The simplest way to output data to the user's terminal is via the DISPLAY statement. The CRT statement is identical to DISPLAY.

The DISPLAY statement names the item to be displayed. This may be a constant, variable or expression. After the item is output, the cursor is moved down to the start of the next line. If no display item is given, a blank line is output.

For example,

```
DISPLAY 'Outstanding Payment Details'
DISPLAY
DISPLAY 'Open invoices ' : NUM.OPEN
DISPLAY 'Overdue invoices ' : NUM.OVERDUE
```

The above statements would produce a display such as

```
Outstanding Payment Details

Open invoices 27
Overdue invoices 3
```

Here we have used a simple expression in the last two DISPLAY statements, concatenating a fixed text string to the variable holding the value to be shown.

The output could be improved by use of a **display list**. This consists of a series of items to be displayed, separated by commas. The comma advances the output to the next tab column across the screen. Tab columns are set at ten character intervals by default though this can be changed.

Our example becomes,

```
DISPLAY 'Outstanding Payment Details'
DISPLAY
DISPLAY 'Open invoices ', NUM.OPEN
DISPLAY 'Overdue invoices ', NUM.OVERDUE
```

The above statements would produce a display such as

```
Outstanding Payment Details

Open invoices       27
Overdue invoices    3
```

A DISPLAY statement that ends with a trailing colon suppresses the normal movement down to the start of the next line, leaving the cursor positioned after the final data character output. One of the lines from the previous example could become

```
DISPLAY 'Open invoices ':
DISPLAY NUM.OPEN
```

This feature is particularly useful when building complex display lines or when displaying a prompt for user input.

When the DISPLAY statement is used to output successive lines of data as in the preceding examples, QM maintains a count of the number of lines output. When a complete screen of data has been output a "Press any key to continue" prompt is displayed automatically. Programmers do not need to include any statements to perform their own line counting and continuation prompt handling.

Taking Control of Screen Layout

Instead of simply displaying successive lines down the screen, we may want to output a formatted screen. Here we need to take control of the cursor position.

Terminal devices allow the cursor to be moved to a specific point on the screen using cursor control sequences. Rather than writing these explicitly into our program (which would make it hard to move between terminal types), we use the @() function to look up the control sequence in the terminal definition library.

Our previous example now becomes,

```
DISPLAY @(0,0) : 'Outstanding Payment Details'
DISPLAY @(0,2) : 'Open invoices' : @(17,2) : NUM.OPEN
DISPLAY @(0,3) : 'Overdue invoices' : @(17,3) : NUM.OVERDUE
```

The @() function takes two arguments, the column and row position, both numbered from zero. The returned value of this function is the control string to move the cursor to the specified position. Although usually used in a DISPLAY statement, the result of the @() function is simply a character string which may be used in any way we wish.

The @() function can be used with just a column number. This positions the cursor to the given column on the current line. Use of this feature is not recommended as not all terminals have this capability.

First use of the @() function to perform cursor positioning within a program turns off QM's line counting. It becomes the programmer's responsibility to manage the screen layout and pagination. No "Press any key to continue" prompts will appear.

Terminal Control Functions

The @() function also provides access a large number of terminal control functions. These are identified by the first argument to the @() function being a negative value. Some of these functions require a second argument to qualify the action to be performed.

The most commonly used terminal control functions are:

@(-1)	Clear screen, leaving the cursor at the top left
@(-2)	Move the cursor to the top left
@(-3)	Clear the screen from the cursor to the end of the screen
@(-4)	Clear the current line from the cursor position onwards
@(-5)	Start flashing mode
@(-6)	End flashing mode
@(-11)	Start half intensity mode
@(-12)	End half intensity mode
@(-13)	Start reverse video mode (exchange foreground / background colours)
@(-14)	End reverse video mode
@(-15)	Start underlining
@(-16)	End underlining

Note how some of these functions operate in pairs to turn on or off a feature.

Not all terminals support all features available via the @() function. Where a terminal does not support a feature, the relevant @() functions return a null string and thus have no effect.

Example

```
DISPLAY @(-1) :
DISPLAY @(30,0) : @(-15) : 'Outstanding Invoices' : @(-16) :
```

These statements clear the screen and display the given text centered on the top line. The text will be underlined if the terminal supports this feature.

Note the use of the trailing colon to suppress unnecessary cursor movements.

The INPUT Statement

The INPUT statement performs input from the user's terminal.

The main elements of the syntax of this statement are:

```
INPUT var {, length { _ } } { : }
```

where

var	is the variable to receive the input data.
length	is the maximum number of characters allowed.

In its simplest form, specifying only the variable name

```
INPUT VAR
```

all data characters entered by the user are echoed back to the display and stored in the named variable until the user presses the return key. The return key itself is not stored but is echoed to the terminal causing the cursor to move to the start of the next line. The program continues with the next statement.

Adding the *length* component specifies a maximum number of characters to be entered.

```
INPUT VAR, 10
```

Fewer characters may be entered by terminating input with the return key. If *length* characters are entered, input is terminated as though the user had pressed the return key. This is likely to be difficult to use because the user needs to know that the return key must not be pressed if the maximum number of characters are entered. The most common use of this form is where the length is specified as one to catch a single keystroke.

The optional underscore after the length specifies that although input is to be limited to the given number of characters, the user must press the return key to terminate the input.

```
INPUT VAR, 10_
```

This time, any excess data entered at the keyboard is not echoed and not stored.

The optional trailing colon suppresses the cursor movement to the start of the next line. This is normally only needed when entering input on the bottom line of the screen where echoing of the return key would otherwise scroll the screen up by one line.

QM supports extended options in the INPUT statement:

```
INPUT VAR, 10_ HIDDEN
```

to echo asterisks in place of each character entered. This is useful for password entry.

```
INPUT VAR, 10_ UPCASE
```

converts input data to uppercase automatically.

There is also a timeout option that is described in the *QM Reference Manual* .

The INPUT @ Statement

The INPUT @ statement performs input from the user's terminal at a given position.

The syntax is

```
INPUT @(col, row) : var,  length { _ } { : }
```

where

col	is the column position for the input field.
row	is the row position for the input field.
var	is the variable to receive the input data.
length	is the maximum number of characters allowed.

The INPUT @ statement is a combination of a display and an input. If the colon after the screen position is present, the current content of the named variable is displayed at the given position in a field of length characters. The cursor is then positioned to the start of the displayed data for entry of new data to be stored in the variable.

As soon as the first character is entered, the original data is cleared from the screen and the new data is displayed in its place. When the return key is pressed, input terminates and the program continues at the next statement.

If the user simply presses the return key without entering any other data characters, the original value of the variable is retained.

This statement allows programmers to offer the user a default value which may be changed by entering new text or retained by pressing the return key.

The INPUT @ statement supports the HIDDEN and UPCASE options described for INPUT plus

APPEND	Position the cursor at the end of the data. Use of this keyword also implies EDIT mode.
EDIT	Starts in "edit" mode, suppressing the normal clearance of the input field if the first character entered by the user is a data character rather than an edit character.
OVERLAY	Starts in "overlay" mode where data entered by the user replaces the character under the cursor rather than being inserted.
PANNING	Allows entry of an unlimited number of characters in a field width of the given length by panning the data if it is longer than the display width. Use of this option requires length to be specified and implies the presence of the underscore.

The PROMPT Statement

Whenever a program requests terminal input using INPUT or INPUT @, a prompt character is displayed to the left of the input field. By default, this prompt is a question mark but it can be changed using the PROMPT statement.

```
PROMPT char
```

or

```
PROMPT ''
```

The first form sets the prompt character to *char*. The second form suppresses display of a prompt character.

The INPUTERR Statement

The INPUTERR statement displays a message on the bottom line of the screen. This message will be removed automatically when the return key is pressed to terminate a subsequent INPUT @ operation.

```
INPUTERR  text
```

The *text* must not include any cursor movement functions.

The INPUTERR statement is intended for displaying error messages in input validation but can be used in any way the programmer finds useful.

Exercise

If it does not already exist, create a BP file in your account by typing

```
CREATE.FILE BP DIRECTORY
```

Create the QM demonstration database by typing

```
SETUP.DEMO
```

Use a suitable editor such as SED or ED to create a program named ORDERS in your BP file. This program is the starting point for the exercises that follow.

This first version should simply display a screen similar to the one shown below and allow the user to enter data for the order number and action fields. Leave two blank lines under the customer number as we will insert additional data here later. There should be five lines available for product details above the order total line. Position the Action prompt text at least one line above the bottom of the screen to allow for use of INPUTERR.

```
                          ORDER PROCESSING

 Order No:          Date:

 Customer No:


 Part   Description                      Price     Qty
Total




                                             Order Total:


 Action(F/D/X):
```

Construct your program so that it does all the DISPLAY operations to paint the fixed text and then does two INPUT operations, one for the order number and one for the action prompt.

Use meaningful names for your variables. As you develop this program further, you will be using variables to hold items that have been read from files. It is often useful where a data item corresponds to a field in a file to use the same name as in the dictionary. This helps to make your program easier to understand.

For this initial stage of your program, think of a suitable name for the order number (we have used ORDER.NO in the worked examples). Order numbers are five digits so allow five characters to be entered.

The action prompt does not correspond to a data item stored in a file. Think of a suitable name for this item in your program. This prompt will be developed further in later exercises. Allow up to three characters to be entered as we will add new features here later.

Compile your program using

```
BASIC BP ORDERS
```

and, assuming that you have no errors, run your program using

```
RUN ORDERS
```

to check that it works.

Suggested solution

Your program may be different and will probably continue to diverge from our examples as you work through the exercises.

```
PROGRAM ORDERS

   PROMPT ''

   * Display fixed parts of screen

   DISPLAY @(-1)
   DISPLAY @(32,0) : 'ORDER PROCESSING'
   DISPLAY @(0,3)  : 'Order No:'
   DISPLAY @(20,3) : 'Date:'
   DISPLAY @(0,5)  : 'Customer No:'
   DISPLAY @(5,8)  : 'Part'
   DISPLAY @(11,8) : 'Description'
   DISPLAY @(48,8) : 'Price'
   DISPLAY @(56,8) : 'Qty'
   DISPLAY @(65,8) : 'Total'
   DISPLAY @(48,14) : 'Order Total:'
   DISPLAY @(0,20) : 'Action (F/D/X):'

   * Perform input

   INPUT @(10,3) : ORDER.NO, 5_
   INPUT @(16,20) : ACTION,3_
END
```

15.3 Conditional Execution and Looping

So far, our program has started at the top and worked its way statement by statement to the bottom. In this module we will learn how to alter the flow through a program by adding conditional statements that determine whether part of our program is to be executed and loops to repeat part of the program several times.

Boolean Values

All conditional actions require a true / false value (a **boolean** value) to determine how they will behave. In QMBasic the value for true is 1 and the value for false is zero. Any action that produces a boolean result will give one of these two values.

When testing a boolean value, QMBasic treats zero and a null string as false and all other values as true. This rather wider definition can be of great use in simplifying our programs.

The relational operators (=, #, <, >, <= and >=) all return boolean values showing the outcome of the test performed. Remember that these operators work by performing a numeric comparison if both data items can be treated as numbers and a character by character string comparison if either cannot be treated as a number.

A = 99
B = 100
The relational test A > B would return false (0)

A = '99A'
B = 100
The relational test A > B would return true (1)

The MATCHES operator tests whether a character string has a structure that matches a template, returning a boolean result. The template consists of any combination of the components below:

...	Zero or more characters of any 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.

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 "".

Example

```
REG = 'N785HJG'
```

The expression

```
REG MATCHES '1A1-3N3A'
```

would return true (1).

The pattern string may contain alternative templates separated by value marks.

```
PATTERN = "'V'1-3N3A" : @VM : "'W'1-3N3A"
V.OR.W = REG MATCHES PATTERN
```

The MATCHES operator tries each template in turn until one is a successful match against the string.

There are many functions that return a boolean result. Two very useful ones are

ALPHA(*str*)	Test if *str* is entirely alphabetic.
NUM(*str*)	Test if *str* can be treated as a number. Note that a null string returns true as it is generally treated as zero.

The NOT() function takes a boolean value as its argument and returns the inverse. Thus the expression

```
NOT(PART.NO MATCHES '3N')
```

tests whether PART.NO is not three digits. Think carefully about how this differs from

```
PART.NO MATCHES '~3N'
```

Boolean values can be combined using the AND and OR operators. For example

```
OK = PART.NO MATCHES '3N' AND PART.NO > 100
```

or

```
DONE = PART.NO = '' OR PART.NO = 'Q'
```

The IF Statement

The IF statement allows conditional execution. The general form of this statement is

```
IF condition THEN statement.1 ELSE statement.2
```

where

condition	evaluates to a boolean value.
statement.1	is the statement to be executed if the condition is true.
statement.2	is the statement to be executed if the condition is false.

The ELSE clause is optional and can be omitted if no action is to be performed if the condition is not true.

Either or both of the THEN or ELSE clauses can start on a new line.

Examples

```
IF NOT(PART.NO MATCHES '3N') THEN INPUTERR 'Bad part no'

IF DUE.DATE < DATE() THEN DISPLAY 'Overdue'
ELSE DISPLAY 'On loan'

IF ITEM.ID[1,1] = '0' THEN DISPLAY 'Sundries'
ELSE IF ITEM.ID[1,1] = '1' THEN DISPLAY 'Furniture'
ELSE DISPLAY 'Other'
```

Note how the ELSE clause of the first IF in the final example contains a second IF statement.

Often we want many statements to be controlled by the same condition. There is an alternative form of the IF statement that allows this.

```
IF condition THEN
   statements
END ELSE
   statements
END
```

Here the entire first block of statements will be executed if the condition is true, the second block of statements if the condition is false. Again, the THEN and ELSE keywords can start a new line.

The conditioned block of statements may contain any BASIC constructs including further IF statements.

The CASE Statement

The CASE statement provides a means to execute one of a series of blocks of conditioned statements.

```
BEGIN CASE
   CASE condition.1
      statements
   CASE condition.2
      statements
   CASE condition.3
      statements
END CASE
```

The CASE statement tests each condition in turn until one is found that is true. The statements for that condition are executed and then the program continues at the statement following the END CASE.

If more than one of the conditions is true, only the statements controlled by the first true condition will be executed.

If none of the conditions is true, none of the conditioned statements is executed.

We frequently want some statements to be executed if none of the conditions is true. QMBasic does not provide a special method to do this. Instead, we simply add a final CASE which will always be true. By convention, this is written as CASE 1 because 1 is our value for true.

Using a CASE statement, one of the earlier examples becomes

```
BEGIN CASE
   CASE ITEM.ID[1,1] = '0'
      DISPLAY 'Sundries'
   CASE ITEM.ID[1,1] = '1'
      DISPLAY 'Furniture'
   CASE 1
      DISPLAY 'Other'
END CASE
```

By use of the semicolon separator to place multiple statements on a single line, this can be made more readable:

```
CLASS = ITEM.ID[1,1]
BEGIN CASE
   CASE CLASS = '0'  ; DISPLAY 'Sundries'
   CASE CLASS = '1'  ; DISPLAY 'Furniture'
   CASE CLASS = '2'  ; DISPLAY 'Paper'
   CASE CLASS = '3'  ; DISPLAY 'Binders'
   CASE CLASS = '4'  ; DISPLAY 'Equipment'
   CASE 1            ; DISPLAY 'Other'
END CASE
```

In the above example, we have added further item classifications. Using the first form where each CASE element extracts the first character of the ITEM.ID and tests it against a constant becomes inefficient with many possible values. Instead, we have extracted the product class just once and stored it in a temporary variable which is then used in the

CASE elements. As a general rule, if a value extracted using a substring or dynamic array reference will be used more than once, it is better to create a temporary variable as in this example.

The LOOP / REPEAT Construct

The simplest way to repeat a group of BASIC statements is to enclose them in a LOOP / REPEAT construct.

```
LOOP
   statements
REPEAT
```

The statements inside the loop will be repeated until something causes the program to terminate or we exit from the loop.

The most common way to exit from the loop is to use either WHILE or UNTIL.

```
LOOP
   statements
WHILE condition
   statements
REPEAT
```

The WHILE statement causes the program to exit from the loop if the condition is false. When this occurs, execution continues at the statement following the REPEAT.

```
LOOP
   statements
UNTIL condition
   statements
REPEAT
```

The UNTIL statement causes the program to exit from the loop if the condition is true. Note how with both WHILE and UNTIL the condition test may appear at any point within the loop structure and is performed at that point.

A simple input data validation loop might appear as

```
LOOP
   INPUT @(10,4) : ITEM.ID, 3_:
UNTIL ITEM.ID MATCHES '3N'
   INPUTERR 'Item code must be three digits'
REPEAT
```

A single loop may contain multiple WHILE or UNTIL statements.

Where a program has loops inside other loops, the WHILE or UNTIL controls exit from the innermost loop. Sometimes it is necessary to include the same test in the outer loop(s) to "walk out" of the loops one by one when some condition occurs.

Counted Loops - The FOR / NEXT Construct

The FOR / NEXT construct allows us to repeat a groups of statements some given number of times by use of a control variable.

```
FOR var = start.expr TO end.expr  { STEP incr.expr }
   statements
NEXT var
```

where

var	is the control variable to be used.
start.expr	evaluates to the first value for *var*.
end.expr	evaluates to the final value for *var*.
incr.expr	evaluates to the value by which *var* is to be incremented for each cycle of the loop. If omitted, a value of 1 is used.

The statements within the loop are executed for successive values of *var* from *start.expr* to *end.expr* . The loop terminates when the next cycle would be for a value greater than *end.expr* .

If *incr.expr* is a negative value, the loop counts backwards and terminates when the next cycle would be for a value less than *end.expr* .

The control variable may be used within the loop but should not be overwritten.

The program should not rely on the value of the control variable on leaving the loop.

Examples

```
FOR I = 1 TO 10
   CRT I
NEXT I
```

The above loop simply displays the numbers 1 to 10.

```
FOR I = 1 TO 5
   INPUT @(16, 5+I) PROD.NO, 3_:
UNTIL PROD.NO = ''
   processing statements
NEXT I
```

This loop shows the use of UNTIL in a counted loop to exit before the count has reached its limiting value. Note also how the control variable is used to calculate the screen position for the input.

The EXIT Statement

The EXIT statement provides another way to exit from a LOOP / REPEAT or from a FOR / NEXT construct.

The above example could be written as

```
FOR I = 1 TO 5
   INPUT @(16, 5+I) PROD.NO, 3_:
   IF PROD.NO = '' THEN EXIT
   processing statements
NEXT I
```

Although, in this case, the UNTIL is probably neater, there are situations that we will meet in later modules when the syntax of the conditional test is such that we would have to use EXIT.

The STOP Statement

The STOP statement terminates the current command, returning to the menu, paragraph, program or command prompt from which it was started.

```
STOP { message }
```

The optional *message* is displayed on the user's terminal.

The STOP statement is intended for termination of programs on successful completion or in the event of minor errors.

The ABORT Statement

The ABORT statement terminates all current activity (programs, menus, paragraphs, etc) and returns to the command prompt. Before this prompt is displayed, QM checks for an executable VOC item named ON.ABORT and, if this is present, executes it.

```
ABORT { message }
```

The optional *message* is displayed on the user's terminal.

The ABORT statement is intended for termination of programs when a major error is detected, for example, when a critical file cannot be opened.

Internal Subroutines

A BASIC program becomes more difficult to understand and maintain as it gets larger. We can use subroutines to break the task of the application into manageable sized pieces.

QM applications make use of two types of subroutine. An **internal subroutine** is appears in the same program module (source record) as the calls to it. An **external subroutine** is a totally separate program module. Typical applications use external subroutines to represent the major functional areas of the application and internal subroutines to break the task of the modules into small steps. We will discuss external subroutines in detail later.

An internal subroutine is simply a series of statements within the program. The first statement within the subroutine has a **label** to allow us to refer to it in a GOSUB statement elsewhere in the module. The subroutine will terminate when it executes a RETURN statement to return to the statement following the GOSUB.

We use internal subroutines either because the same sequence of statements needs to be executed at several places in the flow through the logic of the program or to make the program easier to read and maintain by breaking the task into small steps. Calling an internal subroutine is very fast.

Unlike most other programming languages, internal subroutines do not have their own local variables. The variables used in a program module are common to all statements within that module. We will discuss some extensions provided in QM later that do allow local variables.

Labels

The first statement of an internal subroutine must be labelled so that we can refer to it elsewhere. A label consists of a letter followed by further letters, digits, full stops, dollar signs or percent signs. The label must be terminated by a colon.

Label names should be chosen to reflect the role of the subroutine. For example, in our ORDERS program, we could put all of the processing for the action prompt into a subroutine. This could be named PROCESS.ACTION or some other meaningful name.

Alternatively, for historic reasons, QMBasic allows numeric labels. These start with a digit and may contain only digits and full stops. The trailing colon is optional but makes it easy to locate the subroutine with the editor. Numeric subroutines names are not recommended as they impart no information about the purpose of the subroutine.

The program statements to paint the fixed parts of the screen in our orders program could be made into a subroutine as shown below.

```
* ============================================================
* Paint fixed part of screen

PAINT.SCREEN:
   DISPLAY @(-1) :
   DISPLAY @(32,0) : 'ORDER PROCESSING' :
   DISPLAY @(0,3)  : 'Order No:' :
   DISPLAY @(20,3) : 'Date:' :
   DISPLAY @(0,5)  : 'Customer No:' :
   DISPLAY @(5,8)  : 'Part' :
   DISPLAY @(11,8) : 'Description' :
   DISPLAY @(48,8) : 'Price' :
   DISPLAY @(56,8) : 'Qty' :
   DISPLAY @(65,8) : 'Total' :
   DISPLAY @(48,14) : 'Order Total:'
   RETURN
```

Including the bar across the page and a brief comment at the head of the subroutine aids readability of the program.

The GOSUB Statement

The above example subroutine could be called from elsewhere in the same program module by a statement such as

```
GOSUB PAINT.SCREEN
```

The program continues execution at the first statement of the subroutine. When the RETURN statement is executed, the subroutine exits back to the statement following the GOSUB.

Internal subroutines may call other internal subroutines. There is a limit of 256 on the depth of calls; a limit far higher than any well designed program would need.

The ON GOSUB Statement

Sometimes we wish to call one of a list of subroutines depending on the value in a variable. The ON GOSUB statement can be used to do this.

```
INPUT FLD
ON FLD GOSUB SUBR1,
             SUBR2,
             SUBR3,
             SUBR4
```

The ON GOSUB statement calls the first subroutine (SUBR1) if FLD is 1, the second if FLD is 2, and so on. A real program would include some validation of the input value.

Exercise

Extend your ORDERS program to add the following functionality.

Use a CASE statement to process the data entered at the order number prompt. This needs to provide separate processing paths for a null entry (which will be used to add a new order), an entry of Q (which will terminate the program) and entry of a five digit number (which will display the details of an existing order). Arrange for an error message to be displayed if any other data is entered at this prompt.

After processing the input, loop back to ask for a further order number.

When the user enters Q at the order number prompt, exit from the loop, clear the screen and terminate the program using a STOP statement.

The statements that clear the screen and display the fixed text should be moved into a subroutine. This should be called before entering the main loop of the program and at the end of the paths for a new order or display of an existing order.

Your input of the action code should also be moved into a subroutine. This should be called immediately before repainting the screen at the end of processing a new or existing order.

The action processing subroutine needs to contain validation of the response, allowing D, F and X, each with its own processing path. Any other entry should display an error and repeat the input.

Suggested Solution

Statements in bold face have been added to the program created in the first exercise. Your program may look very different from this but our suggested solution may be a good source of ideas.

```
PROGRAM ORDERS

   PROMPT ''

   GOSUB PAINT.SCREEN

   * Main loop - once per order

   LOOP
      INPUTERR "Enter order number, blank for next, Q to quit"

      ORDER.NO = ''
      INPUT @(10,3) : ORDER.NO, 5_
      BEGIN CASE
         CASE ORDER.NO = ''
            * New order processing to be added here
            GOSUB ACTION.PROMPT
            GOSUB PAINT.SCREEN

         CASE ORDER.NO = 'Q'
            EXIT

         CASE ORDER.NO MATCHES '5N'
            * Existing order processing to be added here
            GOSUB ACTION.PROMPT
            GOSUB PAINT.SCREEN

         CASE 1
            INPUTERR 'Order number must be 5 digits, blank for new
order, Q to quit'
      END CASE
   REPEAT

   DISPLAY @(-1) :

   STOP

* ================================================================
* Paint fixed part of screen

PAINT.SCREEN:
   DISPLAY @(-1)
   DISPLAY @(32,0) : 'ORDER PROCESSING'
   DISPLAY @(0,3)  : 'Order No:'
   DISPLAY @(20,3) : 'Date:'
   DISPLAY @(0,5)  : 'Customer No:'
   DISPLAY @(5,8)  : 'Part'
   DISPLAY @(11,8) : 'Description'
```

```
   DISPLAY @(48,8) : 'Price'
   DISPLAY @(56,8) : 'Qty'
   DISPLAY @(65,8) : 'Total'
   DISPLAY @(48,14) : 'Order Total:'
   DISPLAY @(0,20) : 'Action (F/D/X):'

   RETURN

* ================================================================
* Action prompt

ACTION.PROMPT:
   LOOP
      ACTION = ''
      INPUT @(16,20) ACTION,3_

      BEGIN CASE
         CASE ACTION = 'D'  ;* Delete
            * Order deletion to be added here
            EXIT

         CASE ACTION = 'F'  ;* File
            * Order filing to be added here
            EXIT

         CASE ACTION = 'X'  ;* Exit
            * Exit processing to be added here
            EXIT

         CASE 1
            INPUTERR 'File, Delete, eXit'
      END CASE
   REPEAT

   RETURN
END
```

15.4 File Handling

A QM application typically uses many hundreds or, perhaps, thousands of files, each holding data of some particular type. Our order processing system has only a few files but allows us to explore all of the file handling features of QMBasic.

Files are defined by F-type records in the VOC. These records relate the internal name of the file (the name of the VOC record) to the operating system pathname of the data and dictionary parts of the file. This VOC record should be the only place where the pathnames appear. If we move the file, all we need to change is the VOC entry; the application software is not affected.

Every file normally consists of a data part holding the actual data records and a dictionary part that holds records which describe the format of the data records. BASIC programs do not normally use the dictionary to locate individual data fields. Instead, the application designer normally builds a set of EQUATE tokens from the dictionary and these are used in all operations that process data records. QM provides a tool named GENERATE to construct this file automatically.

Using tokens for each field position makes the application more readable, simplifying maintenance. Applications that refer directly to fields by number are very difficult to understand. We will create a set of tokens for the files used in these exercises later.

A dictionary is just a file with a specific purpose in the system. Everything that we discuss in this section applies to either the data or dictionary parts of the file unless we explicitly say otherwise.

The OPEN Statement

The OPEN statement opens a file, associating it with a file variable.

```
OPEN {dict.expr,} filename.expr {READONLY} TO file.var
{ON ERROR statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

where

dict.expr	evaluates to DICT to open the dictionary portion of the file or to a null string to open the data portion. If omitted, the data portion is opened.
filename.expr	evaluates to the VOC name of the file to be opened.
READONLY	specifies that the file is to be opened for read only access. Attempts to update the file will cause a run time error. Note that QM will automatically adopt read-only mode if the operating system permissions on the file permit reading but not writing.

file.var	is the name of the variable for use in later operations on this file. This variable holds all of the control information used by QM to manage access to the file. You cannot do anything with it except to use it in file handling operations.
statement(s)	are statements to be executed depending on the outcome of the OPEN operation.

The ON ERROR, THEN and ELSE clauses are all optional but must appear in the order specified. At least one of the THEN or ELSE clauses must be present. Each of these clauses follows the same structure as the conditioned clauses of an IF statement; there may be a single statement on the same line as the keyword or a block of statements terminated by END.

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 return an error number identifying the cause of the error. If no ON ERROR clause is present, a fatal error results in an abort. This clause is rarely used because few programs can take any sensible recovery action following such an error. We will not use ON ERROR clauses in our programs.

The THEN clause is executed if the file is opened successfully.

The ELSE clause is executed if the open fails because, for example, the file does not exist. The STATUS() function may be used to determine the cause of the failure.

Typically, an OPEN statement is written as

```
OPEN filename TO file.var ELSE STOP message
```

which will continue with the next statement if the file is opened successfully.

File variable names may be constructed in any way that the developer wishes. A common convention is to shorten the file name to three or four characters and add a suffix of .F to the name to show that it is a file variable. We will use this convention throughout this course such that the file variable for our SALES file becomes SAL.F. A useful extension is then to use names such as SAL.ID for SALES file record ids and SAL.REC for records read from the file. These are conventions only and you may use whatever names you like.

QM allows more files to be open than the underlying operating system limit. This is achieved by closing files at the operating system level if they have not been referenced recently whilst retaining information to reopen them automatically when the next access to the file occurs. This process allows the application designer to ignore operating system limits.

From an application developer's perspective, the file remains open for as long as the file variable remains intact. When the program terminates, local variables are discarded, implicitly closing the file. If a program overwrites the file variable, the file previously open to it will be closed.

Examples

```
OPEN 'SALES' TO SAL.F ELSE STOP 'Cannot open SALES file'

OPEN '', 'SALES' TO SAL.F
```

```
ON ERROR
   STOP 'Fatal error opening SALES file. Error ': STATUS()
END ELSE
   STOP 'Cannot open SALES file. Error ' : STATUS()
END
```

Both of the above examples open the data part of the ORDERS file to a file variable named SAL.F which will be used in later file handling operations.

The second example includes an ON ERROR clause to display the error code returned by the STATUS() function. Similarly, the ELSE clause also displays the error code. Note that this use of the ON ERROR clause may result in a less informative diagnostic message being displayed than would have happened without the ON ERROR clause.

The ON ERROR clause appears in most file handling statements. It is strongly recommended that this clause is only used if there is some sensible recovery action that can be performed by the program.

```
OPEN 'DICT', 'SALES' TO SAL.D
ELSE ABORT 'Cannot open SALES file dictionary'
```

This example opens the dictionary of the SALES file to SAL.D.

The READ Statement

The READ statement reads a record from a previously opened file into a variable as a dynamic array.

```
READ var FROM file.var, record.id
{ON ERROR statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

where

var	is the name of a variable to receive the dynamic array read from the file.
file.var	is the file variable associated with the file.
record.id	evaluates to the id of the record to be read.
statement(s)	are statements to be executed depending on the outcome of the READ operation.

At least one of the THEN or ELSE clauses must be present.

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. This clause is rarely used.

The THEN clause is executed if the READ is successful. The specified record is read into the named variable.

The ELSE clause is executed if the READ fails because no record with the given id is present on the file. The *var* will be set to a null string.

Example

```
READ SAL.REC FROM SAL.F, ORDER.NO THEN
   DISPLAY ORDER.NO
   DISPLAY 'Customer = ' : SAL.REC<2>
END ELSE
   DISPLAY 'Order ' : ORDER.NO: ' not found'
END
```

This program fragment reads a record from the a file previously opened to file variable SAL.F into variable SAL.REC. If successful, the customer number is displayed. If the record is not found, an error message is displayed.

Note that this example uses a field number when extracting the customer number from the sales record. For a simple application with only a few files, each of which has only a few fields, we might be able to remember all the field numbers. For a realistic application, this approach can make maintenance difficult and error prone. It is strongly recommended that EQUATE tokens are used to give names to fields. Typically, these have a prefix that identifies the file and the rest of the name relates to the field name. We will create some tokens that work this way later, using a prefix of SL, ST or CS to correspond to our SALES,

STOCK and CUSTOMERS files. The relevant line of the above example might then become

```
DISPLAY 'Customer = ' : SAL.REC<SL.CUST>
```

Not only does this save us remembering the field numbers, it makes the program easier to read and makes it possible to find all references to a field by searching for the corresponding token name.

When accessing a directory file, QM normally translates newlines to field marks when reading data and conversely translates field marks to newlines when writing data. If the file being processed contains binary data such as a bit-mapped image, this action will corrupt the data. Use of

```
MARK.MAPPING file.var, OFF
```

after opening the file will suppress the translation. A corresponding use of

```
MARK.MAPPING file.var, ON
```

will re-enable it though it is unusual to need to do this.

The READU Statement

The READU statement reads a record from a previously opened file into a variable as a dynamic array. It also takes an update lock on the record to prevent other users updating the record at the same time. Only one user can hold an update lock on any specific record at one time though many different locks may be held simultaneously.

```
READU var FROM file.var, record.id
{ON ERROR statement(s)}
{LOCKED statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

where

var	is the name of a variable to receive the dynamic array read from the file.
file.var	is the file variable associated with the file.
record.id	evaluates to the id of the record to be read.
statement(s)	are statements to be executed depending on the outcome of the READU operation.

At least one of the THEN or ELSE clauses must be present.

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. This clause is rarely used.

The LOCKED clause is executed if we are unable to complete the read because another user has this record locked. The STATUS() function can be used to find the user number of the user owning the lock. If no LOCKED clause is present, the program waits for the lock to be released.

The THEN clause is executed if the READU is successful. The specified record is read into the named variable. The process performing the READU now has an update lock on the record to prevent interactions from other users.

The ELSE clause is executed if the READU fails because no record with the given id is present on the file. The var will be set to a null string. Note that even though the record does not exist, the process performing the READU now has an update lock on the record to prevent interactions from other users.

Programs should always use the READU statement rather than READ when they are going to add, modify or delete a record. Failure to do so may result in data corruptions. The READ statement takes no part in the locking system and always succeeds, even if another user has the record locked.

Examples

```
READU SAL.REC FROM SAL.F, ORDER.NO
LOCKED
   DISPLAY ORDER.NO : ' is locked by user ' : STATUS()
END THEN
   processing statements
END ELSE
   DISPLAY 'Record ' : ORDER.NO : ' not found'
END
```

This program fragment extends the earlier example to include locking. If the record is already locked by another user, the program displays that user's number.

```
READU SAL.REC FROM SAL.F, ORDER.NO
THEN
   processing statements
END ELSE
   DISPLAY 'Record ' : ORDER.NO : ' not found'
END
```

Omitting the LOCKED clause would cause the program simply to wait for the record to become available.

The READL Statement

The READL statement reads a record from a previously opened file into a variable as a dynamic array. It also takes a sharable read lock on the record. Any number of users may hold a sharable read lock on the same record but no other user can own the update lock.

```
READL var FROM file.var, record.id
{ON ERROR statement(s)}
{LOCKED statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

where

var	is the name of a variable to receive the dynamic array read from the file.
file.var	is the file variable associated with the file.
record.id	evaluates to the id of the record to be read.
statement(s)	are statements to be executed depending on the outcome of the READL operation.

At least one of the THEN or ELSE clauses must be present.

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. This clause is rarely used.

The LOCKED clause is executed if we are unable to complete the read because another user has an update lock on this record. The STATUS() function can be used to find the user number of the user owning the lock. If no LOCKED clause is present, the program waits for the lock to be released.

The THEN clause is executed if the READL is successful. The specified record is read into the named variable. The process performing the READL now has a sharable read lock on the record to prevent interactions from other users.

The ELSE clause is executed if the READL fails because no record with the given id is present on the file. The var will be set to a null string. Note that even though the record does not exist, the process performing the READL now has a sharable read lock on the record to prevent interactions from other users.

Programs should use the READL statement when they want to look at a record, knowing that no other user may change it, but don't mind other users also looking at the data.

The WRITE Statement

The WRITE statement writes a record to a file, replacing any existing record with the same record id.

```
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 depending on the outcome of the WRITE.

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. The WRITE statement releases any lock on this record.

The ON ERROR clause is executed for serious fault conditions such as running out of disk space. The STATUS() function can be used to find the cause of the error. If no ON ERROR clause is present, an abort would occur.

Example

```
LOOP
   INPUT ORDER.NO
UNTIL ORDER.NO = ''
   READU SAL.REC FROM SAL.F, ORDER.NO THEN
      INPUT NEW.CUST
      SAL.REC<2> = NEW.CUST   ;* Replace customer field
      WRITE SAL.REC TO SAL.F, ORDER.NO
   END ELSE
      DISPLAY 'Record ' : ORDER.NO : ' not found'
   END
REPEAT
```

The WRITEU statement is similar to WRITE except that it does not release the lock.

The DELETE Statement

The DELETE statement deletes a record from an open file.

```
DELETE 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.

The DELETE statement also releases any lock on the record being deleted.

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.

Example

```
READU SAL.REC FROM SAL.F, ORDER.NO THEN
   DELETE SAL.F, ORDER.NO
END ELSE
   DISPLAY 'Record ' : ORDER.NO : ' not found'
END
```

The above program fragment deletes a record from the file. Note the use of READU to lock the record first.

The DELETEU statement is similar to DELETE except that it does not release the lock.

The RELEASE Statement

The RELEASE statement releases a record lock.

```
RELEASE {ON ERROR statement(s)}

RELEASE file.var  {ON ERROR statement(s)}

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 if the action fails.

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. This is very dangerous as it will release locks acquired by other parts of the application.

With *file.var* but no record.id, is present all locks associated with *file.var* are released. This is still slightly dangerous.

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.

Example

All of the earlier locking examples left the lock in place if they failed to read the record. A RELEASE statement is needed to ensure that we do not collect unwanted locks.

```
READU SAL.REC FROM SAL.F, ORDER.NO
LOCKED
   DISPLAY ORDER.NO : ' locked by user ' : STATUS()
END THEN
   processing statements
END ELSE
   RELEASE SAL.F, ORDER.NO
   DISPLAY 'Record ' : ORDER.NO : ' not found'
END
```

Checking for Locks - The LIST.READU Command

The LIST.READU command (not a program statement) can be used to check for outstanding locks in the system. It should be used when testing applications to ensure that locks are taken when required and released when no longer needed. It can also be used if the system hangs to check if users are waiting for locks.

An example is shown below.

```
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
```

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.

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 (discussed later) 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 in the LIST.READU command.

Deadlocks

A deadlock occurs when one process is waiting for a record locked by a second process and the second process is waiting for a record locked by the first process.

Deadlocks are totally avoidable by good programming methods. Two simple to describe (but often difficult to implement) techniques are available:

- Always lock records in a fixed order so that the situation can never occur.
- Use the LOCKED clause and, on finding that processing is blocked by some other user, release all locks and start again.

The CLOSE Statement

The CLOSE statement closes a file previously opened using the OPEN statement.

```
CLOSE file.var
```

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.

Files do not usually 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 blocks (discussed later) will not be affected.

The main reason to use CLOSE is to help the file sharing mechanism that provides the illusion that we can open more files than the operating system allows. If we know that we will not refer to a file again in the near future, closing it allows it to be removed from the rotation mechanism rather than decaying through the file aging process. This can result in a small performance improvement.

File Locks

Sometimes we find it useful to lock all records in a file. This may be because we are going to update them all or it might be to guarantee a consistent view of the data where no other user can change any records. The file lock allows us to do this.

The FILELOCK and FILEUNLOCK statements can be used to obtain and release the file lock for a given file. A user can only obtain the file lock if no other user has any lock of any type in the file. Conversely, while the file lock is held, no other user can obtain locks on any records in the file.

```
FILELOCK  file.var  { LOCKED statement(s) }

FILEUNLOCK  file.var
```

Clearly, the file lock is likely to have substantial effects on performance of a multi-user application. It should be used with care.

Exercise

This exercise will introduce use of an include record (discussed in the Introduction to QMBasic Programming section) to add definitions of our field positions. Although we will construct this record manually, a real application might use the GENERATE tool to do this automatically from the dictionary. You can learn more about GENERATE from the *QM Reference Manual* or the help system.

Use SED, ED or any other suitable text editor to create a record named FILES.H in your BP file that contains the following:

```
* FILES.H  -  File definitions for order processing
application

* STOCK file
EQU ST.DESCR   TO 1  ;* Item description
EQU ST.QTY     TO 2  ;* Quantity on hand
EQU ST.PRICE   TO 3  ;* Selling price (MD2)

* CUSTOMERS file
EQU CS.NAME    TO 1  ;* Customer name
EQU CS.ADDR    TO 2  ;* Address
EQU CS.TELNO   TO 3  ;* Telephone

* SALES file
EQU SL.DATE    TO 1  ;* Order date
EQU SL.CUST    TO 2  ;* CUSTOMERS id
EQU SL.ITEM    TO 3  ;* Part number (multi-valued)
EQU SL.QTY     TO 4  ;* Order quantity (multi-valued)
EQU SL.PRICE   TO 5  ;* Selling price (multi-valued)
EQU SL.PAYDATE TO 6  ;* Payment date
EQU SL.PAYMENT TO 7  ;* Payment value
```

Add the FILES.H include record to your program with a line

```
$INCLUDE FILES.H
```

just after the PROGRAM statement. This defines names for each field in the data files. Use these in preference to field numbers in your program.

Add file handling statements to your ORDERS program to achieve the following:

Open the CUSTOMERS, STOCK and SALES files at the start, outside the main loop of the program.

Develop a subroutine which, given an order record, will display the data from this record. This subroutine should show the order date and customer number and should call two further subroutines to show the customer details (name and address) and order details (parts, quantities, prices, etc).

At this stage, the subroutine to show the customer details should display only the customer's name. Display of the address will be added later. Your subroutine may assume that any customer referenced by an order must exist in the CUSTOMERS file.

The subroutine to display the order details should display up to five lines of order information. We will explore how to allow more than five items in an order later. The display

should show the part number, description, selling price, quantity ordered and line total value. The part number, selling price and quantity come from the order record. You will need to read the corresponding STOCK record for each item to get the description. The line total value is calculated simply by multiplying the quantity by the selling price.

In the path for processing an existing order, add a READU statement to read the order record.

If the record is locked by another user, display a suitable message.

If the order is successfully read, call subroutines developed above to display the data from this order. On return, display the action prompt and repaint the screen.

If the order is not found, release the update lock and display a message.

In all cases, then return to the order number prompt

If all is working correctly, running your program for order number 12002 should produce a screen similar to the one below.

```
                              ORDER PROCESSING


Order No: 12002     Date: 14400

Customer No: 1002   Ross, Alan


    Part  Description                          Price   Qty     Total
    013   Pencil, blue                         28        2     56
    012   Pencil, red                          28        3     84



                                               Order Total:




Action (F/D/X):
```

The date and the prices are not yet shown correctly or aligned as we would like them. We will address this problem in the next section.

Suggested solution

Again, we have shown changes to the previous program in bold face.

```
PROGRAM ORDERS

$INCLUDE FILES.H

   PROMPT ''

   OPEN 'CUSTOMERS' TO CUS.F
       ELSE ABORT 'Cannot open CUSTOMERS file'
   OPEN 'STOCK' TO STK.F
       ELSE ABORT 'Cannot open STOCK file'
   OPEN 'SALES' TO SAL.F
       ELSE ABORT 'Cannot open SALES file'

   GOSUB PAINT.SCREEN

   * Main loop - once per order

   LOOP
      INPUTERR "Enter order number, blank for next, Q to quit"

      ORDER.NO = ''
      INPUT @(10,3) : ORDER.NO, 5_
      BEGIN CASE
         CASE ORDER.NO = ''
            * New order processing to be added here
            GOSUB ACTION.PROMPT
            GOSUB PAINT.SCREEN

         CASE ORDER.NO = 'Q'
            EXIT

         CASE ORDER.NO MATCHES '5N'
            READU SAL.REC FROM SAL.F, ORDER.NO LOCKED
               INPUTERR 'Record is locked by user ' : STATUS()
            END THEN
               GOSUB PAINT.DATA
               GOSUB ACTION.PROMPT
               GOSUB PAINT.SCREEN
            END ELSE
               RELEASE SAL.F, ORDER.NO
               INPUTERR 'Order ' : ORDER.NO : ' is not on file'
            END

         CASE 1
            INPUTERR 'Order number must be 5 digits, blank for new
order, Q to quit'
      END CASE
   REPEAT

   DISPLAY @(-1) :
```

```
   STOP

* ================================================================
* Paint fixed part of screen

PAINT.SCREEN:
   DISPLAY @(-1)
   DISPLAY @(32,0) : 'ORDER PROCESSING'
   DISPLAY @(0,3)  : 'Order No:'
   DISPLAY @(20,3) : 'Date:'
   DISPLAY @(0,5)  : 'Customer No:'
   DISPLAY @(5,8)  : 'Part'
   DISPLAY @(11,8) : 'Description'
   DISPLAY @(48,8) : 'Price'
   DISPLAY @(56,8) : 'Qty'
   DISPLAY @(65,8) : 'Total'
   DISPLAY @(48,14) : 'Order Total:'
   DISPLAY @(0,20) : 'Action (F/D/X):'

   RETURN

* ================================================================
* Paint data from current order record

PAINT.DATA:
   DISPLAY @(26,3)  : SAL.REC<SL.DATE>
   DISPLAY @(13,5)  : SAL.REC<SL.CUST>
   GOSUB DISPLAY.CUSTOMER.DETAILS

   GOSUB DISPLAY.ORDER.LINES

   RETURN

* ================================================================
* Display customer details

DISPLAY.CUSTOMER.DETAILS:
   CUST.NO = SAL.REC<SL.CUST>
   READ CUS.REC FROM CUS.F, CUST.NO THEN
      DISPLAY @(20,5) : CUS.REC<CS.NAME>
   END

   RETURN

* ================================================================
* Show order details

DISPLAY.ORDER.LINES:
   LN = 9
   FOR IDX = 1 TO 5
      PART.NO = SAL.REC<SL.ITEM,IDX>
   UNTIL PART.NO = ''
      PRICE = SAL.REC<SL.PRICE,IDX>
      QTY = SAL.REC<SL.QTY,IDX> + 0
      DISPLAY @(5,LN) : PART.NO
```

```
      DISPLAY @(45,LN) : PRICE
      DISPLAY @(55,LN) : QTY
      READ STK.REC FROM STK.F, PART.NO THEN
         DISPLAY @(11,LN) : STK.REC<ST.DESCR>[1,30]
         DISPLAY @(61,LN) : PRICE * QTY
      END
      LN += 1
   NEXT IDX

   RETURN

* ================================================================
* Action prompt

ACTION.PROMPT:
   LOOP
      ACTION = ''
      INPUT @(16,20) ACTION,3_

      BEGIN CASE
         CASE ACTION = 'D'  ;* Delete
            * Order deletion to be added here
            EXIT

         CASE ACTION = 'F'  ;* File
            * Order filing to be added here
            EXIT

         CASE ACTION = 'X'  ;* Exit
            * Exit processing to be added here
            EXIT

         CASE 1
            INPUTERR 'File, Delete, eXit'
      END CASE
   REPEAT

   RETURN
END
```

15.5 Conversion and Formatting

We discussed conversion and formatting earlier when we met database dictionaries. Application programs need to use both of these concepts internally.

Data conversion changes the way in which the data is represented. For example, we store dates as a number of days from a reference date.

Formatting positions the data in a given field width, applying justification and other features to control its appearance.

Data Conversion

Data conversion is performed by applying a **conversion code**. The codes that we use in our BASIC programs are the same as those that appear in dictionary definitions of our database files.

We need to be able to convert data from its external form to its internal form when data arrives from the terminal or other sources. This input conversion is performed using the ICONV() function.

We need to convert the internal form of the data back to its external form when displaying it back to the user, printing reports, etc. This output conversion is performed by the OCONV() function.

Both functions have the same form:

```
ICONV(data, code)
OCONV(data, code)
```

where

data is the data to be converted.

code is the conversion code to be applied.

The functions return the converted data as their result.

Conversion may fail for a number of reasons. The STATUS() function can be used to test whether the conversion was successful. This returns:

0 Successful conversion

1 The data cannot be converted using the given conversion code.

2 The conversion code is not recognised.

3 A date conversion contains a faulty date (e.g. 31 April) which has been converted to a likely date (1 May in this example).

Input conversion of dates using the ICONV() function tries to interpret dates even if the format in which they are entered does not fully match the conversion code. This allows some flexibility in user entry of date fields.

Input conversion of dates with two digit year numbers assumes that

30-99	is 19xx
00-29	is 20xx

Example

```
DISPLAY "Time now is " : OCONV(TIME(), "MTS.")
```

This statement displays the current time as a 24 hour clock value with seconds and using the full stop as the separator.

Related Functions

Conversion of a character string to uppercase can be performed using the UPCASE() function. Similarly, conversion to lowercase is provided by the DOWNCASE() function.

A Practical Example of Input Validation with Conversion

The following few lines of QMBasic program can be used for any input validation where a conversion code is to be applied.

```
LOOP
   TEMP = OCONV(data, code)
   INPUT @(col, row) TEMP, width_:
   TEMP = ICONV(TEMP, code)
UNTIL STATUS() = 0
   INPUTERR 'Invalid input'
REPEAT
data = TEMP
DISPLAY @(col, row) : OCONV(data, conv)
```

Note how the data is converted to its external form for display and modification by the user and then immediately converted back to its internal form. By keeping the converted data in the temporary variable, the original data has not been lost if the validation fails.

In most cases, the ICONV() function does all the data validation for us. We need only test that the conversion was successful by using the STATUS() function.

Finally, we redisplay the data to ensure that it is in its correct form (decimal places, currency symbol, etc).

Formatting

The FMT() function performs data formatting according to a format specification. This is identical to the format codes described earlier when we explored dictionaries. The FMT() function is typically used to convert data for display or printing.

```
FMT(expr, fmt.spec)
```

where

expr evaluates to the data to be formatted

fmt.spec evaluates to the format specification.

Shortform Notation

The FMT() function action can also be performed in programs 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'
```

Exercise

Amend your ORDERS program to add conversion of the order date to a suitable format and conversion of the price fields.

To simplify maintenance of your program, you could use an EQUATE token for the two conversion codes. If you later decide to change the conversion, you need only change the value of this token, not every reference within the program.

If all is working correctly, order number 12002 should now appear as below.

```
                                ORDER PROCESSING


 Order No: 12002     Date: 04 JUN 07

 Customer No: 1002   Ross, Alan


     Part  Description                          Price   Qty
Total
     013   Pencil, blue                         0.28      2
0.56
     012   Pencil, red                          0.28      3
0.84



                                                Order Total:




 Action (F/D/X):
```

Suggested solution

Again, we have shown changes to the previous program in bold face.

```
PROGRAM ORDERS

$INCLUDE FILES.H
EQU DATE.CONVERSION TO 'D2DMY[,A3]'
EQU CASH.CONVERSION TO 'MD2'

   PROMPT ''

   OPEN 'CUSTOMERS' TO CUS.F
       ELSE ABORT 'Cannot open CUSTOMERS file'
   OPEN 'STOCK' TO STK.F
       ELSE ABORT 'Cannot open STOCK file'
   OPEN 'SALES' TO SAL.F
       ELSE ABORT 'Cannot open SALES file'

   GOSUB PAINT.SCREEN

   * Main loop - once per order

   LOOP
      INPUTERR "Enter order number, blank for next, Q to quit"

      ORDER.NO = ''
      INPUT @(10,3) : ORDER.NO, 5_
      BEGIN CASE
         CASE ORDER.NO = ''
            * New order processing to be added here
            GOSUB ACTION.PROMPT
            GOSUB PAINT.SCREEN

         CASE ORDER.NO = 'Q'
            EXIT

         CASE ORDER.NO MATCHES '5N'
            READU SAL.REC FROM SAL.F, ORDER.NO LOCKED
               INPUTERR 'Record is locked by user ' : STATUS()
            END THEN
               GOSUB PAINT.DATA
               GOSUB ACTION.PROMPT
               GOSUB PAINT.SCREEN
            END ELSE
               RELEASE SAL.F, ORDER.NO
               INPUTERR 'Order ' : ORDER.NO : ' is not on file'
            END

         CASE 1
            INPUTERR 'Order number must be 5 digits, blank for new
order, Q to quit'
      END CASE
   REPEAT
```

```
   DISPLAY @(-1) :

   STOP

* ================================================================
* Paint fixed part of screen

PAINT.SCREEN:
   DISPLAY @(-1)
   DISPLAY @(32,0) : 'ORDER PROCESSING'
   DISPLAY @(0,3)  : 'Order No:'
   DISPLAY @(20,3) : 'Date:'
   DISPLAY @(0,5)  : 'Customer No:'
   DISPLAY @(5,8)  : 'Part'
   DISPLAY @(11,8) : 'Description'
   DISPLAY @(48,8) : 'Price'
   DISPLAY @(56,8) : 'Qty'
   DISPLAY @(65,8) : 'Total'
   DISPLAY @(48,14) : 'Order Total:'
   DISPLAY @(0,20) : 'Action (F/D/X):'

   RETURN

* ================================================================
* Paint data from current order record

PAINT.DATA:
   DISPLAY @(26,3)  : OCONV(SAL.REC<SL.DATE>, DATE.CONVERSION)
   DISPLAY @(13,5)  : SAL.REC<SL.CUST>
   GOSUB DISPLAY.CUSTOMER.DETAILS

   GOSUB DISPLAY.ORDER.LINES

   RETURN

* ================================================================
* Display customer details

DISPLAY.CUSTOMER.DETAILS:
   CUST.NO = SAL.REC<SL.CUST>
   READ CUS.REC FROM CUS.F, CUST.NO THEN
      DISPLAY @(20,5) : CUS.REC<CS.NAME>
   END

   RETURN

* ================================================================
* Show order details

DISPLAY.ORDER.LINES:
   LN = 9
   FOR IDX = 1 TO 5
      PART.NO = SAL.REC<SL.ITEM,IDX>
   UNTIL PART.NO = ''
      PRICE = SAL.REC<SL.PRICE,IDX>
      QTY = SAL.REC<SL.QTY,IDX> + 0
```

```
      DISPLAY @(5,LN) : PART.NO
      DISPLAY @(45,LN) : FMT(OCONV(PRICE, CASH.CONVERSION), '7R')
      DISPLAY @(55,LN) : FMT(QTY, '4R')
      READ STK.REC FROM STK.F, PART.NO THEN
         DISPLAY @(11,LN) : STK.REC<ST.DESCR>[1,30]
         DISPLAY @(61,LN) : FMT(OCONV(PRICE * QTY,
CASH.CONVERSION), '9R')
      END
      LN += 1
   NEXT IDX

   RETURN

* ================================================================
* Action prompt

ACTION.PROMPT:
   LOOP
      ACTION = ''
      INPUT @(16,20) ACTION,3_

      BEGIN CASE
         CASE ACTION = 'D'  ;* Delete
            * Order deletion to be added here
            EXIT

         CASE ACTION = 'F'  ;* File
            * Order filing to be added here
            EXIT

         CASE ACTION = 'X'  ;* Exit
            * Exit processing to be added here
            EXIT

         CASE 1
            INPUTERR 'File, Delete, eXit'
      END CASE
   REPEAT

   RETURN
END
```

Once you are happy that this is working correctly, modify your program to allow entry of new orders.

When the user enters a blank order number, display '*new*' in the order number field. Think about why we should not generate the next order number at this point.

Set the variable you are using to hold the order record to a null string and then insert today's date into the correct field. You can get the current date in its internal form from the DATE() function. Display this date on the screen.

Call a new subroutine to prompt for a customer number and insert this into the order record. This subroutine should validate that the customer exists in the CUSTOMERS file.

Call your existing subroutine to display the customer's name.

Call a new subroutine to get the order details (part numbers and quantities). This can be constructed by making a copy of the subroutine that displays this information from existing orders and changing it to prompt for the part number and quantity. Include some validation of this input data. The subroutine should allow a maximum of five order lines but should exit if a blank part number is entered.

Modify the handling of the action prompt to write the new order if the user enters 'F'. When writing a new order, you will need to get the next order number from somewhere. Many applications have a central file to store this sort of information. We will use an X-type record in the dictionary of the SALES file.

Add statements for the 'X' response to the action prompt to release the lock if we are processing an existing order.

Check that your program allows entry of new orders and that the generated order number is correctly incremented as each order is entered. Check that you can correctly display an order that you have entered.

Suggested solution

This example uses a record named NEXT.ID in the dictionary of the SALES file to hold the next order number. This should start out as something like

1: X
2: 14000

where 14000 is the next order number to be created. This will be incremented by the program each time a new order is filed.

```
PROGRAM ORDERS

$INCLUDE FILES.H
EQU DATE.CONVERSION TO 'D2DMY[,A3]'
EQU CASH.CONVERSION TO 'MD2'

   PROMPT ''

   OPEN 'CUSTOMERS' TO CUS.F
       ELSE ABORT 'Cannot open CUSTOMERS file'
   OPEN 'STOCK' TO STK.F
       ELSE ABORT 'Cannot open STOCK file'
   OPEN 'SALES' TO SAL.F
       ELSE ABORT 'Cannot open SALES file'
   OPEN 'DICT', 'SALES' TO SAL.D
       ELSE ABORT 'Cannot open SALES dictionary'

   GOSUB PAINT.SCREEN

   * Main loop - once per order

   LOOP
      INPUTERR "Enter order number, blank for next, Q to quit"

      ORDER.NO = ''
      INPUT @(10,3) : ORDER.NO, 5_
      BEGIN CASE
         CASE ORDER.NO = ''
            DISPLAY @(10,3) : '*new*' :

            SAL.REC = ''
            OLD.SAL.REC = ''
            SAL.REC<SL.DATE> = DATE()
            DISPLAY @(26,3)  : OCONV(SAL.REC<SL.DATE>,
DATE.CONVERSION) :

            GOSUB GET.CUST.NO
            GOSUB DISPLAY.CUSTOMER.DETAILS

            GOSUB GET.ORDER.DETAILS
            GOSUB ACTION.PROMPT
            GOSUB PAINT.SCREEN

         CASE ORDER.NO = 'Q'
            EXIT
```

```
         CASE ORDER.NO MATCHES '5N'
            READU SAL.REC FROM SAL.F, ORDER.NO LOCKED
               INPUTERR 'Record is locked by user ' : STATUS()
            END THEN
               GOSUB PAINT.DATA
               GOSUB ACTION.PROMPT
               GOSUB PAINT.SCREEN
            END ELSE
               RELEASE SAL.F, ORDER.NO
               INPUTERR 'Order ' : ORDER.NO : ' is not on file'
            END

         CASE 1
            INPUTERR 'Order number must be 5 digits, blank for new
order, Q to quit'
      END CASE
   REPEAT

   DISPLAY @(-1) :

   STOP

* ================================================================
* Paint fixed part of screen

PAINT.SCREEN:
   DISPLAY @(-1)
   DISPLAY @(32,0) : 'ORDER PROCESSING'
   DISPLAY @(0,3)  : 'Order No:'
   DISPLAY @(20,3) : 'Date:'
   DISPLAY @(0,5)  : 'Customer No:'
   DISPLAY @(5,8)  : 'Part'
   DISPLAY @(11,8) : 'Description'
   DISPLAY @(48,8) : 'Price'
   DISPLAY @(56,8) : 'Qty'
   DISPLAY @(65,8) : 'Total'
   DISPLAY @(48,14) : 'Order Total:'
   DISPLAY @(0,20) : 'Action (F/D/X):'

   RETURN

* ================================================================
* Paint data from current order record

PAINT.DATA:
   DISPLAY @(26,3)  : OCONV(SAL.REC<SL.DATE>, DATE.CONVERSION)
   DISPLAY @(13,5)  : SAL.REC<SL.CUST>
   GOSUB DISPLAY.CUSTOMER.DETAILS

   GOSUB DISPLAY.ORDER.LINES

   RETURN

* ================================================================
* Display customer details

DISPLAY.CUSTOMER.DETAILS:
```

```
   CUST.NO = SAL.REC<SL.CUST>
   READ CUS.REC FROM CUS.F, CUST.NO THEN
      DISPLAY @(20,5) : CUS.REC<CS.NAME>
   END

   RETURN

* ================================================================
* Show order details

DISPLAY.ORDER.LINES:
   LN = 9
   FOR IDX = 1 TO 5
      PART.NO = SAL.REC<SL.ITEM,IDX>
   UNTIL PART.NO = ''
      PRICE = SAL.REC<SL.PRICE,IDX>
      QTY = SAL.REC<SL.QTY,IDX> + 0
      DISPLAY @(5,LN) : PART.NO
      DISPLAY @(45,LN) : FMT(OCONV(PRICE, CASH.CONVERSION), '7R')
      DISPLAY @(55,LN) : FMT(QTY, '4R')
      READ STK.REC FROM STK.F, PART.NO THEN
         DISPLAY @(11,LN) : STK.REC<ST.DESCR>[1,30]
         DISPLAY @(61,LN) : FMT(OCONV(PRICE * QTY,
CASH.CONVERSION), '9R')
      END
      LN += 1
   NEXT IDX

   RETURN

* ================================================================
* Get customer number

GET.CUST.NO:
   CUST.NO = ''
   LOOP
      INPUT @(13,5) : CUST.NO, 4_:
      READ CUS.REC FROM CUS.F, CUST.NO THEN EXIT
      INPUTERR "Customer " : CUST.NO : " is not known"
   REPEAT

   SAL.REC<SL.CUST> = CUST.NO

   RETURN

* ================================================================
* Get new order details

GET.ORDER.DETAILS:
   LN = 9
   FOR IDX = 1 TO 5
      LOOP
         PART.NO = ''
         INPUT @(5,LN) PART.NO, 3_:
      UNTIL PART.NO = ''
         READ STK.REC FROM STK.F, PART.NO THEN EXIT
```

```
         INPUTERR 'Part number is not known'
      REPEAT

   UNTIL PART.NO = ''

      PRICE = STK.REC<ST.PRICE>
      DISPLAY @(11,LN) : STK.REC<ST.DESCR>[1,30] :
      DISPLAY @(45,LN) : FMT(OCONV(PRICE, CASH.CONVERSION),'7R'):

      LOOP
         QTY = ''
         INPUT @(55,LN) QTY,4_:
      UNTIL QTY MATCHES '1-4N' AND QTY > 0
         INPUTERR 'Invalid quantity'
      REPEAT
      DISPLAY @(55,LN) : FMT(QTY, '4R') :
      DISPLAY @(61,LN) : FMT(OCONV(PRICE * QTY, CASH.CONVERSION),
'9R') :

      SAL.REC<SL.ITEM,IDX> = PART.NO
      SAL.REC<SL.QTY,IDX> = QTY
      SAL.REC<SL.PRICE,IDX> = PRICE

      LN += 1
   NEXT IDX

   RETURN

* ======================================================================
* Action prompt

ACTION.PROMPT:
   LOOP
      ACTION = ''
      INPUT @(16,20) ACTION,3_

      BEGIN CASE
         CASE ACTION = 'D'  ;* Delete
            * Order deletion to be added here
            EXIT

         CASE ACTION = 'F'  ;* File
            * If we are creating a new order, we have left
            * generating the order number until now.

            IF ORDER.NO = '' THEN
               READU NEXT.ORDER FROM SAL.D, "NEXT.ID"
               ELSE ABORT 'Cannot find next order number'
               ORDER.NO = NEXT.ORDER<2>
               NEXT.ORDER<2> = ORDER.NO + 1
               WRITE NEXT.ORDER TO SAL.D, "NEXT.ID"
               DISPLAY @(10,3) : ORDER.NO :
            END

            WRITE SAL.REC TO SAL.F, ORDER.NO
```

```
            DISPLAY @(0,23) : 'Order confirmed. Press return' :
            INPUT JUNK, 1_:
            EXIT

         CASE ACTION = 'X'  ;* Exit
            IF ORDER.NO # '' THEN RELEASE SAL.F, ORDER.NO
            EXIT

         CASE 1
            INPUTERR 'File, Delete, eXit'
      END CASE
   REPEAT

   RETURN
END
```

Additional Exercises

If you find the time now or later you might like to try to add some of the following optional extra features to your program.

Refuse to file an order that has no products in it.

When a user enters the quantity against a product, check that there is sufficient stock to meet this order.

Adjust the STOCK file QTY figure to remove items from stock when they are sold. This is not as easy as it sounds as there are locking issues to consider.

When entering a new order, if the user selects the 'X' option to abandon the order, put any ordered items back into stock.

Add statements to process the 'D' option to delete an order, putting any items back into stock.

Add a new option to modify an existing order, putting back into stock any items removed from the order. Again this is not trivial. What happens if the user modifies the order and then selects the 'X' option of the action prompt?

Make the five lines of order details into a rolling window or a paginated display. This is not as difficult as it sounds but requires that you separate the counters that determine screen line position and order value position so that the screen can show any five consecutive parts from the order.

15.6 String Manipulation

This module describes some of the many string manipulation statements and functions available in QMBasic. Most of the operations related to strings which hold dynamic arrays are the subject of another module.

The main string handling statements and functions are listed below. Some of these have already been discussed in earlier modules.

ALPHA()	Test if a string holds only alphabetic characters
CHANGE()	Replace substring in a string
CHAR()	Get ASCII character for a given collating sequence value
COL1()	Start of substring position from FIELD()
COL2()	End of substring position from FIELD()
COMPARE()	Compare strings
CONVERT	Substitute characters with replacements
CONVERT()	Substitute characters with replacements
COUNT()	Count occurrences of a substring in a string
DOWNCASE()	Convert a string to lowercase
FIELD()	Extract delimited fields
FIELDSTORE()	Replace or insert delimited fields
INDEX()	Locate occurrence of a substring within a string
LEN()	Return length of a string
MATCHFIELD()	Return portion of a string matching pattern
NUM()	Test if a string holds a numeric value
SEQ()	Get collating sequence value for a given ASCII character
SOUNDEX()	Form a soundex code value for a string
SPACE()	Create a string of spaces
STR()	Create a string from a repeated substring
TRIM()	Trim characters from a string
TRIMB()	Trim spaces from back of a string
TRIMF()	Trim spaces from front of a string
UPCASE()	Convert a string to uppercase

The CHANGE() Function

The CHANGE() function replaces one sequence of characters with another.

```
CHANGE(str, old, new {, occ {, start }} )
```

where

str is the string in which replacement is to occur.

old is the substring to be replaced.

new is the replacement substring.

occ is the number of occurrences of *old* to replace. If omitted or less than one, all occurrences are replaced.

start specifies the first occurrence of *old* to replace. If omitted, it defaults to one.

Example

```
DISPLAY CHANGE(CUS.REC<2>, @VM, ', ')
```

This statement displays the result of changing all value marks in field 2 of CUS.REC (perhaps the address field of our CUSTOMERS file) to be a comma followed by a space. Thus an address stored internally as

```
121 Stoke RoadVMMansfield
```

would be displayed as

```
121 Stoke Road, Mansfield
```

The CONVERT() Function

The CONVERT() function creates a string which is a copy of some other string in which one set of characters is replaced by another set of characters.

```
CONVERT(old, new, str)
```

where

old is the set of characters to be replaced.

new is the corresponding set of replacement characters.

str is the string in which replacement is to occur.

The CONVERT() function returns the converted string as its result value.

Every occurrence of each character is *old* within *str* is replaced by the corresponding character from *new*. For example,

```
A = "A sample string to be converted"
DISPLAY CONVERT("ert", "XYZ", A)
```

would display

```
A samplX sZYing Zo bX convXYZXd
```

If *old* contains more characters than *new*, occurrences of the old characters for which there is no replacement are removed from the converted string.

```
A = "A sample string to be converted"
DISPLAY CONVERT("ert", "X", A)
```

would display

```
A samplX sing o bX convXXd
```

This provides an easy way to test whether a character string contains only a given set of characters. If we use the CONVERT() function to remove all the valid characters, anything that is left must be an invalid character.

```
LOOP
   INPUT @(16,5) ISBN, 13_:
UNTIL CONVERT('0123456789X-', '', ISBN) = ''
   INPUTERR 'Invalid ISBN'
REPEAT
```

The CONVERT Statement

The CONVERT statement is similar to the CONVERT() function except that the result overwrites the original source string.

```
CONVERT old TO new IN str
```

where

old is the set of characters to be replaced.

new is the corresponding set of replacement characters.

str is the string in which replacement is to occur.

A statement such as

```
CONVERT 'A' TO 'B' IN S
```

is identical in effect to

```
S = CONVERT('A', 'B', S)
```

The COMPARE() Function

The COMPARE() function compares two strings, returning a value indicating their correct order in the character sorting sequence.

```
COMPARE(str1, str2 {, just } )
```

where

str1 is the first string to be compared.

str2 is the second string to be compared.

just is L for a left justified comparison, R for a right justified comparison. If omitted, a left justified comparison is performed.

The COMPARE() function returns

-1 *str1* is before *str2*
0 *str1* is the same as *str2*
1 *str1* is after *str2*

When performing a right justified comparison, spaces are added to the start of the shorter string to make them of equal length and they are then compared character by character from the left hand end.

Note in particular the result of the following two tests:

`'0' = '00'` returns true as a numeric comparison is performed.

`COMPARE('0','00')` returns -1 as a string comparison is performed.

The INDEX() Function

The INDEX() function returns the position of a specified occurrence of a substring within a string.

```
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 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.

Example

```
ISBN = '1-102-42464-3'
N = INDEX(ISBN, "-", 3)
```

This statement assigns N with the character position of the third hyphen in variable ISBN (12).

The COUNT() Function

The COUNT() function counts occurrences of a substring within a string.

```
COUNT(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.

Examples

```
ISBN = '1-463-29233-2'
NUM.HYPHENS = COUNT(ISBN, '-')
```

This program fragment counts the hyphens in ISBN (3).

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.

The FIELD() Function

The FIELD() function returns one or more delimited substrings from a string.

```
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.

The COL1() and COL2() functions can be used to find the character positions of the extracted substring.

Examples

```
A = "1*2*3*4*5"
S = FIELD(A, "*", 2, 3)
```

This program fragment assigns the string "2*3*4" to variable S.

```
ISBN = '1-485-46324-4'
PART3 = FIELD(ISBN, '-', 3)
C1 = COL1()
C2 = COL2()
```

In this example, PART3 is set to "46324", C1 is set to 6 and C2 is set to 12.

Trimming Character Strings

The TRIM() function removes excess characters from a string.

```
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* .
- 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* . This is the default.
- 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 "BRCDBR"

```
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 "BRACADABR"

TRIMB() and TRIMF()

The TRIMB() function removes spaces from the back of a string. The TRIMF() function removes spaces from the front of a string.

```
TRIMB(string)
TRIMF(string)
```

where

string evaluates to the string to be trimmed.

Example

```
A = "  1  2  3  "
DISPLAY '"' : TRIMB(A) : '"'
DISPLAY '"' : TRIMF(A) : '"'
```

This program displays

```
"  1  2  3"
"1  2  3  "
```

The LEN() Function

The LEN() function returns the length of a string including any trailing spaces.

```
LEN(string)
```

where

string is the string for which the length is to be returned.

Example

```
LOOP
   DISPLAY "Enter account number: "
   INPUT ACCOUNT.NO
WHILE LEN(ACCOUNT.NO) # 6
   INPUTERR "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.

The SPACE() Function

The SPACE() function returns a string consisting of a given number of spaces.

```
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

```
DISPLAY SPACE(INDENT) : TEXT
```

This statement displays the contents of TEXT indented by the number of spaces specified by INDENT.

The STR() Function

The STR() function returns a string made up of a given number of repeated occurrences of another string.

```
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.

Example

```
DISPLAY STR("*", 79)
```

This statement displays a line of 79 asterisks.

The MATCHFIELD() Function

The MATCHFIELD() function extracts a portion of a string that matches a pattern element.

```
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
*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 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).

The MATCHFIELD() function returns a null string if the string does not completely match the pattern.

Examples

```
DISPLAY MATCHFIELD("ABC12DEF", "0A2N0A", 2)
```

display the string "12".

```
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).

The CHAR() and SEQ() Functions

The CHAR() function returns the character with a given ASCII collating sequence value. The SEQ() function returns the ASCII collating sequence value for a given character.

```
CHAR(seq)
SEQ(chr)
```

where

seq evaluates to an integer in the range 0 to 255.

chr evaluates to a single character.

The CHAR() function returns a single character string containing the ASCII character with value seq. It is the inverse of the SEQ() function which returns the ASCII sequence value of the supplied character.

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.

Exercise

Modify your ORDERS program to display the customer's address. This should be displayed under the customer's name and with the lines of the address merged, each separated by a comma and a space.

Order 12002 should now appear as shown below.

```
                                ORDER PROCESSING


 Order No: 12002     Date: 04 JUN 07

 Customer No: 1002   Ross, Alan
                     47 Warren Road, Hansworth, Birmingham

     Part  Description                          Price   Qty
Total
     013   Pencil, blue                         0.28      2
0.56
     012   Pencil, red                          0.28      3
0.84




                                                Order Total:




 Action (F/D/X):
```

Suggested Solution

This time, we have only shown the modified subroutine.

```
* ================================================================
* Display customer details

DISPLAY.CUSTOMER.DETAILS:
   CUST.NO = SAL.REC<SL.CUST>
   READ CUS.REC FROM CUS.F, CUST.NO THEN
      DISPLAY @(20,5) : CUS.REC<CS.NAME>
      DISPLAY @(20,6) : CHANGE(CUS.REC<CS.ADDR>, @VM, ', ')
   END

   RETURN
```

15.7 Dynamic Arrays

We have already seen some simple dynamic array handling operations in previous modules and you have used these in your programs. In this module, we will look at some of the more powerful features available with dynamic arrays.

Multi-valued Functions

There are a number of functions that provide multi-valued equivalents of other functions that we have already seen. 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.

For example, if we have two dynamic arrays

A contains ABC$_{FM}$DEF$_{FM}$GHI
and
B contains 123$_{FM}$456$_{FM}$789

We can concatenate these two dynamic arrays in two ways:

```
C = A : B
```
sets C to "ABC$_{FM}$DEF$_{FM}$GHI123$_{FM}$456$_{FM}$789"

```
C = CATS(A, B)
```
sets C to "ABC123$_{FM}$DEF456$_{FM}$GHI789"

The multi-valued string functions available are

CATS()	Concatenate elements of a dynamic array
COUNTS()	Multi-valued variant of COUNT()
FIELDS()	Multi-valued variant of FIELD()
FMTS()	Format elements of a dynamic array
ICONVS()	Perform input conversion on a dynamic array
INDEXS()	Multi-valued equivalent of INDEX()
NUMS()	Multi-valued variant of NUM()
OCONVS()	Perform output conversion on a dynamic array
SPACES()	Multi-valued variant of SPACE()
STRS()	Multi-valued variant of STR()
SUBSTRINGS()	Multi-valued substring extraction
TRIMBS()	Multi-valued variant of TRIMB()
TRIMFS()	Multi-valued variant of TRIMF()
TRIMS()	Multi-valued variant of TRIM()

There are also a number of multi-valued 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 11FM0VM17VMPQRFM2
and B contains 12FM0VM14VMABCFM2

```
C = GTS(A, B)
```

returns C as 0FM0VM1VM1FM0

The multi-valued logical functions are

ANDS()	Multi-valued logical AND
EQS()	Multi-valued equality test
GES()	Multi-valued greater than or equal to test
GTS()	Multi-valued greater than test
LES()	Multi-valued less than test
LTS()	Multi-valued less than or equal to test
NES()	Multi-valued inequality test
NOTS()	Multi-valued logical NOT
ORS()	Multi-valued logical OR

The IFS() function returns a dynamic array constructed from elements chosen from two other dynamic arrays depending on the content of a third dynamic array.

```
IFS(control.array, true.array, false.array)
```

where

control.array	is a dynamic array of true / false values.
true.array	holds values to be returned where the corresponding element of *control.array* is true.
false.array	holds values to be returned where the corresponding element of *control.array* is false.

The IFS() function examines successive elements of *control.array* and constructs a result array where elements are selected from the corresponding elements of either *true.array* or *false.array* depending on the *control.array* value.

Example

A contains 1VM0VM0VM1VM1VM1VM0
B contains 6VM2VM3VM4VM9VM6VM3
C contains 2VM8VM5VM0VM3VM1VM3

```
D = IFS(A, B, C)
```

returns D as 6VM8VM5VM4VM9VM6VM3

The LOCATE Statement

The LOCATE statement searches a dynamic array for a given field, value or subvalue.

Beware: This statement has different forms in the various multivalue products. By default, QM follows the "Information style" model as described below. There is a compiler option to select an alternative form of this statement discussed later.

```
LOCATE string IN array<field {, value {,subvalue}} {BY seq}
{SETTING var}
{THEN statement(s)}
{ELSE statement(s)}
```

where

string	evaluates to the item to be located.
array	is the dynamic array in which searching is to occur.
field	is the field at which the search is to commence.
value	is the value at which the search is to commence. If omitted or zero, the entire *array* is search for a field equal to *string* .
subvalue	is the subvalue at which the search is to commence. If omitted or zero, the specified field of *array* is search for a value equal to *string* .
seq	evaluates to the sequence string as described below. If omitted, no ordering is assumed.
var	is the variable to receive the position value. In QM, the SETTING clause is optional.
statement(s)	are statements to be executed depending on the outcome of the LOCATE action.

At least one of the THEN or ELSE clauses must be present.

The LOCATE statement searches for fields, values or subvalues depending on the position specified by field, value and subvalue. If *value* is not present, a search is performed for a field that matches *string* . If *value* is present but *subvalue* is absent or zero, a search is performed for a value within the specified field that matches string. If *value* and *subvalue* are both present, a search is performed for a subvalue within the specified field and value that matches string.

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>.

Searching commences at the position defined by *field* , *value* and *subvalue* .

- When *value* and *subvalue* are omitted or zero, the LOCATE looks for a field equal to *string* .

- When *value* is non-zero and *subvalue* is omitted or zero, the LOCATE looks for a value equal to *string* within the specified field. Searching does not continue beyond the end of the field.
- When *value* and *subvalue* are non-zero, the LOCATE looks for a subvalue equal to *string* within the specified value. Searching does not continue beyond the end of the value.

If a match is found, *var* is set to the *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 unsequenced LOCATE this will be such that it would be appended to the end of any exiting items.

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. Numeric items are compared as numbers, including negative values.

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.

Left aligned ordering is normally 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 THEN clause is executed if the string is found in array. The ELSE clause is found if the string is not found.

The result of a LOCATE statement with a specific ordering when applied to a dynamic array which does not conform to that ordering is undefined and likely to lead to misbehaviour of the program at run time.

Examples

Consider a college student registration database where the fields are:

1 Name
2 Multi-valued list of course codes for which the student has registered
3 Corresponding multi-valued list of grades scored for each course
4 Multi-subvalued lists of test paper scores for each course

A student might have the following dynamic array of course codes and results:

Smith, J[FM]M101[VM]M143[FM]A[VM]B[FM]97[SM]95[VM]87[SM]88

The LOCATE statement could be used to search the list of course codes:

```
LOCATE 'M143' IN VAR<2,1> SETTING POS
THEN DISPLAY 'Found at value ' : POS
ELSE DISPLAY 'Not found'
```

The above would find M143 as the second value in the specified field and set POS to 2. The THEN clause would be executed.

```
LOCATE 'M122' IN VAR<2,1> SETTING POS
THEN DISPLAY 'Found at value ' : POS
ELSE DISPLAY 'Not found'
```

The above would not find M122 in the specified field. POS would be set to 3 as the position at which the item could be appended to the list. The ELSE clause would be executed.

The list of course codes in field 2 appears to be sorted. Use of the BY clause would change the behaviour of the previous example.

```
LOCATE 'M122' IN VAR<2,1> BY 'AL' SETTING POS
THEN DISPLAY 'Found at value ' : POS
ELSE DISPLAY 'Not found'
```

The above would not find M122 in the specified field. POS would be set to 2 as the position at which the item should be inserted to maintain the correct sorted order of the list. The ELSE clause would be executed.

Alternative Forms

The LOCATE statement is implemented differently in the various multivalue database products. This can lead to confusion when you first start writing programs. Find out what variant is used by your business software and stay with just that one form.

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.

```
LOCATE string IN array{<field {,value }>} {, start} {BY seq}
{SETTING var}
{THEN statement(s)}
{ELSE statement(s)}
```

This format of LOCATE can be selected by including a line

```
$MODE UV.LOCATE
```

in the program on a line preceding the LOCATE statement (usually at the top of the program).

The Pick database uses a very different syntax.

```
LOCATE(string, array{,field {,value }}; var {; seq})
```

```
{THEN statement(s)}
{ELSE statement(s)}
```

This format 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.

QMBasic also supports a function version of LOCATE which can be useful in dictionary I-types:

var = LOCATE(*string*, *array*, *field* {,*value* {, *subvalue* }} {; *seq*})

The LOCATE() function returns as its result the position at which the item was found, or zero if it was not found. Although the *seq* 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.

Inserting Items - The INS Statement

We often want to insert items into an existing dynamic array. The INS statement allows us to do this very easily. The INSERT() function performs the same task but returns a new dynamic array, leaving the original dynamic array unchanged.

```
INS string BEFORE array<field {, value {, subvalue}}>
INSERT(array, field {, value {, subvalue}} , string)
```

where

string	is the string to be inserted.
array	is the dynamic array into which the item is to be inserted.
field	is the number of the field before which insertion is to occur.
value	is the number of the value before which insertion is to occur. If omitted or zero, value 1 is assumed.
subvalue	is 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 *array* variable, overwriting the original data. The INSERT() function returns the result without modifying *array*.

Example

Consider the same dynamic array as in earlier examples.

Smith, J[FM]M101[VM]M143[FM]A[VM]B[FM]97[SM]95[VM]87[SM]88

A program to add a new course to the list might contain statements of the form:

```
INPUT COURSE
LOCATE COURSE IN VAR<2,1> BY 'AL' SETTING POS THEN
   DISPLAY 'Student is already registered for this course'
END ELSE
   INS COURSE BEFORE VAR<2,POS>
   INS '' BEFORE VAR<3,POS>
   INS '' BEFORE VAR<4,POS>
END
```

Adding course M120 would result in

Smith, J[FM]M101[VM]M120[VM]M143[FM]A[VM][VM]B[FM]97[SM]95[VM][VM]87[SM]88

Note that it is the programmer's responsibility to maintain the relationship between associated fields. The null strings inserted into fields 3 and 4 create a space for the results of the new course to appear later. For a simple association with only two or three fields, this would usually be done as in our example. Program maintenance may be simplified with a more complex data structure that has many associated fields by having a subroutine that inserts a row into the association. This subroutine encapsulates maintenance of the association and can be called from any place in the application that needs to add an entry.

Deleting Items - The DEL Statement

The DEL statement deletes a field, value or subvalue from a dynamic array. The DELETE() function performs the same task but returns a new dynamic array, leaving the original dynamic array unchanged.

```
DEL array<field {, value {, subvalue}}>
DELETE(.array, field {, value {, subvalue}})
```

where

array	is the dynamic array from which the item is to be deleted.
field	evaluates to the number of the field to be deleted.
value	evaluates to the number of the value to be deleted. If omitted or zero, the entire field is deleted.
subvalue	evaluates to the number of the subvalue to be deleted. If omitted or zero, the entire value is deleted.

The specified field, value or subvalue of the dynamic array is deleted. The DEL statement assigns the result to the array variable, overwriting the original data. The DELETE() function returns the result without modifying array.

Example

Again, consider our student registration record

Smith, J[FM]M101[VM]M143[FM]A[VM]B[FM]97[SM]95[VM]87[SM]88

A program to delete a course from the list might contain statements of the form:

```
INPUT COURSE
LOCATE COURSE IN VAR<2,1> BY 'AL' SETTING POS THEN
   DEL VAR<2,POS>
   DEL VAR<3,POS>
   DEL VAR<4,POS>
END ELSE
   DISPLAY 'Student is not registered for this course'
END
```

Using this program to delete course M143 would result in

Smith, J[FM]M101[FM]A[FM]97[SM]95

Again, note that it is the programmer's responsibility to maintain the relationship between associated fields.

Replacing Fields, Values and Subvalues

The programs that you have written so far have already included statements to replace the value in a given field of a dynamic array. Equivalent statements allow us to replace values or subvalues.

```
var<field> = new.value
var<field, value> = new.value
var<field, value, subvalue> = new.value
```

If the given field, value or subvalue does not already exist, mark characters are added as necessary to create this new item.

The REPLACE() function returns a modified dynamic array, leaving the original array unchanged.

```
REPLACE(array, field {, value {, subvalue}} , new.value)
```

where

array	evaluates to a string in which the replacement is to occur.
field	evaluates to the field position number. If zero, this argument defaults to one.
value	evaluates to the value position number. If omitted or zero, the entire field is replaced.
subvalue	evaluates to the subvalue position number. If omitted or zero, the entire value is replaced.
new.value	evaluates to the replacement data.

If *field*, *value* and *subvalue* are not all present, the comma before the *new.value* argument must be replaced by a semicolon.

Examples

Perhaps, the grade recorded in our student registration database is incorrect.

Smith, J[FM]M101[VM]M143[FM]A[VM]B[FM]97[SM]95[VM]87[SM]88

A program to modify the grade might contain statements of the form:

```
INPUT COURSE
LOCATE COURSE IN VAR<2,1> BY 'AL' SETTING POS THEN
   INPUT GRADE
   VAR<3,POS> = GRADE
END ELSE
   DISPLAY 'Student is not registered for this course'
END
```

Consider a dynamic array named X containing

AFMB1VMB2FMC

The following two statements

```
X<5> = 'E'
X<2,5> = 'B5'
```

would result in the dynamic array containing

AFMB1VMB2VMVMVMB5FMCFMFME

Before this update, field 4 was implicitly empty. After the update it is explicitly empty. The modification has not changed field 4 in any way. The significance of this is that it does not matter in what order an application populates a dynamic array. The mark characters will be correctly maintained automatically.

Appending to Lists of Unknown Length

There is an easy way to append a new field to the end of a dynamic array, a value to the end of a field, or a subvalue to the end of a value without needing to know how many items are already present. This makes use of a special syntax where the field, value or subvalue number is given as -1.

```
var<-1> = new.value
var<field, -1> = new.value
var<field, value, -1> = new.value
```

Example

Once more, consider our student record

Smith, JFMM101VMM143FMAVMBFM97SM95VM87SM88

The statement

```
VAR<2,-1> = 'M152'
```

would amend the dynamic array to become

Smith, JFMM101VMM143VMM152FMAVMBFM97SM95VM87SM88

The LISTINDEX() Function

The LISTINDEX() function is broadly similar to LOCATE() but searches for an item in a list delimited by any single character.

```
LISTINDEX(list, delimiter, item)
```

where

list	is the list to search
delimiter	is the single character delimiter that separates items in *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.

The DCOUNT() Function

We often want to know how many items there are in a list. The DCOUNT() function counts items separated by some delimiter character.

```
DCOUNT(list, delim)
```

where

list	is the delimited character string in which the count is to be performed.
delim	evaluates to the single character delimiter between each item in *list*.

Examples

For our student registration record

Smith, J[FM]M101[VM]M143[FM]A[VM]B[FM]97[SM]95[VM]87[SM]88

the statement

```
NUM.COURSES = DCOUNT(VAR<2>, @VM)
```

could be used to count the values in field 2.

If the student has taken only one course, there is no value mark present

Smith, J[FM]M101[FM]A[FM]97[SM]95

The statement

```
NUM.COURSES = DCOUNT(VAR<2>, @VM)
```

will still work correctly, returning the value 1.

If the student has registered, but not yet taken any courses, the field holding the list of courses will be empty or may not even exist

```
Smith, J
```

In this case, the statement

```
NUM.COURSES = DCOUNT(VAR<2>, @VM)
```

would return zero. DCOUNT() returns one more than the number of delimiters unless the string is empty, in which case it returns zero.

The REMOVE Statement

The most efficient way to process every element of a dynamic array in turn is by use of the REMOVE statement.

```
REMOVE item FROM var SETTING delim
```

where

item is the variable to receive the extracted data.

var is the dynamic array from which data is to be extracted.

delim is a variable to receive a code indicating the delimiter character the terminated data extraction.

Whenever a character string value is stored in a variable, an associated **remove pointer**, is set to zero. The REMOVE statement extracts characters from the string in *var* starting at the character offset one greater than the remove pointer value. Because there is no searching involved, this operation is very fast.

Each character extracted is copied into *item*.

Extraction terminates when a mark character is found. The mark character is not stored in item. The remove pointer is set to the offset of this mark within the string.

If the end of the string is reached, the remove pointer is set to one more than the length of the string. A further REMOVE would return a null item, leaving the remove pointer unchanged.

The delim variable is set according to the mark character:

0 End of string reached
1 Item mark
2 Field mark
3 Value mark
4 Subvalue mark
5 Text mark

The REMOVE() function performs the same task but returns the extracted data as its result.

```
REMOVE(var, delim)
```

The remove pointer can be repositioned using the SETREM statement:

```
SETREM pos ON var
```

where

pos is the value to be set in the remove pointer

var is the variable to which this applies

Although setting the remove pointer to zero is the best way to position it at the start of the string, SETREM is not found in all multivalue products. Developers often use a strange statement of the form

```
var = var
```

to rewind the remove pointer. This works because assigning a value to a string resets the pointer, even if the value assigned is the current content of the variable.

Examples

The REMOVE statement or corresponding function is frequently used to process a list of items stored in a field of a database file.

Consider our SALES file. This includes a field holding the products purchased by the customer. We could process this in a loop such as

```
PARTS = SAL.REC<SL.ITEM>
IF PARTS # '' THEN
   LOOP
      DISPLAY REMOVE(PARTS, MORE)
   WHILE MORE
   REPEAT
END
```

Note that, although probably irrelevant in this example, we have included a check that the product list is not empty before we enter the loop. This is because we would otherwise perform one cycle of the loop, displaying a blank line.

We can also use REMOVE to break a long text string across multiple lines. The text mode justification of the FMT() function can be used to insert text mark characters such that no portion of the string exceeds some given length. The REMOVE statement or function could then be used to extract these sections in turn.

```
DESC = FMT(STK.REC<ST.DESCR>, '25T')
LOOP
   DISPLAY REMOVE(DESC, MORE)
WHILE MORE
REPEAT
```

The REMOVEF() Function

QMBasic provides a more generalised string function that extracts items using the optimisation provided by the remove pointer.

```
REMOVEF(string {, delim {, count}})
```

where

string is the string from which data is to be extracted.

delim is the delimiter character that separates elements of *string* . If omitted, a field mark is used. If delimiter is more than one character, only the first character is used. If *delim* is a null string, *count* characters are extracted.

count is the number of consecutive delimited elements of *string* to be extracted. If omitted or less than one, this defaults to one.

The REMOVEF() function uses the same optimised method as the REMOVE() function to extract items from string sequentially but uses a specified delimiter character instead of terminating on any mark character.

Whenever a string is assigned to a variable the remove pointer is set to point one character before the start of the string. Subsequent uses of REMOVEF() advance the point by one character and then extract characters from the position of the remove pointer up to the next delimiter character or the end of the string. Because the remove pointer gives immediate access to the position at which the REMOVEF() should commence, this operation requires no searching and is therefore very fast.

Once the end of the string has been reached, the remove pointer remains positioned one character beyond the end of the string and further REMOVEF() operations would return a null string.

The REMOVEF() function uses the STATUS() function to return information about its outcome:

0 Successful
1 Null string
2 End of string

The remove pointer may be repositioned as described above for the REMOVE statement.

Example

```
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.

Numeric Arrays

A numeric array is a dynamic array in which all of the elements contain numbers. QMBasic provides several methods to perform operations on each value of the array in a single statement or function.

The arithmetic operators (*, /, +, -) all work on corresponding pairs of values when the items on either side of the operator are numeric arrays. For example,

A contains 12FM9FM6FM18
B contains 4FM3FM2FM12

```
C = A + B
```
sets C to 16FM12FM8FM30

```
C = A / B
```
sets C to 3FM3FM3FM1.5

What if there are fewer items in B than in A?

A contains 12FM9FM6FM18
B contains 4FM3FM2

```
C = A + B
```
sets C to 16FM12FM8FM18

```
C = A / B
```
sets C to 3FM3FM3FM18

The addition has assumed that the "missing" element of B is zero. This is also true for subtraction and multiplication. For division, the missing item is assumed to be 1 if it is the divisor to avoid a divide by zero error.

Sometimes, we want to reuse the last value in place of the missing item. The REUSE() function allows us to do this.

A contains 12FM9FM6FM18
B contains 4FM3FM2

`C = A + REUSE(B)` sets C to 16FM12FM8FM20

This function is often used when its argument is a single value, perhaps even a constant. For example, to add 17.5% tax to a list of prices, we could write

```
SALE.PRICE = EX.TAX.PRICE * REUSE(1.175)
```

The REUSE() function is frequently used in dictionary I-type items. Combined with the multi-valued functions discussed earlier in this section, it is possible to write some very elegant solutions to apparently complex problems.

Summing the Items

The SUM() function eliminates the lowest level of a numeric array by adding the elements to form an item of the next highest level.

```
SUM(expr)
```

where

expr is 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.

Example

An invoicing system might have a PAYMENTS field containing a list of payments against the invoice. We can find the total of all payments by a statement such as

```
TOTAL.PAID = SUM(PAYMENTS)
```

Exercise

Extend the action prompt in your ORDERS program to allow entry of a part number. Check that the part exists and, if not, display an error message.

Examine the order using LOCATE to see if this part is already included. If so, allow the user to adjust the quantity. If a quantity of zero is entered, use DEL to remove the part and associated quantity and price from the order. This will require redisplay of the order detail lines. Your existing subroutine should do this though it may need some slight modification to remove the final line from the display.

Suggested Solution

Only the modified subroutines are shown below. Depending on the mode settings of QM used on your system, the LOCATE statement may be written differently. The example below is for the default "Information style" syntax.

This example also shows how the next record id in the SALES file is tracked by use of a record named NEXT.ID in the file's dictionary and assumes that the dictionary has been opened earlier in the program using a file variable named SAL.D.

```
* ================================================================
* Paint fixed part of screen

PAINT.SCREEN:
   DISPLAY @(-1) :
   DISPLAY @(32,0) : 'ORDER PROCESSING' :
   DISPLAY @(0,3)  : 'Order No:' :
   DISPLAY @(20,3) : 'Date:' :
   DISPLAY @(0,5)  : 'Customer No:' :
   DISPLAY @(5,8)  : 'Part' :
   DISPLAY @(11,8) : 'Description' :
   DISPLAY @(48,8) : 'Price' :
   DISPLAY @(56,8) : 'Qty' :
   DISPLAY @(65,8) : 'Total' :
   DISPLAY @(48,14) : 'Order Total:'
   DISPLAY @(0,20) : 'Action (F/D/X/part):'

   RETURN


* ================================================================
* Show order details

DISPLAY.ORDER.LINES:
   LN = 9
   FOR IDX = 1 TO 5
      PART.NO = SAL.REC<SL.ITEM,IDX>
      IF PART.NO = '' THEN
         DISPLAY @(0,LN) : @(-4) :   ;* Clear unused line
      END ELSE
         PRICE = SAL.REC<SL.PRICE,IDX>
         QTY = SAL.REC<SL.QTY,IDX> + 0
         DISPLAY @(5,LN) : PART.NO
         DISPLAY @(45,LN) : FMT(OCONV(PRICE, CASH.CONVERSION),
'7R')
         DISPLAY @(55,LN) : FMT(QTY, '4R')
         READ STK.REC FROM STK.F, PART.NO THEN
            DISPLAY @(11,LN) : STK.REC<ST.DESCR>[1,30]
            DISPLAY @(61,LN) : FMT(OCONV(PRICE * QTY,
CASH.CONVERSION), '9R')
         END
      END
      LN += 1
   NEXT IDX
```

```
   RETURN

* ================================================================
* Action prompt

ACTION.PROMPT:
   FINISHED = @FALSE
   LOOP
      ACTION = ''
      INPUT @(21,20) ACTION,3_

      BEGIN CASE
         CASE ACTION = 'D'  ;* Delete
            * Order deletion to be added here
            FINISHED = @TRUE

         CASE ACTION = 'F'  ;* File
            * If we are creating a new order, we have left
generating
            * the order number until now.

            IF ORDER.NO = '' THEN
               READU NEXT.ORDER FROM SAL.D, "NEXT.ID" ELSE
                  STOP "Cannot find NEXT.ID record"
               END
               ORDER.NO = NEXT.ORDER<2>
               NEXT.ORDER<2> = ORDER.NO + 1
               WRITE NEXT.ORDER TO SAL.D, "NEXT.ID"
               DISPLAY @(10,3) : ORDER.NO :
            END

            WRITE SAL.REC TO SAL.F, ORDER.NO
            DISPLAY @(0,23) : 'Order confirmed. Press return' :
            INPUT JUNK, 1_:

            FINISHED = @TRUE

         CASE ACTION = 'X'  ;* Exit
            IF ORDER.NO # '' THEN RELEASE SAL.F, ORDER.NO
            FINISHED = @TRUE

         CASE ACTION MATCHES '3N'  ;* Part number?
            PART.NO = ACTION
            READ STK.REC FROM STK.F, PART.NO THEN
               PRICE = SKT.REC<ST.PRICE>  ;* May need to revise
price

               LOCATE PART.NO IN SAL.REC<SL.ITEM,1> SETTING IDX
               THEN     ;* Modifying quantity
                  LN = 8 + IDX
                  LOOP
                     QTY = SAL.REC<SL.QTY,IDX>
                     INPUT @(55,LN) QTY,4_:
                  UNTIL QTY MATCHES '1-4N'
                     INPUTERR 'Invalid quantity'
                  REPEAT
```

```
                  IF QTY = 0 THEN
                     DEL SAL.REC<SL.ITEM,IDX>
                     DEL SAL.REC<SL.QTY,IDX>
                     DEL SAL.REC<SL.PRICE,IDX>
                     GOSUB DISPLAY.ORDER.LINES
                  END ELSE
                     DISPLAY @(55,LN) : FMT(QTY, '4R') :
                     SAL.REC<SL.QTY,IDX> = QTY
                     SAL.REC<SL.PRICE,IDX> = PRICE

                     DISPLAY @(61,LN) : FMT(OCONV(PRICE * QTY,
CASH.CONVERSION), '9R') :
                  END
               END ELSE
                  INPUTERR 'Part number is not in order'
               END
            END ELSE
               INPUTERR 'Part number is not known'
            END

         CASE 1
            INPUTERR 'File, Delete, eXit, part number'
      END CASE
      UNTIL FINISHED
   REPEAT

   RETURN
END
```

15.8 Matrix File Operations

Matrix Variables

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 in the source program on a line which precedes any other reference. The DIM statement must also be executed at program run time before the variable is used in any other way.

A one dimensional matrix of five elements with index values 1 to 5 is defined by a statement of the form

DIM A(5)

A(1)	A(2)	A(3)	A(4)	A(5)

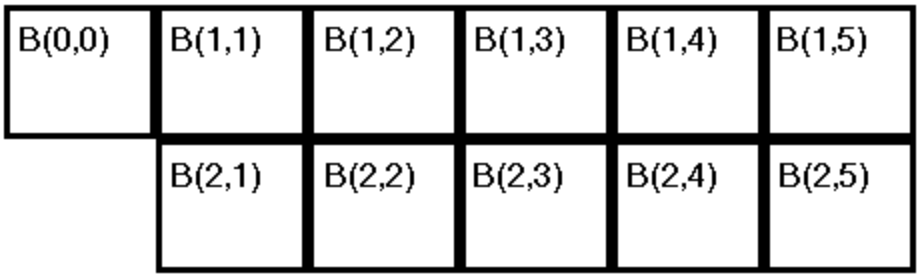
For a two dimensional matrix with 2 rows of 5 columns this becomes

DIM B(2,5)

B(1,1)	B(1,2)	B(1,3)	B(1,4)	B(1,5)
B(2,1)	B(2,2)	B(2,3)	B(2,4)	B(2,5)

By default, all matrices have an additional element, the zero element, which is used by some statements. This is referred to as A(0) or B(0,0).

A(0)	A(1)	A(2)	A(3)	A(4)	A(5)



The elements of a matrix may hold data of differing types. They may be used anywhere that a simple variable can be used

Every element of a matrix can be set to the same value by a statement of the form

```
MAT A = value
```

The zero element is set to a null string regardless of the *value* assigned to the remaining elements.

One matrix can be copied to another by a statement of the form

```
MAT A = MAT B
```

which copies all elements, including the zero element. A single dimensional matrix can be copied to a two dimensional matrix and vice versa.

To understand the effect if the matrices are not of the same dimensionality or size, consider the copy process as walking through the matrix left to right, row by row until the end of either matrix is reached. If the source matrix has more elements than the target matrix, the excess elements are ignored. If the target matrix has more elements that the source matrix, the remaining elements are unchanged.

Matrix File Operations

The file handling operations that we have seen so far read and write data using dynamic arrays. These are very efficient where we perform relatively little processing of each record or where the records have relatively few fields.

If we are going to perform a significant amount of processing on data records with many fields, the cost of searching through the dynamic array to find each field as it is required may be such that it is better to break the record into individual fields. QM provides a set of file handling operations to break the fields of a data record across successive elements of a dimensioned matrix.

The MATREAD Statement

The MATREAD statement reads a database record, placing each field in a separate element of a dimensioned array.

```
MATREAD mat FROM file.var, record.id
{ON ERROR statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

where

mat	is the name of a dimensioned array to receive the data read from the file.
file.var	is the file variable associated with the file.
record.id	evaluates to the id of the record to be read.
statement(s)	are statements to be executed depending on the outcome of the MATREAD operation.

This statement works in the same way as the READ statement except that each field of the record is placed in a separate element of the target matrix.

There are also corresponding MATREADU and MATREADL statements that take a lock on the record. These have an optional LOCKED clause that works in the same way as in their dynamic array counterparts.

Example

```
DIM STK.REC(10)
...
MATREAD STK.REC FROM STK.F, PART.NO THEN
   DISPLAY PART.NO
   DISPLAY 'Description = ' : STK.REC(ST.DESCR)
   DISPLAY 'Stock level = ' : STK.REC(ST.QTY)
END ELSE
   DISPLAY 'Record ' : PART.NO: ' not found'
END
```

In this example we have created a dimensioned array named STK.REC to hold a STOCK file record and we use MATREAD in place of READ to read the record. The number of elements in the STK.REC matrix is greater than the number of fields in a stock record. The unused fields will be set to a null string.

If there are more fields in the data record than there are elements in the dimensioned array, the excess fields are stored in the zero element together with any intervening field marks.

All references to fields in the record within the application now become indexed references to matrix elements. With so little processing of the record, the cost of breaking the record apart during the MATREAD would be greater than the performance gained by removing the need for dynamic array field searches. In a more realistic program with much larger record structures, there may be significant advantage in this technique.

The EQUATE statement allows us to give names to elements of a dimensioned array. We could modify the above example to become

```
DIM STK.REC(10)
EQUATE STK.DESCR    TO STK.REC(1)
EQUATE STK.QTY      TO STK.REC(2)
EQUATE STK.PRICE    TO STK.REC(3)
...etc...
...
MATREAD STK.REC FROM STK.F, PART.NO THEN
   DISPLAY PART.NO
   DISPLAY 'Description = ' : STK.DESCR
   DISPLAY 'Stock level = ' : STK.QTY
END ELSE
   DISPLAY 'Record ' : PART.NO: ' not found'
END
```

By placing the EQUATE token definitions in an include record, they can be imported into all programs that need them.

The MATWRITE Statement

The MATWRITE statement builds a record from successive elements of a dimensioned array and writes it to a file.

```
MATWRITE mat TO file.var, record.id
{ON ERROR statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

where

mat	is the name of the matrix 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 depending on the outcome of the MATWRITE.

MATWRITE constructs a dynamic array by concatenating the 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(s) of the record.

The ON ERROR, THEN and ELSE clauses work as in the WRITE statement.

There is also a corresponding MATWRITEU statement which retains the lock after writing the record.

The way in which MATREAD and MATWRITE use the zero element means that it acts as an "overflow bucket" for fields in the record that the program did not expect to be present. If new fields are added to an existing record structure, only the programs that access those fields need modifying. Other programs will correctly maintain the values of the unexpected fields by use of the zero element as temporary storage.

Pick systems do not have a zero element in a dimensioned matrix. Instead, any unexpected data is stored in the final element of the matrix. For this reason, Pick programmers typically declare the matrix to be at least one element larger than the record size, effectively moving the zero element to the end of the matrix.

Because of an interaction with common blocks (discussed later), QM supports Pick style matrices with no zero element. These are enabled using the PICK.MATRIX setting of the $MODE compiler directive. See the *QM Reference Manual* for more details.

The language includes two statements, MATPARSE and MATBUILD, which can be used to break a dynamic array across a dimensioned matrix or to build a dynamic array from a dimensioned matrix. These statements are not discussed further here.

15.9 Select Lists

A select list is a list of items, usually record ids, to be processed. As we saw earlier, the query processor can be used to construct select lists based on detailed selection criteria. In this section, we will look at the QMBasic SELECT statement that builds a list of all of the records in a file.

Every QMBasic program has access to the eleven numbered select lists (0 to 10) used by the query processor. The QMBasic statements that we meet in this section will use list 0, the default select list, unless we include clauses to direct use of an alternative list.

The QMBasic SELECT statement creates a list of all of the records in an open file for subsequent processing. Record ids are extracted from the list one by one using the READNEXT statement.

The syntax of SELECT is

```
SELECT file.var {TO list.no} {ON ERROR statement(s)}
```

where

file.var	is the file variable associated with an open file.
list.no	is the select list number of the list to be created. If omitted, select list zero is used.
statement(s)	are statement(s) to be executed if a fatal error occurs.

A list of record keys in the file open as *file.var* is created and stored as an active select list *list.no* replacing any previously active list.

Although this may sound inefficient as it seems to require us to read the entire file as part of the SELECT statement, QM uses an optimised method such that each group is examined only when the record keys are extracted from the select list by a subsequent READNEXT or other list processing statement. The SELECT statement actually does no more than set a pointer to the start of the file. It is READNEXT that actually does the work. This reduces disk transfers and gives better application performance than constructing the entire list and then processing the records.

The syntax of READNEXT is

```
READNEXT var {FROM list.no}
{ON ERROR statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

where

var	is the variable to receive the select list item.

list.no	is the select list number. If omitted, select list zero is used.
statement(s)	are statement(s) to be executed depending on the outcome of the READNEXT operation.

At least one of the THEN or ELSE clauses must be present.

The next item in the specified select list is removed from the list and stored in *var*.

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.

Example

```
OPEN 'CUSTOMERS' TO CUS.F
   ELSE ABORT 'Cannot open CUSTOMERS file'
SELECT CUS.F
LOOP
   READNEXT CUS.ID ELSE EXIT
   READ CUS.REC FROM CUS.F, CUS.ID THEN
      DISPLAY CUS.ID : '  ' : CUS.REC<CS.NAME>
   END
REPEAT
```

The above example displays a list of record ids and customer names from the CUSTOMERS file. Note the use of EXIT. We could not have used UNTIL here without introducing a flag variable to handle the end of the list.

The CLEARSELECT Statement

Perhaps we have a program that must find a record that meets some condition but only needs to find one such record. For example, we might want to find any order that was place by a specific customer. We could do this with a loop such as

```
OPEN 'SALES' TO SAL.F ELSE ABORT 'Cannot open SALES file'
SELECT SAL.F
LOOP
   READNEXT SAL.ID ELSE EXIT
   READ SAL.REC FROM SAL.F, SAL.ID THEN
      IF SAL.REC<SL.CUST> = CUST.NO THEN
...         processing of this record...
         EXIT
      END
   END
REPEAT
...further processing...
```

This loop would leave a partially processed list in the default select list. Later processing in the program might not work correctly because of the way in which this list is used automatically by many parts of QM. We must, therefore, discard any remaining items in the list using the CLEARSELECT statement.

```
CLEARSELECT { list.no }
CLEARSELECT ALL
```

The first form clears the specified select list. If *list.no* is omitted, the default list is cleared. The second form clears all select lists.

Transferring Select Lists To/From Dynamic Arrays

Two statements are provided to create a select list from a dynamic array or vice versa.

```
FORMLIST dyn.array { TO list.no }

READLIST dyn.array { FROM list.no }
THEN statement(s)
ELSE statement(s)
```

The dynamic array version of the list is delimited by field marks.

Exercise

Modify the subroutine that handles entry of customer numbers in your ORDERS program to display a list of customers if the user enters a question mark.

Initially assume that the entire list will fit on a single screen. You will need to clear the screen before displaying the list. After the list has been displayed, wait for the user to press the return key, repaint the screen and prompt for the customer number again.

Once this is working, consider how you would need to modify the program to allow for a customer list that spanned many pages.

Suggested Solution

```
* ================================================================
* Get customer number

GET.CUST.NO:
   CUST.NO = ''
   LOOP
      INPUT @(13,5) : CUST.NO, 4_:
      IF CUST.NO = '?' THEN
         DISPLAY @(-1) :
         SELECT CUS.F
         LOOP
            READNEXT CUST.NO ELSE EXIT
            READ CUS.REC FROM CUS.F, CUST.NO THEN
               DISPLAY CUST.NO : '  ' : CUS.REC<CS.NAME>
            END
         REPEAT
         INPUT JUNK,1_:
         GOSUB PAINT.SCREEN
         GOSUB PAINT.DATA
      END ELSE
         READ CUS.REC FROM CUS.F, CUST.NO THEN EXIT
         INPUTERR "Customer " : CUST.NO : " is not known"
      END
   REPEAT

   SAL.REC<SL.CUST> = CUST.NO

   RETURN
```

15.10 External Subroutines

External subroutines are totally separate program modules. They are entered using the CALL statement. The RETURN statement is again used to return to the caller.

A real application will typically have hundreds or perhaps thousands of modules called in this way. Unlike most programming languages, the modules that make up a QMBasic application are not linked together during development to make a single executable file. Instead, this linking takes place dynamically at run time. When the application first attempts to call a subroutine, the system looks for it in the **catalogue**, loads it into memory and continues. This mechanism results in QM processes only loading the parts of the application that they need, often giving a substantial reduction in memory requirements.

An external subroutine has its own local variables which are totally separate from those of the caller. The application designer, therefore, does not have to check for uses of the same names in other modules.

The CALL Statement

There are two styles of call available. A direct call refers directly to the name of the subroutine to be called:

```
CALL MYSUB
```

An indirect call refers to a variable which holds the name of the subroutine:

```
VAR = 'MYSUB'
CALL @VAR
```

Indirect calls allow programs to derive the name of the subroutine to be called from lookup tables, etc. In a real program, the indirection variable would normally be set as a result of some data extraction or calculation, not as a simple literal name. The variable holding the subroutine name should only be reassigned when the name of the subroutine changes.

External subroutines may call further external subroutines to any depth (subject to available memory) and may also include internal subroutines.

Argument Passing

Given that an external subroutine has its own local variables and has no automatic way to access the variables of the calling program, we need some way to pass information in and out of the subroutine. This is achieved by passing **arguments**.

The CALL statement is extended to add a list of variables or expressions to be passed into the subroutine.

```
CALL MYSUB(A, B+6, 99)
```

Subroutines called with arguments must commence with a SUBROUTINE statement in place of the (optional) PROGRAM statement. The SUBROUTINE statement includes a list of names to be used within the subroutine to refer to the items in the CALL.

```
SUBROUTINE MYSUB(P, Q, R)
```

There must be the same number of arguments in the SUBROUTINE statement as in the CALL. There is a limit of 255 arguments in a call.

In this example, the variable that the subroutine references as P would be the variable named A in the calling program, Q would be the result of adding 6 to B and R would be the constant 99.

The argument passing mechanism of QMBasic is what is known technically as a pass **by reference**. The subroutine's P variable is not a copy of the caller's A but is a reference to it. Therefore, if the subroutine changes the value of P, the change will be visible in A on return to the calling program.

The second and third arguments pass across the result of evaluating an expression and a constant. These are passed **by value**. Any changes that the subroutine makes to its variables Q and R have no effect on the calling program's data. Only simple variables are passed by reference.

A CALL statement may force an argument to be passed by value by enclosing it in parentheses, effectively making it into an expression:

```
CALL MYSUB((A), B+6, 99)
```

QMBasic extends the language definition found in other multivalue systems to allow the SUBROUTINE statement to show that an argument is to be passed by value. This is done by enclosing the argument variable name in parenthesis:

```
SUBROUTINE MYSUB((P), Q, R)
```

A PROGRAM is simply a SUBROUTINE with no arguments. It is therefore possible to CALL a program. It is not possible to RUN a subroutine that requires arguments from the command prompt.

The argument list may also pass across a whole dimensioned matrix by prefixing the argument name in the CALL with MAT. The subroutine's argument list must also include the MAT prefix on the relevant argument. Because the compiler needs to know whether this is a one or two dimensional matrix, there must be a DIM statement in the subroutine, though the actual values for the dimensions are ignored and are frequently given as 1 to emphasise that they are meaningless.

Calling program:

```
DIM A(100,5)
CALL MYSUB(MAT A)
```

Subroutine:

```
SUBROUTINE MYSUB(MAT X)
DIM X(1,1)
```

Alternatively, QMBasic allows the dimensionality of the matrix to be included in the SUBROUTINE statement

```
SUBROUTINE MYSUB(X(1,1))
```

Again, the actual values of the dimensions are ignored.

Common Variables

We can define blocks of variables that are to be shared between programs by use of the COMMON statement.

```
COMMON var1, var1, var3...
```

Every program that includes the same COMMON statement will access the same variables *var1*, *var2*, *var3*, etc. Data stored in these variables in one program can be seen by the other programs. Note that the common data is only shared between programs in the same process, not between separate QM sessions.

The variable names are written as a comma separate list that may span multiple lines. Alternatively, the example above could be written as

```
COMMON var1
COMMON var2
COMMON var3
...etc...
```

The COMMON statement should be identical in all programs using the variables. With large common blocks and realistic names it is very easy to write

```
COMMON A, B, C, D
```

in one program and

```
COMMON A, C, B, D
```

in another. The names are only meaningful within the program module. The underlying QMBasic processing simply treats this as a common block with four variables. Now what the first program stores in B will be seen by the second program in C.

One way to ensure that this does not happen is to place the COMMON statement in an include record that is then referenced by every program that needs access to these variables. Any change to the common block requires only that the include record is edited and all programs using it are recompiled.

The common block defined using a COMMON statement of the form above is known as the unnamed common. It is created when first referenced by a program and is deleted on return to the command prompt. There are also named common blocks, defined by a statement of the form

```
COMMON /name/ var1, var2, var3...
```

Named common blocks are created when first referenced by a program and remain in existence until the user leaves QM. A typical application will have several named common blocks, each containing variables related to some aspect of the application (file handling, user interface, etc). By including each common block only in the programs that need it, maintenance becomes easier.

Variables in common blocks are initially zero. We can make use of this to detect whether the data in a common block has been set up. For example, common blocks are frequently used to store file variables so that the files need only be opened once.

```
COMMON /FILEVARS/ SAL.F,        ;* SALES
                  STK.F,        ;* STOCK
```

```
                    CUS.F,        ;* CUSTOMERS
                    FILES.OPEN
```

The name of the common block should be chosen to avoid possible clashes with other application software used from the same account. The name used in the example above might well conflict between applications.

The include record may have any name that the application designer wishes to use. It is useful to base it on the name of the common block that it defines. The name often has a suffix of .H to indicate that it is a header file (a convention that has found its way into QMBasic from the C programming language).

A named common block cannot be redefined with a different size once it has been created. If you modify the structure of a common block you must recompile all programs that use it. Because the block normally persists until you leave QM, you must either quit from QM or use the DELETE.COMMON command to discard any existing memory copy of the block.

We could write a subroutine to open the files as below.

```
SUBROUTINE OPEN.FILES
$INCLUDE FILEVARS.H

   IF NOT(FILES.OPEN) THEN
      OPEN 'SALES' TO SAL.F
         ELSE ABORT 'Cannot open SALES'
      OPEN 'STOCK' TO STK.F
         ELSE ABORT 'Cannot open STOCK'
      OPEN 'CUSTOMERS' TO CUS.F
         ELSE ABORT 'Cannot open CUSTOMERS'
      FILES.OPEN = @TRUE
   END
   RETURN
END
```

Each program module that uses any of these files would contain a $INCLUDE line referencing the common block definition:

```
$INCLUDE FILEVARS.H
```

Each program module that might need to open the files (because it is an entry point to the application) would also contain a call to the OPEN.FILES subroutine:

```
$INCLUDE FILEVARS.H
CALL OPEN.FILES
```

There would be some minor performance benefit in writing the above as

```
IF NOT(FILES.OPEN) THEN CALL OPEN.FILES
```

to save unnecessary calls to the OPEN.FILES subroutine, however, this breaks one of the rules of structured program design by including knowledge of the subroutine's operation in the calling programs.

Cataloguing Programs

An external subroutine must be placed into the system catalogue before it can be called from other programs. This is done using the CATALOGUE command. The American spelling (CATALOG) is supported for compatibility with other multivalue products.

```
CATALOGUE file.name call.name { record.id } { options }
```

where

file.name	is the name of the program file. The .OUT suffix is added automatically.
call.name	is the name to be given to the subroutine in the catalogue.
record.id	is the name of the record in *file.name* . If omitted, the *call.name* is used.
options	are options controlling the way in which the program is catalogued.

Cataloguing operates in three modes:

Local Cataloguing

A V-type (verb) VOC entry is set up to point to the compiled program in the named file. If the program is recompiled it is not necessary to repeat the CATALOGUE command as the pointer will already be in place.

Local mode cataloguing is selected by use of the LOCAL option.

```
CATALOGUE BP MYSUB LOCAL
```

Private Cataloguing

The compiled program is copied to the cat subdirectory under the account. No VOC entry is created for this mode of cataloguing. If the program is recompiled, the CATALOGUE command must be repeated to copy the new version of the program.

Private mode cataloguing is the default and requires no option keyword.

```
CATALOGUE BP MYSUB
```

It is possible to share private catalogues between multiple accounts. See the *QM Reference Manual* for details.

Global Cataloguing

The compiled program is copied to the gcat subdirectory under the QMSYS account. There is no VOC entry for a program catalogued in this way. If the program is recompiled, the

CATALOGUE command must be repeated to copy the new version of the program. Globally catalogued subroutines can be called from all accounts.

Global mode cataloguing is selected by use of the GLOBAL option.

```
CATALOGUE BP MYSUB GLOBAL
```

For compatibility with some other multivalue products, global mode cataloguing can also be selected by adding an asterisk prefix to the call.name.

The *options* also include

NO.QUERY	Suppresses the normal confirmation prompt when cataloguing a program globally that is already in the catalogue.
NOXREF	Removes the cross-reference tables from the compiled program when placing it in the catalogue. This reduces the memory space required to run the program but prevents QM displaying details of the program line number and variable names if a program error occurs.

Automatic Cataloguing

QMBasic includes a compiler directive

```
$CATALOGUE {name} {LOCAL | GLOBAL}
```

to catalogue a program automatically after successful compilation.

Exercise

Create an include record to define a common block for all the file variables used by your program.

Write a file opening subroutine similar to the one in this module.

Modify your ORDERS program to use the include record and to call the subroutine instead of opening the files itself.

15.11 External Command Execution

Any QM command that can be executed from the command prompt can also be executed directly from within a program by use of the EXECUTE statement.

```
EXECUTE command
```

The EXECUTE statement is used most often for executing query processor commands to build lists of records meeting specific criteria.

In an earlier section, we met an example using the QMBasic SELECT statement to print the customer name from all of the records in our CUSTOMERS file.

```
OPEN 'CUSTOMERS' TO CUS.F ELSE ABORT 'Cannot open CUSTOMERS'
SELECT CUS.F
LOOP
   READNEXT CUS.ID ELSE EXIT
   READ CUS.REC FROM CUS.F, CUS.ID THEN
      DISPLAY CUS.ID : '  ' : CUS.REC<CS.NAME>
   END
REPEAT
```

Perhaps, we only want to see the customers for whom we have a telephone number. We could do this by adding a check for the TELNO field being non-null.

```
OPEN 'CUSTOMERS' TO CUS.F ELSE ABORT 'Cannot open CUSTOMERS'
SELECT CUS.F
LOOP
   READNEXT CUS.ID ELSE EXIT
   READ CUS.REC FROM CUS.F, CUS.ID THEN
      IF CUS.REC<CS.TELNO> # '' THEN
         DISPLAY CUS.ID : '  ' : CUS.REC<CS.NAME>
      END
   END
REPEAT
```

Alternatively, we could use the query processor to do the record selection for us.

```
OPEN 'CUSTOMERS' TO CUS.F ELSE ABORT 'Cannot open CUSTOMERS'
EXECUTE 'SELECT CUSTOMERS WITH TELNO
LOOP
   READNEXT CUS.ID ELSE EXIT
   READ CUS.REC FROM CUS.F, CUS.ID THEN
      DISPLAY CUS.ID : '  ' : CUS.REC<CS.NAME>
   END
REPEAT
```

This latter method becomes particularly attractive when the condition to be tested is complex.

Return Codes

Most, but not all, QM commands set a status value into a system variable named @SYSTEM.RETURN.CODE. This can be tested after an EXECUTE to determine whether the action was successful. The values set into this variable are described in the *QM Reference Manual* .

For user written programs, there is an equivalent status variable named @USER.RETURN.CODE which may be used in any way the application designer wishes. This variable is initially zero and is never changed by QM.

Both of these status variables can also be tested from QM paragraphs to control processing of stored command sequences.

Other Features of EXECUTE

The EXECUTE statement has several other features.

The CAPTURING option allows the screen output of a command to be captured in a dynamic array for subsequent processing. Each line of output is stored as a separate field.

```
EXECUTE 'LIST.READU' CAPTURING VAR
```

The PASSLIST option passes a Pick style select list variable into the command as the default select list. Select list variables are described in the *QM Reference Manual* .

```
EXECUTE 'RUN SALES.ANALYSIS' PASSLIST CLIENT.LIST
```

The RTNLIST option returns the default select list in a Pick style select list variable.

```
EXECUTE 'RUN FIND.OVERDUE' RTNLIST OVERDUE
```

The SETTING or RETURNING option copies the value of @SYSTEM.RETURN.CODE to a named variable after the command has been executed.

```
EXECUTE 'DATE.FORMAT INFORM' SETTING UKDATE
```

The TRAPPING ABORTS option causes an abort in the executed command to return to the program performing the EXECUTE instead of totally aborting the session.

```
EXECUTE 'RUN MYPROG' TRAPPING ABORTS
```

Exercise

Modify the subroutine that handles entry of order details in your ORDERS program to display a list of products if the user enters a question mark. Use the query processor to do this by executing the sentence

```
LIST STOCK DESCR HDR.SUP COUNT.SUP
```

The two keywords at the end of this query suppress display of the page heading and the record count.

This time, rebuilding of the screen requires more data to be displayed but your existing subroutines should do this correctly. Depending on how your subroutines interact, you may need to recalculate the current order line number after redisplaying the data.

Suggested Solution

```
GET.ORDER.DETAILS:
   FOR IDX = 1 TO 5
      LN = IDX + 8

      LOOP
         PART.NO = ''
         INPUT @(5,LN) PART.NO, 3_:
      UNTIL PART.NO = ''
         IF PART.NO = '?' THEN
            EXECUTE 'LIST STOCK DESCR HDR.SUP COUNT.SUP'
            INPUT JUNK,1_:
            GOSUB PAINT.SCREEN
            GOSUB PAINT.DATA
            IDX = DCOUNT(SAL.REC<SL.ITEM>,@VM) + 1
            LN = IDX + 8
         END ELSE
            READ STK.REC FROM STK.F, PART.NO THEN EXIT
            INPUTERR 'Part number is not known'
         END
      REPEAT
      ...etc...
```

15.12 QMBasic Use of Alternate Key Indices

An alternate key index provides a means of locating records in a data file by the value in a data field rather than by record id.

Consider our SALES file but scaled up to a realistic size with, perhaps, 100,000 orders where 10,000 customers have each placed 10 orders. If we want to find all the orders placed by a particular customer, we would need to read all 100,000 records, rejecting those that are for other customers. If we have an index of orders for each customer, we can read one index record and then go directly to the SALES records of interest. The performance improvement using such an index is likely to be very large.

QM allows us to create indices on one or more fields defined in the dictionary. Once created and built, these indices are maintained completely automatically and used by the query processor completely automatically. The application requires no changes.

The commands to create and build the index are CREATE.INDEX and BUILD.INDEX, both of which take a file name and one or more field names. These fields may correspond to any dictionary data definition item. The MAKE.INDEX command combines the create and build processes into a single operation. These commands were all discussed in an earlier section.

The SELECTINDEX Statement

Although an executed query processor SELECT from a program will automatically use any relevant index, sometimes a programmer may want to construct a select list directly from the index entries. This can be achieved using the SELECTINDEX statement.

```
SELECTINDEX index.name FROM file.var { TO list.no }
```

This form of the SELECTINDEX statement constructs a select list containing the values in the named index. For an index of customers in our SALES file, it would create a list of the customers that are referenced in the file.

```
SELECTINDEX index.name, value FROM file.var { TO list.no }
```

This form of the SELECTINDEX statement constructs a select list containing the ids of records in the data file which have the given value for the indexed field. For an index of customers in our SALES file, it would create a list of the orders for a specific customer.

In either case, because of the way in which the indices are stored, the list comes back in sorted order.

Example

The following program would print a list of orders and order dates for each customer. The customers and orders would both be in numerical order. Each customer would be separated by a blank line.

```
PROGRAM ORDER.LIST
$INCLUDE FILES.H

OPEN 'SALES' TO SAL.F ELSE ABORT 'Cannot open SALES'

SELECTINDEX 'CUST' FROM SAL.F TO 1
LOOP
   READNEXT CUS.ID FROM 1 ELSE EXIT
   SELECTINDEX 'CUST', CUS.ID FROM SAL.F
   DISPLAY 'Customer ' : CUS.ID
   LOOP
      READNEXT ORDER.NO ELSE EXIT
      READ SAL.REC FROM SAL.F, ORDER.NO THEN
         DISPLAY ORDER.NO : '  ' :
         DISPLAY OCONV(SAL.REC<SL.DATE>, 'D2DMY[,A3]')
      END
   REPEAT
   DISPLAY
REPEAT
END
```

Note the use of two select lists in this program. List 1 holds the customer numbers, the default list (list 0) holds the order numbers for the customer processed for each cycle of the outer loop.

Scanning Indices

QM alternate key indices are implemented using balanced tree (B-tree) files which use an internal sorted tree structure. This is slower than a hashed file for insertion and deletion of records but has significant advantages when constructing look up systems that require sorted data.

Programmers can use special operations within QMBasic to navigate the structure of the index to extract items in sequence.

The starting point for a scan of the index can be set using SELECTINDEX with some specific indexed value or with SETLEFT or SETRIGHT to position at either end of the index.

```
SETLEFT index.name FROM file.var
```

where

index.name is the name of the index

file.var is the file variable for the file to which the index relates

Once the start position has been set, the program can walk through the index using SELECTLEFT or SELECT RIGHT

```
SELECTRIGHT index.name FROM file.var {SETTING key} {TO
list.no}
```

where

index.name	is the name of the index
file.var	is the file variable for the file to which the index relates
key	is the variable to be set to the key value associated with the returned list
list.no	is the select list number to create. This defaults to 0 if omitted.

Example

```
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
UNTIL STATUS()
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.

15.13 User Written Functions

We have seen many examples of functions that are part of the QMBasic language. In this module we will find out how to add new functions of our own.

A typical QM application may include many functions. Some may be very simple data manipulation tasks, others may involve complex file searches. What all functions have in common is that they return a value as their result.

The FUNCTION Statement

A function commences with a FUNCTION statement in place of the (optional) PROGRAM statement. As with the SUBROUTINE statement, this includes a list of names to be used to refer to the data items passed into the function.

```
FUNCTION MYFUN(P, Q, R)
```

There is a limit of 254 arguments in a function.

The function may include any elements of the BASIC language. The value of the function's result is passed back using a modified form of the RETURN statement.

RETURN *value*

Example

The function below takes a country code and a VAT number, validates the VAT number based on rules in the country database and returns true/false to indicate if it is valid. Do not worry too much about exactly what the function is doing but look at it simply as an example of how functions work.

```
FUNCTION VALID.VAT(COUNTRY, IN.VAT.NO)
$INCLUDE COMMON.H

   VAT.NO = TRIM(IN.VAT.NO, ' ', 'A')  ;* Strip all spaces

   READ CTR.REC FROM CTR.F, COUNTRY ELSE NULL
   COUNTRY.PREFIX = CTR.REC<CT.VAT.PREFIX>
   IF COUNTRY.PREFIX = '' THEN RETURN (@TRUE) ;* Not EEC

   VAT.TEMPLATE = CTR.REC<CT.VAT.TEMPLATE>

   * Template may be multivalued. Form a composite including
the prefix
   TEMPLATE = CATS(REUSE(COUNTRY.PREFIX), VAT.TEMPLATE)

   RETURN VAT.NO MATCHES TEMPLATE
END
```

The DEFFUN Statement

Before a user written function can be used in a program, it must be defined using the DEFFUN statement so that the compiler knows how the function should appear.

```
DEFFUN function(arg1, arg2...) { CALLING cat.name }
```

where

function	is the name of the function.
arg1 , etc	are the arguments. The actual names given are ignored. The compiler simply counts them.
cat.name	is the name of the function in the system catalogue if it is different from *function* .

Typically, an application would use an include record to hold the definitions of all functions used within the application.

A function is actually a subroutine which returns its value through a hidden first argument. The VALID.VAT function is identical to the subroutine below.

```
SUBROUTINE VALID.VAT(OK, COUNTRY, IN.VAT.NO)
$INCLUDE COMMON.H

   OK = @TRUE

   VAT.NO = TRIM(IN.VAT.NO, ' ', 'A')  ;* Strip all spaces

   READ CTR.REC FROM CTR.F, COUNTRY THEN
      COUNTRY.PREFIX = CTR.REC<CT.VAT.PREFIX>
      IF COUNTRY.PREFIX # '' THEN
         VAT.TEMPLATE = CTR.REC<CT.VAT.TEMPLATE>

         * Template may be multivalued. Form a composite
including the prefix
         TEMPLATE = CATS(REUSE(COUNTRY.PREFIX), VAT.TEMPLATE)

         OK = VAT.NO MATCHES TEMPLATE
      END
   END

   RETURN
END
```

15.14 QMBasic and Virtual Attributes

A dictionary I-type item (virtual attribute) is effectively a little subroutine that is called by the query processor to calculate a value used in the report. User programs can also call I-types in this way by use of the ITYPE() function. This is particularly useful for complex calculations where it guarantees compatibility between the program and the query processor and also simplifies maintenance by having only a single definition of the expression.

Although intended for use with I-types, the ITYPE() function in QMBasic will work with all dictionary data definition items. Use of a simple field reference such as a D-type item will not be efficient when performed in this way but this flexibility does allow programs to evaluate dictionary items without needing to handle each record type differently. This can be particularly useful with programs that allow the user to enter the data item name interactively.

Because the ITYPE() function was originally designed for use within the query processor, its interface relies on some system variables used within the query processor.

```
ITYPE(dict.rec)
```

where

dict.rec is the dictionary record to be evaluated.

The expression in *dict.rec* is compiled automatically when it is first referenced in a query or explicitly by use of the COMPILE.DICT (CD) command. The compiled form of this program is stored in fields 16 onwards of the dictionary record. The *dict.rec* argument to the ITYPE() function is the whole of this dictionary record, not its name.

The ITYPE() function evaluates the expression in *dict.rec* . The record to be used as the source data for this expression must be in the @RECORD variable. If the I-type uses the record id, this must be in the @ID variable. These two variables are maintained automatically within the query processor. When evaluating an I-type from a program, the programmer must set up these items.

To execute an I-type from a program, the programmer must

1. Open the dictionary holding the I-type to be executed.
2. Read the I-type dictionary record.
3. Set up @RECORD and, perhaps, @ID.
4. Execute the ITYPE() function.

Steps 1 and 2 only need to be performed once in the program as the I-type dictionary record does not change. Steps 3 and 4 need to be performed for each data record to be processed.

Example

The dictionary of our SALES file includes an I-type named SALE.VALUE to calculate the total value of the order. We could use this in our ORDERS program to display this value.

Assuming that the SALES file dictionary is open as SAL.D, we can read the I-type dictionary record with a statement such as

```
READ ORDER.VALUE FROM SAL.D, 'SALE.VALUE'
   ELSE ABORT 'Cannot read SALE.VALUE I-type'
```

This should be done outside the main loop of the program so that we only read the dictionary entry once.

We can display the total value for an existing order by including the following statements after exit from the loop that displays each order detail line.

```
   @RECORD = SAL.REC
   TOTAL = ITYPE(ORDER.VALUE)
   DISPLAY @(61,14) : FMT(OCONV(TOTAL, CASH.CONVERSION), '9R')
:
```

The same three lines inserted into the bottom of the loop that enters detail lines for new orders will display a running total value as the order is created.

Exercise

Add statements similar to the above to your ORDERS program to display the order total value.

15.15 Printing

All of our programs so far have sent their output to the screen. Realistic applications would also need to send output to printers.

A QM application does not usually take direct control of a printer. Instead, application software sends its output to a numbered print unit with no knowledge of where this output might actually go. The association between a print unit and its destination is made from outside the application using the SETPTR command.

Each QM session has 256 print units, numbered from 0 to 255, available to it. These might be configured to be different printers or, perhaps, different paper types on the same printer. Unless an application specifically requests a different destination, printed output is sent to print unit 0, the default printer.

The PRINT Statement

The PRINT statement is identical in format to the DISPLAY statement that we have been using so far. The specified data item is output, followed by a carriage return and line feed to move to the start of the next line.

Our earlier example of screen output becomes

```
PRINT 'Outstanding Payment Details'
PRINT
PRINT 'Open invoices ' : NUM.OPEN
PRINT 'Overdue invoices ' : NUM.OVERDUE
```

As before, the output from this would be slightly untidy

```
Outstanding Payment Details

Open invoices 27
Overdue invoices 3
```

Again, we can use a series of items to be printed, separated by commas to advance to the next tab column across the page.

Our example becomes,

```
PRINT 'Outstanding Payment Details'
PRINT
PRINT 'Open invoices ', NUM.OPEN
PRINT 'Overdue invoices ', NUM.OVERDUE
```

The above statements would produce output such as

```
Outstanding Payment Details

Open invoices       27
Overdue invoices    3
```

As with DISPLAY, we can use a trailing colon to suppress the carriage return and line feed, allowing us to build up a line in stages.

Perhaps surprisingly, the above examples would display the output on the screen. The PRINT statement using the default print unit (0) directs its output to the screen unless we have previously executed a PRINTER ON statement to send the output to the default print unit. A corresponding PRINTER OFF statement would switch output back to the screen.

The reason for this behaviour is that it allows us to write a program that can send its output either to the screen or to the printer depending on whether we have executed the PRINTER ON. Typically, once a program has turned on printer output, it remains on for the entire program. The DISPLAY statement can be used for output that is to go to the screen regardless of whether printer output has been turned on or not.

Output directed to a printer is not sent to the device as it is generated by the program. Instead, it is put aside in the file system and only sent to the printer when the program executes a PRINTER CLOSE statement. This allows many users to generate simultaneous printer output. The printer is closed automatically on return to the command prompt and hence may not be necessary in all programs.

We cannot use the @() function to control the page position in PRINT statements that may be directed to a printer as this function returns the control sequence appropriate to the terminal device, not the printer. QM does not provide an equivalent function for printer control sequences though it would not be difficult to write one for a specific printer model. Instead, we must format each line using the FMT() function or the substring assignment operator.

For example, to print a report of product numbers, stock levels and product descriptions, we could use either of the following program fragments.

```
SELECT STK.F
LOOP
   READNEXT PART.NO ELSE EXIT
   READ STK.REC FROM STK.F, PART.NO THEN
      PRINT PART.NO : '  ' : FMT(STK.REC<ST.QTY>, '4R') :
      PRINT '  ' : FMT(STK.REC<ST.DESCR>, '30L')
   END
REPEAT
```

In this example, we have used FMT() for the variable length fields to pad them out to a fixed column width. The part number is of fixed length and therefore does not need a formatting. The FMT() function for the description field is also not really necessary as this is the final data item in the line and hence does not need trailing spaces added to bring it up to 30 characters.

An alternative way to format tabular data is to use substring assignment.

```
SELECT STK.F
LINE = SPACE(50)
LOOP
   READNEXT PART.NO ELSE EXIT
   READ STK.REC FROM STK.F, PART.NO THEN
      LINE[1,3] = PART.NO
      LINE[6,4] = FMT(STK.REC<ST.QTY>, '4R')
      LINE[12,30] = STK.REC<ST.DESCR>
      PRINT LINE
   END
REPEAT
```

We have started by setting a variable named LINE to be 50 spaces. This is because, in its default form, the substring assignment operation can only overwrite characters. It cannot extend the data item.

Within the loop, we then use substring assignment to drop each item into the required position within LINE. Note how we still need to use FMT() to right align the quantity.

These two methods of constructing tabular data both have their merits. Imagine a much more complex report with many columns. Using FMT(), we can adjust the width of a column by altering just one number but it is hard to see where on the page an item appears without adding up all the field widths. We also have to insert the inter-column gaps explicitly.

Using substring assignment, we can see where an item is positioned immediately but changing the width of a column would require all later substring positions to be updated. We do not have to insert the inter-column gaps as these are provided by our initialisation of LINE to be space filled.

Note also that we do not need to set LINE back to spaces each time around the loop as substring assignment truncates or space fills to fit exactly into the specified portion of the target variable.

The HEADING and FOOTING Statements

QM can automatically add headers and footers to each page of output. The programmer does not need to include any line counting to determine when these are to appear.

The HEADING statement defines text to be printed or displayed at the top of each page of output.

```
HEADING {NO.EJECT} text
```

where

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 text may be changed at any time by executing a further HEADING statement. This will start a new page immediately unless the NO.EJECT option is used.

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 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.
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.
L	Inserts a new line at this point in the text.
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.
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*.

Examples

```
HEADING "LOANS REPORT"
```

This statement sets up a simple fixed text heading.

```
HEADING "LOANS REPORT ON 'DL'"
```

This statement adds the current date to the heading and includes a blank line between the heading and the first data line.

```
HEADING "'C'LOANS REPORT'LDGPL'"
```

This statement outputs a two line heading on each page with a blank line before the first data line. The first heading line has LOANS REPORT centred on the page. The second line has the date at the left of the page and the page number at the right.

The FOOTING statement is identical in structure except that the NO.EJECT option does not apply. The text appears at the bottom of each page of output.

The PAGE Statement

Programs do not normally need to do any line counting to control pagination but there are times when it is required to force output to move to a new page. The PAGE statement does this.

```
PAGE { page.no }
```

where

page.no is the number to be assigned to the next page. If omitted, page numbering continues from the preceding page.

One common use of PAGE is when output consists of several lines that need to remain together on the same page. A program can determine the number of lines remaining on the page by use of the SYSTEM(4) function. This applies only to the default printer (printer 0). For other printers, use the GETPU() function that can return much information about a printer. Both of these functions are described in the *QM Reference Manual* .

Printing to Non-default Print Units.

Programs often use non-default print units. This might be to select a specific printer from a number of available devices, the select a paper type on a printer or to produce multiple reports simultaneously.

The PRINTER, PRINT, HEADING, FOOTING and PAGE statements all have an optional ON unit clause to identify the print unit concerned.

The PRINTER ON and PRINTER OFF statements do not have an ON clause as they control only the actions of output sent to print unit 0.

```
PRINT { ON unit }  { print.list }

PRINTER CLOSE ON unit

HEADING { ON unit }  heading.text

FOOTING { ON unit }  footing.text

PAGE { ON unit }  { expr }
```

The SETPTR Command

The SETPTR command (not a program statement) sets up the characteristics of a print unit, defining the physical dimensions of the page (lines and width) and the destination for the output.

The SETPTR command has a very large number of options. This section only shows the general structure of the command. For full details see the *QM Reference Manual* .

```
SETPTR unit, width, lines, top.margin, bottom.margin, mode,
{ options }
```

where

unit	is the print unit number.
width	is the page width in characters.
lines	is the number of lines per page, including the margins described below.
top.margin	is the number of lines to be left blank at the top of each page.
bottom.margin	is the number of lines to be left blank at the bottom of each page.
mode	1 to output to a printer. 3 to output to the hold file ($HOLD). 4 to output to stderr. 5 to output to the terminal auxiliary port. 6 to output to a printer and the hold file.
options	comma separated keywords specifying further details such as the printer name, paper type, number of copies, defer time, etc.

Output using mode 3 is saved as a record in the hold file. Mode 3 is useful for

- Reports that may not need to be printed.
- Diagnostic output.
- Testing printed output formats.

Exercise

Write a new external subroutine to print an invoice for a given order. It should be called from your ORDERS program whenever you write an order to the file.

The subroutine should take the order number as its only argument and produce its output on the default print unit. You will need to do a SETPTR command to direct the output to the hold file. For this exercise, to ensure that this will happen again automatically if you leave QM and then re-enter, use EXECUTE to execute the following SETPTR command at the top of your subroutine:

```
SETPTR 0,80,66,0,0,3,BRIEF
```

The BRIEF option omits the normal confirmation prompt from SETPTR. The output from your subroutine should be formatted to fit an 80 character wide page.

Your invoice should have a format similar to the one shown below. The invoice number is the same as the order number. The due date is calculated as 30 days from the order date.

```
Write Right Stationery Supplies

Invoice 12002                                                    Page 1


Order date: 04 JUN 07

Ross, Alan
47 Warren Road
Hansworth
Birmingham


 Part no.   Description                    Price     Qty       Total
    013      Pencil, blue                    0.28       2        0.56
    012      Pencil, red                     0.28       3        0.84
                                                              =========
                                                                  1.40


Payment due by: 04 JUL 07
```

Suggested Solution

```
SUBROUTINE INVOICE(ORDER.NO)

$INCLUDE FILES.H
$INCLUDE FILEVARS.H

EQU DATE.CONVERSION TO 'D2DMY[,A3]'

   READ SAL.REC FROM SAL.F, ORDER.NO ELSE RETURN

   * The SETPTR command is for testing only
   EXECUTE "SETPTR 0,80,66,0,0,3,BRIEF"

   PRINTER ON
   HEADING "Write Right Stationery Supplies'LL'Invoice
":ORDER.NO:"'G'Page 'SLL'"

   PRINT 'Order date: ' : OCONV(SAL.REC<SL.DATE>, DATE.CONVERSION)
   PRINT

   * Customer name and address

   READ CUS.REC FROM CUS.F, SAL.REC<SL.CUST> THEN
      PRINT CUS.REC<CS.NAME>
      ADDR = CUS.REC<CS.ADDR>
      LOOP
         PRINT REMOVE(ADDR, MORE)
      WHILE MORE
      REPEAT
   END

   PRINT
   PRINT

   * Part details

   PRINT '  Part no.  Description                          Price
Qty       Total'

   ORDER.TOTAL = 0
   IDX = 1
   LOOP
      PART.NO = SAL.REC<SL.ITEM,IDX>
   UNTIL PART.NO = ''
      READ STK.REC FROM STK.F, PART.NO THEN
         PRICE = SAL.REC<SL.PRICE,IDX>
         QTY = SAL.REC<SL.QTY,IDX>
         ITEM.TOTAL = PRICE * QTY
         ORDER.TOTAL += ITEM.TOTAL

         LINE = SPACE(72)
         LINE[5,3] = PART.NO
         LINE[14,30] = STK.REC<ST.DESCR>
         LINE[47,7] = FMT(OCONV(PRICE, 'MD2'), '7R')
         LINE[57,4] = FMT(QTY, '4R')
```

```
            LINE[64,9] = FMT(OCONV(ITEM.TOTAL, 'MD2'), '9R')
            PRINT LINE
         END

         IDX += 1
      REPEAT

      PRINT SPACE(63) : '========='
      PRINT SPACE(63) : FMT(OCONV(ORDER.TOTAL, 'MD2'), '9R')

      PRINT
      PRINT

      PRINT 'Payment due by: ' : OCONV(SAL.REC<SL.DATE> + 30,
   DATE.CONVERSION)

      PRINTER CLOSE

      RETURN
   END
```

15.16 Sequential Files

QM directory files are represented by an operating system directory and 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 worth processing the data line by line. QMBasic provides statements to perform sequential reading or writing of text data. These can only be used with directory files.

The OPENSEQ Statement

A record to be accessed sequentially must be opened using the OPENSEQ statement.

```
OPENSEQ file.name, id {mode} TO file.var {ON ERROR
statement(s)}
{LOCKED statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

where

file.name evaluates to the VOC name of the directory file holding the record to be opened.

id evaluates to the name of the record to be opened.

mode may be

APPEND	Opens the item to append new data, positioning at the end of any existing data
OVERWRITE	Any old data is deleted
READONLY	Opens the item for read only access

If omitted, the item is open for read/write access and any existing data is retained

file.var is the name of a variable to be used in later statements accessing this record.

statement(s) are statement(s) to be executed depending on the outcome of the OPENSEQ statement.

At least one of the THEN or ELSE clauses must be present.

An alternative syntax allows an item to be opened by pathname instead of file name and id:

```
OPENSEQ pathname {mode} TO file.var {ON ERROR statement(s)}
```

In either syntax, the named item is opened and associated with *file.var* for later operations.

If the item already exists, the THEN clause is executed. An update lock will be set on this item.

If the item does not already exist, the ELSE clause is executed and a subsequent WRITESEQ using the *file.var* will create it. The ELSE clause is also executed if the file does not exist or if it exists but is not a directory file. These three cases can be distinguished by the value returned by the STATUS() function immediately after the OPENSEQ. This may be

0 The record does not exist but can be created. An update lock will be set on the record.

1 The file exists but is not a directory file.

2 The file does not exist.

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.

Examples

```
OPENSEQ "BP", "BOOKS" TO SEQ.F ELSE
   STOP "Cannot open BOOKS program source"
END
```

This program fragment opens the BOOKS record in the BP file. No test of the STATUS() value is included in the ELSE clause as we are opening the record to read it.

```
OPENSEQ "C:\LOGS\AUDIT" APPEND TO AUD.F ELSE
   IF STATUS() THEN
      STOP "Cannot open audit trail file"
   END
END
```

This example is opening an item to record an audit trail. It is valid that it does not already exist and hence the ELSE clause must not cause the program to terminate for a status value of zero.

The READSEQ Statement

The READSEQ statement reads the next line from a directory file record previously opened for sequential access.

```
READSEQ var FROM file.var {ON ERROR statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

where

var	is the variable to receive the data read from the file.
file.var	is the file variable associated with the record by a previous OPENSEQ statement.
statement(s)	are statement(s) to be executed depending on the outcome of the READSEQ.

At least one of the THEN or ELSE clauses must be present.

The next line of text from the file is read into *var*. If successful, the THEN clause is executed and the STATUS() function would return zero.

If there is no further data to be read, the ELSE clause is executed.

Example

```
LOOP
   READSEQ REC FROM SEQ.F ELSE EXIT
   DISPLAY REC
REPEAT
```

This program fragment reads lines from the record previously opened for sequential access via the SEQ.F file variable and displays each line on the terminal. The loop terminates when the ELSE clause is executed when all data has been processed.

The WRITESEQ Statement

The WRITESEQ statement writes data to a directory file record previously opened for sequential access. WRITESEQF is identical except that it force writes the data to disk before continuing with the next statement.

```
WRITESEQ var TO file.var {ON ERROR statement(s)}
{THEN statement(s)}
{ELSE statement(s)}
```

where

var	is the variable containing the data to be written.
file.var	is the file variable associated with the record by a previous OPENSEQ statement.
statement(s)	are statement(s) to be executed depending on the outcome of the WRITESEQ.

The keyword TO may be replaced by ON. At least one of the THEN or 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 WRITESEQF statement is identical to WRITESEQ except that execution of the next statement does not occur until the data has been written to disk. With WRITESEQ, the data may still be in internal buffers. WRITESEQF is useful when generating audit trails.

Example

```
WRITESEQ REC TO SEQ.F ELSE STOP "Write error"
```

This statement writes the data in REC to the record open for sequential access via the SEQ.F file variable.

The WEOFSEQ Statement

The WEOFSEQ statement truncates a record open for sequential access at the current position.

```
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 and is equivalent to use of the OVERWRITE option of OPENSEQ. 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 "$HOLD", "STOCKS" TO STOCK.LIST THEN
   WEOFSEQ STOCK.LIST
ELSE
   IF STATUS() THEN ABORT "Cannot open stocks list"
END
```

This program fragment opens the record STOCKS in the $HOLD file for sequential access. If it already exists, the THEN clause of the OPENSEQ is taken and the existing data is removed using WEOFSEQ. If it does not already exist, the ELSE clause is executed. The test of the STATUS() function causes the program to abort if the record cannot be created by a subsequent WRITESEQ.

The CLOSESEQ Statement

The CLOSESEQ statement closes a record of a directory file previously opened for sequential access.

```
CLOSESEQ file.var
```

where

file.var is the file variable previously associated with the directory file record by use of the OPENSEQ statement.

The directory file record is closed, flushing the buffers and releasing the update lock.

Example

```
CLOSESEQ SEQ.F
```

This statement closes a directory file record previously opened using OPENSEQ using SEQ.F as the file variable.

Other Sequential File Processing Operations

In this section, we have looked only at the most frequently used sequential file processing operations. There are more, some of which are unique to QMBasic.

NOBUF	Suppress buffering
READBLK	Read a specific number of bytes instead of a text line
READCSV	Read a comma separated line of text, parsing it into separate variables
SEEK	Position the file pointer
WRITEBLK	Write a byte sequence instead of a text line
WRITECSV	Assemble a comma separated line of text from multiple variables

Full details of these operations can be found in the *QM Reference Manual* .

Exercise

Write a new program that exports the records from the STOCK file as a comma separated variable (CSV) format item that could be read into the Excel spreadsheet program. The output data should be written to a record named DATA in your BP file.

Each line should contain the part number and data from the corresponding STOCK record. The price field should be converted to external format.

Suggested Solution

You may have chosen to use the OPEN.FILES subroutine instead of opening the file in the program.

```
PROGRAM EXPORT
$INCLUDE FILES.H

   OPEN 'STOCK' TO STK.F ELSE STOP 'Cannot open STOCK'

   OPENSEQ 'BP', 'DATA' to SEQ.F ELSE
      IF STATUS() THEN STOP 'Cannot open sequential item'
   END

   SELECT STK.F
   LOOP
      READNEXT STK.ID ELSE EXIT
      READ STK.REC FROM STK.F, STK.ID THEN
         S = STK.ID : ','
         S := STK.REC<ST.DESCR> : ','
         S := STK.REC<ST.QTY> : ','
         S := OCONV(STK.REC<ST.PRICE>, 'MD2')
         WRITESEQ S TO SEQ.F ELSE STOP 'Write error'
      END
   REPEAT
END
```

QMBasic has an easier way to do this exercise by using the WRITECSV statement:

```
PROGRAM EXPORT
$INCLUDE FILES.H

   OPEN 'STOCK' TO STK.F ELSE STOP 'Cannot open STOCK'

   OPENSEQ 'BP', 'DATA' to SEQ.F ELSE
      IF STATUS() THEN STOP 'Cannot open sequential item'
   END

   SELECT STK.F
   LOOP
      READNEXT STK.ID ELSE EXIT
      READ STK.REC FROM STK.F, STK.ID THEN
         WRITECSV STK.ID, STK.REC<ST.DESCR>, STK.REC<ST.QTY>,
            OCONV(STK.REC<ST.PRICE>, 'MD2') TO SEQ.F ELSE STOP
'Write error'
      END
   REPEAT
END
```

If the exercise had not asked for the price to be converted to external format, the THEN clause of the READ could be reduced to:

```
WRITESEQ STK.ID:',':CHANGE(STK.REC, @FM, ',') TO SEQ.F ...
```

or, more simply,

```
WRITECSV STK.ID, STK.REC TO SEQ.F ...
```

relying on the fact that we are emitting all the data fields and they can be in the same order as in the dynamic array.

15.17 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.

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.

15.18 Triggers

Updating one record frequently requires other records to be modified too. For example, if we were to delete an unfulfilled order from our SALES file, this would require us to update the STOCK records to increment the quantity on hand.

Although this can be done by the application (and frequently is), QMBasic provides an alternative mechanism by which this sort of action can be handled totally automatically. This is known as a **trigger** because it is used to trigger related events.

Triggers can also be used to provide data validation, refusing to write a record if it fails the validation rules. Because the trigger is associated directly with the file rather than being embedded in the application, it will be executed regardless of what element of QM or the application is making the update.

A trigger function written as a subroutine declared as

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

where

name is the trigger subroutine name.

mode indicates the point at which the trigger function is being called:

1	FL$TRG.PRE.WRITE	before writing a record
2	FL$TRG.PRE.DELETE	before deleting a record
4	FL$TRG.POST.WRITE	after writing a record
8	FL$TRG.POST.DELETE	after deleting a record
16	FL$TRG.READ	after reading a record
32	FL$TRG.PRE.CLEAR	before clearing the file
64	FL$TRG.POST.CLEAR	after clearing the file

Other values may be used in the future. Trigger functions should be written to ignore unrecognised values.

id is the id of the record to be written or deleted.

data is the data. This is a null string for a delete or clearfile action.

on.error indicates whether the program performing the file operation has used the ON ERROR clause to catch aborts.

fvar is the file variable that can be used to access the file. Beware that reading, writing or deleting records via this file variable may cause a recursive call to the trigger function. This argument can be omitted for compatibility with earlier releases.

When writing trigger functions, the original data of the record to be written or deleted can be examined by reading it in the usual way. Trigger functions should not attempt to write the

record for which they are called. Neither should they release the update lock on this record as this could cause concurrent update of the record.

If 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.

The trigger subroutine must be catalogued like any other subroutine. It is then linked to the file using the SET.TRIGGER command.

```
SET.TRIGGER file.name subr.name {modes}
```

where

file.name is the name of the file to which the trigger is to apply.

subr.name is the catalogue name of the trigger subroutine.

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

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

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

After it has been set up, the trigger function is loaded into memory when the file is opened and is called for all operations defined by *modes* . Modifying and recataloguing the trigger function will have no effect on processes that have the file open until they close and reopen it.

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. Files that are to be accessed via QMNet require that associated trigger functions are globally catalogued.

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.

15.19 Sockets

QMBasic includes a set of functions that allow network connections to be established with other systems or other software on the same system. This section gives a brief overview of the socket interface of QMBasic.

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. We use domain name servers partly because names are usually easier to remember than numbers and also because the service may be moved to a different server (and hence different network address) without changing the name.

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. Full details of the functions described here can be found in the *QM Reference Manual* .

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. The first argument can be used to specify the network address on which to listen. Use of a null string as in the above example listens on all network addresses defined for the system on which the program is run. The second argument is the port number so this example is listening for all connections to port 4000 on the local system. SRVR.SKT becomes a socket variable that is 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, zero implying no timeout. 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 reads a message of up to 100 bytes and 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
```

The OPEN.SOCKET() function opens an outgoing connection to a server (194.118.13.14 in this example) that is listening on the specified port (4000).

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 simpler:

```
SRVR.SKT = CREATE.SERVER.SOCKET("", 4000, SKT$DGRM+ SKT$UDP)
IF STATUS() THEN STOP 'Cannot initialise server 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
```

15.20 Local Subroutines and Functions

We have seen internal subroutines that are part of a program module, sharing all the variables within that module, and we have seen external subroutines that have their own separate variables. QMBasic extends the language definition found in other multivalue products to add the concept of **local subroutines** that can have their own private variables but can also see the variables used elsewhere in the program module. There is also a **local function** that works in much the same way but returns a result.

Local subroutines appear at the end of the program, after the other internal subroutines but before the final END statement. The local subroutine starts with a line

```
LOCAL SUBROUTINE name
```

and ends with

```
    RETURN
END
```

A local subroutine must have a RETURN as the END carries an implied STOP just like the one at the end of the main program.

The subroutine is entered using GOSUB just like an internal subroutine. In fact, so far, there doesn't seem to be any difference.

The first key feature of a local subroutine is that it can have **private variables**. These are declared by one or more statements of the form

```
PRIVATE var1, var2, ...
```

immediately after the LOCAL SUBROUTINE declaration. The variables may be simple scalar items of dimensioned matrices:

```
PRIVATE A, B, C(2,5)
```

The variables named in the PRIVATE statement are private to the one local subroutine. They cannot be accessed from outside it. All variables defined in the main body of the module can be used within the local subroutine unless they have the same name as a private variable.

Private variables are not just names with limited scope; they are dynamically created on entry to the subroutine and discarded on return. The impact of this is that a local subroutine can call itself recursively and each instance has its own separate copy of the private variables.

The second difference between internal and local subroutines is that a local subroutine may have arguments.

```
LOCAL SUBROUTINE name(arg1, arg2)
```

Each argument may be a scalar variable or a matrix name prefix by MAT, exactly as in a SUBROUTINE declaration. When using a matrix argument, there must be a DIM statement to define the dimensionality of the matrix. Again, as with internal subroutines, an argument may be specified as being passed by value by enclosing the argument name in parenthesis.

The GOSUB statement is extended to support arguments when calling local subroutines.

A local function is created in much the same way, using a declaration of the form

```
LOCAL FUNCTION name(arg1, arg2)
   ...function body...
   RETURN value
END
```

Local functions must be defined using DEFFUN just like their external counterparts:

```
DEFFUN name(arg1, arg2) LOCAL
```

15.21 Object Oriented Programming

Object Oriented (often abbreviated to OO) programming provides a different way to develop applications. QM integrates the object oriented programming features into the QMBasic language rather than requiring developers to learn a whole new skill set. If you have experience of other object oriented languages, you will find the concepts to be familiar though there may be some differences in usage.

What is an Object?

An easy way to approach this question is to compare an object with a subroutine.

- A subroutine is a set of program operations that work on data typically provided via its argument variables.
- An object is a set of data items that have associated program operations.

In many ways, the object can be treated as a "black box" that has a number of operations that can be performed without any external understanding of how they are executed. This is no different from a subroutine but, because the data is embedded within the object, it is impossible for an application to perform operations that are not provided by the object. This **encapsulation** results in programs that are usually significantly easier to understand and maintain than their non-object oriented counterparts. Having said that, object oriented programming is not a replacement for everything we have seen so far. Rather it is an additional tool that has use in specific places in an application.

Another important point about objects is that there may be more than one **instance** of the object. Consider an object that represents a network connection to another system. This might support operations to open the connection, read data, write data and close the connection. If we implemented this as a subroutine (or, more likely, a separate subroutine for each operation), we would need to maintain some form of table of connection data, probably as a set of dimensioned matrices, and each subroutine call would need to include the relevant table index.

In the object oriented version of this, there would be a separate instance of the object for each connection but all of the data for management of the connection would be hidden within the object itself. It would also be impossible for a developer to perform any actions on the connection that were not provided by the object.

Class Modules

An object is defined by a **class module**, a QMBasic program that is introduced by the CLASS statement instead of the PROGRAM, SUBROUTINE or FUNCTION statements that we have met so far. This module defines the **persistent data** that lives within the object and a set of **public subroutines** and **public functions** that are the program operations provided by the object.

An object is a runtime instance of the class, **instantiated** by use of the OBJECT() function.

```
obj = OBJECT(name)
```

where

obj	is the variable to represent the instantiated object
name	is the catalogue name of the class module

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 of the object.

Persistent Data

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 can use these just like any other program but it 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**.

PRIVATE and PUBLIC variables are set to unassigned when the object is instantiated.

Public Subroutines and Functions

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).

In either case, 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. For more information on this, see the *QM Reference Manual* .

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 in the *QM Reference Manual* 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**.

The INHERIT statement

```
INHERIT name
```

can be used to define this relationship. At run time, a link is formed between the object that contains the INHERIT statement and a new instance of the named class. An object may inherit many other classes.

Use of the INHERITS clause of the CLASS statement

```
CLASS MYCLASS INHERITS inherited.class
```

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. The *inherited.class* element can be a comma separated list of class names.

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
```

Note that the above summary shows how a class module may contain internal and local subroutines just like any other program.

15.22 QMBasic Debugger

As you worked through the programming exercises, there were probably times when your program didn't behave as you wanted. Unless you looked ahead at this section, you may have found it difficult to see exactly what was happening.

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. You can even condition the DEBUG statement:

```
IF @LOGNAME = "GEORGE" THEN DEBUG
```

so that the program runs as normal for everyone except the user identified by the condition.

The debugger can also be entered from the break key action prompt if any program currently being executed has been compiled in debug mode.

During application development it is often worth compiling the entire application in debug mode. Execution of the program with the RUN command will not invoke the debugger unless a DEBUG statement is encountered. There is a small performance impact of running a debug mode program in this way but it is usually not significant.

When used with QMConsole on a Windows system, via the QMTerm terminal emulator or the bundled version of 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.

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.

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
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.

ABORT	Quit the program, generating an abort.
BRK *n*	Set a breakpoint on line *n*.
CLR	Clear all breakpoints.
CLR *n*	Clear breakpoint on line *n*.
DUMP *var path*	Dumps a variable to an operating system level file.
EP	Exit program, returning to parent program or external subroutine.
EXit	Exit subroutine, returning to parent program, internal or external subroutine.
Goto *n*	Continue execution at line *n*.
HELP	Display help page.
PDUMP	Generate a process dump file.
Quit	Quit the program, generating an abort.
Run	Free run.
Run *n*	Run to line *n*.

SET *var*=*value*	Change content of a program variable
STACK	Display the call stack. The current program is shown first.
Step *n*	Execute *n* lines.
Step .*n*	Execute *n* elements.
StepOver	Step over subroutine call
STOP	Quit the program, generating a stop.
UnWatch	Cancels an active watch action.
View	Display user screen (normal program output)
Watch *var*	Watches the named variable.
XEQ *command*	Execute *command* . Note that some commands may interfere with correct operation of the debugger.

The following commands apply only to full screen mode debugging:

SRC	Revert to default program source display
SRC *name*	Show source of program *name* .
SRC *n*	Display around line *n* of currently displayed program.
SRC +*n*	Move display forward *n* lines in program.
SRC -*n*	Move display backward *n* lines in program.

The following commands apply only to non-full screen mode debugging:

SRC	Display current source line
SRC *n*	Display source line *n*. Entering a blank debugger command line after this command will display the next source line.
SRC *n*,*m*	Display *m* lines starting at source line *n*. The value of *m* is limited to three lines less than the screen size. Entering a blank debugger command line after this command will display the next m 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 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.

SET *var = value*	to set a numeric value

`SET var = "string"`	to set a string value. Double quotes, single quotes or backslashes may be used to enclose the string.
`SET var(row,col) = value`	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.

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.

Debugging Phantom Processes

Phantom processes and those acting as the server side of a QMClient connection can be debugged using the PDEBUG command:

```
PDEBUG {command}
```

If no *command* is specified, the PDEBUG command waits for a phantom or QMClient process running in the same account 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.

If *command* is specified, PDEBUG starts a phantom process to execute *command* and then enters the debugger as above.

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. See the *QM Reference Manual* for more details.

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. Locks
   The report shows all task locks, file locks and record locks owned by the process.

4. 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.

Exercise

Add a DEBUG statement at the top of your ORDERS program. Run the program and explore some of the commands listed above.

16 What Next?

If you have worked step by step through this course material, you should now have a good understanding of OpenQM and be competent to develop and maintain applications. There is, however, much more for you to discover once you are completely comfortable with this material. Areas to explore in the *QM Reference Manual* include:

Data encryption - How to secure your sensitive data against theft.

QMClient - How to access QM from other environments such as Visual Basic, C, web servers, etc.

QMNet - Allows access to data on other QM servers with full concurrency control (locking).

In addition, there are many more commands and QMBasic statements that we have not discussed here.

If your application originated in a Pick style environment, it is likely to have PQ (Proc) type VOC entries. Proc is the predecessor of paragraphs and, whilst it has some useful capabilities, it is discouraged for new developments in favour of paragraphs or QMBasic programs. Procs are documented in the VOC section of the *QM Reference Manual* .

There is a whole section of the *QM Reference Manual* covering system administration topics.

Take time to skim through the manuals looking for topics that we have not covered. You learn best by practising so try things by writing simple test programs. You will rapidly become an expert in all that QM has to offer.