Getting Started in OpenQM - Part 2

Programming with AccuTerm

Version 1.0

March 2010

Brian Speirs

0 Preface

0.1 Purpose Of This Book

This book is a follow-on to *Getting Started in OpenQM – Part 1*. It aims to provide a resource allowing new users of *OpenQM* to understand how they go about creating a working application.

As with the first book, it was initially intended that this book would cover a greater range of programming topics. However, as the size of the book increased, it was decided to leave some topics for a future publication.

This book concentrates on illustrating the use of the *AccuTerm* GUI interface. Topics that have not been covered include: connecting *OpenQM* to *Visual Basic* or *C* using the `QMClient` interfaces; creating a web interface for *OpenQM*; and object-oriented programming.

Some other topics have been touched upon, but have not been illustrated in coding examples. The use of local variables and local subroutines in `QMBasic` is one obvious example of this. These features restrict variable scope thereby allowing much greater structure to be enforced in `QMBasic` programs than is available in other multi-value databases. However, such structure comes at the expense of code portability to other multi-value databases. Essentially, the choice has been made in this book to emphasise portability of code rather than producing the best code for *OpenQM*.

Even in those areas that this book seemingly covers in reasonable depth, there is much that is either glossed over or omitted completely. The *AccuTerm* `ATGUIxxx` subroutines are used extensively to create an application but the syntax of these subroutines is not really covered. Nor are the full range of `ATGUIxxx` subroutines covered. The reader is directed to the *AccuTerm* documentation for more information on these matters.

It should also be noted that this book does not cover areas such as database design – although it does make some digressions into some areas relevant to database design for multi-value databases. Readers are encouraged to learn about database design from other sources.

Although this book primarily covers *OpenQM*, much of what is shown here applies to other multi-value databases too. In particular, *OpenQM* uses syntax that is almost identical to that of *UniVerse*. So this book could be used as an alternate introduction to *UniVerse*. While the syntactical differences are greater to the other multi-value databases, the broad principles remain similar.

This book will also be a useful guide to the use of *AccuTerm* on any of the multi-value platforms. While the code that the programmer has to use within the event handlers will change slightly from one multi-value system to another (and AccuTerm's internal programs contain platform specific code), the *AccuTerm* interface that is presented to the programmer (and the user) is consistent between platforms.

0.2 About The Author

Brian Speirs is now on his second career. At the time of writing Part 1 of *Getting Started in OpenQM*, Brian was an economist. However, even in that role, he had used multi-value databases on a daily basis for over 20 years – mostly with Meat & Wool New Zealand and its predecessor organisations. The databases used during that time were mostly derived from the ADDS lineage – *Mentor*, *MOE*, *PC/OS*, and *mvBASE*, although there was some exposure to *UniVerse* also.

Brian's use of *OpenQM* stemmed largely from wanting a multi-value database for his own personal use. In practice, this also became his primary database for developing *UniVerse* applications – write it in *OpenQM* and then test it to make sure it works in *UniVerse*.

In 2008, Brian left New Zealand for the United Kingdom. He arrived just before recession struck the country. Fortunately, he found work reasonably quickly.

Brian now works as a Software Engineer with CACI based in Twickenham (on the outskirts of London). CACI software is widely used by organisations involved in inspection services or scheduled activities such as delivery of homecare. CACI won the "Best Use of Technology in Business" category at the UK National Business awards in 2008 for its use of mobile devices for delivery of a real-time roster for care providers. CACI software uses a *UniData* database running SB+ along with a CACI web front end.

As with Part 1, this book has been written in Brian's spare time.

Brian would welcome feedback on this book. He can be contacted at brian@rushflat.co.nz

0.3 Contents of This Book

This book is organised as follows:

Chapter 1 provides some general notes on the software required for use with this book, and about what the reader should understand before progressing through the book. The notation conventions used in this book are also outlined.

Chapter 2 looks at *OpenQM* files. This begins by providing a background on how hashed files work, and the problems that are traditionally associated with hashed files. It then looks at how *OpenQM*'s dynamic files solve some of these problems. Some examples are given showing how to use the configurable file parameters. It then covers how *OpenQM* stores data on disk and in memory, and how you can use this knowledge to handle the data more efficiently. Finally, the use of alternate key indices to increase performance is covered.

Chapter 3 is an introduction to `QMBasic`. It begins with a discussion of programming style, before moving on to creating the necessary files to begin programming. The obligatory “Hello world” program is covered, followed by a brief introduction to the main elements of a program. The chapter continues with a more elaborate example program, and an analysis of how it works. Finally, there is an overview of some of the important statements and functions that are found in multi-value programs.

Chapter 4 is quite brief. It simply serves to discuss what application should be created to provide an example of programming in `QMBasic`. There is some brief discussion of green screen programming before introducing the elements of *AccuTerm* that we will use in the remainder of this book.

Chapter 5 begins the application development process. There is a brief outline of what we want to achieve and some discussion of how we want the application to operate. The application form is then created using these design elements. A program is then created from the form design using the *AccuTerm* tools. The method of adding events to the form is illustrated along with updating the code to reflect the form events. Finally, code is added to the form so that it correctly responds to the form resize event.

Chapter 6 starts making the form functional. Code is added to the form that populates combo boxes with data, and then to respond to the user choices. Data is read from files on disk and displayed in the grid on the form. Users are allowed to edit the data, and then the updated data is saved back to disk. Various bugs are tracked down and eliminated.

In Chapter 7, the application is interfaced with an external application to download data from the internet. We then use AccuTerm scripting to drive *OOBasic* in *OpenOffice* to manipulate the downloaded data and save it in `CSV` format. The *AccuTerm* file transfer routines are then used to import the data into *OpenQM*.

Chapter 8 reviews the application created and suggests improvements that could be made. *Windows* help is added to the application.

Chapter 9 looks at other ways to create an *AccuTerm* GUI program, and introduces concepts such as using `COMMON` variables to pass information between subroutines, and using programs to write programs.

Chapter 10 looks at options for cataloguing your programs, along with various other options available within *OpenQM*. Finally, *OpenQM*’s menu utilities are covered along with an introduction to menu security.

Chapter 11 provides a short session of Questions and Answers.

Finally, the Appendix gives a complete listing of the main application and scripts that are developed in the course of the book.

0.4 Thanks

Thanks have to go to Ladybridge Systems. Without them, there would be no *OpenQM* either commercial or under the GPL licence.

Ladybridge also set up the *OpenQM* Google groups, and regularly answer questions from users in those fora. Some of that information has found its way into this book, so in one sense, all of the contributors to the *OpenQM* Google groups have indirectly contributed to this book.

Similar thanks go to AccuSoft Enterprises for the development of *AccuTerm*, and the support of AccuTerm through the forum on the AccuSoft website.

Other information has been gleaned from the *comp.databases.pick* newsgroup, although that newsgroup covers all multi-value databases, and not just OpenQM.

Martin Phillips of Ladybridge and Peter Schellenbach of AccuSoft Enterprises have both reviewed a draft of this book, and suggested a number of changes and corrections. All remaining errors are of course the responsibility of the author.

0.5 Trademarks and Copyright

0.6 Copyright of this Publication

1 Introduction

1.1 General

This book should be read in conjunction with Part 1 of *Getting Started in OpenQM*. While both books can be read "standalone", this book does assume knowledge of the material covered in Part 1.

Part 1 was mostly concerned with the basics of installing *OpenQM*, creating dictionary items, and then using the inbuilt query language to retrieve information from the database.

This book is primarily concerned with programming. In particular, it covers building applications using the *AccuTerm* GUI interface.

Despite being "about" programming, this book doesn't really aim to teach the reader how to write programs – although a lot of that ground is covered. Rather, this book is aimed more at people who already know how to write computer programs in another language, but who haven't used any of the variants of `BASIC` used by multi-value databases.

In particular, assumptions are made that readers are familiar with the terminology associated with GUI programming and event-driven programming. Ideally, readers should already understand what is meant by the terms: forms, controls, properties, events, and event handlers.

1.2 Software Configuration

This book does not assume any particular version of *OpenQM*. However, *OpenQM* has developed significantly over time, and a few constructs presented here do not work in early versions[1] (such as the GPL release).

The environment used for developing the applications in this book was a standalone notebook running *Windows Vista* with a commercial version of

1 These include trailing sub-string extraction (see 'Operators' in Section 3.2.2), the `CHILD` function (see Section 7.2.2) and `RETURN FROM PROGRAM` (see Section 9.1).

OpenQM. The version was kept current throughout the writing process and increased from 2.6-9 at the completion of Part 1 of *Getting Started* to 2.10-1 at present.

Some of the material presented here uses the GPL version of *OpenQM*. This version remains at version 2.6-6, and was installed on Ubuntu version 8.10 running inside a virtual machine (using Sun *Virtual Box*) on the same notebook as the commercial *Windows* version.

The applications in this book are wholly reliant on *AccuTerm*. While there are other terminal emulators for multi-value databases, *AccuTerm* is one of the most widely used emulators, and is bundled with the commercial version of *OpenQM*. If you wish to use a different emulator, then most of the material presented in Sections 5 through 9 will need to be rewritten to run on your software.

The current version of *AccuTerm* is 5.3c SP2 – although this is soon to be replaced by *AccuTerm* 7. *AccuTerm 7* is expected to be backward compatible with the current version, so no changes to the code presented here are expected. No testing of the GUI application or scripting has been done with earlier versions of *AccuTerm*.

A minimum installation required for the applications in this book would be a standalone *Windows* computer running the Personal version of *OpenQM* and the Personal version of *AccuTerm*. This has to be on a *Windows* computer because *AccuTerm* only runs on *Windows*, and the personal version of *AccuTerm* can only connect to `localhost`.

If you wish to use the GPL version of *OpenQM*, or any network connection to an *OpenQM* database, then you will need to use a commercial version of *AccuTerm*.

None of the files used in the applications are particularly large[2]. The largest file used by the applications is the `FX.DAILY` file which is currently around 162 KB. This means that the Personal version of *OpenQM* is quite adequate for the purposes of this book. (The Personal version cannot write to files larger than 0.5 MB).

If you don't have a computer of your own, you can still do everything in this book by using a USB installation of *OpenQM* and *AccuTerm*. This means that you can simply take your USB drive and use whatever (*Windows*) computer is available. The best way to load this combination of software is to download the USB Demonstration package from the *OpenQM* website[3]. This will load a basic installation of *OpenQM* and *AccuTerm* onto your USB drive.

Most of the programming material presented in this book is fairly standard across all multi-value implementations. This means that you could use another multi-value database (such as *UniVerse* or *UniData*) without any particular difficulties.

Installation of *OpenQM* on a *Windows* system was covered in Part 1 of *Getting Started in OpenQM*. This also covered installation of *AccuTerm*.

Notes on installing the GPL version of *OpenQM* can be found on the Rush Flat website (www.rushflat.co.nz).

1.3 Terminology

Multi-value databases come with their own jargon. In many cases, there are multiple terms for the same feature, and those terms are likely to be used interchangeably throughout this book.

Some of the common multi-value terms are listed below along with their equivalent meanings:

2 Except for the example file used for indexing in Section 2.7 which was 120 MB.
3 www.openqm.com

MV Term	*Translation*
Item	A record
Item-ID	Primary key
Attribute	A field
Value	Sub-field
Sub-value	Sub-sub-field
Frame	A storage bucket within a file
Buffer	Another name for a frame
Group	A base frame plus any linked overflow frames within a file
Dynamic array	A multi-valued variable
Delimited string	Technical description of a dynamic array
Correlative	Processing codes used by the query language to transform the data before display.

In this book, the main interchange of terms involves item and record; and attribute and field.

1.4 Documentation and Help

OpenQM and *AccuTerm* both come with thorough help systems and documentation. Use them.

To access the *OpenQM* help system, simply type **`HELP`** from the command prompt. This help system is well structured, letting you quickly find the syntax for any command. It is also useful to simply browse the help to make yourself aware of other commands, functions, and statements.

There are many references through this books to check the online help. This means the *Windows* help system referred to above. This is also available on the internet on the *OpenQM* website. See 'QM Help Pages' beneath the 'Help and Support' heading.

OpenQM comes with a complete set of PDF help files. You should find these in the `QMSYS\docs` folder.

The *AccuTerm* manuals can be downloaded from www.asent.com. In particular, you will want the Programmers Guide.

The *AccuTerm* Windows help file is also useful for checking syntax of the `ATGUIxxx` subroutines while programming. See the Section titled 'Features for MultiValue (Pick) Users'.

You can also get help from:

- The *OpenQM* Google group (`http://groups.google.co.uk/group/OpenQM`) is a good source of information. Search the group first to see if your problem has been covered already, and then ask a question there if things are not clear.
- If you are using the GPL version of *OpenQM*, then try the open source google group (`http://groups.google.co.uk/group/openqm-opensource`)
- There is a fork of the GPL version called Scarlet DME. It has a Google group at `http://groups.google.co.uk/group/scarletdme` and a knowledge base at `http://www.scarletdme.org/wiki/Main_Page`

- The *AccuTerm* forum (`http://www.asent.com/forum/default.asp`) provides a similar resource for *AccuTerm*.
- The `comp.databases.pick` newsgroup is a general multi-value forum on usenet. The best way to view this resource is via a news reader (Many email programs such as Outlook Express and Thunderbird provide news reader functionality). Alternatively, you can view the newsgroup as a Google group. See: `http://groups.google.co.uk/group/comp.databases.pick`
- Pickwiki is website covering a range of topics related to multi-value databases. See: `http://www.pickwiki.com`

1.5 Conventions in this document

OpenQM is a command-driven environment. User commands are shown in bold **`Bitstream Vera Sans Mono`** typeface, while the responses from *OpenQM* are shown in normal `Courier New`:

```
ANALYSE.FILE IRATES
Account            : D:\QM\QMINTRO
File name          : IRATES
Path name          : D:\QM\QMINTRO\IRATES

Type               : Dynamic, version 2
Group size         : 1 (1024 bytes)
Large record size  : 819
Minimum modulus    : 1
Current modulus    : 17
Load factors       : 80 (split), 50 (merge), 76 (current)
File size (bytes)  : 23552 (18432 + 5120)
```

Where the syntax of commands is shown, curly brackets[4] denote optional components, while a pipe symbol denotes that only one of the separated options should be used:

```
LOOP
  {statements}
{WHILE | UNTIL condition {DO}}
  {statements}
REPEAT
```

OpenQM keywords within the text will be in `Bitstream Vera Sans Mono` typeface. Variable and program names will be in the same font, but be *`italicised`*.

Code fragments are contained in a frame as shown below:

```
debug_msg:
*
CALL ATGUIMSGBOX(debug.txt, 'Debug', MBIICON, MBOK, '', debugok,
guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

RETURN
```

Note that some lines are long and will wrap at the page end. There is no identification of these lines in the text, so use your judgement (and the compiler error messages) to determine when these lines should be joined up. In the above example, the call to `ATGUIMSGBOX` has wrapped. You should be able to identify this because the parentheses do not close, and the line ends with a comma.

4 In many computer manuals, optional components are often shown in square brackets. However, square brackets are used within the OpenQM query language and programming language. Therefore, curly brackets are used both here and in the documentation to avoid confusion.

2 *OpenQM* Files

2.1 Background

Part 1 of this book gave a brief overview of the files in the *OpenQM* system, and also made passing reference to one of the advantages of the multi-value database file structures. This advantage was that the database was able to find any record in the database, no matter what the size of the database, with a single disk read – as long as the database was given the `ID` of the record. However, it didn't really explain the mechanics of the process used to achieve this.

This section will look at how multi-value databases structure their files. This will explain how the database can find any record in the database, and look at other characteristics of the multi-value file structure.

The explanation will initially use the fixed-size files found in other multi-value databases because these illustrate the principles clearly. *OpenQM* uses dynamic files, where the file size changes as data is added or deleted. The data storage principles are the same with dynamic files, but the database system does some extra work in the background when dynamic files are resized.

Note that this is a somewhat technical discussion. Readers can skip most of this section without missing any essential items from a practical usage perspective. However, most people should read Section 2.6 which deals with the `VOC` entries for files.

2.2 Hashed Files

2.2.1 Traditional hashed files

Multi-value databases use what are termed 'hashed files' for data storage. Hashed files use the following principles:

- The file is made up of a series of storage containers (frames or buckets)
- The database "hashes" the `ID` of the record to generate a large number. This number is then divided by the number of buckets in the file (the modulo) and the remainder calculated.
- The record is then stored in the bucket indicated by the remainder.

Consider the following example:

A file has a base size of 7 frames. We want to store an item with an `ID` of *`12345`* in the file. The `ID` hashes to a value of 545,673[5]. Dividing this hash value by 7 and taking the remainder gives us a value of 2. This means that the item will be stored in the third bucket (because we start counting from zero)[6].

The above procedure is always followed when the record is saved to the file. It will also be followed if the database is given the record `ID` and asked to get the record. However, it won't be followed if the file is being processed in sequential order – in that case, the system simply reads the records in the order in which they appear in the database.

Following this process a little further, we can see that each frame (bucket) in the file will contain multiple records. How do we know this? Well, records have to start in one of the buckets defined by the file size. If there are more records than buckets, then there have to be multiple records per bucket.

So what happens when the database system looks for a record in the bucket? Basically, it reads the whole bucket, and does a search through the contents of the bucket for the record being requested.

What happens when the bucket overflows? This will happen when the combined size of all the records assigned to the bucket exceeds the bucket size. In this case, the excess data is loaded into an overflow frame, and that frame gets linked back to the base frame.

An overflowed file imposes a performance penalty on the database system. It increases the number of disk reads to find a given record, and severely overflowed files face greater risks of corruption.

Traditionally, multi-value databases used fixed file sizes. Consequently, overflowed files could become a severe problem. Regular checks on file sizes were recommended to ensure that files did not become severely overflowed.

Oversized files could also be a problem. In that case, a file would have many empty frames and use excess disk space. More importantly, any process that required all data in the file would take longer than necessary because all the empty frames would need to be read as the system gathered the data.

There are other aspects of hashed files you should be aware of:

- Hashed files work best when the item-id's hash to an even distribution
- Hashed files work best where all records have similar sizes
- Hashed files work best when the average record size is only a small proportion of the bucket size (say, less than 20 per cent)
- Large records pose particular problems

Let's consider these:

5 Using the standard PICK hashing algorithm. *OpenQM* probably uses a different algorithm. To find the PICK hashing algorithm, google the comp.databases.pick newsgroup for "hashing algorithm".

6 This is a bit easier to understand when traditional MV databases defined the file as starting at a given frame number (file base). The group that the item belonged to was then determined by adding the remainder derived in this example to the file base. Therefore, if the remainder from the division was zero, the item would be stored in the frame specified by file base.

A hashed file will (almost) always have some unused space. Continuing the example used above, the file has a modulo of 7. If each bucket is 2KB in size, then the total primary file size (i.e. excluding any overflow) is 14KB. Further, the primary file will always be this size regardless of the number of items in the file.

Ideally, we seek to size the file such that the records occupy most of the primary file space. There are two basic requirements to do this:

- Each bucket within the file should be allocated a similar number of records
- The records should be of a size that allows efficient utilisation of the bucket

Say we had 140 records to go into our file of modulo 7. Ideally, we would want 20 records to go into each bucket. However, if the item-ids did not hash evenly, then we may some buckets with 50 items, while others have only 5 or 10.

How do we know if a set of item-ids will hash evenly? In most cases, we don't know. We create a file, populate it, and then analyse the file to find its distribution of records. If there is excessive variability in the distribution of records, then we would test other file sizes using a test utility. Once we find a suitable file size, then we would resize the file to match the item-ids.

However, we do know that item-ids in a sequential series will always hash evenly. How do we know this? When the item-id is incrementing by one for each new item, then the hashing algorithm will effectively increment the bucket pointer by one for each new item. Therefore a sequential series of item-ids will produce an equal number of records going to each bucket[7].

What about the utilisation of space within the bucket? Consider a file with a bucket size of 1KB (the default size for *OpenQM*). If our average record size is 150 bytes, then we can fit six records (900 bytes) into each bucket before that bucket goes into overflow. If we try to fit a seventh record, then the total size of all records in the group[8] (1,050 bytes) exceeds the bucket size. Therefore, an overflow bucket gets allocated to that group and we end up with 1,024 bytes used in primary file space, and 26 bytes in overflow.

What if the average item size was 350 bytes? Then we'll only get two items into the bucket before it goes into overflow. But two items is only 700 bytes, so the file will never be efficiently utilised. For this size record, we would be better using a bucket size of 2KB (5 records before overflow), or 4KB (11 records before overflow).

What if the average record size was 600 bytes? In the case of 1KB frames, then we'd only get one record per group before using overflow frames, and once again, the file space will never be efficiently used.

What if the record size was variable? Well, hopefully the pseudo-random allocation of records to buckets will largely cancel out the random variation in record size. In that case, each file group will have a similar size. The greater the number of records in each group, the more likely it becomes that each group will have a similar overall size[9] – provided that record sizes are evenly distributed, and the records hash evenly. However, if large items cluster together, then some groups will go into overflow while others remain relatively unutilised.

Finally, what if the record size was larger than the bucket size? In this case, the first item into a bucket will completely fill the primary file space and extend into overflow. If other

7 See page 12 for a slight qualification of this.

8 A group consists of the base bucket in primary file space plus any associated buckets in overflow. Therefore, a group has a minimum size of one bucket, but may consist of multiple buckets.

9 For example, Rocket Software recommends that *UniData* group sizes should be at least 10 times the average item size. Even after allowing for some unused space, this implies there should be at least 8 records per group, meaning that groups are more likely to be evenly sized than if there were only 2 items in the group.

records are subsequently allocated to the same bucket, then all of these records will be stored in overflow.

Consider also the outcome if the file does not hash evenly. In this case, some groups will be hugely overflowed, while other groups remain empty.

Therefore, there are two challenges when dealing with large record sizes. Firstly, you want to minimise the number of records being allocated to each bucket so that the number of records in overflow space is minimised. Secondly, you want to minimise the number of empty buckets. The difficulty is that minimising the number of empty buckets tends to come at the expense of putting more records into overflow.

Most multi-value databases have got around this large record problem by moving large items to their own special file space, and storing only a pointer to the large record in the primary file space. This has the advantage of ensuring the primary file space is used efficiently, but comes at the expense of requiring an additional disk read to read the record – the first disk read will retrieve the pointer from the primary file space, while the second will read the record itself.

2.2.2 Dynamic hashed files

OpenQM uses dynamic hashed files. The ‘dynamic’ part of the name means that the files resize themselves automatically as records are added or deleted.

Dynamic hashed files retain the key advantage of normal hashed files of single read access of a given record even without indexing, while removing (most of) the file administration procedures that accompany traditional hashed files.

The QM help system gives a brief overview of how dynamic hashed files work, and there is a more comprehensive Technical Note on their operation on the *OpenQM* web site[10].

A quick summary of how dynamic files work follows:

- *OpenQM* continually calculates a load factor for each file. This is the total file size as a percentage of the primary file space.
- The load factor will change as records are added, modified, or deleted.
- If the load factor exceeds the `SPLIT.LOAD` parameter (default = 80%), then *OpenQM* adds another bucket to the primary file space, and splits an existing bucket to provide records for the new bucket.
- If the load factor falls below the `MERGE.LOAD` parameter (default = 50%), then *OpenQM* will merge some of the buckets.
- If a record is larger than the `LARGE.RECORD` parameter (default = 80% of bucket size) is encountered, then this record will be stored in the indirect record space, with only a pointer to this record stored in the primary file space.

In many cases, you can simply leave *OpenQM* to manage file sizes for you. However, if you are seeking to maximise performance and/or you have unusual record sizes, then you may wish to change some of the default settings. Some of the possible settings have been alluded to above.

In addition to the `SPLIT.LOAD`, `MERGE.LOAD`, and `LARGE.RECORD` parameters, you may also want to change the `GROUP.SIZE` parameter. `GROUP.SIZE` sets the number of 1KB blocks that make up each bucket in the file. In some cases, you may also want to define a `MINIMUM.MODULUS` for the file.

There are a couple of questions which spring to mind regarding these parameters:

- How can we tell how the file is structured?

10 See: http://www.openqm.com/. Click on ‘Sales and Downloads’, and then ‘Technical Notes’

- How do we set or change these parameters?

Analysing a file

OpenQM provides a program to analyse a file to help you determine whether it is appropriately configured. This program is `ANALYSE.FILE`:

```
ANALYSE.FILE filename {STATISTICS}
```

For example:

```
ANALYSE.FILE IRATES
Account           : D:\QM\QMINTRO
File name         : IRATES
Path name         : D:\QM\QMINTRO\IRATES

Type              : Dynamic, version 2
Group size        : 1 (1024 bytes)
Large record size : 819
Minimum modulus   : 1
Current modulus   : 17
Load factors      : 80 (split), 50 (merge), 76 (current)
File size (bytes) : 23552 (18432 + 5120)
```

This starts with basic information about where the file is located in the host filesystem, and then provides information about the file configuration and status. In this case, the file is using a bucket size of 1 KB, a `LARGE.RECORD` size of 819 bytes (80% of the bucket size), a `SPLIT.LOAD` of 80 per cent, and a `MERGE.LOAD` of 50 per cent. The current load factor is recorded as 76 per cent.

Other information notes that there are 17 buckets in the file at the moment (Current modulus), that the primary file space is 18,432 bytes, and the overflow file space is 5,120 bytes, for a total file space of 23,352 bytes. If you view the file through *Windows* Explorer, you will find that these reported file sizes match the sizes of the %0 and %1 files respectively.

Adding the `STATISTICS` option provides additional information about the file:

```
ANALYSE.FILE IRATES STATISTICS
Account           : D:\QM\QMINTRO
File name         : IRATES
Path name         : D:\QM\QMINTRO\IRATES

Type              : Dynamic, version 2
Group size        : 1 (1024 bytes)
Large record size : 819
Minimum modulus   : 1
Current modulus   : 17 (0 empty, 0 overflowed, 0 badly)
Load factors      : 80 (split), 50 (merge), 76 (current)
File size (bytes) : 23552 (18432 + 5120), 13392 used
Total records     : 269 (269 normal, 0 large)

          Per group: Minimum    Maximum    Average
Group buffers     :        1          1       1.00
Total records     :        9         20      15.82
Used bytes        :      440       1016     787.76

  Bytes per record: Minimum    Maximum    Average
All records       :       44         56      49.78
Normal records    :       44         56      49.78
Large records     :
```

```
Histogram of record lengths

                                                                100.0%
    Bytes ------------------------------------------------------------
up to 16 |
up to 32 |
up to 64 | >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
up to 128|
up to 256|
up to 512|
up to 1K |
up to 2K |
up to 4K |
up to 8K |
over 8K  |
```

Extra information included tells us that the file contains 269 records, which take up 13,392 bytes. Of the 17 buckets in the file, none are empty and none are overflowed.

You may think it curious that this reports no buckets overflowed when there are 5,120 bytes used in the overflow file. The answer here is that in its current state, no buckets are overflowed. However, buckets were overflowed at various stages during the process of building the file. Therefore, overflow space was allocated to the file, and remains allocated even though it is currently not used.

We can use this information to verify the load factor. `ANALYSE.FILE` tells us that the data uses 13,392 bytes and the primary file space is 18,432 bytes. This suggests the load factor should be 72 per cent rather than the 76 per cent shown. The answer here is that *OpenQM* uses the first block of the primary file for administrative purposes. If we calculate the primary file space from the modulus, we find that there is really only 17,408 bytes (17 x 1,024 bytes) available in the primary file, and the actual load factor is 76.9 per cent.

The next section of information tells us a bit about the use of file space. The ‘Group buffers’ line tells us that each group in the file uses a minimum of one buffer (bucket), and a maximum of one buffer, for an average of one buffer per group. If there were some empty buckets, then the minimum value would be reported as zero. If some buckets were overflowed, then the maximum value would be two or more.

The ‘Total records’ line tells us that each group contains between 9 and 20 records, with an average of nearly 16 records per group. Likewise, the ‘Used bytes’ line tells us that each group contains between 440 and 1,016 bytes, with an average of 788 bytes.

In the ‘Bytes per record’ section, we see that records occupy between 44 and 56 bytes each, at an overall average of 50 bytes per record. All of these records are “normal” - i.e. there are no large records.

Finally, the histogram shows that all the record sizes fit into a a single size range.

This information is telling us that the file is fairly efficiently utilised. No groups are overflowed, although at least one primary bucket is nearly full. A load factor of 76 per cent is quite good. Similarly, records are of fairly even size.

Could we get better utilisation? Maybe. The smallest group is less than half full (440 bytes), while the largest group is just below the point of going into overflow (1,016 bytes). There is a similar range in the number of records per group. This suggests that the ID’s are not hashing evenly. You will recall from part one of this book that this file contained monthly interest rate indicators and used an `ID` of `YYYYMM`. Given that this `ID` is not random, and that the `MM` part of the `ID` will repeat over a short period (01 to 12), it is not surprising that the hashing is not even.

Overall, we don’t need to do anything with the configuration of this file. Utilisation is good, it is not overflowed, and it is so small that configuration options have minimal effect.

Let’s consider another file used in part one of this book:

```
ANALYSE.FILE FX.DAILY STATISTICS
Account            : D:\QM\QMINTRO
File name          : FX.DAILY
Path name          : D:\QM\QMINTRO\FX.DAILY

Type               : Dynamic, version 2
Group size         : 1 (1024 bytes)
Large record size  : 819
Minimum modulus    : 1
Current modulus    : 52 (0 empty, 12 overflowed, 0 badly)
Load factors       : 80 (split), 50 (merge), 80 (current)
File size (bytes)  : 73728 (54272 + 19456), 42768 used
Total records      : 1065 (1065 normal, 0 large)

         Per group: Minimum    Maximum    Average
Group buffers      :       1          2       1.23
Total records      :      14         36      20.48
Used bytes         :     180        820     822.46

  Bytes per record: Minimum    Maximum    Average
All records        :      16         56      40.16
Normal records     :      16         56      40.16
Large records      :

Histogram of record lengths

                                                                   68.
8%
    Bytes
------------------------------------------------------------
up to 16 | >>>>>>>>>>>>>>>>>>>>>>>>>>>>
up to 32 |
up to 64 |
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
up to 128|
up to 256|
up to 512|
up to 1K |
up to 2K |
up to 4K |
up to 8K |
over 8K  |
```

Amongst other things, this tells us that:

- there are 52 groups of which 12 are overflowed. This is also reflected in the Group analysis with a maximum of 2 buckets per group, and an average of 1.23.
- the current load factor is 80 per cent
- the number of records per group is quite variable (range 14 to 36)
- record sizes are variable (range 16 to 56 bytes). This is also reflected in the bi-modal distribution of record sizes in the histogram.

The bi-modal distribution of record sizes isn't surprising. This file contained daily foreign exchange transactions for New Zealand – with no transactions being recorded on weekends. This means that Saturday and Sunday records are essentially empty, while weekday records contain data for each of the recorded currencies.

The uneven number of records per group is surprising because the ID was a serial number (the internal date for that day). There may be two answers here:

Firstly, *OpenQM*'s internal management of dynamic files will provide its most even hashing when the actual file size is exactly a power of two (1, 2, 4, 8, 16, 32, etc). When the actual modulus is in between those points, some groups will be allocated more records than others.

In this case, the modulus is 52. Internally, *OpenQM* will initially calculate the group of a record using a modulus of 64 (being the next power of two). If that group does not exist (e.g. group 56), then *OpenQM* recalculates the group number using a modulus of 32 to derive the actual group that the record is stored in.

Once again, there is nothing in this file's structure that we would really need to change. However, we could change the file size to 64 (the next power of two) to see the effect on the number of records per group. We will try that shortly.

Secondly, hashing algorithms typically use the `ASCII` value of the characters to derive an overall hash value. When using a series of sequential numbers as item-id's, the values in the hash series will only be sequential for ID's of the same length. When the length of the item-id changes (e.g. from 999 to 1000), the hash value will jump by an order of magnitude (from 6327 to 54328) even though we see only a single value increment. This explanation clearly doesn't apply in this case as all the ID's are the same length.

Setting or changing parameters

File parameters may be set at the time of file creation using the `CREATE.FILE` verb, or changed at some later time using the `CONFIGURE.FILE` verb.

Additionally, the default `GROUP.SIZE` parameter may be changed for the whole *OpenQM* system using the Configuration Editor (Start | Programs | QM | QM Config Editor), or individually for specific users using the `CONFIG` command:

```
CONFIG GRPSIZE value
```

where value = 1, 2, 4, or 8 and represents the number of 1 KB blocks making a bucket in the dynamic file.

Note the different name given to `GROUP.SIZE` when set via the configuration options. To see the current setting for `GRPSIZE` (and all other configuration options), simply type:

```
CONFIG
```

To set the values of these configurable parameters at the time of file creation, use the options in the `CREATE.FILE` verb:

```
CREATE.FILE filename {GROUP.SIZE size} {LARGE.RECORD bytes}
{SPLIT.LOAD pct} {MERGE.LOAD pct}
```

where: size = 1, 2, 4, or 8 representing the number of 1 KB block in each bucket
bytes = size of record before the record is stored indirectly
pct = load percentage before split or merge operations

For example:

```
CREATE.FILE SALES GROUP.SIZE 2 LARGE.RECORD 800
```

Reconfiguring an existing file uses similar parameters:

```
CONFIGURE.FILE filename {GROUP.SIZE size} {LARGE.RECORD bytes}
{SPLIT.LOAD pct} {MERGE.LOAD pct}
```

Note that changing the group size of an existing file requires exclusive access to the file – i.e. other users will be denied access to the file while the file restructures itself – but changes to other parameters will take place in the background.

Both `CREATE.FILE` and `CONFIGURE.FILE` have other configurable parameters, most of which are not covered here. See the online help for more information on these parameters.

We can use a couple of these other options to resize the `FX.DAILY` file analysed earlier. We want to see if the file will hash evenly if the modulus is exactly 64. We can set the file to this size using the following command:

```
CONFIGURE.FILE FX.DAILY MINIMUM.MODULUS 64 IMMEDIATE
```

After running this command, the file analysis looks like:

```
ANALYSE.FILE FX.DAILY STATISTICS
Account            : D:\QM\QMINTRO
File name          : FX.DAILY
Path name          : D:\QM\QMINTRO\FX.DAILY

Type               : Dynamic, version 2
Group size         : 1 (1024 bytes)
Large record size  : 819
Minimum modulus    : 64
Current modulus    : 64 (0 empty, 0 overflowed, 0 badly)
Load factors       : 80 (split), 50 (merge), 65 (current)
File size (bytes)  : 86016 (66560 + 19456), 42768 used
Total records      : 1065 (1065 normal, 0 large)

         Per group: Minimum    Maximum    Average
Group buffers     :       1          1       1.00
Total records     :      13         20      16.64
Used bytes        :     472        896     668.25

  Bytes per record: Minimum    Maximum    Average
All records       :      16         56      40.16
Normal records    :      16         56      40.16
Large records     :
```

This change has moved all of the records in overflow back into primary file space, and reduced the load factor to 65 per cent. It has also evened up the distribution of records (range 13 to 20 records per group, down from 14 to 36), but it is still not completely even.

To return the file to its default configuration:

```
CONFIGURE.FILE FX.DAILY DEFAULT IMMEDIATE
```

Example file configuration

Let's consider an example where you might want to use a file configuration that is different from the *OpenQM* default:

Let's say we want to store the daily transactions of a medium size supermarket. Because this file will be growing constantly during the day, we want to create the file with an initial size large enough to hold all the expected transactions for the day – this way *OpenQM* won't have to devote resources to file resizing during the day.

The supermarket has 20 checkouts, and will have an average of 12 of them open throughout the day. It is open from 7.00 in the morning until 11.00 at night. Each checkout processes an average of 25 customers per hour. Each transaction has an average of 35 line items. What characteristics should we assign to the file?

First we need to calculate the average item size. From this, we will be able to set the group size, and calculate the total file size. Given those two pieces of information, we should be able to derive the minimum modulo to set for the file.

Each record will consisit of:

Field	**Size**
ID	4 Key
Date	5
Time	5
Checkout No	2
Operator ID	4
Payment Reference	16 or 1
Loyalty card number	16 or 0
Total Value (excl tax)	5
Tax	4
Sub-total	**61 or 30**

Item reference	16
Quantity (number)	1 or
Quantity (weight)	4
Price	4
Sub-total	**22 average**

The key for this file is simply a sequential transaction number. We can quickly calculate from the numbers given so far that we expect 4,800 transactions per day. On this basis, the ID will usually be a 4 digit number.

Our average item size will be:

Header	61
35 lines @ 22 bytes	770
Attribute marks	12
Value marks	140
Total	**983 bytes**

and our total file size will be:

12 checkouts x 16 hours x 25 customers x 983 bytes = 4,718,400 bytes

How should we configure this file?

These records are nearly 1 KB each on average. If we want 10 records per group (or more), then the group size needs to be in excess of 10 KB. The nearest we can come to this is to use the 8 KB group size.

We can now calculate the modulo for the file:

4,718,400 bytes / 8,096 bytes per group / 75% load factor = 777

Or if you want to be a bit more aggressive in your file sizing, you could use a load factor of 80 per cent to derive a modulo of 728.

Therefore, our `CREATE.FILE` command would look like:

```
CREATE.FILE DS,DS20081028 GROUP.SIZE 8 MIMIMUM MODULUS 780
```

Note this assumes use of a shared dictionary named `DS` (daily sales).

The biggest problem with this file configuration is the inherent variability in the size of supermarket orders. There are two particular aspects to this variability – firstly, the daily variation in the amount of business transacted along with seasonal variations; and secondly, the variability of transaction sizes within any given day. Let's consider the variability in transaction size first:

Individual purchases may range from a single item to a whole trolley load of 100 items. This effectively means that our record size will vary from 59 bytes (single item purchased using cash and without a loyalty card) up to 1,959 bytes for a 100 item purchase. Clearly, this range of record sizes will create potential problems for the filing system.

Some options for dealing with this variability include:

- ignoring it
- using the `LARGE.RECORD` setting to move the large items into indirect file space
- normalising the file to move the transaction lines to a separate file.

Ignoring the issue may not be ideal, but it is the simplest option, and may work well enough.

Using the large record setting does not fix the problem either, but does move the largest records (say those over 1,000 bytes) out of primary file space. This will effectively reduce the variability of the record sizes – but will have the side effect of reducing the total amount of primary file space required. Therefore, the minimum modulus could be specified a little smaller than was calculated above.

The final solution of suggesting file normalisation is akin to heresy when applied to multi-value databases, but is actually a good solution to the issue of sizing this data. On the other hand, it may complicate reporting and programming. Further, because of the necessity of providing a key for each item in the transactions file, this will actually use more disk space. However, it is well worth considering.

In this case, the file structures would consist of a header file with records of 30 to 61 bytes (plus 8 bytes for field marks), and a transaction lines file with records of 22 bytes plus a key of about 8 bytes (transaction number, separator, transaction line number, and field mark). Both files would consist of evenly sized records which should assist in achieving good file utilisation, and the records are all small enough to allow the default group size of 1 KB.

The file sizes would be:

> 12 checkouts x 16 hours x 25 customers x 60 bytes = 288,000 bytes (header)
>
> 12 checkouts x 16 hours x 25 customers x 35 lines x 30 bytes = 5,040,000 bytes (transactions)

and modulos would be:

> 288,000 bytes / 1,024 bytes per group / 80% load factor = 351 (header)
>
> 5,040,000 bytes / 1,024 bytes per group / 80% load factor = 6,152 (transactions)

Accordingly, the file creation commands would be:

```
CREATE.FILE DS,DS20081028 MIMIMUM.MODULUS 350
CREATE.FILE DT,DT20081028 MIMIMUM.MODULUS 6150
```

Once again, this assumes the use of shared dictionaries named `DS` (daily sales) and `DT` (daily transactions).

Note the following points:

- total file size has increased (5,328,000 bytes c.f. 4,718,400 bytes)
- the modulo of the transactions file is much larger than the previously calculated modulo (6,152 c.f. 780)
- the group size for each file is much smaller than calculated earlier (1 KB c.f. 8 KB).

Essentially, this file structure uses many small groups rather than fewer large groups.

The "right" structure of the data files will depend on local considerations. The above example is intended to give some guidance as to how the different file configuration options work, and when to use them.

Variability in the number of transactions per day could be handled by a set of factors which rate a given day relative to a "normal" day. Mondays and Saturdays may be rated at 120 per cent of normal, while the remaining days are about 90 per cent. In the two weeks before Christmas, days are given an additional factor starting at 100 per cent of normal 14 days prior to Christmas, increasing to 300 per cent of normal for the last full day of shopping before Christmas.

2.3 Directory Files

Hashed files are good for storing records which are relatively homogeneous in terms of size. Ideally, the records shouldn't be too large either. Traditional multi-value databases cannot store binary items (such as images) in hashed files either, but *OpenQM* does not have this limitation.

For data that doesn't meet these criteria, you should use directory files. These are simply operating system directories (folders) which are also defined in the `VOC` of *OpenQM* so that *OpenQM* can access them.

Within multi-value environments, the most common usage of directory files is for storing programs. This allows the use of external editors to write and modify programs used in the multi-value environment.

Similarly, because directory files are accessible from outside the multi-value environment, such files are frequently used to exchange data between the multi-value environment and external systems.

However, there are some limitations associated with directory files that should be noted. In large part, these limitations are the flip-side of the benefits of the directory files:

- file performance is lower than for hashed files (sorting and selecting data)
- directory files cannot be indexed
- you cannot apply triggers to directory files.

The restrictions on indexing and triggers are obvious enough – if processes outside *OpenQM* add, modify, or delete records in directory files, then *OpenQM*'s indexing and triggers cannot keep track of those changes. Therefore, data would become inconsistent (because trigger processes have not been applied), and indexes would become out of date.

The lack of indexing partly explains the lower performance of directory files. However, at a more fundamental level, it is more difficult for *OpenQM* to sort and process directory files.

In summary then, use directory files for:

- programs
- some large items
- transferring data between *OpenQM* and other environments.

2.4 Data Storage

2.4.1 Variable length fields

So far, there has been almost no mention of one of the key advantages of multi-value databases – variable length fields (and by association, records). This provides several benefits:

- display restrictions do not limit the amount of data that can be entered into a field
- empty fields do not take up any space in the database
- changing field lengths do not require any restructuring of the database.

These benefits are best shown by contrasting the situation with a a typical SQL database:

SQL databases use a schema that rigidly defines the data entered into the database[11]. A data field that is defined as containing 20 characters can only take a maximum of 20 characters, regardless of the actual length of the data attribute. In contrast, you can enter

11 Some modern SQL databases have removed some of these restrictions. For example, Oracle allows fields to be redefined without a complete restructure of the database, while MS Access uses variable width fields for text fields, making the field width restriction advisory rather than definitive.

any amount of data into a multi-value database field – even if the dictionary item describing that field specifies a lesser field width. This highlights the fact that multi-value database dictionary items are descriptive (rather than prescriptive). Of course, the way this "extra" data is displayed will depend on the formatting instructions in the dictionary item, but at least the data is available in the database.

Likewise, when an SQL database defines a field with a width of 20 characters, the database will always allocate 20 characters to that field – even if that field is empty. In contrast, a multi-value database will not allocate any space to an empty field (other than a field marker). This has potential to make a multi-value database smaller than an equivalent SQL database.

So what happens when the database design needs to change? For example, the size of a data field needs to increase. In an SQL database, you usually need to restructure the whole database to allocate more space to the field. In a multi-value database, nothing changes at the database level – the database can already take any amount of data (within reason) in any field.

To sum things up, variable length fields add a great deal of flexibility to database design. You can focus on entering, processing, and displaying the data rather than worrying about the size of the data field that is to be entered.

The flip side to this flexibility is that it is easy to start doing things in multi-value environments without thinking through the design issues properly.

2.4.2 String representation

The other unusual aspect of multi-value storage is that the data is all stored as an ASCII string – that is, using alphanumeric characters. The number 12348 is literally stored in the database as '12348'.

Once again, compare this with the situation in other databases. Typically, those databases will store numbers in fields defined as having a numeric format. Importantly, those numbers are converted to hexadecimal before being stored. Therefore, 12348 is actually stored as '303C' - i.e. 4 bytes instead of 5 – but the 4 bytes may well be stored within an 8 byte field.

What is the benefit of this? There is no great advantage in storage space requirements – particularly with the large capacity hard disks that are common on all modern hardware. Multi-value environments generally use less space for any given storage requirement, but this is more due to the variable length fields than to differences in numeric storage methods. It does make the contents of the database human-readable – but looking directly at the contents of the database files (through a file utility) is not recommended.

The major advantage (or disadvantage depending on your viewpoint) is that you gain flexibility by treating everything as a string:

- you don't need to define the fields as strictly integer, real, or string
- you can store string data in fields that are mostly numeric, and vice-versa.

The first point simply means that you don't need to formally define your fields – but you should do so anyway. The whole point of a field is to group data of similar types.

The second point is also not recommended. Storing data of different types within a field makes analysis of that data difficult.

Overall then, multi-value environments store data as strings largely because there is no formal database schema that says that a particular field must be of a specific data type. While it does give some advantages in terms of reducing the overhead of database design, this isn't a step that you should skip. You should design your database with the intention of storing particular data types in specific fields.

However, there are some advantages when it comes to programming in the database environment. These advantages are that variables can be re-used with changing the data type, and that string and numeric values can be combined together without type conversion (see page 34 for an example of this). Once again, these advantages do not come without disadvantages – notably the danger of trying to perform numeric operations on alphabetic characters.

2.4.3 Internal Data Storage

While it may seem unnecessary to know how *OpenQM* stores data internally, this knowledge can actually help you (as a developer) to handle the data better.

The way that data is stored follows on directly from the previous two points – data is stored as a string in variable length fields. *OpenQM* uses a number of special delimiter characters to identify the boundaries of fields, values, sub-values, and records. By knowing this, you can use *OpenQM*'s special string handling functions to extract data fields, and groups of fields.

The full list of delimiter characters, their normal ASCII values, and their internal tokens are shown below:

Mark	*ASCII*	*Token*	*Representation*
Item	255	`@IM`	
Attribute or Field	254	`@AM or @FM`	^
Value	253	`@VM`	]
Sub-value	252	`@SM or @SVM`	\
Text	251	`@TM`	

While the `ASCII` values and tokens are functionally identical, it is recommended that you use the tokens in your programs. This is partly because the code is more readable, and partly because it allows the delimiter character to be changed in the future (to take advantage of Unicode character sets for example).

The representation column in the table shows how delimiter characters are often displayed in text. Note, this is different from the way that your terminal may display the character. A terminal will use the character associated with the character set that you are using.

Prior to the introduction of standard tokens, many multi-value `BASIC` programs used constants for the same purpose:

```
EQUATE AM TO CHAR(254)
EQUATE VM TO CHAR(253)
EQUATE SM TO CHAR(252)
```

The program would then use the constants defined above in the rest of the program.

Now let's consider what a stored record looks like, and how we might use that information to write more efficient programs. In general, a stored record will look like:

```
item-id^a1^a2-v1]a2-v2-sv1\a2-v2-sv2^a3^a4-v1]a4-v2]a4-v3 etc
```

where:

a = attribute
v = value
sv = sub-value

When the item is read into a variable, the item-id is stripped off, while the variable contains the rest of the string (from a1 onwards) as is.

Let's say the item has 16 attributes, and is stored in the variable `rec`. We want to create a new item consisting of attributes 11 through 16. We could do this as follows:

```
newrec = ''
FOR ii = 1 TO 6
  newrec<ii> = rec<ii + 10>
NEXT ii
```

or we could do it as:

```
newrec = FIELD(rec, @AM, 11, 6)
```

Both methods get attributes 11 through 16 and assign them to a new record, but the second method, which treats the entire record as a delimited string, is much more efficient.

Older code may use a technique like:

```
xpos = INDEX(rec, @AM, 10)
newrec = rec[xpos + 1, 9999]
```

Similarly, if we want our new record to have attributes 1 to 6 and 12 to 16, then we could:

```
newrec = ''
FOR ii = 1 TO 11
  IF ii GT 6 THEN
    jj = ii + 5
  END ELSE
    jj = ii
  END
  newrec<ii> = rec<jj>
NEXT ii
```

or:

```
dummy = FIELD(rec, @AM, 7, 5)
newrec = rec[1, COL1() - 1]:rec[COL2(), LEN(rec)]
```

or:

```
newrec = FIELD(rec, @AM, 1 6):@AM:FIELD(rec, @AM, 12, 5)
```

Once again, the second and third methods treat the existing record as a string, and use string extraction techniques to build the new record from the constituent parts of the existing record.

2.5 Binary Data

As noted above, *OpenQM* can store binary data in hashed files. This is slightly unusual in multi-value databases because binary data is likely to contain some characters that match the system delimiters (ASCII 251 to 255).

Just because this is possible, it doesn't mean that it is always a good idea. Use your judgement – large binary items such as images or movie files probably shouldn't be stored in hashed files. A more appropriate arrangement for such items would be to rename the binary item to a sequential numeric name and store the item in a directory file. The hashed file should then contain the real file name and the path to the renamed item in the directory file (or the *OpenQM* filename and itemname).

Even when you do store binary data in directory files, you need to be aware of how *OpenQM* handles such items:

OpenQM assumes that an item stored in a directory files is a text item, and that the appropriate representation of that item within the multi-value world is to replace the line-ends with field marks. Clearly, that is the wrong thing to do if the item is actually a binary item.

To prevent this happening, `MARK.MAPPING` should be turned `OFF` prior to reading or writing the binary item, and then turned back `ON` after the read/write.

For more information on this, see the topics 'Directory Files' and `MARK.MAPPING` in the online help.

2.6 File VOC Entries

2.6.1 Basic concepts

A key feature of multi-value databases is that most things are defined in the `VOC` file. This is really how `QMQuery` works – all the elements of a `QMQuery` must be present either in the `VOC` or in the dictionary of the file being queried.

This is true for files too. It is the file entry in the `VOC` that lets each part of *OpenQM* find the file to operate on. Let's prove this:

In Part 1 of Getting Started in *OpenQM*, we created a file named `FX.DAILY`. The `VOC` entry for this file looks like:

```
CT VOC FX.DAILY
VOC FX.DAILY
1: F
2: FX.DAILY
3: FX.DAILY.DIC
```

The 'F' on line 1 tells *OpenQM* that this is a file. It is only the first character that is relevant to *OpenQM*, so you can add a description of the file on this line:

```
VOC FX.DAILY
1: File of daily FX transactions
2: FX.DAILY
3: FX.DAILY.DIC
```

This description will be displayed when you list the files in the account using the `LISTF` or `LISTFL` commands.

Line 2 of the `VOC` entry tells *OpenQM* the name of the data file, while line 3 has the name of the dictionary file.

`QMQuery` uses the information in this item to find both the dictionary and the data file so that it can report on the file:

```
SORT FX.DAILY WITH YYYYMM = "200701" DATE AUD USD GBP ID.SUP
Date........   Aus Dollar   US Dollar.   GB Pound..
 01 JAN 2007
 02 JAN 2007
 03 JAN 2007       229.29       878.90        32.90
 04 JAN 2007       142.45       831.10        32.07
 05 JAN 2007       115.40     1,542.80         8.10
 06 JAN 2007
 07 JAN 2007
 08 JAN 2007       143.24       985.50         5.20
 09 JAN 2007       121.84     1,194.80        28.80
 10 JAN 2007       121.10       808.30         7.30
 11 JAN 2007       117.44       552.92        45.50
 12 JAN 2007       195.70       923.50        13.20
```

Now, let's copy the VOC entry to a temporary item, and delete the main VOC entry:

```
COPY FROM VOC FX.DAILY,TEMPFXD
1 record(s) copied.
```

```
CT VOC TEMPFXD
VOC TEMPFXD
1: F
2: FX.DAILY
3: FX.DAILY.DIC
```

```
DELETE VOC FX.DAILY
1 record(s) deleted
```

Note that although we've deleted the `VOC` entry for the file, we haven't deleted the file itself. You can check this by looking at the `QMINTRO` account using *Windows Explorer*. You can also display the contents of the file by using the `TEMPFXD` file pointer that we created:

```
SORT TEMPFXD WITH YYYYMM = "200701" DATE AUD USD GBP ID.SUP
Date........   Aus Dollar   US Dollar.   GB Pound..
 01 JAN 2007
 02 JAN 2007
 03 JAN 2007       229.29       878.90        32.90
 04 JAN 2007       142.45       831.10        32.07
 05 JAN 2007       115.40     1,542.80         8.10
 06 JAN 2007
 07 JAN 2007
 08 JAN 2007       143.24       985.50         5.20
 09 JAN 2007       121.84     1,194.80        28.80
 10 JAN 2007       121.10       808.30         7.30
 11 JAN 2007       117.44       552.92        45.50
 12 JAN 2007       195.70       923.50        13.20
```

But, if we try to reference the file using the `FX.DAILY` name, `QMQuery` will not be able to find the file:

```
SORT FX.DAILY WITH YYYYMM = "200701" DATE AUD USD GBP ID.SUP
File not found
```

OK – Now let's put things back the way they should be:

```
COPY FROM VOC TEMPFXD,FX.DAILY
1 record(s) copied.
```

```
CT VOC FX.DAILY
VOC FX.DAILY
1: F
2: FX.DAILY
3: FX.DAILY.DIC
```

```
DELETE VOC TEMPFXD
1 record(s) deleted
```

So, what have learnt from this:

- *OpenQM* relies on `VOC` entries to find files
- `VOC` entries exist independently of the data and dictionary files, and can have a different name
- Files can exist without `VOC` entries and vice-versa
- A file can have more than one `VOC` entry at any given point in time
- We can manually create `VOC` entries for existing files.

2.6.2 Different types of VOC entries

Dynamic and directory files

Both dynamic files and directory files have the same type of `VOC` entry. We can see this by comparing the `VOC` entry for `FX.DAILY` (a dynamic file) with that of the `QUERIES` file (a directory file), also created in Part 1 of *Getting Started With OpenQM*:

```
CT VOC QUERIES
VOC QUERIES
1: F
2: QUERIES
3: QUERIES.DIC
```

Multi-files

Multi-files have a slightly different `VOC` entry:

```
CREATE.FILE DICT TEMP
Created DICT part as TEMP.DIC
Added default '@ID' record to dictionary

CT VOC TEMP
VOC TEMP
1: F
2:
3: TEMP.DIC

CREATE.FILE DATA TEMP,TEMP1
Created DATA part as TEMP\TEMP1

CT VOC TEMP
VOC TEMP
1: F
2: TEMP\TEMP1
3: TEMP.DIC
4: TEMP1

CREATE.FILE DATA TEMP,TEMP2
Created DATA part as TEMP\TEMP2

CT VOC TEMP
VOC TEMP
1: F
2: TEMP\TEMP1²TEMP\TEMP2
3: TEMP.DIC
4: TEMP1²TEMP2
```

Note the following points from this sequence of commands:

- Following the initial dictionary creation, the `VOC` entry does not contain any reference to a data file
- When a multi-file data portion is added to the file, the format of the second attribute changes, and a new attribute (attribute 4) is added:
  - Attribute 2 now contains the path name (relative to the account base) to the data file(s)
  - Attribute 4 contains the sub-file name of the multi-file
- As more data portions are added to the file, attributes 2 and 4 become multi-valued to store the path and file names.

The documentation notes that F-type `VOC` entries also have a fifth attribute. This controls the way the `ACCOUNT.SAVE` and `FILE.SAVE` commands treats this file. See the documentation for more details on this attribute.

Q-Pointers

Q-pointers are an alternate method of referencing a file. They are typically used when:

- the file resides in another account
- the file is part of a multi-file, and you wish to use a simpler filename
- the has a long filename and you wish to use a simpler filename.

Q-pointers have the following structure:

```
1: Q
2: Account name
3: File name
```

To reference the `BP` file in the `QMSYS` account, the Q-pointer will look like:

```
1: Q
2: QMSYS
3: BP
```

We might name this item: `BP.QMSYS`

We would create this item using one of the editors available in *OpenQM* (`ED`, `SED`, or `MODIFY`; `WED` if you are using *AccuTerm*; or any other editor that you have enabled). Don't include the line numbers when entering the item! We could also use the `SET.FILE` command to create the Q-pointer.

A Q-pointer to one of the data portions of the multi-file created on the previous page would look like:

```
1: Q
2: QMINTRO
3: TEMP,TEMP2
```

If we called this item `T2`, then we could use the filename `T2` in our `QMQuery` sentences, or within `BASIC` programs, whenever we wanted to access the data in `TEMP,TEMP2`.

Note that Q-pointers can be chained. That is, one Q-pointer can point to another Q-pointer, which in turn could point to another Q-pointer, or could point to an F-type entry. While Ladybridge do not recommend this structure, it can be quite useful.

Consider the situation where you wish to move an important file from one account to a new account. You already have a number of accounts pointing to the file using Q-pointers. Rather than change all of the existing Q-pointers (and risk missing one of them), simply move the file to its new location, and place a Q-pointer in the `VOC` of the old account pointing to the new location. When one of the other accounts wish to access the file, they will look up their own Q-pointer which will direct them to the old location. They will find a Q-pointer there which will direct them to the new location.

Manual creation of F-type VOC entries

It is clear that Q-type `VOC` entries are created independently of the file itself. On the other hand, F-type `VOC` entries are maintained automatically by *OpenQM* when we create and delete files.

What is less apparent is that we can create F-type entries manually to point to existing files on the system. Such files may be within the current account, or have a totally different path.

For example, we created a Q-pointer to the `BP` file in the `QMSYS` account earlier. We could have defined that as an F-type record as follows:

```
1: F
2: C:\QMSYS\BP
```

or:

```
1: F
2: @QMSYS\BP
```

This is a slightly unusual example because we haven't created a third line in the `VOC` entry for the dictionary path. The reason for this is quite simple – the `BP` file in `QMSYS` does not have a dictionary, and if we specify a path for it, `QMQuery` will report that it cannot open the dictionary. However, we would normally specify the path to the dictionary in attribute 3 of the `VOC` entry.

The second example uses the predefined token `@QMSYS`. It is preferable to use this token than specify the pathname for `QMSYS` because some administrators will place *OpenQM* in a non-standard location. By using a token for the location of the `QMSYS` account, we can transfer our applications between systems with confidence that they will still work in the new system.

OpenQM automatically defines two more tokens to be used in this manner - `@TMP` and `@HOME`. See the documentation for further information on these tokens.

We could also create an F-type entry for a directory on the system so that *OpenQM* can access the directory as if it were a file created by *OpenQM*. For example:

```
1: F
2: C:\TEMP
```

Once again, we haven't included a reference to a dictionary file because the 'Temp' directory doesn't have a dictionary.

Given that we can manually create F-type `VOC` entries for existing files, and such entries serve a similar purpose as Q-pointers, when should we create F-type `VOC` entries for existing files, and when should we create Q-type `VOC` entries?

As with many issues in multi-value databases, the answer is one of principle, rather than what is enforced by the database. The database lets you use either type of `VOC` entry, but what is the principle behind each type of entry?

An F-type entry is created when *OpenQM* creates a data file. The file is created within the account, and the F-type `VOC` entry represents an ownership of that file by the account.

A Q-type entry can point to anywhere – the current *OpenQM* account, another *OpenQM* account, or directory file elsewhere within the operating system. It does not confer ownership of the file – rather it simply says that this account uses that file.

Using these basic principles:

- ➢ Let *OpenQM* manage the F-type entries for files within an account
- ➢ Use a manually created F-type entry to create a reference to a file that is not part of the *OpenQM* file system
- ➢ Only have one F-type entry per file used by *OpenQM*
- ➢ Use Q-type entries to refer to *OpenQM* files in other accounts, or to provide files with simpler names (in any account).

2.7 Alternate Key Indices

An alternate key index is a means of increasing the speed of accessing a set of records from a file. It does this by indexing fields from the file, or expressions based on the file data, in a special lookup file. When you select on the main file using one of the indexed fields, *OpenQM* can access the indexed data and return the list of matching records without having to search the entire main file.

Indexing becomes more important as:

- ➢ the main file gets larger
- ➢ the number of records to be selected becomes a small proportion of the total number of records in the file
- ➢ the load on the system becomes higher
- ➢ the need for rapid response times increases.

Let's see what performance gains we can achieve by indexing a file:

We'll use a file named `PRODUCT-DATA`. This file currently contains 2.14 million records. `ANALYSE.FILE` produces the following statistics:

```
Type              : Dynamic, version 2
Group size        : 2 (2048 bytes)
Large record size : 1638
Minimum modulus   : 1
Current modulus   : 59572 (0 empty, 8871 overflowed, 81 badly)
Load factors      : 80 (split), 50 (merge), 80 (current)
File size (bytes) : 153077760 (122005504 + 31072256), 98821548 used
Total records     : 2140774 (2140774 normal, 0 large)

         Per group: Minimum     Maximum     Average
Group buffers     :       1           3        1.15
Total records     :      10          99       35.94
Used bytes        :      36        2048     1658.86

  Bytes per record: Minimum     Maximum     Average
All records       :      24         116       46.16
Normal records    :      24         116       46.16
Large records     :
```

We can see that the modulo is nearly 60,000 groups, and that the file has some groups in overflow. Primary file space is around 120 MB, and there is 30 MB of overflow space.

The file has a key of:

yyyywwttnnnlll

where:	yyyy	= season
	ww	= week
	ttnnn	= plant identifier
	lll	= product line identifier

Typical selection processes involve one or more of these key components. Dictionary items to extract these components are:

Dictname	Type	Expression	Conv	Name	Format	S/M
SEASON	I	@ID[1,4]		Season	6R	S
WEEK	I	@ID[5,2]		Week	4R	S
ME.NO	I	@ID[7,5]		ME No	6L	S
LINE3	I	@ID[12,3]		Line	3R	S

We'll create a simple paragraph to time a selection from this file:

```
CT VOC INDEX.TEST
VOC INDEX.TEST
1: PA
2: TIME
3: SSELECT PRODUCT-DATA WITH ME.NO EQ "ME078" AND WITH SEASON EQ
"2007"
4: CLEARSELECT
5: TIME
```

Running this (twice) on a freshly rebooted system gave the following results:

```
INDEX.TEST
21:41:34  4 FEB 2010
7697 record(s) selected to list 0
Cleared numbered select list 0
21:42:46  4 FEB 2010
```

```
INDEX.TEST
21:42:55  4 FEB 2010
7697 record(s) selected to list 0
Cleared numbered select list 0
21:43:46  4 FEB 2010
```

On the first run, the selection took 1m 12s. This time decreased to 51s on the second run. This decrease in time is due to a portion of the file remaining in memory after the first run.

Now, we want to create the indices. As we are using the *ME.NO* and *SEASON* dictionary items, we will create indices on these fields:

```
CREATE.INDEX PRODUCT-DATA SEASON ME.NO
Added index for SEASON
Added index for ME.NO
```

```
BUILD.INDEX PRODUCT-DATA ALL
Building index 'SEASON'...
2140774 records processed
Populating index...
Building index 'ME.NO'...
2140774 records processed
Populating index...
```

We could have used the `MAKE.INDEX` command rather than the combination of `CREATE.INDEX` and `BUILD.INDEX`.

We can use the `LIST.INDEX` command to view information about the indices:

```
LIST.INDEX PRODUCT-DATA ALL STATISTICS
Alternate key indices for file PRODUCT-DATA
Number of indices = 2

Index name...... En Tp Nulls SM Fmt NC Field/Expression
SEASON            Y  I  Yes   S  R  N  @ID[1,4]
Index entries      Key values     Min Recs        Avg Recs        Max Recs
      2140774              12       147286     178397.8333          219143

Index name...... En Tp Nulls SM Fmt NC Field/Expression
ME.NO             Y  I  Yes   S  L  N  @ID[7,5]
Index entries      Key values     Min Recs        Avg Recs        Max Recs
      2140774             108            1      19821.9815          105665
```

After restarting the system, the test is run again:

```
INDEX.TEST
22:05:24  4 FEB 2010
7697 record(s) selected to list 0
Cleared numbered select list 0
22:06:10  4 FEB 2010
```

```
INDEX.TEST
22:06:26  4 FEB 2010
7697 record(s) selected to list 0
Cleared numbered select list 0
22:06:30  4 FEB 2010
```

This time, the first selection took 46s, while the second selection took just 4s. Subsequent restarts confirmed this pattern.

In practice, selection times will be between these two values – depending on how frequently the `PRODUCT-DATA` file is used. The more frequently it is used, the greater the probability that the index will be in memory and the shorter the selection times.

`QMQuery` will automatically use any indices that are available. You can also utilise the indices from `QMBasic` (see the online help for more information on this).

2.8 Summary

This chapter has covered a wide range of topics. You should understand by now:

- *OpenQM* uses two types of files – dynamic hashed files and directory files
- the basic principles of hashed files
- what makes the dynamic files used by *OpenQM* different from static hashed files used by traditional multi-value databases
- how to analyse the dynamic hashed files
- how configure the dynamic hashed files for improved performance
- the way that *OpenQM* stores data on disk
- the different types of `VOC` entry relating to files, how to create them manually, and when to use each type
- how to create alternate key indices on a file.

3 Introduction to QMBasic

3.1 General Considerations

3.1.1 What is QMBasic

`QMBasic` is the programming language that comes bundled with *OpenQM*. As its name implies, it is a dialect of the `BASIC` language. This dialect is closely related to the `BASIC` dialects that come with other multi-value databases.

`QMBasic` has its own set of enhancements that are not in other multi-value databases. These include:

- local variables
- object-oriented programming
- socket connections
- inbuilt encryption

plus enhancements to many of the standard inbuilt functions.

`QMBasic` is a server-side language. Some people have described `QMBasic` as a 'server side scripting language'.

OpenQM does not support any other server side language. However, *OpenQM* does offer two interfaces to external languages – one for *Visual Basic* and one for *C*. These interfaces allow a variety of languages to communicate with *OpenQM*. Further, applications developed using these interfaces may split the processing between the server and the client.

3.1.2 What is covered here?

This is not intended to be a programming tutorial – it is assumed that the reader is generally familiar with programming principles. Further, it is assumed that the reader is familiar with event-driven programming principles.

Nevertheless, a reasonable amount of basic programming information is covered, In particular, differences from other `BASIC` dialects will be covered.

3.1.3 Coding styles

Ladybridge Systems recommend[12] the following coding style for *OpenQM* applications:

- Write source code in lowercase for improved readability. We have partially adopted a convention that equate token names should be in uppercase.
- Use the standard command parser (`!PARSER`) rather than writing your own.
- Use token names from include records rather than literal values where appropriate.
- Use `CRT` or `DISPLAY` rather than `PRINT` unless you want the output to go to a printer.
- *QM* is reasonably case independent. Try to preserve this by looking up `VOC` or dictionary items as typed and, if not found, by trying again in uppercase.
- Whatever the programming purists may say, limited careful use of `GOTO` is fine.
- Use meaningful label names. Numeric label names are not allowed in the master source.
- The standard message handler should be used for all output text. Use message numbers in the range 10000 - 19999. We will change these to the final message number on integration where appropriate. Pick style messages from the `ERRMSG` file (and hence the Pick syntax variant of `STOP` and `ABORT`) may not be used.
- Programs must conform to the locking rules such that all writes and deletes, including adding a new record, are covered by a suitable lock. Programs must operate correctly with the `MUSTLOCK` configuration parameter set to 1.
- Programs must not rely on settings in the `$BASIC.OPTIONS` record for correct compilation.
- Programs must be adequately commented that their operation can be clearly understood for maintenance purposes.

Their purpose in publishing this guide is so that any submissions of improvements can be incorporated into the *OpenQM* codebase with a minimum of refactoring. Most of these points can be readily adopted.

These guidelines really apply if you are coding for *OpenQM* only. However, if there is any possibility that your application will be ported to another database, then you need to consider compatibility. You may say that you have no intention of moving to another database, but that isn't the only reason to consider compatibility – your application may be so good that users of *UniVerse*, *UniData*, *D3*, *Reality*, *jBASE*, or *mvBASE* may want to use it on their systems.

What is important for compatibility? Here are some suggestions:

- Use upper case keywords – some multi-value databases (e.g. *mvBASE*) cannot handle lower case keywords

12 http://*OpenQM*.com/cgi/lbscgi.exe?X=dcuvn8x953&t0=h&t1=devinfo

- Be consistent with your casing of variable names. For simplicity, it is best to use either all uppercase or all lowercase. The reason for this is that variable names in other databases are usually case sensitive – therefore, *`TEMP`*, *`Temp`*, and *`temp`* are all different variable names in most other databases, but are treated as being the same variable in *OpenQM*.
- Think carefully before using features of `QMBasic` that are not standard across the multi-value flavours. For example, no other multi-value database supports local variables or object oriented programming.
- Use keywords that are consistent between the various multi-value databases. Even where different databases have the same keyword, the keyword may not always have the same syntax, or have the same functionality. For example, in *OpenQM*, `SWAP` is a synonym for `CHANGE` (as it is in *mvBASE*). In *UniData*, `SWAP` has the same functionality, but has a different syntax. In *UniVerse*, `SWAP` has an entirely different function. In this case, do not use the keyword `SWAP` if you want to guarantee compatibility with other databases.

In this guide, the following conventions will be used:

- Keywords will be in upper case
- Variables will be in lower case
- Constants will be in upper case

You may also want to read the coding standards article on the PickWiki website: http://www.pickwiki.com/cgi-bin/wiki.pl?CodingStandards

3.1.4 Where is the GUI?

OpenQM does not directly support GUI programming. While this may surprise some, this is consistent with its role as a database server system. Few (if any) database servers directly support GUI programming – most rely on external toolsets to provide a GUI front-end to the database using the inbuilt API's.

There are various ways to add a GUI to *OpenQM* applications. These include:

- Use QMClient to connect an external language to *OpenQM* using either the *VB* or *C* interface
- Use *AccuTerm*'s GUI interface
- Use *AccuTerm*'s multi-value server interface to connect an external language
- Use one of the 4GL's available for *OpenQM* (and other multi-value databases). These include:
  - *mv.NET*
  - *designBAIS*

This book covers the use of the *AccuTerm* GUI interface.

3.2 General Programming Issues

3.2.1 Creating, Compiling, and Running Programs

Programs are stored in an *OpenQM* file. In the multi-value world, this file is traditionally named `BP`, but may take on any valid filename. Programs may be stored in multiple files, thereby assisting application development by keeping programs logically grouped together.

Let’s create a file for programs:

```
CREATE.FILE BP DIRECTORY
Created DICT part as BP.DIC
Created DATA part as BP
Added default '@ID' record to dictionary
```

By convention, programs are often stored in a file named `BP` (meaning Basic Programs), but there is no restriction on filenames for programs. You could use `PROGRAMS`, `PROGS`, `UTILS`, or any other valid filename.

Note that we’ve created this file as a directory. This type of file is suitable for text items, and allows the items in the file to edited directly from the operating system level – i.e. you can use your favourite text editor to edit the programs. In the meantime, we’ll start by assuming that you are using *AccuTerm*, and that you therefore have access to the `WED` editor.

Now, let’s create the standard “Hello world” program:

```
WED BP HELLO
```

This will start the `WED` editor. If the *`HELLO`* program exists, then `WED` will display the existing program; otherwise, it will ask if you wish to create the new item. Answer ‘Yes’ to this prompt, and then type in the following program:

```
PROGRAM HELLO
CRT "Hello World!"
STOP
END
```

The program starts with a declaration that this is a program. This declaration is optional for programs but is required for external subroutines (`SUBROUTINE`), functions (`FUNCTION`), and classes (`CLASS`). If the name declared here differs from the name by which the program is stored in the file, then the compiler will give a warning message. While it is permitted for these names to be different, it is good practice to make them the same, and certainly makes it easier to debug applications.

The second line of the program does all the work. This prints the string “Hello World!” to the terminal.

The `STOP` command is not strictly necessary, but it is good practice to use them at the expected end of the executable code.

The `END` statement is also not strictly necessary, but the compiler will inform you if it is not present. The `END` statement marks the end of the program.

Close the `WED` editor, and save the program when prompted. In `WED`, you can also save the program by clicking on the save icon, or by selecting ‘Save’ or ‘Save as’ from the ‘File’ menu.

Now compile the program:

```
BASIC BP HELLO
Compiling BP HELLO
0 error(s)
Compiled 1 program(s) with no errors
```

and run it:

```
RUN BP HELLO
Hello World!
```

Finally, (assuming this is a program we want to keep), we will want to `CATALOGUE` the program. This lets us run the program simply by using the program name, rather than by using the `RUN` command.

```
CATALOGUE BP HELLO
HELLO added to private catalogue
```

```
HELLO
Hello World!
```

OpenQM provides a number of options for cataloguing. The default method is private cataloguing as shown. Cataloguing options are covered in more detail in Section 10.1.

By default, you will need to recatalogue the program every time it is compiled. One way to get around this it to use a compiler directive that directs the compiler to automatically catalogue the program. The compiler directive can be placed into each program individually or placed in a `$BASIC.OPTIONS` item so that it applies to all programs within a file, or all programs within an account. See Section 10.1 for more information.

In this book, the compiler directive will be placed in each program. This makes our intentions explicit to anyone who subsequently edits the program.

Note that the commands invoking the editors have the same basic structure as virtually any other *OpenQM* command:

command filename itemname

Therefore, if you are using an editor other than `WED`, simply substitute the name of the editor you are using:

```
SED BP HELLO
```

Likewise, if you are using a different filename:

```
WED PROGS HELLO
```

Note that `WED` allows you to compile the program from within the editor itself. This is done using the “play” button at the far right of the toolbar, or by selecting ‘Compile’ from the ‘Tools’ menu, or by pressing the F11 key. Note that the program must be saved before compiling.

3.2.2 Statements, variables, tokens, constants, and operators

Statements

A program is made up of a series of statements which collectively tell the computer what to do. Generally, programs should be written with one statement per line – although multiple statements may be written on a single line if they are separated by a semi-colon.

```
statement 1 ; statement 2 ; statement 3
```

Statements beginning with an asterisk (*), an exclamation mark (!), or the keyword `REMARK` (or `REM`) are considered to be comments. Comments are often appended to the end of an active statement by using a semi-colon to denote a new statement, followed by the comment marker.

As noted above, programs are normally written with one statement per line. However, there are always exceptions:

- Some statements cover several lines
- Comments are often placed as a second statement on a line
- Sometimes, it is more convenient to place related statements together on a single line.

An example of all three of these possibilities is shown below:

```
BEGIN CASE
  CASE papersize = 'A4';        papersizeindex = 9
  CASE papersize = 'LETTER';    papersizeindex = 1
  CASE papersize = 'EXECUTIVE'; papersizeindex = 7
  CASE papersize = 'LEGAL';     papersizeindex = 5
  CASE papersize = 'B5';        papersizeindex = 13
  CASE papersize = 'ENV10';     papersizeindex = 15
  CASE papersize = 'DL';        papersizeindex = 19
  CASE 1;                       papersizeindex = 9;   * A4 default
END CASE
```

This code fragment sets the variable *papersizeindex* based on the literal value of *papersize*.

- This whole block of statements makes up a `CASE` statement.
- There is a comment at the end of the line that starts with `CASE 1`.
- Each of the individual `CASE` lines has a second statement that assigns the papersizeindex variable.

Variables

`QMBasic` variables have relatively few naming restrictions.

- They must start with a letter
- They must not end with an underscore
- They consist of letters, digits, dollar signs, percentage signs, periods (full stops) and underscores.

The documentation provides a short list of reserved names which cannot be used as variable names. This means that it is technically possible to use the other keywords as variables. However, this is not a good idea as it will lead to confusion over whether you are using a variable name or a keyword.

There is no restriction on the length of variable names – although for practical purposes you should avoid very long names.

`QMBasic` variables can hold data of any type, and can change their type during the course of program execution. Therefore, we could write:

```
temp = '05'     ; * Define temp as string variable
CRT temp
temp = temp + 0 ; * Redefine as numeric
CRT temp
temp = 'X':temp ; * Redefine as string
CRT temp
```

and the output from this program would be:

```
05
5
X5
```

Variables should be declared before they are used. However, unlike some languages, this declaration does not need to be formal – it is simply an assignment with an initial value:

```
temp = 0
```

If this assignment does not take place, the program will abort at runtime with a variable not assigned error. For example:

```
PROGRAM TEST
FOR ii = 1 TO 10
  jj += 1
  CRT jj
NEXT ii
END
```

```
BASIC BP TEST
RUN BP TEST
000000B1: Unassigned variable JJ at line 3 of
D:\QM\QMINTRO\BP.OUT\TEST
```

In this example, the variable `jj` had no value when the program encountered it for the first time.

Tokens

Tokens are predefined language elements that you use to represent a value in the programming environment. They aren't variables because you (usually) cannot assign new values to them, nor are they constants because they may change.

In *OpenQM* (as with other multi-value environments), tokens are prefixed by the '@' symbol. The tokens dealing with multi-value delimiters have already been covered in Section 2.4.3. A few of the other frequently used tokens are shown in the table below:

Token	***Meaning***
`@TRUE`	A logical true value – nominally 1
`@FALSE`	A logical false value – nominally 0
`@LOGNAME` or `@USER`	The user's operating system logon name
`@SENTENCE`	The most recently executed QM sentence
`@WHO`	The current account

The full list of tokens (or @-variables) can be found in the *OpenQM* help files under `QMBasic`.

Constants

Constants are language elements that are assigned values that never change. Further, if assignment of a new value is attempted, an error will result.

Constants are assigned using the `EQUATE` statement:

```
EQUATE pi TO 3.141592654
```

Thereafter, constants may be used within `QMBasic` expressions:

```
area = pi * PWR(radius, 2)
```

Multi-value Variables

OpenQM is a multi-value database, and accordingly, variables used in the programming language may be multi-valued. Such variables may also be termed 'dynamic arrays' or delimited strings, and are denoted as:

```
varname<amc {,vmc {,svmc}} >
```

In this syntax, the angle brackets '<' and '>' indicate the use of a multi-valued variable. Multi-valued variables can have up to 3 dimensions corresponding to attributes[13] (fields), values, and sub-values. The following are valid multi-value variables:

```
abc<3>
def<2, 1>
ghi<ii, jj, kk>
```

The final example uses variables to denote the index positions of each dimension of the variable.

If you wish to append data to the array, you can use an index value of -1. For example:

```
abc<-1>
def<-1, 1>
```

13 They syntax description here uses abbreviations of amc, vmc, and svmc. These mean "attribute mark count", "value mark count", and "sub-value mark count". These abbreviations are common in the PICK world.

```
ghi<3, 1, -1>
```

Note that you do not need to explicitly set the dimensions of the variable before use. You can simply add dimensions, and add elements to each dimension as required:

```
score = ''
FOR die1 = 1 TO 6
  FOR die2 = 1 to 6
    score<die1, die2> = die1 + die2
  NEXT die2
NEXT die1
```

This creates a 2-dimensional dynamic array holding all the possible scores from rolling 2 dice. This could then be used to return the combined value of two die:

```
rollvalue = score<die1, die2>
```

Referencing positions in dynamic arrays is often done by the numeric position in the array. However, as dynamic arrays often hold data that has been read from a file, it is often useful to use an equated constant that gives you a better idea of what the data is in the variable. Consider the following statements:

```
rec<10> = rec<8> + rec<9>
```

```
sales.rec<SL.TOTAL> = sales.rec<SL.SUBTOTAL> + sales.rec<SL.VAT>
```

There are two points to take from these statements:

- Using equated constants has made the statement much more understandable – you don't have look up what data is held in positions 8, 9, and 10 in the dynamic array to understand the meaning of the statement
- Mistakes in programming logic are much easier to spot because you can read the purpose of the statement. For example, the first statement may have been written as:

  ```
  rec<10> = rec<8> + rec<19>
  ```

  The mistake is not immediately obvious. This mistake would be unlikely if equated constants had been used.

OpenQM can create a set of equated constants for you from a file dictionary. The command for this is GENERATE:

```
GENERATE XRATES
Type (D=dynamic array, M=matrix, or DM=both): D
Prefix for dynamic array tokens  (excluding separator): XR
```

This creates an item in the BP file that can be included in programs:

```
CT BP XRATES.H
BP XRATES.H
01: * BP XRATES.H
02: * Generated from DICT XRATES at 18:16:42 on 16 May 2009
03:
04: equate XR.TWI     to    1
05: equate XR.TWI.CNV to "MR1,Z"
06: equate XR.USD     to    2
07: equate XR.USD.CNV to "MR4,Z"
08: equate XR.GBP     to    3
09: equate XR.GBP.CNV to "MR4,Z"
10: equate XR.AUD     to    4
11: equate XR.AUD.CNV to "MR4,Z"
12: equate XR.JPY     to    5
13: equate XR.JPY.CNV to "MR2,Z"
14: equate XR.EUR     to    6
15: equate XR.EUR.CNV to "MR4,Z"
```

```
16: equate XR.GDM     to    7
17: equate XR.GDM.CNV to "MR4,Z"
```

Note this has created constants not only for the field numbers in the record, but also the conversion codes used to convert data from internal to external format.

Operators

OpenQM has the usual set of operators for writing expressions and relational conditions. In addition, it has a set of substring extraction operators, a pattern matching operator, and a set of alternative relational operators.

Substring extraction

OpenQM uses square brackets ([and]) to denote substring extraction as follows:

```
substring = string[startpos, numchars]
```

For example:

```
st = 'AbcDefGhi'
ss = st[4,3]
```

In the above program fragment, the substring 'Def' is assigned to the variable `ss`.

To extract the last n characters, use:

```
substring = string[n]
```

For example:

```
st = 'AbcDefGhi'
ss = st[3]
```

In this example, the substring 'Ghi' is assigned to the variable `ss`.

In some other languages, this substring extraction functionality is provided by the `LEFT`, `MID`, and `RIGHT` functions.

Pattern matching

Pattern matching is often used in the context of an `IF` statement:

```
IF var MATCHES pattern THEN ...
```

For example:

```
xx = '(04) 456 7890'
IF xx MATCHES '"("2N") "3N" "4N' OR xx MATCHES '"("2N") "3N"-"4N' THEN
```

The above program fragment matches the test string to see if it is a phone number. Note, however, that not all phone numbers will match the test patterns. Clearly, some thought needs to be put into the appropriate tests for phone numbers.

For more information on how to define the test patterns, search for `MATCHES` in the online help.

Alternative relational operators

Most languages have a set of mathematical symbols for use in relational comparisons. *OpenQM* has a set of mnemonic codes that act as synonyms for these mathematical symbols. These are shown in the table below:

Symbol	*Alternatives*
<	LT
>	GT
=	EQ
#	NE <> ><
<=	LE =< #>
>=	GE => #<
MATCHES	MATCH
AND	&
OR	!

For example:

```
IF xx LT 1 THEN xx = 1
```

Assignment

Simple assignment takes the form:

```
var = value
var = expression
```

Assignment shortcuts

In common with some other languages, *OpenQM* offers shortcuts for repeated operations. For example:

```
xx = xx + 1
```

could be written as:

```
xx += 1
```

This not only saves keystrokes in typing, but is actually more efficient in execution. These shortcuts take the general form:

```
var shortcut value
```

The following set of assignment shortcuts are available:

+=	Add expression to the original value
-+	Subtract expression from the original value
*=	Multiply original value by expression
/=	Divide original value by expression
:=	Concatenate expression as a string to the original value

For example:

```
outputstring := thisline:CR:LF
```

This takes the current value of outputstring and appends the string value of thisline and a carriage return/line feed sequence.

Note this also introduces the use of the colon (:) as the means of concatenating two strings together. For example:

```
newstring = string1:string2
```

Whitespace can be introduced into the statement to make the operators more visible without changing the functionality of the statement. For example:

```
newstring = string1 : string2
```

This separation makes each component of the statement more visible, and can aid with understanding.

Substring assignment

Substring extraction was outlined in the section on operators above. Similar techniques can be used to assign values to substrings:

```
var[startpos, numchars] = expression
```

For example:

```
st = 'Abcdefghi'
st[4,3] = '123'
```

The new value of st would be: Abc123ghi

Now consider the following example:

```
st = 'Abcdefghi'
st[4,3] = '12'
```

In this case, the new substring is shorter than the existing substring. *OpenQM*'s behaviour (and the resulting value of *`st`*) here is dependent on the `$MODE` settings used to control the compiler. See the online help for more information.

A shortcut method is available to assign the trailing characters of a string:

```
var[numchars] = expression
```

For example:

```
st = 'Abcdefghi'
st[3] = '123'
```

The new value of *`st`* would be: Abcdef123

Finally, *OpenQM* has another form of substring assignment particularly suited to use with delimited strings:

```
var[delimiter, firstgroup, numberofgroups] = expression
```

For example:

```
key = '1234*DEF*14996'
key['*', 3, 1] = DATE()
```

This would replace the 14996 part of the key with the internal value of the current date. For the 14th of April, 2009, this would make the key value: *`1234*DEF*15080`*

Null values

In common with the most other multi-value databases, *OpenQM* does not have a special character representing the null value. For practical purposes, an empty string is considered a null value.

3.3 A simple program

The best way to illustrate the statements and functions available in *OpenQM* is to show them in operation. Therefore, we will now consider a short program and examine how it works.

Introduction to QMBasic

Part 1 of '*Getting Started in OpenQM*' used several files to illustrate how to use `QMQuery`. These files contained interest and exchange rate for New Zealand. Building on this base, an appropriate first program will allow this data to be viewed.

3.3.1 Program

```
PROGRAM SHOW.XRATES
**********************************************************************
* Program to display exchange rates for a year nominated by the user.
*
$CATALOGUE

PROMPT ''
OPEN 'XRATES' TO xrates ELSE STOP 201, 'Xrates'
OPEN 'DICT','XRATES' TO xrates.dict ELSE STOP 201, 'Dict Xrates'

! Define constants

$INCLUDE SYSCOM KEYS.H
$INCLUDE BP XRATES.H

! Read conversions and headings from dictionary

dnames = 'TWI,USD,GBP,AUD,JPY,EUR'
CONVERT ',' TO @AM IN dnames

cnvs = ''
hdgs = ''
FOR cc = XR.TWI TO XR.EUR
  dictname = dnames<cc>
  READ drec FROM xrates.dict,dictname THEN
    conversion = drec<3>
    IF conversion EQ '' THEN conversion = 'MR0,Z'
    cnvs<cc> = conversion
    hdgs<cc> = drec<4>
  END
NEXT cc

! Page heading

CRT @(IT$CS):
CRT @(00,00):'Display exchange rates'
CRT @(00,01):'======================'

! Main loop

LOOP
  CRT @(00,03):@(IT$CLEOS):'Enter year: ':
  INPUT yyyy:
  IF (yyyy EQ '') OR (UPCASE(yyyy) EQ 'X') THEN EXIT

  IF NOT(yyyy MATCHES '4N') THEN
    CRT @(18,03):'Invalid year':
    INPUT pause,1:
    CONTINUE
  END

!  Read data from file and display

  first = @TRUE
  FOR mm = 1 TO 12
    xid = yyyy:mm 'R%%'
    READ xrec FROM xrates, xid THEN
      IF first THEN
        CRT @(00,05):'Month':
        FOR cc = XR.TWI TO XR.EUR
          CRT hdgs<cc> 'R#12':
        NEXT cc
        CRT
        first = @FALSE
      END

      xdate = '01/':mm:'/':yyyy
      CRT OCONV(ICONV(xdate, 'D'), 'DMAL[3]') 'L#5':
      FOR cc = XR.TWI TO XR.EUR
        CRT OCONV(xrec<cc>, cnvs<cc>) 'R#12':
      NEXT cc
      CRT
    END
  NEXT mm

  IF NOT(first) THEN
    CRT
    CRT 'Press any key to continue ':
  END ELSE
    CRT @(00,05):'No data on file for year ':yyyy:
  END
  INPUT pause,1:
REPEAT

STOP
END
```

3.3.2 Analysis

This program asks the user to enter a year. It then displays the exchange rate data on file for that year. The data is displayed until the user presses enter, and then the program returns to the prompt for a year. The user exits the program by pressing enter at the year prompt. Sample output is shown below:

```
Display exchange rates
======================

Enter year: 2004

Month       TWI      USD      GBP      AUD      JPY      EUR
Jan        66.4   0.6724   0.3690   0.8728    71.55   0.5332
Feb        68.0   0.6916   0.3703   0.8891    73.68   0.5473
Mar        66.3   0.6614   0.3616   0.8811    71.90   0.5388
Apr        64.7   0.6419   0.3556   0.8622    68.94   0.5350
May        63.1   0.6156   0.3445   0.8731    69.00   0.5127
Jun        64.2   0.6293   0.3440   0.9058    68.81   0.5184
Jul        65.4   0.6466   0.3508   0.9030    70.67   0.5268
Aug        66.5   0.6542   0.3594   0.9209    72.19   0.5370
Sep        67.1   0.6588   0.3674   0.9384    72.49   0.5392
Oct        68.5   0.6825   0.3782   0.9326    74.47   0.5471
Nov        68.3   0.6993   0.3764   0.9084    73.29   0.5387
Dec        69.0   0.7142   0.3702   0.9315    74.18   0.5337

Press any key to continue
```

Let’s step through the program to see how it works:

The first functional line is `$CATALOGUE`. This is a compiler directive rather than a statement, and tells the compiler to automatically catalogue the program in the private catalogue every time it is compiled.

`PROMPT ''` sets the prompt character to nothing. By default, *OpenQM* displays a '?' character whenever the program reaches an `INPUT` statement. By setting it to null, the developer has full control over how the program appears to the user.

The two `OPEN` statements open the `XRATES` data file and dictionary. This lets the program read from these files later.

The two `$INCLUDE` statements include code fragments from elsewhere on the system. The first of these is the keys definition file included with *OpenQM*. The second is the file definition item we created for the `XRATES` file using the `GENERATE` command in section

3.2.2. By including these files, we can use mnemonic codes for file references and system functions rather than numbers.

The next section reads the column headings and conversions to apply to the data from the file dictionary. First, we create a dynamic array that contains the ID's of the dictionary items. This is done in two steps – firstly by creating a string variable containing the ID's, and then by converting the string to a dynamic array by changing the commas to attribute marks. This approach is used because it is easier than writing a line such as:

```
dnames = 'TWI':@AM:'USD':@AM:'GBP':@AM:'AUD':@AM:'JPY':@AM:'EUR'
```

Next, the conversions and column headings for each of the columns are read from the dictionary. These conversions transform the stored data from its internal format to its external format. While these conversions were also defined in the include file created using the `GENERATE` command, reading them from the dictionary puts them into a more general structure which makes them easier to use.

The screen is then cleared, and some page headings put in place.

The rest of the program is encompassed by the main loop. This starts with a statement that clears the screen from line 3 onwards, and displays a prompt asking the user to enter a year. This year is then tested to see if it matches a 4 digit pattern. If the year does not match this pattern, then an error message is displayed, and the loop is started again.

If the year does contain 4 digits, then the program will attempt to read data for that year. Note that this will happen even if the year entered is nonsensical – such as 0000 or 2050. However, the read is structured so that all years are handled appropriately, whether they contain data or not.

An attempt is made to read the data for each month of the year. If data for this month is found on file, then the data is displayed. If this is the first month found for the year, then the column headings are written to the screen first. This means that if no data are found for the year, then no column headings are displayed.

Each line of data is displayed by first calculating the month name. Once the month is displayed, then the program loops through the record converting the data to external format, and displaying it.

Finally, a message is displayed, either asking the user to press any key to continue, or informing them that there was no data on file for that year. This message is displayed until the user presses enter, when control returns to the top of the loop.

3.3.3 Detail points

The program uses defined constants wherever possible to avoid incorrect numbers being accidentally used. Example of this are:

```
FOR cc = XR.TWI TO XR.EUR
```

```
CRT @(IT$CS):
```

The constants *`XR.TWI`* and *`XR.EUR`* were defined in the `XRATES.H` include item, and evaluate to values of 1 and 6 respectively. Therefore, the `FOR` statement evaluates to:

```
FOR cc = 1 TO 6
```

Likewise, the `CRT` statement evaluates to:

```
CRT @(-1):
```

This expression is used to clear the screen. Similarly, the line:

```
CRT @(00,03):@(IT$CLEOS):'Enter year: ':
```

firstly, positions the cursor, then clears the screen from the position onwards using the `@(-3)` function.

The constants used in these screen functions are defined in the `KEYS.H` item in the `SYSCOM` file. This item contains many more constants for use within *OpenQM*.

These `CRT` statements are terminated by a colon. A colon in an expression is used to concatenate two or more elements together. When used at the end of a `CRT` statement, it acts to hold the cursor at that position. If the colon were not present at the end of the "Enter year" statement, then an automatic end of line sequence would be executed, causing the cursor to go to the left hand edge of the following line.

A number of lines have formatting expressions:

```
xid = yyyy:mm 'R%%'
```

```
CRT hdgs<cc> 'R#12':
```

In the first of these, the format expression `'R%%'` forces the variable *`mm`* to be formatted right-justified in a field 2 characters wide and padded with zeroes. Therefore, a value of *`mm`* of 1 (i.e. month 1 or January) would be formatted as `01`. This is concatenated to the 4 digit year (say `2009`) to produce an exchange rate identifier of `200901`. This identifier is then used to read the data for that month from the exchange rates file.

The second use of the format expression is to format the output of the column headings (and later of the rows of data). The `'R#12'` expression means format the data right-justified in a field of 12 blank characters.

The printing of the month name is accomplished by the following two lines:

```
xdate = '01/':mm:'/':yyyy
CRT OCONV(ICONV(xdate, 'D'), 'DMAL[3]') 'L#5':
```

The first line is clear enough. This creates an international format date string from the month number and year. Therefore, for a month number of 1 and a year of 2009, *`xdate`* would be '01/1/2009'. If you are in North America, you would need to change this date string, or specify a European date conversion in the following statement.

The second line is a little more complicated, but is really just three operations carried out in a single line. First, the date string is converted to an internal date:

```
ICONV(xdate, 'D')
```

So, '01/01/2009' would be converted to 15128. (Type in: `DATE INTERNAL 01/01/09` from the command prompt).

The next part gets a month representation from that date:

```
OCONV(15128, 'DMAL[3]')
```

The `'DMA'` portion of the conversion expression returns the month name as 'JANUARY'. The `'L'` converts this to Title case (January), and the `'[3]'` truncates this to 3 characters (Jan).

Finally, this is displayed left-justified in a field of 5 blank characters:

```
CRT 'Jan' 'L#5':
```

Note the presence of the colon to keep the cursor on the current output line.

The final details involve printing a line of data:

```
FOR cc = XR.TWI TO XR.EUR
  CRT OCONV(xrec<cc>, cnvs<cc>) 'R#12':
NEXT cc
CRT
```

This code fragment displays each of the columns in a field of 12 characters wide on the screen. Each field is placed immediately to the right of the preceding field because the cursor is held on the line by the colon following the format expression. Once the loop is complete for each row, the cursor is still held on the output line. The `CRT` statement on its own outside the loop moves the cursor to the next line to print a new month's data.

The actual expression used to display the data is also interesting. This is easier to understand if we substitute the variable *`cc`* for one of its actual values:

```
CRT OCONV(xrec<XR.TWI>, cnvs<XR.TWI>) 'R#12':
```

This expression says take the TWI value from the exchange rate record and apply the output conversion defined for the TWI field, then display it right-justified in a field of 12 spaces. If we go back to the sample output displayed above, we can see that the TWI for January 2004 was 66.4 – this is the output format. This was actually stored as 664, and the output conversion of `'MR1,Z'` converted this internal representation of 664 to an external representation of 66.4. To check the actual conversion used by TWI, check the conversion stored in the dictionary:

```
CT DICT XRATES TWI
```

The conversion appears on line 3 of the dictionary. We read these output conversions from the dictionary at the start of the program to fill the *`cnvs`* dynamic array. Because the structure of the *`cnvs`* dynamic array matches that of the *`xrec`* dynamic array that holds the exchange rate, we could use these in a matched fashion in the display loop.

It is worthwhile considering how we could have used the conversions defined in the `$INCLUDE` statement to display the same data. To have used those conversions, we would have had to refer to them by their name. That would have meant the display statement would have been something like:

```
CRT OCONV(xrec<XR.TWI>, XR.TWI.CNV>) 'R#12':
CRT OCONV(xrec<XR.USD>, XR.USD.CNV>) 'R#12':
CRT OCONV(xrec<XR.GBP>, XR.GBP.CNV>) 'R#12':
etc
```

So, by putting the conversions in a dynamic array, we were able to make the display portion of the program much more compact. Of course, we could have accomplished this by creating the *`cnvs`* dynamic array from the defined conversions rather than reading from the dictionary:

```
cnvs = ''
cnvs<XR.TWI> = XR.TWI.CNV
cnvs<XR.USD> = XR.USD.CNV
cnvs<XR.GBP> = XR.GBP.CNV
etc
```

In fact, this is probably a better method of loading the conversions than reading from the dictionary. The reason for this is that *OpenQM* supports dictionary structures used in other multi-value databases that have the conversion in a different place in the dictionary. Further, even if we adapted the read section to get the conversion from the correct position, we would also need to check that the value did not contain some other processing codes. Therefore, our dictionary read section would become something like:

```
cnvs = ''
hdgs = ''
FOR cc = XR.TWI TO XR.EUR
  dictname = dnames<cc>
  READ drec FROM xrates.dict,dictname THEN
    dtype = drec<1>[1,1]
    BEGIN CASE
      CASE (dtype EQ 'D') OR (dtype EQ 'I') OR (dtype EQ 'C')
        conversion = drec<3>
        hdgs<cc> = drec<4>
      CASE (dtype EQ 'A') OR (dtype EQ 'S')
        conversion = drec<7>
        hdgs<cc> = drec<3>
      CASE 1
        conversion = ''
    END CASE
    dc = DCOUNT(conversion<1>, @VM)
    IF dc GT 1 THEN
      conversion = conversion<dc>
    END
    IF conversion = '' THEN conversion = 'MR0,Z'
    cnvs<cc> = conversion
  END
NEXT cc
```

This code fragment gets the heading and conversion from the appropriate position for each dictionary type. Note the following points:

- Some dictionary items (usually A and S types) may have multi-valued conversion fields. The code therefore checks for the presence of value marks and gets the last value as the conversion code.
- If the dictionary item is not one of the specified types, then the conversion is explicitly set to a null value. If we didn't do this, then the variable 'conversion' would still contain the conversion from the previous dictionary item (or would be undefined if this was the first loop).
- Headings are only set for the specified dictionary types. If the dictionary is of some other type, then no heading is set.

The alternative to this code is to use the `GENERATE` command to create an include file containing the conversions. You can then check that include file to make sure the conversions are valid, and edit them if necessary.

Other improvements that could be made include reading the display width for each column from the dictionary (field 5 of a D, I, or C type dictionary), and allowing for multi-line column headings.

Overall, this is a fairly simple program. However, it introduces many of the basic elements that are present in larger programs – data input, validation, reading data from a file, formatting data, and outputting formatted data to an output device. All that is really missing is writing data to a file.

3.4 Some useful functions and statements

There are a number of functions that are frequently used in multi-value databases. It is useful to outline some of these here so that you will understand them as we cover other programming structures.

CRT and DISPLAY

`CRT` and `DISPLAY` are the same function. In this book, `CRT` will be used, but in other programs you may find `DISPLAY` being used.

As its name suggests, `CRT` outputs data to the screen. Its syntax is:

```
CRT printitem {,printitem}
```

The printitem may be a variable, expression, or a literal string. If there are multiple printitems separated by commas, the commas are replaced by TAB characters to produce a simple formatted display.

You can include a cursor positioning command as part of the CRT statement:

```
CRT @(col,line)
```

```
CRT @(col)
```

Line and column numbering starts at zero, so @(0,0) will position the cursor at the top left of the screen. The second version shown will position at 'col' on the current line.

The CRT @(x) function also accepts negative arguments for terminal control purposes. Some useful examples of these include:

```
CRT @(-1)       Clear screen
CRT @(-3)       Clear to end of screen
CRT @(-4)       Clear to end of line
```

Tokens to use with the CRT function are contained in the KEYS.H include item in the SYSCOM file. Tokens are useful because they make it less likely that you will use an incorrect numeric value, and they make the code easier to read. The tokenised equivalents of the three CRT statements shown above are:

```
CRT @(IT$CS)
CRT @(IT$CLEOS)
CRT @(IT$CLEOL)
```

To use these tokens, you need to $INCLUDE the KEYS.H item in your program.

Note that the CRT function issues an automatic end of line sequence, so if you want to hold the cursor at that position, you need to suppress this sequence. This is done by appending a colon (:) to the command:

```
CRT @(20,05):
```

It is common for a CRT statement to contain an expression which builds a formatted string for display:

```
CRT @(sx,sy):qty<mthno> 'R#12Z':
```

The above statement outputs the value contained in the *mthno* attribute of the *qty* variable at position *(sx, sy)* on the screen, displayed right-justified in a field of 12 spaces. See the online help for more information on format specifications.

PRINT

PRINT is closely related to the CRT function. However, whereas the CRT function always outputs to the screen, PRINT will output to an output device – whether that is the screen or a print unit (printer). Its syntax is:

```
PRINT {ON printunit} printitem {,printitem}
```

Output is directed as follows:

- If a print unit is specified, then output will be directed to that print unit
- If the print unit specified is -1, then output will go to the screen
- Print unit 0 (the default) is switchable between the screen and the print unit by use of the PRINTER ON/OFF statement

- Printunit may be a number between 0 and 255. The characteristics of the print unit may be set using the SETPTR command from the *OpenQM* command environment, or the SETPU statement from within QMBasic.

```
PRINT "This text will go to the screen"
PRINTER ON
PRINT "This text will go to the print unit"
PRINT ON -1 "This text will go to the screen"
PRINTER OFF
```

As with the CRT statement, you can include cursor positioning commands with the PRINT statement – but these will have no effect if output is directed to a print unit.

INPUT

INPUT accepts input from the keyboard or data queue and stores that input in a variable:

```
INPUT variable {,length}
```

INPUT accepts a number of other parameters not shown above – see the online help for more details.

If length is specified, then input of characters is automatically terminated when that number of characters has been input:

```
INPUT pause,1
```

A more sophisticated version of INPUT is the INPUT @(x,y) statement. This version allows cursor positioning, formatting of the displayed/entered variable, and various modes of editing – see the online help for more details.

DCOUNT

DCOUNT is used to count the number of components in a delimited string. As multi-value variables are simply delimited strings, it is frequently used to count the number fields, values, or sub-values in a variable. Its syntax is:

```
number = DCOUNT(string, delimiter)
```

For example:

```
menucnt = DCOUNT(menuitems<1>, @VM)
```

This example counts the number of values in the first field of the variable *menuitems*.

There is a closely related function named COUNT which simply counts the delimiters:

```
number = COUNT(string, delimiter)
```

Usually, DCOUNT returns a number that is one greater than that COUNT.

FIELD

FIELD extracts a component of a delimited string. This isn't usually used for extracting fields, values, and sub-values as the language provides for direct extraction and assignment of these elements. However, we are frequently confronted by strings delimited by other characters.

For example, a compound key may have the structure:

```
plant * model no * date
e.g.  123*A-4567-IJK*15096
```

Each of these components may have variable length (although date should be stable at 5 characters for some time to come), so we cannot extract by position. This is where we use the FIELD function. This has the following syntax:

```
component = FIELD(string, delimiter, occurrence {, count})
```

Therefore to extract each of the components of the variable ‘key’, we would use:

```
plant = FIELD(key, '*', 1)      returns '123'
modelno = FIELD(key, '*', 2)    returns 'A-4567-IJK'
pdate = FIELD(key, '*', 3)      returns 15096
```

Note that the strings returned by the field statement exclude the delimiter – unless more than one field is returned:

```
plantmodel = FIELD(key, '*', 1, 2)   returns '123*A-4567-IJK'
```

Similar functionality can be achieved by using a group extraction conversion in an `OCONV` function:

```
PROGRAM TEST
key = '123*A-4567-IJK*15096'
plant = FIELD(key, '*', 1)
modelno = FIELD(key, '*', 2)
pdate = FIELD(key, '*', 3)
plantmodel = FIELD(key, '*', 1, 2)
pm2 = OCONV(key, 'G0*2')

CRT 'Plant = ':plant
CRT 'Model = ':modelno
CRT 'Pdate = ':pdate
CRT 'PlantModel = ':plantmodel
CRT 'PM2 = ':pm2

END
```

```
RUN BP TEST
Plant = 123
Model = A-4567-IJK
Pdate = 15096
PlantModel = 123*A-4567-IJK
PM2 = 123*A-4567-IJK
```

`FIELD` has two associated functions – `COL1()` and `COL2()` – which return the character positions immediately prior to (`COL1()`) and immediately following (`COL2()`) the extracted string. These are useful for extracting the strings prior to and/or following the extracted field:

```
key = '123*A-4567-IJK*15096'
modelno = FIELD(key, '*', 2)
priorstring = key[1,COL1()]
followstring = key[COL2(), LEN(key)]
```

See the online help for more information.

ICONV / OCONV

`ICONV` and `OCONV` are transformation functions – `ICONV` converts data to internal format, while `OCONV` converts data to external format. The `'I'` and the `'O'` stand for input and output, representing the type of conversion that takes place at these times. An input conversion will be done at the time of data input, and converts data from external to internal format. An output conversion will be done prior to display, and converts data from internal to external format.

Dates and times are obvious examples of data with differing input and output representations, but any data may have conversions applied using these functions. Some examples are shown below:

Conversion	**Result**
`x = OCONV(15100, 'D')`	`04 MAY 2009`
`x = ICONV('4 MAY 2009', 'D')`	`15100`
`x = OCONV(32000, 'MTS')`	`08:53:20`
`x = ICONV('14:20', 'MTS')`	`51600`
`x = OCONV('SAMPLE TEXT', 'MCL')`	`sample text`
`x = OCONV('sample text', 'MCU')`	`SAMPLE TEXT`

```
x = OCONV('SAMPLE TEXT', 'MCT')       Sample Text
x = ICONV(123.456', 'MR33')           123456
x = OCONV(123456, 'MR22,$')           $1,234.56
```

See 'Conversion Codes' in the online help for more information on possible conversions.

LOCATE

`LOCATE` is used for finding a value within a delimited string. `LOCATE` is frequently used where a variable contains sets of related data.

The basic concepts of the `LOCATE` statement are:

- A search for a string is made through the specified part of a delimited variable returning:
    - whether the string was found
    - the position in the variable that the variable occupies, or would occupy if it existed
- `THEN` or `ELSE` clauses are executed depending on whether the string was found.

OpenQM allows several variants for the syntax of the `LOCATE` statement. The variant shown here is the *UniVerse* version. This requires a `MODE` setting in the program:

```
$MODE UV.LOCATE
LOCATE string IN dyn.array {<field {,value}>} {BY order} SETTING pos
{THEN statements } {ELSE statements}
```

The `$MODE` statement need only be declared once in the program. Alternatively, it can be specified in a `$BASIC.OPTIONS` item in the programs file, or in the `VOC` of the account.

Say we want to search through the days sales and return the value of sales by country.:

```
OPEN 'SALES' TO sales ELSE STOP 201, 'Sales'
countries = ''
SELECT sales
eof = @FALSE
LOOP
  READNEXT sales.id ELSE eof = @TRUE
UNTIL eof DO
  READ sales.vec FROM sales, sales.id ELSE sales.vec = ''
  sales.ctry = sales.vec<SL.CTRY>
  IF sales.ctry = '' THEN sales.ctry = 'GB'
  sales.value = sales.vec<SL.VALUE>
  IF sales.ctry AND sales.value THEN
    LOCATE sales.ctry IN countries<SS.CTRY> BY 'AL' SETTING cpos THEN
      countries<SS.CTRYVALUE, cpos> += sales.value
    END ELSE
      INS sales.ctry BEFORE countries<SS.CTRY, cpos>
      INS sales.value BEFORE countries<SS.CTRYVALUE, cpos>
    END
  END
REPEAT
```

This code fragment builds the dynamic array *`countries`* by searching for the country identifier in the relevant field of the dynamic array. This search uses an ascending left-justified sort order (i.e. alphabetic). If the country already exists, then the position of the country in the field is returned in *`cpos`*, and the value of the sale is added to the existing sales values for that country in the appropriate field. If the country does not exist, then *`cpos`* contains the location within the dynamic array where it should be inserted, and both the country identifier and the sales value are inserted into the dynamic array.

Note that for this code fragment to work, there needs to be an `$INCLUDE` statement that defines the *`SL.CTRY`*, *`SL.VALUE`*, *`SS.CTRY`* and *`SS.CTRYVALUE`* constants.

`LOCATE` can also be used without a sort order. In this case, the `LOCATE` portion of the above code fragment would look like:

```
LOCATE sales.ctry IN countries<SS.CTRY> SETTING cpos ELSE
  countries<SS.CTRY, cpos> = sales.ctry
END
countries<SS.CTRYVALUE, cpos> += sales.value
```

This code is a bit simpler because it doesn't have to insert values into the dynamic array. If the country is found in the dynamic array, then all that needs to happen is to add the sales value to the appropriate position. If the country is not found, then the value returned by *cpos* is at the end of the existing data – therefore, you can simply assign the country to that position and add the sales data. Because the sales data is added to a position in the dynamic array whether the country is found or not, this is removed from the locate statement logic and made an unconditional statement.

Note the following points about the use of `LOCATE`:

- Searching within a dynamic array will slow down as the array size gets larger. For this reason, avoid building large lists if possible.
- A sorted list will be more efficient than an unsorted list. This is because once the location of the target is found (whether or not it currently exists), searching will stop. However, with an unsorted dynamic array, searching needs to continue right through the list to determine whether it is present in the list.
- `LOCATE` finds an entire value within the dynamic array. If you want to find part of a string, then you need to use a different function.

Now, while the above example usage of `LOCATE` may be a reasonable demonstration of using `LOCATE`, it is not an efficient piece of code. This is because the `LOCATE` will run for every item in the sales file. This is quite unnecessary – and inefficient. Consider the following alternative:

```
OPEN 'SALES' TO sales ELSE STOP 201, 'Sales'
EXECUTE \SSELECT SALES BY CTRY\
countries = ''
sales.value = 0
sales.ctry.prev = ''
eof = @FALSE
LOOP
  READNEXT sales.id ELSE eof = @TRUE
UNTIL eof DO
  READ sales.vec FROM sales, sales.id THEN
    sales.ctry = sales.vec<SL.CTRY>
    IF sales.ctry = '' THEN sales.ctry = 'GB'
    IF sales.ctry NE sales.ctry.prev THEN GOSUB addctry
    sales.value += sales.vec<SL.VALUE>
  END
REPEAT

IF sales.value THEN GOSUB addctry
*
RETURN

addctry:
IF sales.value THEN
  countries<SS.CTRY, -1> = sales.ctry.prev
  countries<SS.CTRYVALUE, -1> = sales.value
END
sales.value = 0
sales.ctry.prev = sales.ctry
*
RETURN
```

This code fragment does not use `LOCATE` at all, but will build the same dynamic array of sales by country. The code works by sorting the data first, and then loops through the data accumulating the sales figures. The accumulated data is only added to the dynamic array when the country changes, or on final exit from the loop.

Now, say the dynamic array that is created by this code is saved to a sales summary file using a key of the internal date value. The following code fragment shows LOCATE being used to query the item and return the value of sales for a given country:

```
PROMPT ''
OPEN 'SALES.SUMMARY' TO sales.summary ELSE STOP 201,
'Sales.summary'
LOOP
  CRT 'Enter date: ':
  INPUT sdate
  IF (sdate = '') OR (UPCASE(sdate) = 'X') THEN EXIT
  CRT 'Enter country: ':
  INPUT ctry
  IF (ctry = '') OR (UPCASE(ctry) = 'X') THEN EXIT

  idate = ICONV(sdate, 'D')
  IF (idate = '') OR NOT(NUM(idate)) THEN
    CRT 'Invalid date'
    CONTINUE
  END
  sdate = OCONV(OCONV(idate, 'D'), 'MCT')
  READ sales.vec FROM sales.summary, idate ELSE
    CRT 'Sales data for ':sdate:' not on file'
    CONTINUE
  END

  LOCATE ctry IN sales.vec<SS.CTRY> SETTING cpos THEN
    sales.value = sales.vec<SS.CTRYVALUE, cpos>
  END ELSE
    sales.value = 0
  END
  sdate = OCONV(OCONV(idate, 'D'), 'MCT')
  CRT 'Sales for ':sdate:' were ':OCONV(sales.value, 'MR2,')
REPEAT
```

This program gets a date and a country from the user, then reads the summary item for that date. If no data exists for that date, then an error message is displayed, and the CONTINUE statement causes the program to start another LOOP. If data is found, then the country is located within the dynamic array, and the matching sales value returned. Finally, the sales value for the country for that date is displayed.

Note that the date value has some interesting conversions in this program. It is initially converted to an internal format, then the internal date value has a double output conversion applied to it. The first of these conversions converts the internal value to a standard date string (e.g. 08 MAR 2010), while the second converts this to a mixed case string (08 Mar 2010). This ensures consistency of display of date values.

As LOCATE is such an important statement within the multi-value databases, we'll give another example of its usage:

Say we have a banking application. Account balances are held in file ACCBALS. This file records the account balance of an account on any day that a transaction occurs. In addition, an account balance will be stored in the file on the last day of the month regardless of whether any transaction has occurred during the month. Our problem is: How do we efficiently find the account balance on any given day?

An account that is used daily will have as many account balances stored as there are (working) days in the month. An account that has no transactions during the month will only have a single account balance stored for the month.

The brute force method of getting the account balance is simply to attempt to read the account balance for the given day. If the item doesn't exist, then step back a day at a time until an account balance is found. The problem here is that we may need to attempt 30 reads from the file before we find the balance. This is not efficient.

A more efficient method involves indexing the monthly transactions. Let's set this out in a bit more detail:

The ACCBALS file has the following structure:

ID:	account*date	e.g.	12345678*15317
F1:	account balance	e.g.	1534682

The date in the account balance is 7 December, 2009. We know there will also be an item in the file with an ID of 12345678*15310 (being 30 November, 2009).

We also have an index file named ACCBALS.NDX. This has the following structure:

ID:	account*yyyymm	e.g.	12345678*200912
F1:	mv-list of dates	e.g.	15317]15314]15310

Note that the last date in the list is actually the November month-end date. The closing square brackets (]) represent value marks. The three dates shown are 7 December, 4 December, and 30 November.

When a transaction occurs, the application maintains the ACCBALS.NDX file, recording the dates on which new balances are written to the ACCBALS file. The month-end process has to create the index item for the next month and seed it with the date of the month-end just past.

Now, let's see how to use this structure to obtain the account balance:

```
PROMPT ''
OPEN 'ACCBALS' TO accbals ELSE STOP 201, 'Accbals'
OPEN 'ACCBALS.NDX' TO accbals.ndx ELSE STOP 201, 'Accbals.Ndx'
LOOP
  CRT 'Enter date: ':
  INPUT sdate
  IF (sdate = '') OR (UPCASE(sdate) = 'X') THEN EXIT
  idate = ICONV(sdate, 'D')
  IF (idate = '') OR NOT(NUM(idate)) THEN
    CRT 'Invalid date'
    CONTINUE
  END

  CRT 'Enter account: ':
  INPUT account
  IF (account = '') OR (UPCASE(account) = 'X') THEN EXIT

  ndxid = account:'*':OCONV(idate, 'DY'):OCONV(idate, 'DM') 'R%%'
  READ ndx.rec FROM accbals.ndx, ndxid ELSE
    CRT 'This account was not open on this date'
    CONTINUE
  END

  LOCATE idate IN ndx.rec<1> BY 'DR' SETTING datepos ELSE NULL
  baldate = ndx.rec<1, datepos>
  IF baldate THEN
    accdate = account:'*':baldate
    READV balance FROM accbals, accdate, 1 THEN
      CRT 'Accont balance was: ':OCONV(balance, 'MR2,$'
    END ELSE
      CRT 'Balance not found'
    END
  END ELSE
    CRT 'Index file is corrupt'
  END
REPEAT
```

This requires two reads to determine the account balance – regardless of the date requested. The first read gets the index item, while the second read gets the account balance.

Let's run through the code:

- We enter a date – let’s say it is 14 December, 2009. This is internal date 15324. In practice, we would need to validate this date to ensure it is not in the future
- We enter an account – let’s say it is: 123456. In practice, this would need some further validation
- We generate an ID for the index file. This is: 123456*200912
- We read the index item. This is as shown on the previous page
- We `LOCATE` the entered date in the index item. This returns position 1 because this is the position that we would need to insert the entered date (15324) to maintain the sequence in descending order
- We generate the ID for the `ACCBALS` file based on the account number and the date in the first position in the index file. This would be: 123456*15317
- We read and display the balance from the `ACCBALS` file.

Let’s say the date entered was the 2nd of December. This has an internal value of 15312.

If we `LOCATE` this date in our index item, the position returned is 3. The date at position 3 is 15310 which is November 30 – the previous month-end value.

Note that in this usage of `LOCATE`, it doesn’t actually matter whether we find the value in the list or not. Therefore, we simply set the `ELSE` action to `NULL`.

This banking application example is a typical usage of `LOCATE`. Of course, this only shows one part of the usage within the application – there will be an equivalent usage of `LOCATE` during the maintenance of the index item so that the index item is always in sorted order.

FIND

`FIND` is closely related to `LOCATE.` It finds a data element within a dynamic array. However, it does so in a different manner than `LOCATE`:

- `LOCATE` searches within one dimension of a dynamic array, and returns a position for the search string whether or not it is found. `THEN` and/or `ELSE` statements are executed depending on whether the string was found.
- `FIND` searches in one, two, or three dimensions – depending on the number of variables specified – and returns the position of the string in that number of dimensions. `THEN` and/or `ELSE` statements are executed depending on whether the string is found.

To make things a little clearer: `FIND` returns the position of the string in 1, 2, or 3 dimensions. In contrast, `LOCATE` only ever returns the position of the string within a single dimension of the dynamic array.

The syntax for `FIND` is:

```
FIND string IN dyn.array {,occurrence) SETTING field {,value
{,subvalue}} {THEN statements } {ELSE statements}
```

As with `LOCATE`, the search string should make up the entire field, value, or subvalue. Unlike `LOCATE`, the values returned do not represent the position in the dynamic array where the string should be inserted – they only represent the string’s actual position.

Warning: If the string is not found, then the values of field, value, and subvalue are not changed. Therefore, you cannot assume that the presence of data in these variables indicates that the string has been found – unless the variables were empty before the `FIND` statement was executed.

3.5 Program control structures

QMBasic supports variants on the common program control structures. In many cases, the QMBasic variants are more flexible than those found in languages such as *Visual Basic*, or *C/C++*.

IF-THEN-ELSE

OpenQM supports both single and multi-line versions of IF statements:

```
IF condition {THEN statement} {ELSE statement}

IF condition {THEN
  statements
END} {ELSE
  statements
END}
```

In either construct, at least one of the THEN or ELSE branches must exist. This means that you can write statements such as:

```
IF ok ELSE CONTINUE
```

While this is a perfectly valid statement, it is difficult to read. Most people logically expect a THEN condition as the primary branch of an IF statement, and would find the following statement easier to read:

```
IF NOT(ok) THEN CONTINUE
```

Multi-line IF statements must have an END terminating each block of conditional statements:

```
IF condition THEN
  statements
END
```

The conditions shown in the examples use Boolean values. *OpenQM* takes any non-null, non-zero value to be true, and null or zero to be false. Therefore, if the variable *ok* shown in the example above contained the value 'Y', then this would be true[14], whereas if it contained an empty string, then this would be false. **Warning:** If the variable contained 'N', then this would also evaluate to true.

Boolean tests are good for evaluating the presence or absence of data. They should not be used for testing between two different data values. In that case, use a construct like:

```
IF UPCASE(answer) = 'Y' THEN
  statements
END ELSE
  statements
END
```

Alternatively, you could convert the original data to a Boolean value:

```
ok = (UPCASE(answer) EQ 'Y')
IF ok THEN ...
```

CASE

CASE statements provide an alternative method of branching for conditional execution. While IF-THEN-ELSE statements provide a two-way branch (or more if conditions are nested), CASE statements allow multi-way branching. The basic format of the statement is:

```
BEGIN CASE
  CASE condition-1
```

14 Note that some multi-value databases require Boolean values to be strictly numeric or null. Therefore, a value of 'Y' will result in a non-numeric value error, with zero assumed – i.e. false.

```
    statements
{  CASE condition-2
    statements}
{ CASE condition-n
    statements }
{ CASE 1
    statements }
END CASE
```

The statement begins with the `BEGIN CASE` declaration, and ends with the `END CASE` declaration. In between these two declarations, there are as many `CASE` conditions as required.

In operation, code execution in a `CASE` statement always takes the first condition that evaluates to `TRUE`. Once that conditional block of statements has been executed, all the remaining cases are skipped and execution continues with the statement immediately following the `END CASE` statement.

Optionally, you can include a `CASE 1` condition. This condition always evaluates as `TRUE` and therefore acts as a default execution branch (or `ELSE` clause).

Note the following points about `CASE` statements:

- The conditions do not need to be related to each other – although they often are. This is in contrast to the `SELECT CASE` statements in some other languages that can only branch on the value of a single variable
- If no condition is matched, then execution will continue with the first statement following `END CASE`

Consider the following code fragment:

```
cntcrlf = COUNT(descdata, CR:LF)
cntcr = COUNT(descdata, CR)
cntlf = COUNT(descdata, LF)

BEGIN CASE
  CASE cntcrlf EQ cntcr AND cntcrlf EQ cntlf;  fdelim = CR:LF
  CASE cntcr GT cntlf;                         fdelim = LF
  CASE cntlf GT cntcr;                         fdelim = CR
  CASE 1;                                      fdelim = ''
END CASE
```

This code counts the number of carriage returns and line feeds in the variable ***`descdata`***, then assigns the variable ***`fdelim`*** on the basis of these counts. Although a `CASE 1` condition is present, this should be impossible to reach.

Loops

Conditional loops

The basic structure of a conditional loop is:

```
LOOP
  {statements}
{WHILE | UNTIL condition {DO}}
  {statements}
REPEAT
```

The `DO` keyword is not required in *OpenQM*, but is required by other multi-value databases, and so is shown here for compatibility.

The above structure shows two blocks of optional statements. The first set of optional statements will always be executed within the loop. However, the second set will only be executed when the condition statement allows.

A typical usage of this structure is in the sequential processing of records in a file:

```
EQUATE CUST.SURNAME   TO 4
EQUATE CUST.FIRSTNAME TO 5
OPEN 'CUSTOMERS' TO customers ELSE STOP 201,'Customers'

eof = @FALSE
SELECT customers
LOOP
  READNEXT custid ELSE eof = @TRUE
UNTIL eof DO
  READ custrec FROM customers,custid THEN
    CRT custrec<CUST.FIRSTNAME>:' ':custrec<CUST.SURNAME>
  END
REPEAT
```

The assignment of the *CUST.FIRSTNAME* and *CUST.SURNAME* references, and the opening of the CUSTOMERS file are shown for completeness.

This program fragment works as follows:

- The SELECT statement selects the customers file and returns a select list for processing
- The READNEXT statement reads the next customer id from the select list. If there are no more items in the select list then the *eof* variable is set to @TRUE
- The UNTIL clause tests the value of the *eof* variable. If *eof* is @FALSE, then processing moves to the next part of the loop. If *eof* is @TRUE, then loop processing ends, and processing continues with the statement following REPEAT
- The inner part of the loop reads the customer record, and displays the customer's firstname and surname
- When the REPEAT statement is encountered, processing returns to the LOOP statement, and the next customer id is read from the select list.

Loops can also be written with only one internal block of statements:

```
ii = 0
LOOP
  ii += 1
  more statements
UNTIL ii GE maxii DO REPEAT

ok = @TRUE
LOOP WHILE ok DO
  more statements
REPEAT

ok = @TRUE
LOOP
  more statements
WHILE ok DO REPEAT
```

Note that the WHILE/UNTIL conditions may occur at any point during the loop – unlike some other languages that require WHILE conditions to be placed at the top of the loop, and UNTIL conditions at the end. Similarly, you may have more than one WHILE/UNTIL conditions within a loop – although such structures may be difficult for others to read.

They can also be written without any WHILE/UNTIL conditions:

```
SELECT customers
LOOP
  READNEXT custid ELSE EXIT
  READ custrec FROM customers,custid THEN
    CRT custrec<CUST.FIRSTNAME>:' ':custrec<CUST.SURNAME>
  END
REPEAT
```

In this case, the `EXIT` statement is used to terminate the loop when the list of customer ids has been exhausted. More information on the `EXIT` statement will be given shortly.

For-Next loops

If you know the number of loops that you require, you can use a `FOR-NEXT` loop rather than testing a condition on every loop. The basic structure of `FOR-NEXT` loops in *OpenQM* is similar to that in other languages, but does have some variations:

```
FOR var = start TO end { STEP stepsize }
  statements
NEXT var
```

If `STEP` is omitted, then 'stepsize' is assumed to be 1.

For example:

```
IF st GT '' THEN
  dc = DCOUNT(st, @AM)
  FOR ii = 1 TO dc
    CRT msgs<ii>
    EXECUTE st<ii> CAPTURING junk
  NEXT ii
END
```

In this example, the variable *`st`* may contain a series of executable statements. First, the variable is tested to see if it contains any data. If so, the statements are counted, and a loop started which executes each statement and displays a message associated with each statement.

OpenQM also allows `WHILE` and `UNTIL` statements to be included in the `FOR-NEXT` loop. Consider the following program:

```
PROGRAM TEST
kk = 0
FOR ii = 1 TO 4
  FOR jj = 1 TO 6
    kk = ii * jj
    WHILE kk LE 12
      CRT kk 'R#6':
  NEXT jj
  CRT
NEXT ii
STOP
END
```

The output from this program is:

```
RUN BP TEST
     1     2     3     4     5     6
     2     4     6     8    10    12
     3     6     9    12
     4     8    12
```

In this case, the inclusion of the `WHILE` clause has made the display of the number conditional on its value.

Recent versions of *OpenQM* have two new syntax options for the `FOR-NEXT` loop:

```
FOR var = value1 {,value2 ...}
  statements
NEXT var
```

and:

```
FOR EACH var IN string
  statements
NEXT var
```

The first of these two new variants allows you to specify a list of values to use in the FOR-NEXT loop. The second allows you to use the values contained in a dynamic array within the loop. See the online help for more information on these two variants.

Some programs may be written to use the final value of the index variable after the loop has terminated. For example:

```
PROGRAM TEST
FOR ii = 1 TO 5
  CRT ii
NEXT ii
CRT 'Final value is ':ii
END
```

This produces the output:

```
RUN BP TEST
1
2
3
4
Final value is 4
```

Some other multi-value databases may display 5 as the final value. *OpenQM* can mimic this behaviour by using the compiler mode FOR.STORE.BEFORE.TEST. This could be included directly in the program or stored in a $BASIC.OPTIONS item either in the program file or in the account VOC:

```
PROGRAM TEST
$MODE FOR.STORE.BEFORE.TEST
FOR ii = 1 TO 5
  CRT ii
NEXT ii
CRT 'Final value is ':ii
END
```

```
RUN BP TEST
1
2
3
4
Final value is 5
```

While it is possible to make OpenQM emulate this behaviour, you should really question whether this is sensible. The index value of a FOR-NEXT loop only has meaning within the loop. You should not rely on its value after loop termination. It isn't difficult to write your code in such a manner that you don't have to worry about this difference in behaviour between multi-value systems.

EXIT and CONTINUE

EXIT and CONTINUE statements may be used to modify the behaviour of both conditional loops and FOR-NEXT loops.

- EXIT causes the loop to terminate.
- CONTINUE skips the remaining statements in this loop and starts a new loop

Example usage of both EXIT and CONTINUE is shown below:

```
SELECT customers
LOOP
  READNEXT custid ELSE EXIT
  READ custrec FROM customers,custid THEN
    IF custrec<CU.ACTIVE> NE 'Y' THEN CONTINUE
    GOSUB processcust
  END
REPEAT
```

In this loop, if there are no more customer id's to process, then the EXIT statement will cause the loop to terminate. Valid customers are checked to see if they active. If they are not active, then the CONTINUE statement causes processing to skip the processing of the customer record and jump to the REPEAT statement.

Subroutines

Subroutines are a means of breaking your program into small blocks that:

- allow you to structure to your program
- encourage re-use of code sections.

The key concepts of a subroutine are:

- it is a block of code that carries out a specific action or set of actions
- it is called from elsewhere in the program via the GOSUB statement for internal subroutines or the CALL statement for an external subroutine
- once processing of the subroutine is complete, control returns to the statement immediately following the calling statement.

Subroutines may call other subroutines, which in turn may call other subroutines. There are limits to the depth of such nesting (the default setting is 1,000 subroutine calls), but you are unlikely to reach them unless an error of program logic causes recursive or circular subroutine calls.

Program structure

An example of structured programming is shown in the loop above. What we see is a relatively small block of code, the purpose of which is readily apparent.

In contrast, consider what would happen if we included the customer processing code inside the loop. Say the code for the customer processing code was 200 lines long. In that case, we would not be able to quickly see the start and end points of the loop, and we would probably lose track of flow of logic.

Well structured programs usually have logic flows that look like:

```
GOSUB initialise
GOSUB getuserresponses
GOSUB dojob
GOSUB finalise
GOSUB shutdown
```

or:

```
GOSUB initialise
LOOP
  GOSUB getuserresponses
UNTIL quit DO
  GOSUB dojob
REPEAT
GOSUB finalise
GOSUB shutdown
```

Likewise, each of the major functional blocks referenced in the GOSUB statements above will in turn be broken down into small blocks.

Code re-use

As you write your programs, you will find some parts of the program are used many times. Subroutines allow you to put this code into one place, and then call it from many places within the program. For example, the following fragment is used to set screen colours and repeatedly calls a subroutine to convert a colour name to a colour number:

```
decodethreeparams:
```

```
*
emsg = ''
thiscolour = FIELD(ss, ' ', 2);       *  Get background colour
GOSUB getcolournum;                   *  Convert to colour number
bgc = colournum;                      *  Store

thiscolour = FIELD(ss, ' ', 3);       *  Get foreground colour
GOSUB getcolournum
fgc = colournum

IF fgc NE bgc THEN;                   *  bgc and fgc not same
  CALL SY.SET.COLOUR(bgc, fgc, emsg); *  Set colours
END ELSE
  emsg = 'Error - Foreground colour same as background colour'
END

IF emsg GT '' THEN;                   *  Errors encountered
  CRT emsg;                           *  Display error message
  GOSUB showusage
  STOP
END
*
RETURN
```

The above code fragment calls the internal subroutine *getcolournum* twice, and the external subroutine *SY.SET.COLOUR* once. Both of these subroutines are also called from elsewhere in the program, and the *getcolournum* subroutine itself calls another external subroutine:

```
getcolournum:
*
CALL SY.GET.AT.COLOUR.NUM(thiscolour, colournum, emsg)

IF emsg GT '' THEN;          *  Colour not found - error message
  CRT emsg
  GOSUB showusage
  INPUT pause,1
  STOP
END
*
RETURN
```

Internal subroutines

Internal subroutines are defined within the main body of the program. They start with a label that identifies the subroutine, and end with a RETURN statement. Labels are normally[15] an alphanumeric string followed by a colon (:), or a number (which may optionally be followed by a colon). For example:

```
initialise:
  statements
RETURN
```

or:

```
1000
  statements
RETURN
```

You can put a comment on the same line to identify the purpose of the subroutine if it is not immediately apparent:

```
1000 ;* Initialise variables
  statements
RETURN
```

The above subroutines would be called with the following statements:

```
GOSUB initialise
```

15 A third label format is also available. See the online help for more information.

```
GOSUB 1000
```

External subroutines

External subroutines are stored outside the main program – they are program modules in their own right. This means that external subroutines may be called by any program, whereas an internal subroutine can only be called from within its parent program.

External subroutines have another distinguishing characteristic – they can be defined to take a list of parameters that define their action. In contrast, internal subroutines make use of the same set of variables that are used by the parent program.

An external subroutine begins with the `SUBROUTINE` declaration, and ends with a `RETURN` statement.

```
SUBROUTINE subname{(parameter {,parameter ...})}
  statements
RETURN
```

For example:

```
SUBROUTINE SY.GET.SETTING(ctrldata, identifier, settings, found)
* -------------------------------------------------------------- *
*
*  Copyright 2008 Rush Flat Software
*
* Version: 1.0.0
* Author : BSS
* Created: 20 Mar 2006
* Updated: 20 Mar 2006
*
* Subroutine to search the passed control data for a particular
* identifier. Subroutine passes back the settings and found
* variables.
*
* Assumes that the control data is in the form:
*
*   identifier1=settings1
*   identifier2=settings2
*   etc
*
* -------------------------------------------------------------- *
*
$CATALOGUE GLOBAL

progname = 'SY.GET.SETTING'

upctrldata = OCONV(ctrldata, 'MCU')
upidentifier = OCONV(identifier, 'MCU')
numctrllines = DCOUNT(ctrldata, @AM)

found = @FALSE
settings = ''
IF numctrllines GT 0 THEN
  ii = 0
  LOOP
    ii += 1
  UNTIL ii GT numctrllines OR found DO
    thisline = ctrldata<ii>
    upthisline = OCONV(thisline, 'MCU')
    temp = TRIM(FIELD(upthisline, '=', 1))
    IF temp = upidentifier THEN
      settings = TRIM(FIELD(thisline, '=', 2))
      found = @TRUE
    END
  REPEAT
END
*
*
RETURN
```

```
*
* ------------------------------------------------------------- *
*
END
```

And this subroutine would be called as follows:

```
identifier = 'colours'
CALL SY.GET.SETTING(sysctrl.userdata, identifier, colours, found)
colours = OCONV(colours, 'MCU')
```

So, if the control data that is passed to the subroutine looks like:

```
Colours=darkblue,yellow
NormCols=132
NormRows=35
ExtCols=160
ExtRows=40
ScrMode=Normal
```

then, the above call would return the string ‘darkblue,yellow’ in the variable *colours*, and @TRUE in the variable ‘found’.

Let’s go through the subroutine and call in a bit more detail:

- The subroutine was declared with the SUBROUTINE statement, and the declaration contained four variables to be passed to or from the subroutine.
- The subroutine contained a short description of what it does, followed by the actual code. This description could have been more explicit in the values that need to be passed to the subroutine, and those that will be returned.
- This code ended with a RETURN statement.
- The call to the subroutine passed four variables as part of the call. Note that the names of these variables do not have to match the names declared in the subroutine code. Inside the subroutine, the passed variables take on the names declared in the subroutine heading. Therefore, the variable *sysctrl.userdata* that is passed to the subroutine is referred to as *ctrldata* inside the subroutine.
- While the description implies that you should pass *ctrldata* and *identifier*, and the subroutine will pass back *settings* and *found*, this is a human interpretation of what happens. The subroutine itself makes no distinction between the variables in the declaration. *If a variable declared in the subroutine header is changed in the subroutine, it will be passed back in its changed state.* Therefore, if the subroutine changed the variable *identifer* to the literal value ‘junk’, then the value ‘junk’ would be available to the calling program when control returns there.

This last point is quite important. You need to be clear about the way subroutines change the variables passed to them. There are various strategies for ensuring that some variables are not changed by the subroutine:

- Pass a copy of the variable to the subroutine:

  ```
  temp = sysctrl.userdata
  CALL SY.GET.SETTING(temp, identifier, colours, found)
  ```

- Pass the variable by value. This ensures that only the value of the variable is passed to the subroutine – not the variable itself. To pass the variable by value, enclose the variable in parentheses in the the call:

  ```
  CALL SY.GET.SETTING((sysctrl.userdata),identifier,colours,found)
  ```

- Define the subroutine parameters as being passed by value. This ensures that the variable passed to the subroutine will not be changed. To declare the variable as being passed by value, enclose the variable in parentheses in the declaration:

```
SUBROUTINE SY.GET.SETTING((ctrldata),identifier,settings,found)
```

Which approach you use depends on what you expect the subroutine to do. Often, the point of the subroutine is to change the value of the variable, in which case you don't need the strategies outlined above.

Local subroutines

OpenQM has a third type of subroutine which has features of both internal and external subroutines:

- The subroutine is defined within the main body of the program
- The subroutine can be defined to take parameters

These subroutines are defined using the `LOCAL` keyword, and have an `END` statement after the `RETURN`:

```
LOCAL SUBROUTINE subname{(parameter {,parameter …})}
    statements
  RETURN
END
```

Essentially, the `RETURN` statement terminates the `SUBROUTINE` declaration, while the `END` is required to terminate the `LOCAL` declaration.

Local subroutines may also employ local variables – but these must be explicitly declared using the `PRIVATE` keyword. Otherwise, variables in local subroutines are global in scope.

Local subroutines are called using the `GOSUB` statement.

```
PROGRAM TEST
dt = ''
st = '28/2/10'
GOSUB DATATYPE(st, dt)
CRT 'Datatype of ':st:' is ':dt

st = '123.45'
GOSUB DATATYPE(st, dt)
CRT 'Datatype of ':st:' is ':dt
CRT temp
STOP

LOCAL SUBROUTINE DATATYPE(datastring, datatype)
  PRIVATE temp, datetest
    temp = datastring
    CONVERT ',' TO '' IN temp
    datetest = OCONV(ICONV(temp, 'D'), 'D2/')
    IF datetest NE '' AND LEN(temp) < 6 THEN datetest = ''
    BEGIN CASE
      CASE temp = '';                 *  NULL is Text
        datatype = 'T'
      CASE INDEX(temp,' ',1);         *  At least one space
        datatype = 'T'
      CASE OCONV(temp,'MCA') NE '';   *  Alpha is not null
        datatype = 'T'
      CASE NUM(temp);                 *  Is numeric
        datatype = 'N'
      CASE datetest NE '';            *  OCONV DATE is not null
        datatype = 'D'
      CASE 1;                         *  Anything else = text
        datatype = 'T'
    END CASE
  RETURN
END
END
```

```
BASIC BP TEST
Compiling BP TEST
***
WARNING: TEMP is not assigned a value
0 error(s)
Compiled 1 program(s) with no errors
```

```
RUN BP TEST
Datatype of 28/2/10 is D
Datatype of 123.45 is N
00000109: Unassigned variable TEMP at line 10 of
D:\QM\QMINTRO\BP.OUT\TEST
```

The above example shows how to define and call a local subroutine. It also shows how the `PRIVATE` variables contained within the local subroutine are not available to the main program.

User Defined Functions

User defined functions are broadly similar to external subroutines (and like subroutines, can also be defined as `LOCAL)`. However, unlike subroutines, they use a general assignment syntax rather than a call:

```
result = MYFUNCTION(arg-list)
```

Like external subroutines, functions usually occupy their own operating system file, and have the general form:

```
FUNCTION functionname{(parameter {,parameter …}) {VAR.ARGS}}
  statements
RETURN varname
```

For example:

```
FUNCTION SY.EXCELDATE(datestring)
*
* Version: 1.0.0
* Author : BSS
* Created: 08 Mar 2007
* Updated: 08 Mar 2007
*
*  Copyright 2008 Rush Flat Software
*
* ----------------------------------------------------------------
*
*
$CATALOGUE GLOBAL

  internaldate = ICONV(datestring, 'D')
  xldate = internaldate + 24837
RETURN xldate
*
END
```

This function takes a passed date in external format (e.g. 25 Apr 2009), and returns this as an Excel date number.

To use the function in a program, we must define it so that the compiler knows this is a valid function and can validate the number of arguments passed at compile time. We use the `DEFFUN` keyword to define the function in a program:

```
PROGRAM TEST
DEFFUN SY.EXCELDATE(datestring)
testdate = '25 Apr 2009'
CRT SY.EXCELDATE(testdate)
END
```

And the output is:

```
RUN BP TEST
39928
```

Note that we have been able to call the function directly in the CRT statement, although a more normal usage of this function may have been:

```
xdate = SY.EXCELDATE(testdate)
```

As noted above, *OpenQM* allows functions to be defined locally within a program through use of the LOCAL statement. See the subroutines section above for more information on this, or lookup LOCAL in the online help.

The other twist provided by *OpenQM* is to allow a variable number of arguments through use of the VAR.ARGS keyword in the function definition:

```
FUNCTION FNTEST(arg1, arg2, arg3, arg4) VAR.ARGS
  x = ARG.COUNT()
RETURN x
END
```

This function has been defined to accept up to 4 arguments. However, all it does is return the count of arguments passed to it. The following program shows this function in use:

```
PROGRAM TEST
DEFFUN FNTEST(arg1, arg2, arg3, arg4) VAR.ARGS
numargs = FNTEST('ABC')
CRT 'Number of arguments = ':numargs
END
```

```
RUN BP TEST
Number of arguments = 1
```

Note that if you wish to use functions with a variable number of arguments, the VAR.ARGS keyword should be included in both the function definition and the DEFFUN declaration – although if you test this, you can omit it from the function definition, but it must be included in the DEFFUN declaration.

As with subroutines, variables are passed to the function by reference. This means that changes to the variables in the function will be carried back to the calling program. Note this is different from many other computer languages which pass arguments to functions by value. You can use any of the strategies outlined in the section on subroutines to pass the arguments as values.

3.6 Files

OpenQM applications use files extensively. Therefore, it is vital to understand how to use files in *OpenQM*.

In general, the processes involved in dealing with files are:

- opening the files
- selecting records in the files
- reading the records from the files
- writing records to the files
- closing files.

There are several related issues to consider also:

- error handling
- record locking
- handling special file types

The following sections will give a brief coverage of these issues.

Most file handling statements have an {ON ERROR statements} clause within them. These statements are executed when a serious error condition is encountered in the file structure. As this clause is common to most statements, it will be omitted in the following descriptions.

Likewise, most file handling statements have optional THEN and ELSE clauses. While these are noted as being optional, in reality, they must have at least one of these clauses present.

In all cases, see the online help for more information.

3.6.1 Opening files

Files must be opened before they are available within an *OpenQM* program. Opening the file associates the file's operating system filename with a variable within the program:

```
OPEN filename TO filevar {THEN statements } { ELSE statements }
```

The filename may contain a reference to a dictionary so that you can open the dictionary itself, or the data portion of a multifile:

```
OPEN 'CUSTOMERS' TO customers ELSE GOTO fileopenerror
```

```
OPEN 'DICT','SALES' TO sales.dict ELSE GOTO fileopenerror
```

```
OPEN 'SALES','FY2009' TO sales ELSE GOTO fileopenerror
```

The dissociation of the database filename with the internal file variable means that you could use the same file variable for multiple files (one at a time of course). In the multifile example given above, we may have the year (say 2008) we wish to open in the variable *fyear*. Therefore:

```
fname = 'FY':fyear
OPEN 'SALES',fname TO sales ELSE GOTO fileopenerror
```

This is a particularly valuable technique for two reasons:

- It allows you to write generalised software that operates on a number of files. The physical file reference that you provide to the software is converted to a file variable for the actual operations.
- The error handling can be generalised. Once again, pass the error handler the name of the file in a variable, and it can output an appropriate message.

The file variable is just another variable that should conform to *OpenQM* naming rules. However, as file variables have specific roles within *OpenQM*, you should try to be consistent with your naming of file variables. Some strategies are:

- Use the file name as the name of the file variable:

  ```
  OPEN 'STAFF' TO staff ELSE ...
  ```

- Use a file variable that indicates purpose and source:

  ```
  OPEN 'CUSTOMERS' TO cust.file ELSE ...
  OPEN 'DICT','REPORTS' TO reports.dict ELSE ...
  ```

  - A common variant on this is to use '.f' as a suffix (or 'f.' as a prefix) to denote a file variable:

    ```
    OPEN 'CUSTOMERS' TO customers.f ELSE ...
    ```

Consistency of naming file variables will aid subsequent programming enormously, as you don't have to continually check what name was given to each file.

The ON ERROR clause will only get executed if severe errors are encountered when opening the file. A THEN clause will be executed if the file is opened successfully, while an ELSE clause will be executed if the file cannot be opened. At least one of the THEN or ELSE clauses must be present.

Error handling

A common form of error handling on OPEN statements is simply to stop the program with an error message. This is fine in simple applications, but is not appropriate in larger applications where the OPEN error may occur deep in the application.

A typical example of this type of error handling is:

```
OPEN 'SALES' TO sales ELSE STOP 201, 'Sales'
```

201 refers to the error message number in the ERRMSG file. If the OPEN statement fails, then this produces the following error message:

```
[201] 'Sales' IS NOT A FILE NAME
```

Importantly, the STOP statement actually stops the program, so this is a drastic form of error handling. Ideally, we want to open files in a way that captures errors and gives a chance to handle them appropriately. One way to do this is to wrap the standard OPEN statement in a custom subroutine (or function) that allows you set the desired action when you call the subroutine, and it passes back appropriate messages:

```
SUBROUTINE FILE.OPEN(filename, fileptr, errorlevel, etext)
***********************************************************************
* Bp File.Open - A subroutine to open files in a standardised manner.
*
*  Copyright 2008 Rush Flat Software
*
* Author : BSS
* Created: 09 Aug 2008
* Updated: 06 Aug 2009
* Version: 1.0.1
*
* Pass   : filename
* Return : fileptr, errorlevel, etext
*
* Errorlevel: 0 - No errors
*             1 - Errors encountered
*             2 - Severe error encountered
*
* -------------------------------------------------------------------- *
*
$CATALOGUE GLOBAL

errorlevel = 0
etext = ''
fileopened = @FALSE

CONVERT ' ' TO '' IN filename
ofilename = filename

dictname = ''
IF INDEX(filename, ',', 1) THEN
  dictname = FIELD(filename, ',', 1)
  filename = FIELD(filename, ',', 2)
END

fileptr = ''
BEGIN CASE
  CASE (dictname EQ '') OR (dictname = filename)
    GOSUB opendata
  CASE dictname EQ 'DICT'
    GOSUB opendict
  CASE 1
    filename = dictname:',':filename
    GOSUB opendata
```

```
END CASE

RETURN
STOP
*
* ------------------------------------------------------------------- *
*
opendata:
*
OPEN filename TO fileptr ON ERROR
  GOSUB openerror
  errorlevel = 2
END THEN
  fileopened = @TRUE
END ELSE
  GOSUB openerror
END

RETURN
*
* ------------------------------------------------------------------- *
*
opendict:
*
OPEN 'DICT',filename TO fileptr ON ERROR
  GOSUB openerror
  errorlevel = 2
END THEN
  fileopened = @TRUE
END ELSE
  GOSUB openerror
END

RETURN
*
* ------------------------------------------------------------------- *
*
openerror:
*
errorlevel = 1
errcode = STATUS()
CALL !ERRTEXT(etext, errcode)
etext = 'Err ':errcode:': Error opening file: ':ofilename

RETURN
*
* ------------------------------------------------------------------- *
*
END
```

And a “wrapping” subroutine for use with the AccuTerm GUI:

```
SUBROUTINE GUI.FILE.OPEN(filename, fileptr, options, errorlevel,
guierrors, guistate)
***********************************************************************
* Bp Gui.File.Open - Subroutine to open files from AT GUI.
*
*  Copyright 2008 Rush Flat Software
*
* Author : BSS
* Created: 09 Aug 2008
* Updated: 06 Aug 2009
* Version: 1.0.1
*
* Pass   : filename, options
* Return : fileptr, errorlevel
*          guierrors, guistate
*
* Options: 0 - Stop processing and call stop message
*          1 - Display error message and return to calling program
*          2 - Pass control back to calling program without message
*
```

```
* Errorlevel: 0 - No errors
*             1 - Errors encountered
*             2 - Severe error encountered - stop unconditionally
*
* ------------------------------------------------------------------- *
*
$CATALOGUE GLOBAL
$INCLUDE GUIBP ATGUIEQUATES

errorlevel = 0
errcode = ''
etext = ''
errortext = ''

CALL FILE.OPEN(filename, fileptr, errorlevel, etext)

IF errorlevel THEN
  ok = ''
  BEGIN CASE
    CASE errorlevel GT 1;  *  Unconditional stop
      CALL ATGUIMSGBOX(etext, 'File open error', MBXICON, MBOK, '', ok,
guierrors, guistate)
      IF guierrors<1> GE 2 THEN CRT 'Errors encountered during file
open: ':filename
      CALL ATGUISHUTDOWN
      STOP

    CASE options GT 1;     *  No error message (but pass back err level)
      NULL

    CASE options EQ 1;     *  Display warning message
      CALL ATGUIMSGBOX(etext, 'File open error', MBIICON, MBOK, '', ok,
guierrors, guistate)
      IF guierrors<1> GE 2 THEN CRT 'Errors encountered during file
open'

    CASE 1;                *  No errors tolerated - stop
      CALL ATGUIMSGBOX(etext, 'File open error', MBXICON, MBOK, '', ok,
guierrors, guistate)
      IF guierrors<1> GE 2 THEN CRT 'Errors encountered during file
open: ':filename
      CALL ATGUISHUTDOWN
      STOP
  END CASE
END

RETURN
STOP
*
* ------------------------------------------------------------------- *
*
END
```

The above subroutines allow you to specify how to respond when an error occurs. You can tell it to unconditionally stop (*`options = 0`*); to display a warning message, but return control to the calling program (*`options = 1`*); or to simply pass control back to the calling program so you can handle the error there (*`options = 2`*).

If the `ON ERROR` clause is executed, then this subroutine will always stop further processing.

Once again, the option to stop processing is fairly harsh. This calls the `ATGUISHUTDOWN` subroutine which will stop all AccuTerm GUI subroutines – not just the one currently being processed. Think carefully about the error handling that will be necessary in your application.

The *`FILE.OPEN`* subroutine gets the error description using the `!ERRTEXT` standard subroutine that is supplied with *OpenQM*. This subroutine converts error numbers (supplied by the `STATUS()` function) to descriptive text.

The *GUI.FILE.OPEN* subroutine is specifically for use with the *AccuTerm* GUI. This subroutine “wraps” the *FILE.OPEN* subroutine, and adds message box functionality (using ATGUIMSGBOX) to respond to errors.

You can use these subroutines to test for the existence of a file, and create it if necessary:

```
CALL GUI.FILE.OPEN('SALES', sales, 2, err, guierrors, guistate)
IF err THEN
  EXECUTE \CREATE.FILE SALES\ CAPTURING junk
  CALL GUI.FILE.OPEN('SALES', sales, 0, err, guierrors, guistate)
END
```

This code fragment attempts to open the SALES file. If a major error is encountered, then the subroutines will display a message and stop the program. If the file doesn’t exist, then (because our calling variable *options* was set to 2) control simply returns to back to the calling program with the *err* variable set to 1. The calling code then creates the file tries again to open the file. If it fails again, then the subroutines display an error message and shut the program down (because they were called with *options* set to 0).

The above subroutines are a good example of the way that core functionality can be placed into one subroutine and then enhanced in a second subroutine (by adding the message boxes). Importantly, the core subroutine does not interact with the interface. This means that if you ever move your application to another interface (such as the web), then this core subroutine can be used without changes. This separation of functionality and interface will make porting your application to alternative interfaces much easier.

3.6.2 Selecting data in files

Selecting data from a file is generally understood to mean selecting a set of the available records from a given file. In the multi-value world, it has the added implication that the ID’s of the selected records will be available in a list (a select-list).

There are two basic ways of creating a select-list – using an internal select, and using an external select. An internal select is carried out within the BASIC program, while an external select is executed outside the program.

Once the program has a select-list, it will usually loop through the ID’s in the list, and process the associated records.

Internal select

An internal select has the syntax:

    SELECT *var* {TO list-num}

The behaviour of this statement can be altered via a $MODE compiler directive to select to a list variable[16] rather than a list number. *OpenQM* also has variants of the SELECT statement that always select to a list number (SELECTN) or a list variable (SELECTV).

At this stage, the option to select to a list number or list variable will be ignored. This only needs to occur if multiple select lists could be concurrently active. However, it is good practice to always select to a list number/variable for two reasons:

- multiple select lists can be more easily handled when they occur; and
- to ensure that no active select lists are left behind by your programs.

This second point needs a bit more explanation. Consider the case where a program generates a select-list, and does not associate it with a list number or variable. This means

16 The choice of whether to use list numbers or list variables is usually made on the basis on the developers background. If they have come from a PICK background, they will usually choose list variables, while those from an Information background will choose list numbers. They are functionally similar but have slight differences in usage.

that it will be the default select list. The program then begins processing the list, but stops before all items in the list are exhausted. The remaining items in the list will remain active, and will be processed by any subsequent `READNEXT` command. Further, the list can even continue to exist after the program terminates, and will be processed by any subsequent `QMQuery` commands or programs.

In short, a default select-list that that has not been exhausted can cause programs or `QMQuery` commands to behave in an unexpected fashion. To avoid the problems described above, always select to a list number or variable.

The *`var`* that is selected by `SELECT` may be either the file variable of an open file, or a variable containing a field-mark delimited list of record ID's. In either case, an internal select simply selects all records referenced by *`var`* into a select-list. The statement has no ability to select a subset of the records, nor to sort them into any order. The order of the record ID's in the list simply reflect the order of the records in the file or variable.

OpenQM has another variant of the `SELECT` statement which does a simple sort of the record ID's (`SSELECT`). However, as this sort is limited to a left-justified ascending sort order, this is of limited use.

The advantage of an internal select is that it is fast.

External select

An external select uses the `QMQuery` selection commands to select some or all of the record ID's in a given file. The command may also sort the records into a specific order.

`QMQuery` commands were covered in Part 1 of *Getting Started in OpenQM*. This document will only cover the means by which these commands are used within `QMBasic`.

The issue for `QMBasic` is how to run a `QMQuery` command (or any other command that is normally run from the command line). This is done using the `EXECUTE` or `PERFORM` statements. `EXECUTE` has a number of optional clauses which are not covered here – see the online help for more information.

```
EXECUTE command {CAPTURING display}
```

For example:

```
EXECUTE \SSELECT IRATES WITH YEAR EQ "2006"\ CAPTURING junk
```

This would return a select-list of record ID's in the `IRATES` file where the year was 2006. It uses the `SSELECT` command without specifying a sort order, so the record ID's would be sorted into their ID order – which in this case is ascending month order within the year.

If we run this command from the command prompt, we get:

```
:SSELECT IRATES WITH YEAR EQ "2006"
12 record(s) selected to list 0
::
```

The '`CAPTURING junk`' part of the `EXECUTE` statement captures the message that is reported by `QMQuery`. This stops the message from being displayed, and upsetting any screen formatting that you have.

Note that the command that is executed must be passed as either a quoted string, or as a variable. In the above example, the backslash character (\) has been used to quote the string. The backslash is useful for this purpose as it allows both single and double quotes to be used within the `QMQuery` command.

The same command could be executed using a variable as follows:

```
cmd = \SSELECT IRATES WITH YEAR EQ "2006"\
EXECUTE cmd CAPTURING junk
```

Any of the QMQuery selection commands can be used in this manner. The usual ones are SELECT, SSELECT, and QSELECT. However, the stored list commands can also be used – e.g. GET-LIST.

The advantages of an external select are that it allows you to select a subset of records, and return the list in a sorted order.

Which selection should I use?

In general:

- If you want to select most or all records in a file and the order of the records is not important, then use an internal select
- If you want to select a small subset of records and/or you want the record ID's in a specific order, then use an external select.

3.6.3 Reading from files

Once you have a record ID, you can READ the associated record from the file:

```
READ var FROM filevar, record-id {THEN statements} {ELSE statements}
```

For example:

```
READ sales.vec FROM sales.summary, idate ELSE
  CRT 'Sales data for ':OCONV(idate, 'D'):' not on file'
END
```

This looks up data from the sales summary file which has a key of the internal date number. If the record is not found on the file, then the ELSE clause is executed which displays an error message.

The file must have been opened before the READ statement is attempted. Note also that the READ statement accesses the file variable – not the physical file name.

There are a number of READ statements:

- READV reads a single field from the record rather than the whole record. The syntax for this statement requires that the field number be included.
- READL and READU read the whole record and place a lock on the record to prevent other users from updating it. Record locking will be covered later under 'multi-user issues'.

It is important to recognise that the READ statement will ALWAYS read the record – regardless of the state of any record locks. If your application is only reading the record to obtain a value from the file, then READ is appropriate. However, if the application is going to update the record, then you should use the READU statement which applies an update lock to the record as part of the read process. This will be covered in more detail later.

3.6.4 Getting the ID from the select list for the READ

The above two sections cover creating a select-list of ID's, and then using an ID to read from the file. However, we need an intermediate step to get an ID from the select-list for use in the READ statement.

The READNEXT statement takes an ID from a select-list and stores it in a variable:

```
READNEXT var {FROM list} {THEN statements}{ELSE statements}
```

For example:

```
EQUATE CUST.SURNAME   TO 4
EQUATE CUST.FIRSTNAME TO 5
OPEN 'CUSTOMERS' TO customers ELSE STOP 201,'Customers'
```

```
eof = @FALSE
SELECT customers
LOOP
  READNEXT custid ELSE eof = @TRUE
UNTIL eof DO
  READ custrec FROM customers,custid THEN
    CRT custrec<CUST.FIRSTNAME>:' ':custrec<CUST.SURNAME>
  END
REPEAT
```

The `READNEXT` statement in the unconditional portion of the `LOOP` gets an ID from the select-list, and assigns it to the variable *`custid`*. If there are no more ID's in the list, then the statement assigns a value of `@TRUE` to the variable *`eof`*.

Note that you can only read FORWARD through a select-list. You cannot back up through a select-list. (There is no `READPREV` statement to get the previous item-id).

An alternative way of processing the select-list is to use `READLIST` statement to read the entire list into a variable, and then process the variable:

```
SELECT customers
READLIST custlist THEN
  LOOP
    REMOVE custid FROM custlist SETTING delim
    READ custrec FROM customers,custid THEN
      CRT custrec<CUST.FIRSTNAME>:' ':custrec<CUST.SURNAME>
    END
  WHILE delim DO REPEAT
END
```

`READLIST` reads the entire select-list into a variable, while the `REMOVE` statement extracts the next part of the dynamic array. See the documentation for more information.

3.6.5 Writing to files

Writing a record to a file uses the `WRITE` statement:

```
WRITE var TO filevar, record-id
```

The keyword `ON` may be used instead of `TO`.

For example:

```
WRITE sales.vec ON sales.summary, idate
```

Note the `WRITE` statement has no `THEN` or `ELSE` clauses.

If the record-id already exists in the file, then the existing record will be overwritten. Indeed, the `WRITE` statement offers no way for the programmer to determine whether an item already exists. Any such management of existing items must be done using the `THEN` or `ELSE` clauses of the `READ` statement.

There are also `WRITE` statements that match the variants of the `READ` statements:

- `WRITEV` writes a single field to the record rather than the entire record.
- `WRITEU` writes the whole record and maintains the record lock.

All of these `WRITE` statements will write the item regardless of the state of any record locks. See the online help, or the 'Multi-user issues' section later in this document for more information.

3.6.6 Closing files

Multi-value systems do not usually need to explicitly close files. Nevertheless, there is a `CLOSE` statement which will close the file – or remove the association between the physical file and the file variable:

```
CLOSE filevar
```

3.6.7 Other methods of file handling

The file handling so far has dealt with files defined within the local account, or which have a `Q-pointer` in the `VOC` pointing to the file in another account. In these cases, *OpenQM* uses the data stored in the `VOC` entry to find the location of the file in the file system.

But what if the file isn't defined in the `VOC`? Then you can supply the path to the file directly:

```
OPENPATH pathname TO filevar {THEN statements} {ELSE statements}
```

This is similar to the `OPEN` statement[17] except that you are supplying a pathname rather than an *OpenQM* filename. This method can be used to open both dynamic files and directory files in other *OpenQM* accounts. It can also be used to open other folders in the file system:

```
OPENPATH 'C:\Temp' TO temp.folder THEN ...
```

You can also quickly read a single item in an operating system file (rather than opening the file and then reading the item):

```
OSREAD var FROM path {THEN statements} {ELSE statements}
```

For example:

```
OSREAD txt FROM 'C:\TEMP\TEST.TXT' ELSE txt = ''
```

There is a corresponding `OSWRITE` statement to match `OSREAD`:

```
OSWRITE var TO path
```

Unlike the `WRITE` statement, you may not use the keyword `ON` instead of `TO`.

You can also read a text file sequentially – that is record by record or block by block by using the sequential file processing commands. These include:

```
OPENSEQ pathname {THEN statements} {ELSE statements}
READSEQ var FROM filevar {THEN statements} {ELSE statements}
WRITESEQ var TO filevar {THEN statements} {ELSE statements}
READBLK var FROM filevar, bytes {THEN statements} {ELSE statements}
WRITEBLK var TO filevar {THEN statements} {ELSE statements}
READCSV FROM filevar TO var1, var2 etc {THEN statements} {ELSE
statements}
WRITECSV var1, var2 etc TO filevar {THEN statements} {ELSE
statements}
CLOSESEQ filevar
```

See the online help for more information on these statements.

17 Pathnames can also be used within the `OPEN` statement, but only if an extended syntax is "allowed". See the `FILERULE` configuration parameter for more information.

3.6.8 Multi-user issues

OpenQM is a multi-user database system. As such, there will be times when multiple users try to access or update the same record at the same time. A good application will ensure that such contention issues are handled in a standard manner that preserves data integrity while not inconveniencing users too much. This is achieved through record locking.

Before looking closer at the individual locking statements, it is important to understand the following points:

- Locking is maintained by the application – not by the database. If two different applications access the same file, then it is up to the developer to ensure that the locking is consistent between the applications
- Locking is (normally) advisory only – that is, the locking does NOT prevent applications from reading or writing to the file – UNLESS those applications have been written to respect the locks
- *OpenQM* has a configuration parameter that changes this default behaviour. If the `MUSTLOCK` parameter is set to a value of 1, then any attempt to write or delete an item from a program where the program does not hold an update lock will result in a program abort. While use of this configuration parameter enforces better structure within programs, it will probably break many existing multi-value applications.

The practical implication of these points is that you should write all applications to use and respect locks. This way, when new applications are added to the system, you can be certain that all applications are handling locks correctly.

OpenQM offers locking at two levels – whole file locking, and individual record locking. Given that locking the entire file has potential to inconvenience many users, this option should be used with care.

When should locking be used

Any program that updates data files – or may update data files – should use some form of locking. Even if the system is written as a single user application, it is still good practice to build in locking as (a) this will make it easy to convert it to a multi-user application; and (b) it will maintain a single style of coding between applications.

In some cases, you may also want to employ locking even when data is not being updated. For example, you may want to report on the state of the system at a particular instant in time. To be sure that the data is totally consistent for that instant, you may want to lock the entire file(s) for reporting.

Note that neither of these locking scenarios actually stops applications from reading or writing to the file – a `READ` will ALWAYS read from the file, and a `WRITE` will ALWAYS write to the file. However, if the application is written to test for locks (using `READU`), then the reads and writes will only occur in accordance with those locks.

File locks

To lock an entire file, use the `FILELOCK` statement:

```
FILELOCK filevar {LOCKED statements}{THEN statements}{ELSE
statements}
```

The `LOCKED` clause will be executed if another user already has a file lock or a record lock on any record within the file. Unusually, the `THEN` and the `ELSE` clauses are completely optional, and neither need be present.

The lock should be released once processing of the file is complete. This is done using either the `FILEUNLOCK` or the `RELEASE` statements:

```
FILEUNLOCK filevar {LOCKED statements}{THEN statements}{ELSE
statements}
```

```
RELEASE filevar
```

Note that this form of the `RELEASE` statement will release all locks associated with *`filevar`* – not just the file lock. See the online help for more information.

The usage of these statements will be something like:

```
OPEN 'SALES' TO sales ELSE STOP 201, 'Sales'
err = @FALSE
FILELOCK sales LOCKED err = @TRUE
IF NOT(err) THEN
  ...
  process file here
  ...
  FILEUNLOCK sales
END
```

This example simply bypasses the file processing if a lock already exists on the file. In practice, error handling should be more sophisticated than this.

Record locks

OpenQM supports two types of record locks – read (or shared) locks, and update locks. This section will mostly cover the use of update locks.

The purpose of an update locks is reasonably obvious – you place an update lock on a record when you want to update the record. The purpose of the read lock is a little less obvious – its basic action is to prevent an update lock from being applied. This allows an application to process the entire file without any risk that another user will update it during the processing. See the Locks section in the online help for more information on read locks.

The essential purpose of update locks is to allow the developer to structure the system so that multiple users read and write data in a consistent fashion. What we want to avoid is something like:

```
User A reads record
User B reads record
User B updates record
User A updates record
```

This will leave the system looking the way that User A expects, but User B’s changes have been lost. The use of record locking would have allowed the above situation to have been trapped and action taken to avoid problems.

There are two basic approaches to the use of record locks. These approaches are sometimes termed optimistic and pessimistic locking:

Under pessimistic locking, the record is locked at the time of the original read, and the lock is maintained until the update has been completed. This ensures that no other user can obtain a lock on the item until such time as the original user has updated or released the record. The downside of this type of locking is that the lock may be maintained for a considerable period of time – at least the duration of any amendments to the record, plus distraction time. Users have been known to go out for lunch leaving a record locked on their screen – much to the annoyance of other users.

Optimistic locking works on the premise that in most cases, there will be no contention between users for a particular record. Therefore, the record is only locked immediately prior to update. When this occurs, the record on disk is compared with the original record. If the records are the same, then no one else has updated the record, and it is safe to update the record. If the record has changed, then program needs to offer choices about how to handle the situation.

The advantage of optimistic locking is that records are only locked for brief periods of time. The disadvantage is that more programming is required to handle the situation where records have changed between the original read and the read done immediately prior to update.

Pessimistic locking looks like:

```
READ record setting update lock
process record
WRITE record
```

Optimistic locking looks like

```
READ record
process record
READ record setting update lock
If the record is unchanged from the original READ then
  WRITE the updated record
Else
  do something else
END
```

An update lock is obtained by using the READU statement. This is a variant of the READ statement:

```
READU var FROM filevar, record-id {LOCKED statements} {THEN
statements} {ELSE statements}
```

For example:

```
ok = @TRUE
READU cust.rec FROM customers, cust.id LOCKED
  ok = @FALSE
  EMSG = 'Record ':cust.id:' is locked by user ':STATUS()
END ELSE
  cust.rec = ''
END
IF ok THEN
  GOSUB updaterec
  WRITE cust.rec ON customers, cust.id
END
```

Or:

```
READ cust.rec FROM customers, cust.id ELSE cust.rec = ''
cust.rec.orig = cust.rec
GOSUB updaterec
ok = @FALSE
tries = 0
LOOP
  tries += 1
  READU cust.rec.curr FROM customers, cust.id LOCKED
    NAP 10
  END THEN
    ok = @TRUE
  END ELSE
    ok = @TRUE
    cust.rec.curr = ''
  END
UNTIL ok OR (tries GE 5) DO REPEAT
IF ok THEN
  IF cust.rec.curr EQ cust.rec.orig THEN
    WRITE cust.rec ON customers, cust.id
  END ELSE
    RELEASE cust.rec, cust.id
    EMSG = 'Record ':cust.id:' has been changed by another user'
  END
END ELSE
  EMSG = 'Could not obtain lock on record: ':cust.id
END
```

The first code fragment is an example of pessimistic locking. If the code cannot obtain the lock, then no update takes place and an error message is returned.

The second fragment is an example of optimistic locking where the record is read from the file and updated (in memory) before checking whether it is OK to write the record. Given that locks will only be held momentarily in an optimistic locking scenario, the `READU` is placed in a short loop. If the record is locked, then the process sleeps for 10 milliseconds before trying again. The process will loop in this manner until it successfully reads the record, or it has five unsuccessful reads. Once a lock has been obtained, the record that is read is compared with the one read prior to the update process. If the record has not been changed, then the updated record is written to the file; otherwise the lock is released and an error message returned to the user.

Note that any time that a lock is obtained, then it must be released. Locks can be released by:

- the `RELEASE` statement
- the `WRITE` statement (unless `WRITEU` is used which maintains the lock)
- terminating the program.

Failure to release locks could result in the system running out of locks in the lock table. To check the number of locks available, type CONFIG from the command prompt, and check the value of NUMLOCKS. The number of locks available to the system can be changed by using the Configuration Editor. See the online help for more information.

4 Building Applications

4.1 What to Build?

What application you build in *OpenQM* is up to you. Some possible examples are:

- A home inventory application
- A business inventory application
- Staff records

In general, *OpenQM* can be used to build an application wherever a database is required. The application may be for a single-user PC, or could be put on a server for use by hundreds of users.

Clearly, we need something relatively simple for this book. Part 1 of *Getting Started in OpenQM* used several files to illustrate how to use `QMQuery`. These files contained interest and exchange rate for New Zealand. Building on this base, an appropriate first application will allow this data to be viewed and edited.

The application we will build here will:

- allow the user to display, add, and edit data in the files created in Part 1. These were:
  - interest rates files
  - exchange rate files
  - foreign exchange transaction files
- automate the update process by:
  - getting the update spreadsheet from the internet
  - manipulating the retrieved spreadsheet
  - use the file transfer procedures to update the database.

4.2 Green Screen Applications

4.2.1 General

Many multi-value applications are available in green-screen form, also known as a Character User Interface (CUI). The term green-screen dates back to the days when computer terminals only had monochrome displays, although the colour was not always green.

Later CUI variants introduced colour, menus, and mouse support. However, the term "green screen" was still applied to these interfaces.

Today, CUI interfaces are considered to be a sign of a "legacy" system. This doesn't make them bad – in fact, many people consider CUI interfaces to be superior to GUI interfaces for data entry applications. Nevertheless, building a new application with only a character interface is somewhat anachronistic in today's world. Accordingly, this book will only give brief comments on green screen development.

4.2.2 How to build a Character User Interface in *OpenQM*

Character user interfaces are built using the following statements:

- @(xx,yy) to position the cursor on the screen
- `CRT` or `DISPLAY` to display data on the screen
- Various `@(-x)` commands to perform other screen operations
- `INPUT`, `INPUT @`, `KEYIN`, or a user defined function to get input from the user

These statements are contained within a series of loops and subroutines. These may be structured in the following manner:

```
GOSUB initialise
GOSUB display.screen
LOOP
  GOSUB getinput
UNTIL quit DO
  GOSUB validate
  IF inputok THEN
    GOSUB process
  END ELSE
    GOSUB displaymessage
  END
REPEAT
GOSUB closedown
```

A simple green screen program was presented in section 3.3. This wasn't structured as shown above, but was rather written in a "top-down" style. This style is adequate for short programs, but any complex programs should be structured as shown above.

As character interface programming is the exception rather than the norm these days, this document will not spend any further time covering building a CUI program.

4.3 AccuTerm GUI Applications

4.3.1 *AccuTerm* components

So far *AccuTerm* has been used as a simple terminal emulator – that is, as a piece of software that emulates a variety of traditional computer terminals. In this mode of operation, *AccuTerm* displays text and some graphic characters (depending on the capabilities of the terminal, and the character set in use).

But *AccuTerm* is much more than this. It's features include:

- client-side scripting
- multi-value integration tools
  - file transfer
  - a multi-value server to allow connection of COM applications
  - an Object Bridge to allow control of COM applications
- a graphical user interface (GUI) development environment.

It is this last feature that we will now use to develop a GUI application.

Overall, *AccuTerm*'s capabilities mean that you can split applications into client-side and server-side parts and thereby achieve things that would be difficult to do solely in a server environment.

4.3.2 *AccuTerm* documentation

AccuTerm has many features which will not be covered in this document. Likewise, even those features that are covered will only be glossed over lightly. In order to gain a full appreciation of the features available in *AccuTerm* and how to access them, it is essential that you download a copy of the *AccuTerm* documentation.

There are three manuals for *AccuTerm*. These are:

- the Users Guide
- the Programmers Guide
- the VBA Reference Manual.

For our purposes, it is the Programmers Guide which is of most relevance. This gives technical information on:

- how to access *AccuTerm*'s features via script commands
- the features available under *AccuTerm*'s various terminal emulations
- how to use the multi-value server and object bridge components
- the commands available for use in *AccuTerm* GUI programming.

This last aspect is particularly important, because while we will be using the *AccuTerm* GUI commands to build an application, this document will not be explaining the syntax or the available options in any great detail.

The VBA Reference Manual is important if you are going to do a lot of scripting with *AccuTerm*. *AccuTerm* uses *Winwrap BASIC* (a clone of *Visual Basic for Applications*) for scripting purposes. We will be touching on scripting later in this document.

The User's Guide may be of interest, but there is no important information in there from an application design perspective.

The manuals can be downloaded from the Accusoft Enterprises website (www.asent.com). Follow the links to 'Support', and then 'Manuals'.

Also of interest on this site is the 'Support Forum'. If you have problems with *AccuTerm*, you can post questions here and other forum users may be able to assist you. Don't forget to search the Forum for answers before you post your own question – it may well have been asked on previous occasions.

5 Creating the Application

5.1 First Steps

5.1.1 Application Design

We outlined what we want the application to do in Section 4.1. Now, we want to define how the application should look, and how we should interact with it. Therefore:

- We should select the file to view/edit with a drop-down box. An alternative here would be to use radio-buttons, but this would restrict any future expansion of the list of files available for viewing/editing.
- The data should be displayed in a grid
  - We should have some means of limiting the data display – probably using a drop-down list list of years and/or months
  - The data in the grid should be read-only by default, but can be unlocked for editing/entry.
  - When editing/entering data, should data be saved on a record-by-record basis, or should the whole grid be saved?
- We will need a button to initiate the auto-update of data from the internet.

5.1.2 Creating the necessary files

AccuTerm applications have two (or more) parts:

- An item containing descriptive data about the GUI application. There will be one of these items for each application. These items are stored in their own file.

- The `BASIC` program files used by the application. This should NOT be named `GUIBP`, as this is the *AccuTerm* file containing many of the supporting subroutines for the GUI environment.

So, let's create these files:

```
CREATE.FILE APPS
CREATE.FILE BP.GUI DIRECTORY
```

Now, check that you can access the *AccuTerm* `GUIBP` file. This should be referenced in the `QMINTRO` account as a Q-pointer:

```
SORT GUIBP
```

If you can't `SORT` the contents of the file, then check your installation of *AccuTerm*:

- Did you install the multi-value host programs when you installed *AccuTerm*? If not, then load them now.
- Log to the `ACCUTERM` account and list the files there. Is there a file named `GUIBP`? Does this file contain any items? Load the multi-value host programs again if necessary.

Now we can create the form that will be used by the application.

5.1.3 When you strike problems

As you build applications, you will inevitably strike situations where things don't work as you expect. Here are some issues that you may encounter:

- By default, *OpenQM* is intolerant of encountering variables that have not been previously defined in the program, and the program will abort. The compiler will warn of some of these, but unassigned variable errors can still occur when execution takes a different path from that intended.

To minimise the problems caused by program aborts, you can set the `UNASS.WARNING` option to `ON`. Type the following command from the command prompt:

```
OPTION UNASS.WARNING
```

The unassigned variable error will still occur, and a warning message will be displayed, but the program will still continue. Ladybridge recommend that you only use this option while developing the application, and that it should not be used for production code.

- If the program aborts while the *AccuTerm* GUI is displayed on the screen, then the GUI will remain on the screen and you will be unable to close it via normal means. Use the option 'Reset | Terminal' from the *AccuTerm* menu to close the GUI application and clean up the GUI runtime components.

5.2 Creating the Initial Form

5.2.1 Using the GUI Designer

Forms are designed in *AccuTerm* using the GUI Designer (or `GED`). This can be invoked by:

- selecting 'Tools | GUI Designer' from the menu
- clicking on the 'GUI Designer' icon in the toolbar
- starting `GED` from the command prompt. As with other multi-value commands, the syntax for this is:

```
command filename item
GED APPS item
```

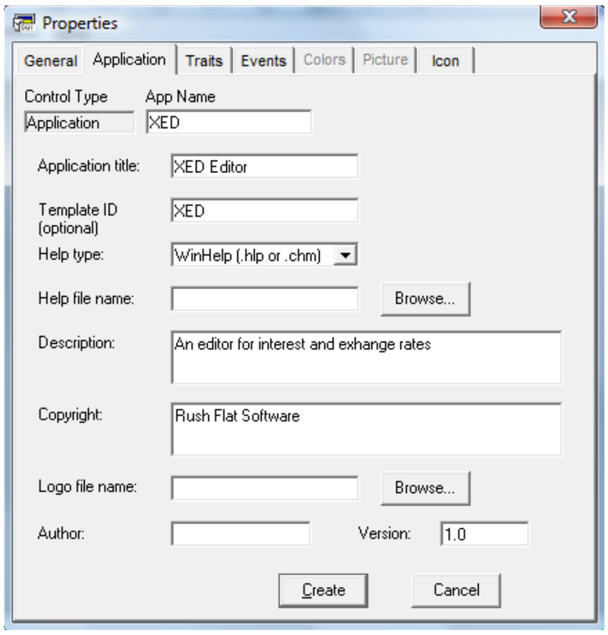
Let’s call this application *XED*. Therefore:

```
GED APPS XED
```

The GUI Designer will start, and display a prompt saying:

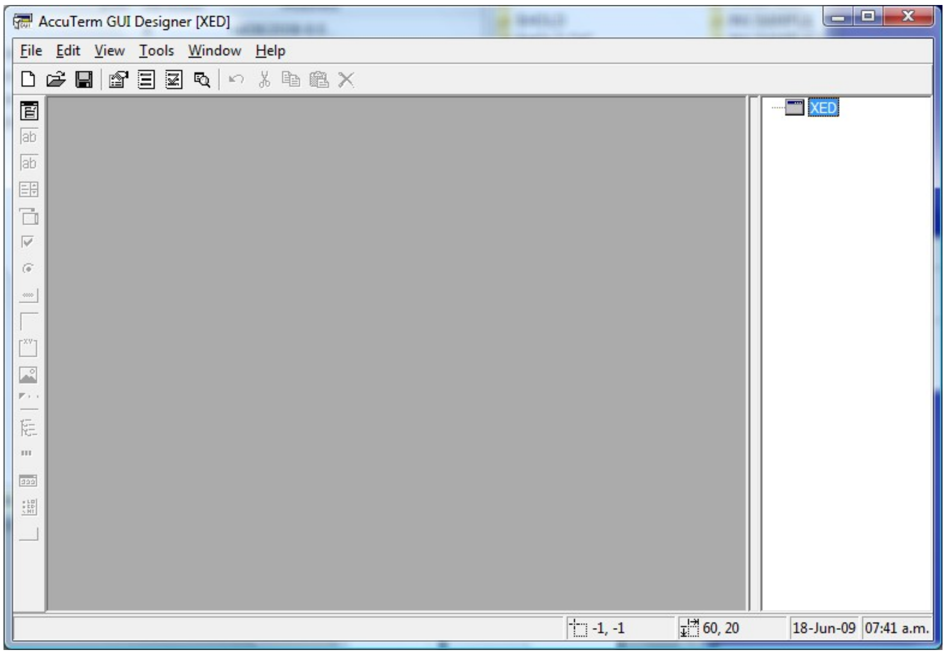
```
Item 'XED' not on file. Create new project?
```

Click on ‘OK’ and the project properties screen will be displayed. On the ‘Application’ tab, fill in any of the descriptive fields, then click on ‘OK’.



Control Type: Application
App Name: XED
Application title: XED Editor
Template ID (optional): XED
Help type: WinHelp (.hlp or .chm)
Help file name:
Description: An editor for interest and exhange rates
Copyright: Rush Flat Software
Logo file name:
Author:
Version: 1.0

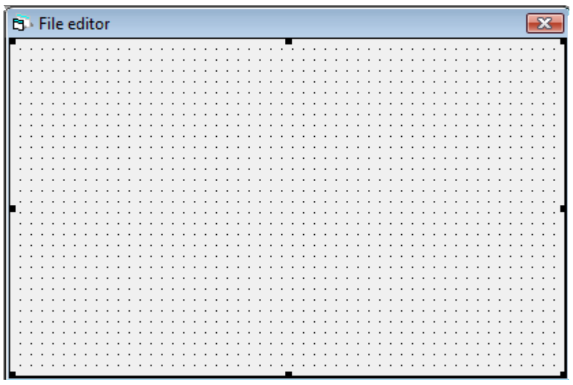
Now, click on ‘Create’. The GUI designer will now display a blank window as follows:



This window has a number of tools on the left hand side, a work area in the centre, and a list of objects and controls on the right. At this stage, the only object is the application itself.

Creating the Application

Right-click on the application in the object list, and choose ‘New | Form’. Another properties screen will appear. Change the form name to `frmmain` and set the window caption to ‘File editor’. You may also wish to change the default font for the form on the ‘Font’ tab – try using 10 point text rather than the default 8 point. Click on ‘Apply’ and the form will appear in the designer work area:



Now add the following controls and set properties as follows:

Control	*Name*	*Parent*	*Properties*
Frame	fracontrols	frmmain	Border style: none
Label	lblfile	fracontrols	Caption: File:
Combo box	cmbfile	Fracontrols	Events: Change
Label	lblyear	fracontrols	Caption: Year:
Combo box	cmbyear	fracontrols	Events: Change
Label	lblmonth	fracontrols	Caption: Month:
Combo box	cmbmonth	fracontrols	Events: Change
Grid	grddata	frmmain	Style: Editable
Frame	frabtns	frmmain	Border style: none
Button	btnupdate	frabtns	Caption: &Update
Button	btnedit	frabtns	Caption: &Edit
Button	btncancel	frabtns	Caption: &Cancel
Button	btnsave	frabtns	Caption: &Save
Button	btnexit	frabtns	Caption: E&xit

Drag the controls to position and size them. When this stage is complete, the form (and the GUI Designer window) should look as shown on the next page.

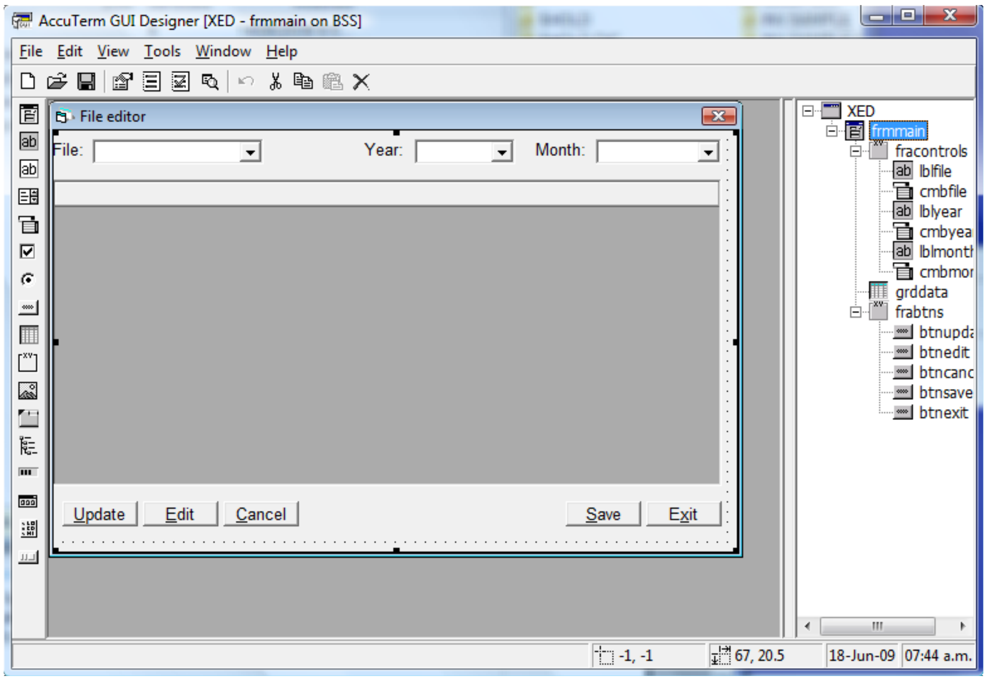
The frames have been used as container objects for the labels, combo boxes, and buttons. These probably aren’t necessary, but they keep the overall design tidy and similar to that of other GUI tools.

The properties of any of these controls can be adjusted by right-clicking on the control (either the control itself or the control name in the control list on the right), and then choosing ‘Properties’ from the shortcut menu.

Note the following points about the settings so far:

- We needed to set events for the combo boxes, but not for the buttons. This is because the ‘Click’ event for the buttons is enabled by default, but combo boxes do not have any events enabled by default.

- The ampersand (&) in the caption of the command buttons identifies the keyboard letter used to activate that button in conjunction with the 'Alt' key. Therefore, 'Alt-x' will activate the Exit button.



- There are many other properties that we could have set. Some of these we will be setting from the `BASIC` subroutines as we build the application. You can investigate the others by looking through the *AccuTerm* manuals and help files.

You can preview how the form will look by choosing 'Tools | Preview' from the menu. You can close the displayed form by clicking the close icon in the top right corner, opening the command menu by clicking on the icon in the top left corner and then choosing 'Close', or by choosing 'Close' from the Form Preview window. Note that clicking on the 'Exit' button on the form does not do anything because we have not assigned any action to that button.

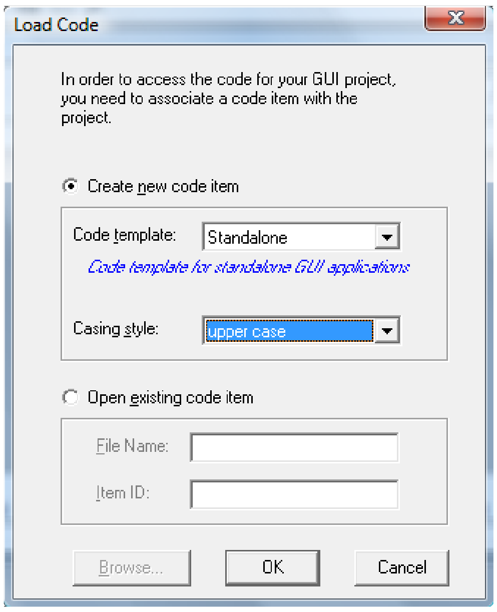
Now save the form by clicking on the 'Save' icon, or choose 'File | Save' from the menu.

5.2.2 Creating the code skeleton

The GUI Designer works closely with the `WED` code editor. The two applications allow you to move back and forth updating the form design and the matching code.

To create the initial code for the form, click on the 'Display code window' icon, or choose 'Tools | Code editor' from the menu. This displays the following window:

Creating the Application


Load Code

In order to access the code for your GUI project, you need to associate a code item with the project.

(●) Create new code item

Code template: Standalone

Code template for standalone GUI applications

Casing style: upper case

( ) Open existing code item

File Name:

Item ID:

Use the ‘Standalone’ code template, and choose the coding style that you want, then click on ‘OK’. The GUI Designer will then generate the code for the form and display it in the `WED` editor.

The basic structure of this code is as follows:

```
Open files
Read GUI template
Initialise the GUI engine
Load the GUI definition
Display the GUI form
LOOP
  Wait for an event from GUI
UNTIL quit
  Process event
REPEAT
Shut down the GUI
```

This code takes up about the first 60 lines of the skeleton program. The rest of the program is made up a switching subroutine that sends each event to the correct event handler, and the individual event handler subroutines. In total, the generated program code for our application totals around 300 lines.

The event handler subroutines are created as empty subroutines. It is up to you as the developer to add the code to the subroutine. The empty code handler for the ‘Exit’ button is shown below:

```
*-->BEGIN EVENT HANDLER<--*
GUI.XED.FRMMAIN.BTNEXIT.CLICK: *
*
* Add your event code here...
RETURN
*
*-->END EVENT HANDLER<--*
```

NOTE: It is IMPORTANT that you keep the text marking the beginning and end of each event handler in the program. The code generator uses these markers to identify whether code sections are present or not. If you remove the markers, the code generator will report that that event handler is not present in the code, and will re-create it for you.

Even though there is no code to perform our desired functions in the skeleton, the program will compile and run to display a working *Windows* form. First we need to save the program. Click on the ‘Save’ icon, and navigate to the `BP.GUI` program file. Save the program as *XED*. Now close WED, save the project file in GED, and exit from the GUI Designer. You should now be back at the command prompt.

Now compile the program:

```
BASIC BP.GUI XED
Compiling BP.GUI XED
********************************************************************
************
WARNING: Final END statement is missing
0 error(s)
Compiled 1 program(s) with no errors
```

The asterisks indicate the progress of compilation. We have got a warning that there is no END statement to mark the end of the program, but the compiler didn't find any errors. We will ignore the warning for now.

Now, try running the program:

```
RUN BP.GUI XED
```

A *Windows* form should now be displayed. You can move this form around the screen like any other window, but you can't do much else with it. You can click on the buttons, but they don't do anything. The minimise button works, but the maximise button is disabled. Finally, you can close the form by clicking on the close icon in the top right corner, choosing 'Close' from the control menu, or pressing 'Alt-F4'.

Adding code to the skeleton

We need to add many sections to the code skeleton to build the application. However, we need to make sure that the code we add does not interfere with the sections of code that are generated by the GUI Designer.

The best way to achieve this is to:

- only add new code sections at the end of the auto-generated code
- only add code in the auto-generated sections where indicated.

By adhering to these rules, the code generator can update the auto-generated code as we build the application without impacting on the code we have added .

In order to assist finding subroutines as you write the application, put the subroutines in alphabetic order. You will still tire of scrolling up and down, but at least you will know where to look!

Debugging

As we build the application, there will be times when you want to see what is happening to a variable. One way to do this is to put messages into the code which will pop up to display the value. We'll put a subroutine in now to do this, so that you can call it whenever you want later on:

```
debug_msg:
*
CALL ATGUIMSGBOX(debug.txt, 'Debug', MBIICON, MBOK, '', debugok,
guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

RETURN
```

To use this subroutine, set the variable `debug.txt` to the message you want to display, and call it like a normal subroutine. For example:

```
debug.txt = 'First: ':start_yyyymm:'  Last: ':last_yyyymm
GOSUB debug_msg
```

5.2.3 Making the form functional

Now, we need to convert this skeleton into a fully working program. You don't need to go through the GUI Designer to get to the code – you can simply use `WED` directly:

```
WED BP.GUI XED
```

Let's start with the 'Exit' button. If you look through the code, you will see there is already an event handler to close the form. So, our first step will be to attach this event handler to the 'Exit' button.

Add the following line to the event handler for the 'Exit' button:

```
GOSUB GUI.XED.FRMMAIN.CLOSE
```

The event handler will now look like:

```
*-->BEGIN EVENT HANDLER<--*
GUI.XED.FRMMAIN.BTNEXIT.CLICK: *
*
GOSUB GUI.XED.FRMMAIN.CLOSE
*
RETURN
```

Add an `END` statement right at the very end of the program. This will identify the end of the program to the compiler, and stop the compilation warning message about the final `END` statement being missing from being displayed.

You can test this works by saving the modified code, compiling it, and running it again. Clicking on the 'Exit' button now closes the form and exits the application.

Compiler directives

Compiler directives are switches that tell the compiler how to operate. This allows *OpenQM* to emulate the behaviour of a variety of other multi-value databases.

We want a couple of compiler directives within this project. Open the code for the project in a code editor, and enter the following lines in the first section (where the comments in the skeleton say "Add your equates and code to open files here"):

```
$CATALOGUE
$MODE UV.LOCATE
```

The `$CATALOGUE` directive tells the compiler to catalogue the program in the private catalogue each time it is compiled. This means that we can run the program by simply typing its name, and that we do not need to manually re-catalogue the program after each compile.

The `$MODE` directive tells the compiler that we will use *UniVerse* syntax for any `LOCATE` statements in the program. Check the online help for `LOCATE` for more information on this directive.

Add these compiler directives and recompile the program. You should now get a new message that `XED` has been added to the private catalogue. You can now run the program simply by typing:

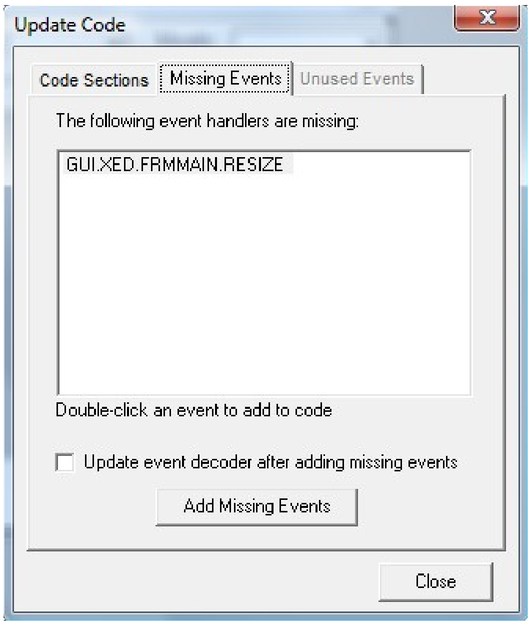
```
XED
```

Resizing the form

If you run the program now, you will find that you cannot resize the form. Clearly, this isn't desirable. Making the form properly resizeable involves several steps.

First of all, re-open the form in the GUI Designer. In the tree-list of controls, right-click on *`frmmain`* and choose 'Properties'. Change the form style to 'Sizable'. Now click on the 'Events' tab, and check the 'Resize' event. Apply the changes.

Now, click on the ‘Update code’ icon (or select ‘Tools | Update code’ from the menu. This will open the same window that you saw when first creating the code skeleton. This time, the existing file and item names will be already filled in. Click on ‘OK’, and the existing code will be opened in the `WED` editor. Note that the ‘Load Code’ window changes to an ‘Update Code’ window. Click on the ‘Missing Events’ tab – you should see something like this:



The following event handlers are missing:

GUI.XED.FRMMAIN.RESIZE

Double-click an event to add to code

☐ Update event decoder after adding missing events

This tells us that we have added a new event to the form in the GUI Designer, but there is no handler for that event in the code.

Click on ‘Add Missing Events’, then go back to the ‘Code Sections’ tab and click on the ‘Update’ button. This updates the code skeleton for the new event. Finally close the ‘Update Code’ window.

Now look at the code in the `WED` code window. Find the event decoder, and around line 80, you should see a couple of lines like:

```
CASE guievt=GERESIZE
GOSUB GUI.XED.FRMMAIN.RESIZE;guievt=0
```

Similarly, later in the skeleton, you should find the (empty) event handler for the resize event.

If you have formatted any of the code in the skeleton so far, you may have noted that updating the skeleton using the ‘Update Code’ window has reformatted the event decoder. To put this back to properly indented code, click on the ‘Format code’ icon (or choose ‘Tools | Format code’ from the menu). The settings for the formatter can be changed from the ‘Options | Tools’ menu option.

Save and close the code from the `WED` window, and the project from the GUI Designer window. Compile and run the program and see what happens:

The form is now resizeable, but none of the controls move as the window is resized. Maximising the window does not increase the usable area in grid, while dragging the window smaller hides some of the buttons.

What we need now are some rules for what happens when we resize the form. These are:

- The form will have minimum sizes so that the buttons and combo boxes do not get hidden

- When the form is resized horizontally, the grid and the two frames will expand with it. Further, the ‘Save’ and ‘Exit’ buttons will maintain their position relative to the right-hand border of the form
- When the form is resized vertically, the lower frame will maintain its position relative to the bottom border of the form. Similarly, the grid will expand to occupy the usable area.

We will get the minimum form sizes from the design-time values which will be in effect when we first create the form. The code for that will go in an *initialise* subroutine which we’ll call during the program startup. This code looks like:

```
initialise:
*
guiapp = 'XED'
guifrm = 'FRMMAIN'

CALL ATGUIGETPROP(guiapp, guifrm, '', GPWIDTH, 0, 0,
form_width_min, guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

CALL ATGUIGETPROP(guiapp, guifrm, '', GPHEIGHT, 0, 0,
form_height_min, guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error
*
RETURN
```

Put the call to the *initialise* subroutine after the macro has been run to create the form, but before the form is displayed. This means that the standard variables *guiapp* and *guifrm* used by the `ATGUIxxx` property subroutines won’t have been initialised (because no GUI events have fired to create them). Therefore, we define those variables here before we call the get property subroutines so that those subroutines don’t return errors.

Now, we need to react to the resize event. Put the following code in the resize event handler:

```
*-->BEGIN EVENT HANDLER<--*
GUI.XED.FRMMAIN.RESIZE: *
*
form_width = guiargs<1, 1>
form_height = guiargs<1, 2>
GOSUB resize_form

RETURN
*
*-->END EVENT HANDLER<--*
```

and add a ‘resize_form’ subroutine:

```
resize_form:
*
IF (form_width LT form_width_min) OR (form_height LT
form_height_min) OR do_resize THEN
  IF form_width LT form_width_min THEN form_width = form_width_min
  IF form_height LT form_height_min THEN form_height =
form_height_min
  CALL ATGUIMOVE(guiapp, guifrm, '', '', '', form_width,
form_height, guierrors, guistate)
  IF guierrors<1> GE 2 THEN GOTO gui.error
  do_resize = @FALSE
END

ctrlid = 'FRACONTROLS'
property = GPWIDTH        ;   prop.value = form_width
ctrlid<-1> = 'FRABTNS'
property<-1> = GPWIDTH    ;   prop.value<-1> = form_width
ctrlid<-1> = 'FRABTNS'
```

```
property<-1> = GPTOP     ;   prop.value<-1> = form_height - 2.5
ctrlid<-1> = 'BTNEXIT'
property<-1> = GPLEFT    ;   prop.value<-1> = form_width - 9
ctrlid<-1> = 'BTNSAVE'
property<-1> = GPLEFT    ;   prop.value<-1> = form_width - 17
ctrlid<-1> = 'GRDDATA'
property<-1> = GPWIDTH   ;   prop.value<-1> = form_width
ctrlid<-1> = 'GRDDATA'
property<-1> = GPHEIGHT  ;   prop.value<-1> = form_height - 5

CALL ATGUISETPROPS(guiapp, guifrm, ctrlid, property, prop.value,
guierrrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error
*
RETURN
```

Finally, add the following line to the `initialise` subroutine:

```
do_resize = @FALSE
```

Let's go through what is happening here.

Firstly, the main code for resizing the form is separated from the resize event handler. This is so that we can call this subroutine from other parts of the application.

When you resize the form, it fires the resize event. The event decoder picks up that event and directs control to the resize event handler.

Note that when an event occurs, the event handler defines the *`guiapp`*, *`guifrm`*, and *`guictl`* variables. These variables can then be used for subsequent calls to `ATGUIGETPROP` or `ATGUISETPROP` (to get or set properties respectively).

When a form is resized, the resize event leaves the new form dimensions in the *`guiargs`* variable. So, the first thing we do in the resize event handler is to retrieve those dimensions. Then we call the *`resize_form`* subroutine to resize the form.

In the *`resize_form`* subroutine, we test the new form sizes against the minimum form sizes we established in the *`initialise`* subroutine. If either of the dimensions is less than the minimum, we reassign that dimension to the minimum size, and resize the form using the `ATGUIMOVE` subroutine.

Now we need to resize/move the controls within the form to match the forms new dimensions. The next block of code forms three dynamic arrays of control id's, property names, and property values, with each array having an entry for each property that we want to change. Forming these arrays lets us change all the properties in one hit by calling the `ATGUISETPROPS` subroutine.

Now, what about the *`do_resize`* variable? In most cases, any resizing of the form is initiated by the user by clicking on the 'maximise' button, or dragging the window borders. However, if we want to resize the form from within the program, we need a way to get to the `ATGUIMOVE` subroutine. This is where the *`do_resize`* flag comes in. If this is set to `@TRUE`, then the *`resize_form`* subroutine will call the `ATGUIMOVE` subroutine with the new form dimensions. At the end of this code block, the *`do_resize`* variable is reset to `@FALSE` so that the code does not automatically follow this branch next time through.

Finally, we needed to put the *`do_resize`* variable in the *`initialise`* subroutine so that the variable has a value when the *`resize_form`* subroutine is called.

There are a few points to note here:

- The property names (`GPHEIGHT`, `GPLEFT`, `GPTOP` etc) are NOT literal string values. Nor are these variables. These are constants that have been defined in the `ATGUIEQUATES` include file (see the GUI Header section in the code skeleton), so it is important that you do not put quotes around these "names"

- Although we defined the control names in the GUI Designer in lower case, the program refers to them in upper case. This is because the control loop in the skeleton program converts these names to upper case before sending the event to the event decoder. [While *OpenQM* treats upper and lower case variable names identically, many other multi-values systems do not, so it is best to keep cases consistent]
- You will note we did not have to adjust the vertical position of any of the buttons at the bottom of the form. That is because those buttons were contained within the *`frabtns`* frame. Moving the frame vertically moved the buttons contained within it
- If we don't move a control, it maintains its position relative to the top left corner. Therefore, we don't need to worry about the left or top values of the top frame or the data grid
- We need to check the value of *`guierrors`* after every call to an `ATGUIxxx` subroutine.

You may need to change the offsets from *`form_height`* and *`form_width`* to get your form looking right. The actual values you need will depend on the size of your upper and lower frames, and widths of your buttons. However, in general:

- The top of the lower frame (*`frabtns`*) should be at the form height less the height of the frame control at design time
- The left of the 'Exit' button should be the form width less the width of the button, less the distance from the right hand side of the button to the edge of the form
- The left of the 'Save' button should deduct another button width and the spacer distance between the buttons
- The grid height should be the form height less the combined heights of the top and bottom frames.

At this stage, the form functions like a normal *Windows* form – which it is. You can move it, resize it, minimise it, and maximise it. You can close it by using various standard *Windows* methods, and the 'Exit' button is functional. All we need to do now is add some real functionality to the form.

6 Adding Functionality

6.1 Tasks

This is the main part of making our application functional. It helps here to list the things that we need to do. These include:

- Populating the combo boxes to allow the user to select the files and items to display and edit
- Get the user responses and open the correct files
- Reading the dictionary of the selected file to get the column headings and other formatting information
- Read the data portion of the selected file to get the requested data
- Display the data in the grid
- Provide editing facilities
- Save any changed data back to the data file.

6.2 Populating the Combo Boxes

6.2.1 Control information

The first question we face in populating the combo boxes is: "What are we going to populate them with?" Or, which files are we going to edit?

There are three basic ways of providing a list of files for the user to choose from. We could:

- hard code the file names into the program
- store the allowable file names in a control item

- select the files from the VOC using a SELECT statement.

The first method is easy, but lacks flexibility. If we add new files to the system that we want to edit (and which are in a format that the program can handle), we would need to edit the program to add those new files.

The last method requires that we have some means to identify those files that we wish to select. Further, we need to give the program some information about each file. All this is rather difficult to achieve simply by selecting files from the VOC.

The second method ticks the boxes of giving us a way to identify files available for editing and allowing us to pass information to the program about each file. Further, we can add files to the system without recompiling the program. This begs the question: "What information should be stored in the control item?" Let's look at the relevant files:

The following table lists the relevant files that we created in Part 1 of 'Getting Started in *OpenQM*', and some of their characteristics:

Filename	*Key structure*	*Field count*	*Record count*
IRATES	YYYYMM	9	269
XRATES	YYYYMM	8	292
FX.DAILY	ddddd	8	1065
FX.MONTHLY	YYYYMM	8	35

A few things stand out quickly from this table:

- Three of the files have the same key structure
- All of the files have a similar count of fields
- Most of the files have too much data to display in a grid without filtering. (This isn't to say the grid can't handle this much data – simply, that for editing purposes, we don't want a large number of records in the grid).

The difference in key structure is something that needs to be known by the application. This will let the program break the data into appropriate groups for display.

The field count column raises another issue: "How will the application know which fields to display?"

We could select the D-type dictionary items. However, some dictionaries have multiple D-type items for each field. How are we to distinguish which D-type item describes the data correctly for display purposes? Further, some files may use A- or S-type items in the dictionary rather than D-types. Finally, we may not want all fields in the file to be editable by this application.

As we go through the possibilities, it quickly becomes apparent that it would be useful to store the field names that we wish to display/edit in the control item.

At this stage, it appears the control item should contain (at least) the following information:

- File names
- Key structures
- Field names

To this, I would also add:

- ID of first record
- ID of last record

These pieces of information will be used to determine the values used to populate the year and month combo boxes.

So far, we have referred to a single control item to hold all of this information. While we can store all this data in a single item, we need to think whether this is the best way to store the information. So, let's consider an alternative:

One option is to have a primary control item that contains the names of the files available for editing. The data in this item will be used to populate the 'File' combo box.

Each file named in the primary control item would also have its own control item containing descriptive data about the file (key structure, field names etc). The data in these items will be used to populate the 'Year' and 'Month' combo boxes, and to provide any other information that the application requires about that file.

It is this second option that we will use in this application. This is mainly because a single item containing all the information becomes complex as we add new files to the application – we need to ensure that all the file specific information stays matched to its filename. Separating this information into separate items removes this complexity.

6.2.2 Creating the control items

First of all, we need a file to store our control items in. Create a file named `XED.VAR`.

```
CREATE.FILE XED.VAR
```

The `VAR` part of the filename indicates that we expect this file to contain "various" bits of information. i.e. this is intended as a relatively unstructured file.

Now use `WED` to create an item named *`XED.FILES`*. Normally, we could use `MODIFY`, but because we haven't created any dictionary items, `MODIFY` can't determine the nature of the fields to create.

```
WED XED.VAR XED.FILES
```

Enter the file names in the first field, with each file name separated by a Value Mark. You can enter a Value Mark in `WED` by clicking on the `VM` button in the toolbar. This will display on screen as a broken ']' symbol. Save the item once all files have been entered.

Check the names of the dictionary items that you wish to display by typing:

```
SORT DICT XRATES
```

and repeat for the other files. Now create a new item for each of the files named in the *`XED.FILES`* item:

- Enter a file description in the first field
- Enter the key structures in the second field. Just use 'D' to represent the date format
- Enter the dictionary names to display for each file in the third field. The names should be space delimited
- Save the item.

To check the contents of the items, type:

```
LIST XED.VAR F1 F2 F3 FMT 'L#50'
XED.VAR...  F1............  F2.......  F3.......................................
XRATES      Exchange rates  YYYYMM     @ID TWI USD GBP AUD JPY EUR GDM
            (month avg)
IRATES      Interest rates  YYYYMM     @ID INTERBANK 30DAY 60DAY 90DAY 1YR 5YR 10YR
            (monthly avg)
FX.MONTHLY  Foreign         YYYYMM     @ID ALL USD GBP AUD JPY EUR OTHER
            Exchange
            (monthly)
FX.DAILY    Foreign         D          DATE ALL USD GBP AUD JPY EUR OTHER
            Exchange
```

```
               (daily)
XED.FILES      IRATES
               XRATES
               FX.DAILY
               FX.MONTHLY

5 record(s) listed
```

This `QMQuery` command uses the default dictionary items (*`F1`*, *`F2`*, *`F3`* etc) to display the contents of each item. We can read the values for each file across the display. So, the `FX.DAILY` file has a key structure of ‘D’, and dictionary items of *`@ID ALL USD GBP AUD JPY EUR OTHER`*. Note that the file names in the *`XED.FILES`* item all appear in the *`F1`* field as these are value-mark delimited rather than field-mark delimited.

6.2.3 Reading the control data

Now, to make use of this data.

First of all, we need to open the control file. Add the following line to the first section of skeleton program:

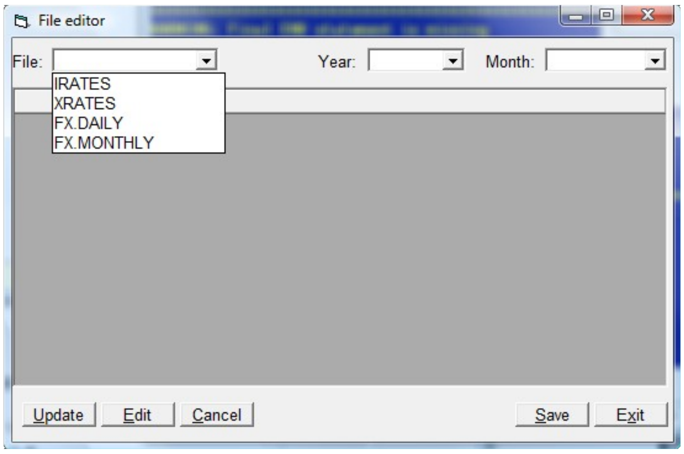
```
OPEN 'XED.VAR' TO xed.var ELSE STOP 201, 'Xed.Var'
```

and in the *`initialise`* section, add the following code:

```
READ xed.files FROM xed.var, 'XED.FILES' ELSE xed.files = ''

CALL ATGUISETPROP('XED', 'FRMMAIN', 'CMBFILE', GPITEMS, 0, 0,
xed.files, guierrrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error
```

You may wonder why we didn’t use the *`GUI.FILE.OPEN`* subroutine that was shown in Section 3.6.1. The simple reason is that program called the `ATGUIMSGBOX` subroutine if an error was encountered – and that requires that the AccuTerm GUI to be running. At the time we are opening the *`XED.VAR`* file, we haven’t actually started the AccuTerm GUI. So, if we attempted to use the *`GUI.FILE.OPEN`* subroutine and there was a problem opening the file, we would generate a set of programming errors by attempting to call the `ATGUIMSGBOX` subroutine without the GUI running.

Save, compile, and run the program. At first glance, the application doesn’t appear any different, but if you click on the down arrow in the files combo box, you will find the list of files available:



Now, the files are displayed in the order in which they were found in the control item. Is this the best order in which to display the files, or would it be better to display them in a sorted (alphabetic) order? On the other hand, if we use a sorted order, then we will need to be very careful adding another file to the control item to make sure that we keep everything in the right order.

To get around these issues, we will put a sort routine into the program. This means that if we need to add another file to the control item, then the sort routine will make sure it appears in the correct place in the visible list.

Add the following two subroutines to the program:

```
sort_files:
*
ofiles = xed.files

xed.files = ''
dc = DCOUNT(ofiles<1>, @VM)
FOR ii = 1 TO dc
  thisfile = ofiles<1, ii>
  LOCATE thisfile IN xed.files<1> BY 'AL' SETTING fpos ELSE
    READV filedesc FROM xed.var, thisfile, 1 THEN
      INS thisfile BEFORE xed.files<1, fpos>
    END
  END
NEXT ii

IF xed.files NE ofiles THEN
  xid = 'XED.FILES'
  xrec = xed.files
  GOSUB update_xed_files
  ofiles = ''
END
*
RETURN
*
* -------------------------------------------------------------- *
*
update_xed_files:
*
writeok = @TRUE
READU dummy FROM xed.var, xid LOCKED
  writeok = @FALSE
END ELSE
  NULL
END

IF writeok THEN
  WRITE xrec ON xed.var, xid
END ELSE
  RELEASE xed.var, xid
END
*
RETURN
```

Now, change the *initialise* subroutine where we read the control item to:

```
READ xed.files FROM xed.var, 'XED.FILES' ELSE xed.files = ''
GOSUB sort_files

CALL ATGUISETPROP('XED', 'FRMMAIN', 'CMBFILE', GPITEMS, 0, 0,
xed.files, guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error
```

The *initialise* subroutine now calls the sort routine with the line `GOSUB` *sort_files*, and passes the sorted list of files to the `ATGUISETPROP` subroutine. In the sort subroutine, the code tests to see if the sorted order of the files is different from that stored on disk. If it is different, then the new sorted order is written back to disk in the *update_xed_files* subroutine.

Note that the *update_xed_files* subroutine uses indirect references for the item-id and item to be saved. This means that this subroutine can be used to save any item into the `XED.VAR` file – not just the *XED.FILES* item.

Compile and run the program. The files should now appear in the combo box in sorted order. Now, exit the program and list the control item using `QMQuery` again. You should find that the control file is now in sorted order too.

6.3 Responding to a file selection

6.3.1 Actions

What needs to happen when a user selects one of the available file names? Some of the things that need to be done include:

- open the dictionary and data portions of the selected file
- read the dictionary to get descriptive information about the data
- update the grid column headings
- populate the year combo box
- activate or de-activate the month combo box as necessary.

Let's start on these:

6.3.2 Opening files

When a user selects a new file from the file combo box, it will trigger the change event (we enabled this event when we first designed the form). Find the event handler for this event in the code skeleton and add the following code:

```
*-->BEGIN EVENT HANDLER<--*
GUI.XED.FRMMAIN.CMBFILE.CHANGE: *
*
CALL ATGUIGETPROP(guiapp, guifrm, guictl, GPVALUE, 0, 0, selfile,
guierrrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

ok = @TRUE
OPEN 'DICT',selfile TO selfile.dict ELSE ok = @FALSE
IF ok THEN
  OPEN selfile TO selfile.data ELSE ok = @FALSE
END

IF ok THEN
  CALL ATGUIMSGBOX('File ':selfile:' opened', 'Open file', MBIICON,
MBOK, '', ok, guierrors, guistate)
  IF guierrors<1> GE 2 THEN GOTO gui.error
END ELSE
  CALL ATGUIMSGBOX('There was a problem opening file ':selfile,
'File error', MBXICON, MBOK, '', ok, guierrors, guistate)
  IF guierrors<1> GE 2 THEN GOTO gui.error
END

RETURN
*
*-->END EVENT HANDLER<--*
```

This block of code gets the selected file name from the combo box using the `ATGUIGETPROP` subroutine. It then tries to open the dictionary and then the data portion of the selected file. If it can open both of these, then the variable *`ok`* will be set to `@TRUE`; otherwise it will be set to `@FALSE`. Finally, it displays a message box telling us whether the files were opened successfully or otherwise.

Save the program, compile it, and run it. Try selecting a file. A message box should appear telling you that the file was opened. Try it with each of the files in turn.

6.3.3 Read the file dictionary

Now, we want to actually do something rather than just display a message when the files are opened correctly. Update the `IF-THEN-ELSE` block with:

```
IF ok THEN
  LOCATE selfile IN xed.files<1> BY 'AL' SETTING fpos THEN
    READ xfiledef FROM xed.var, selfile ELSE xfiledef = ''
    xkeystruct = xfiledef<2>
    xfields = xfiledef<3>
    xfirst = xfiledef<4>
    xlast = xfiledef<5>
  END
  GOSUB read_dict
END ELSE
  CALL ATGUIMSGBOX('There was a problem opening file ':selfile,
'File error', MBXICON, MBOK, '', ok, guierrors, guistate)
  IF guierrors<1> GE 2 THEN GOTO gui.error
END
```

and add the *`read_dict`* subroutine:

```
read_dict:
*
xdamcs = ''
xdnames = ''
xdconvs = ''
xdfmts = ''

colwidths = ''
colaligns = ''
colsizeables = ''

CONVERT ' ' TO @VM IN xfields
dc = DCOUNT(xfields<1>, @VM)
FOR ii = 1 TO dc
  xdid = xfields<1, ii>
  READ xdrec FROM selfile.dict, xdid ELSE xdrec = ''

  xdamcs<1, ii> = xdrec<2>
  xdconvs<1, ii> = xdrec<3>
  xdname = xdrec<4>
  CONVERT @VM TO ' ' IN xdname
  xdnames<1, ii> = TRIM(xdname)
  colfmt = xdrec<5>
  xdfmts<1, ii> = colfmt

  colwidth = OCONV(colfmt, 'MCN')
  temp = LEN(xdname)
  IF temp GT colwidth THEN colwidth = temp
  colwidths<1, ii> = colwidth

  colalign = OCONV(colfmt, 'MCA')
  CONVERT 'LTURC' TO '00012' IN colalign
  colaligns<1, ii> = colalign
  colsizeables<1, ii> = 1
NEXT ii

form_width = SUM(RAISE(colwidths)) + 3
do_resize = @TRUE
GOSUB resize_form

ctrlid     = 'GRDDATA'
property     = GPCOLUMNS     ;   prop.value     = dc
ctrlid<-1> = 'GRDDATA'
property<-1> = GPFIXEDCOLS   ;   prop.value<-1> = 1
ctrlid<-1> = 'GRDDATA'
property<-1> = GPCOLWIDTH    ;   prop.value<-1> = colwidths
ctrlid<-1> = 'GRDDATA'
property<-1> = GPCOLALIGN    ;   prop.value<-1> = colaligns
ctrlid<-1> = 'GRDDATA'
property<-1> = GPCOLHEADING  ;   prop.value<-1> = xdnames
```

```
ctrlid<-1> = 'GRDDATA'
property<-1> = GPCOLSIZABLE  ;   prop.value<-1> = colsizeables

CALL ATGUISETPROPS(guiapp, guifrm, ctrlid, property, prop.value,
guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error
*
RETURN
```

The first thing that happens is that we `LOCATE` the selected file in our control variable. If the file is found, then we read the file definition from the control file, and extract the key structure and fieldnames. Note we also read two variables named *`xfirst`* and *`xlast`* representing the first and last item ids. These aren't in the file yet, but soon will be. Finally, we call the subroutine to read the file dictionary.

This subroutine begins by initialising a number of variables, before counting the number of fields to display, and setting up a `FOR-NEXT` loop to read the fields from the dictionary. Once a dictionary item has been read, the various parts of the item are assigned to variables for later use in the program.

Once all fields have been read, the subroutine calculates a total width for the form, and calls the *`resize_form`* subroutine to set the form to the new width. Note that the *`do_resize`* variable is set to `@TRUE` to allow the *`resize_form`* subroutine to do the resize.

Finally, the grid headings and other grid properties are updated using the `ATGUISETPROPS` subroutine.

Let's look at some bits in a bit more detail:

```
  xdname = xdrec<4>
  CONVERT @VM TO ' ' IN xdname
  xdnames<1, ii> = TRIM(xdname)
```

This fragment gets the column heading text from the dictionary, and then converts any value marks to spaces. Finally, the text is trimmed and stored for later use. The conversion in the middle is convert any multi-line column headings into a single line of text, while the `TRIM` in the final line removes any excess whitespace.

```
  colfmt = xdrec<5>

  colwidth = OCONV(colfmt, 'MCN')
  temp = LEN(xdname)
  IF temp GT colwidth THEN colwidth = temp
  colwidths<1, ii> = colwidth
```

This code fragment starts by getting the format expression from the dictionary item. This is typically a string like `7R`, meaning right-justify the data if a field 7 characters wide. The `MCN` conversion in the `OCONV` function retrieves the numeric part of the format expression, thus setting the initial value of the *`colwidth`* variable. Then it calculates the length of the column heading, and assigns that value to *`colwidth`* if it is greater than the column width already established.

```
  colalign = OCONV(colfmt, 'MCA')
  CONVERT 'LTURC' TO '00012' IN colalign
  colaligns<1, ii> = colalign
```

This fragment extracts the alphabetic characters from the format expression. This should be the justification of the field. The convert function then converts that justification to the alignment value expected by *AccuTerm*.

This `CONVERT` function probably needs a little more explanation. Each of the characters in the in the left-hand string (*`LTURC`*) are converted to the matching characters by position in the right-hand string (*`00012`*) in the *`colalign`* dynamic array. Therefore, if a column is left-aligned, it will have a value in the `colalign` variable of `L` and this will be converted to `0`. See ‘Format specifications’ in the online help to find out what each of the other alignment codes do.

If you study the documentation for format expressions, you will find that this code will only work with relatively simple expressions. If more complex expressions are used in the dictionary items, then either the code could be extended to correctly interpret those expressions, or alternative dictionary items could be provided for use with this application.

If you compile and run the program now, you will find that the column headings and the form width will change as you select each of the files.

6.3.4 Populate the combo boxes

Now we need to populate the `Year` and `Month` combo boxes. There are a few issues to sort out here:

- How do we find the start and end points of the dataset so that we can populate the combo boxes quickly?
- We don’t actually want the `Month` combo box active all the time. Filtering data by month is clearly too restrictive if there is only one observation per month
- For those files where the `Month` combo box will be active, we don’t actually know the number of months to display until the year has been selected. So, all we’ll do at this stage is clear any list data from the control.

All of the above will run out of a new subroutine called `setup_combo_boxes`. So start by putting the `GOSUB` for this after the `GOSUB` for `read_dict` in the file change event handler.

```
GOSUB read_dict
GOSUB setup_combo_boxes
```

Now, we need to create this new subroutine. Enter the following code:

```
setup_combo_boxes:
*
IF xfirst = '' THEN
  GOSUB get_first_last
END

IF xkeystruct = 'D' THEN
  firstyear = OCONV(xfirst, 'DY')
  lastyear = OCONV(xlast, 'DY')
  enabled = @TRUE
END ELSE
  firstyear = xfirst[1, 4]
  lastyear = xlast[1, 4]
  enabled = @FALSE
END

years = ''
FOR thisyear = firstyear TO lastyear
  years<1, -1> = thisyear
NEXT thisyear

CALL ATGUICLEAR(guiapp, guifrm, 'CMBYEAR', guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

CALL ATGUICLEAR(guiapp, guifrm, 'CMBMONTH', guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

ctrlid      = 'CMBYEAR'
property      = GPITEMS    ;   prop.value     = years
ctrlid<-1> = 'LBLMONTH'
```

```
property<-1> = GPENABLED  ;   prop.value<-1> = enabled
ctrlid<-1> = 'CMBMONTH'
property<-1> = GPENABLED  ;   prop.value<-1> = enabled

CALL ATGUISETPROPS(guiapp, guifrm, ctrlid, property, prop.value,
guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error
*
RETURN
```

This subroutine starts by testing the value of `xfirst`. This value will only be null if we haven't previously run the editor for that file, so most of the time, the logic will skip over the subroutine call here.

If the value of `xfirst` is null, then there is a call to a subroutine to create the values for `xfirst` and `xlast`. We'll return to create this subroutine shortly. In the meantime, we'll simply assume that those values have been created.

We now want to create a list of years with which to populate the combo box. As we have two different item-id formats, we need to put the logic for deriving the start and end years into an `IF-THEN-ELSE` block. We also set an `enabled` variable in this block to enable or disable the `Month` combo box.

Once we have the first and last years, the list of years is created using a '`FOR-NEXT`' loop.

Finally, there are a set of calls to `ATGUIxxx` subroutines. The first two calls clear any existing values for the `Year` and `Month` combo boxes. The next call loads the year list into the `Year` combo box, and sets the enabled status of the `Month` combo box (and its label).

Now, we need to go back and create the `get_first_last` subroutine that we referenced in this subroutine. Enter the following code:

```
get_first_last:
*
SELECT selfile.data
LOOP
  READNEXT data.id ELSE EXIT
  BEGIN CASE
    CASE xfirst = ''
      xfirst = data.id
      xlast = data.id
    CASE data.id LT xfirst
      xfirst = data.id
    CASE data.id GT xlast
      xlast = data.id
  END CASE
REPEAT

xfiledef<4> = xfirst
xfiledef<5> = xlast

xid = selfile
xrec = xfiledef
GOSUB update_xed_files
*
RETURN
```

The first part of this code selects the data file that we opened in the file change event handler. It then loops through the select list reading the item-ids. If an item-id is less than the current value of `xfirst` then it assigns that value to `xfirst`. Likewise, if an item-id is greater than the current value of `xlast` then it assigns that value to `xlast`. Of course, we need to make sure that both of these variables have values to test against, and that is done by the first branch in the `CASE` statement. This branch is taken if `xfirst` is null (which will only be true on the first loop), and assigns the first item-id to both `xfirst` and `xlast`.

The next two lines assign the *xfirst* and *xlast* variables back to the file definition item. Finally, the *update_xed_files* subroutine is called to write this control item back to disk.

If you run the program now, you should find that:

- the list of years should be populated when you choose a file to edit, and that the list of years should match the years in the data file
- the *Month* combo box should be enabled and disabled according to the file you have selected.

If you now exit from the program, and copy the contents of the control item to the terminal, you should see that the first and last item values have been saved for each file:

```
CT XED.VAR XRATES
XED.VAR XRATES
1: Exchange rates (month avg)
2: YYYYMM
3: @ID TWI USD GBP AUD JPY EUR GDM
4: 198501
5: 200904
```

6.4 Responding to a year selection

When a user selects a year to display, one of two things must happen. Either:

- the data for that year will be displayed in the grid; or
- the *Month* combo box will be populated with the months for that year.

We'll start with populating the combo box. For the first year in the list, the months to display will start from the month of the first item-id, and go to the last month of the year. For the last year in the list, months will start with the first month of the year, and go the month of the last item-id. Of course, the first year could be the same as the last year, in which case the start and end points may not be the end months of the year – or there may be no data at all. We'll start with the "normal" situations and deal with any issues later.

Enter the following code into the change year event handler:

```
*-->BEGIN EVENT HANDLER<--*
GUI.XED.FRMMAIN.CMBYEAR.CHANGE: *
*
CALL ATGUIGETPROP(guiapp, guifrm, guictl, GPVALUE, 0, 0, selyear,
guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

IF xkeystruct EQ 'D' THEN
  GOSUB setup_months
END ELSE
  NULL
END

RETURN
*
*-->END EVENT HANDLER<--*
```

and create the the *setup_months* subroutine:

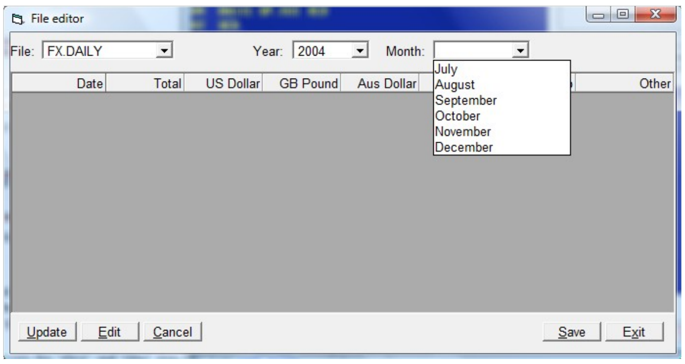
```
setup_months:
*
start_ym = OCONV(xfirst, 'DY') * 100 + OCONV(xfirst, 'DM')
last_ym = OCONV(xlast, 'DY') * 100 + OCONV(xlast, 'DM')

months = ''
FOR mth = 1 TO 12
  this_ym = selyear * 100 + mth
  IF (this_ym GE start_ym) AND (this_ym LE last_ym) THEN
```

```
    months<1, -1> = OCONV(ICONV('15/':mth:'/':selyear, 'D'),
'DMAL')
  END
NEXT mth

CALL ATGUICLEAR(guiapp, guifrm, 'CMBMONTH', guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

IF months THEN
  CALL ATGUISETPROP(guiapp, guifrm, 'CMBMONTH', GPITEMS, 0, 0,
months, guierrors, guistate)
  IF guierrors<1> GE 2 THEN GOTO gui.error
END
*
RETURN
```

The event handler is straight-forward. We get the selected year from the combo box. Then, if the key structure is day based, then we set up the *`Month`* combo box. If we have a month based key structure, then we put a `NULL` statement in as a place holder. We will fill this in properly later.

The *`setup_months`* subroutine converts the *`xfirst`* and *`xlast`* item-id's to a *`YYYYMM`* representation of those dates. It then loops through the selected year, creating a *`YYYYMM`* representation of each month. If that month is within the range defined by the first and last months, then we include that month in our list of months. Finally, it loads the month list into the combo box.

If we test this code, we find that it gives partial lists of months for the start and end years, but a full list of months for the intervening years:


File editor

File: FX.DAILY Year: 2004 Month:

Date	Total	US Dollar	GB Pound	Aus Dollar	Other

July
August
September
October
November
December

Now we need to fill in the other branch from the event handler – the one where we simply entered `NULL` earlier. Update the event handler to read:

```
IF xkeystruct EQ 'D' THEN
  GOSUB setup_months
END ELSE
  GOSUB read_data_months
  GOSUB load_grid
END
```

This adds to two new subroutines. These look like:

```
read_data_months:
*
gdata = ''
rowno = 0
gridcols = DCOUNT(xfields<1>, @VM)
FOR mth = 1 TO 12
  dataid = selyear * 100 + mth
  READ datarec FROM selfile.data, dataid THEN
    rowno += 1
    FOR colno = 1 TO gridcols
      amc = xdamcs<1, colno>
      IF amc EQ 0 THEN
        gdata<1, rowno, colno> = OCONV(dataid, xdconvs<1, colno>)
      END ELSE
        gdata<1,rowno,colno> = OCONV(datarec<amc>,xdconvs<1,colno>)
      END
    NEXT colno
  END
NEXT mth
*
RETURN
```

and:

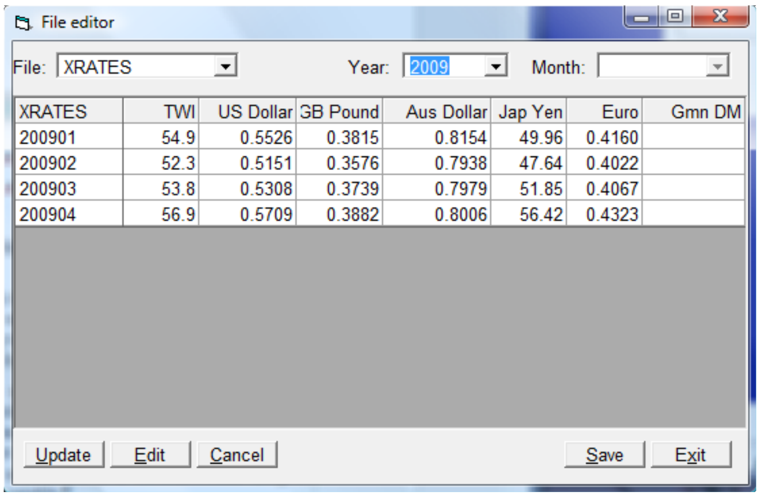
```
load_grid:
*
CALL ATGUISETPROP(guiapp, guifrm, 'GRDDATA', GPVALUE, 0, 0, gdata,
guierrrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error
*
RETURN
```

The read data subroutine reads a years worth of data from the monthly data files – as long as the relevant item is on disk.

It uses two loops to do this. The first loop builds the item-id from the selected year and the month number, while the second loop loads the selected fields into the variable *gdata*. As the first loop doesn't actually know if the item exists until such time as it attempts to read the item, we use another variable *rowno* as a counter to load the data into the *gdata* variable.

The second loop is only executed if the item exists on disk. Once this has been read successfully, the loop goes through the list of fields that were taken from the control item and gets the field number from the information that was read from the dictionary. If the field number is zero, then we store the item-id in the relevant position in the *gdata* variable; otherwise, we get the field data from the record and convert it to external format using the OCONV function before storing it in the *gdata* variable.

Now compile and run the program. Select a monthly file and a year, and the grid loads with the selected data. Change the year and the data changes to match. Everything appears to be OK – or is it?



File: XRATES Year: 2009 Month:

XRATES	TWI	US Dollar	GB Pound	Aus Dollar	Jap Yen	Euro	Gmn DM
200901	54.9	0.5526	0.3815	0.8154	49.96	0.4160	
200902	52.3	0.5151	0.3576	0.7938	47.64	0.4022	
200903	53.8	0.5308	0.3739	0.7979	51.85	0.4067	
200904	56.9	0.5709	0.3882	0.8006	56.42	0.4323	

Try changing the file. The grid column headings change, and the grid resizes itself … but the data stays the same! So what should happen here?

When we change the file name, we reset the combo boxes for year and month. So the simple thing to do is to clear the grid at the same time. The alternative would be to:

- check that the new file has the same key structure as the previously selected file; and

- check that the new file has the same year (and month) available in its dataset as the previously selected file.
- If the above conditions are met, then read the equivalent data into the grid; otherwise, clear the grid.

At this stage, it is much easier to simply clear the grid. There are a couple of ways we could do this:

- simply issue an `ATGUICLEAR` command
- set the value of *`gdata`* to an empty string, and call the *`load_grid`* subroutine.

Either of these options is fine.

You may have noted that the `ATGUICLEAR` subroutine is called at a number of points through this application. In all cases so far, the only difference between the calls is the name of the control. So we could put this external call into a separate internal subroutine, and simply call that subroutine with the appropriate control name. This would look like:

```
clear_control:
*
CALL ATGUICLEAR(guiapp, guifrm, ctrlid, guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error
*
RETURN
```

and we would call this as:

```
ctrlid = 'GRDDATA'
GOSUB clear_control
```

So, replace the occurrences of `ATGUICLEAR` with appropriate internal subroutine calls, and then add a call to clear the grid in the file change event handler.

6.5 Responding to a month selection

The form should now respond correctly to all the user inputs for the monthly files. Now, we have to do the same for the daily files.

The *`load_grid`* routine should work just fine for the daily data, so all we need to do is write a *`read_data_daily`* routine, and call it from the month change event handler. This would make the event handler code look like:

```
*-->BEGIN EVENT HANDLER<--*
GUI.XED.FRMMAIN.CMBMONTH.CHANGE: *
*
CALL ATGUIGETPROP(guiapp, guifrm, guictl, GPVALUE, 0, 0, selmonth,
guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

GOSUB read_data_daily
GOSUB load_grid

RETURN
*
*-->END EVENT HANDLER<--*
```

and the *`read_data_daily`* routine could look like:

```
read_data_daily:
*
gdata = ''
dataid = ICONV('01 ':selmonth:' ':selyear, 'D')
monthno = OCONV(dataid, 'DM')
rowno = 0
```

```
gridcols = DCOUNT(xfields<1>, @VM)
LOOP
  READ datarec FROM selfile.data, dataid THEN
    rowno += 1
    FOR colno = 1 TO gridcols
      amc = xdamcs<1, colno>
      IF amc EQ 0 THEN
        gdata<1, rowno, colno> = OCONV(dataid, xdconvs<1, colno>)
      END ELSE
        gdata<1,rowno,colno> = OCONV(datarec<amc>,xdconvs<1,colno>)
      END
    NEXT colno
  END

  dataid += 1
  chkmonth = OCONV(dataid, 'DM')
  IF chkmonth NE monthno THEN EXIT
REPEAT
*
RETURN
```

Note that the central portion of this subroutine is identical to the equivalent portion of the *read_data_months* subroutine. Let's pull that section out to create a new subroutine called *load_row*. Note this subroutine requires a test for the key structure so that the item-id is displayed appropriately:

```
load_row:
*
rowno += 1
FOR colno = 1 TO gridcols
  amc = xdamcs<1, colno>
  IF amc EQ 0 THEN
    gdata<1, rowno, colno> = OCONV(dataid, xdconvs<1, colno>)
  END ELSE
    gdata<1, rowno, colno> = OCONV(datarec<amc>, xdconvs<1, colno>)
  END
NEXT colno
*
RETURN
```

This shrinks the *read_data_daily* routine to:

```
read_data_daily:
*
gdata = ''
dataid = ICONV('01 ':selmonth:' ':selyear, 'D')
monthno = OCONV(dataid, 'DM')
rowno = 0
gridcols = DCOUNT(xfields<1>, @VM)
LOOP
  READ datarec FROM selfile.data, dataid THEN
    GOSUB load_row
  END

  dataid += 1
  chkmonth = OCONV(dataid, 'DM')
  IF chkmonth NE monthno THEN EXIT
REPEAT
*
RETURN
```

and the *read_data_months* subroutine can have the central portion replaced by the call to the *load_row* subroutine also.

Let's examine the *read_data_daily* subroutine a little more closely. This essentially works by converting the first day of the selected month to a day number, and then looping through the month by incrementing the day number by one on each loop. The loop terminates when the month changes.

Why do it this way?

- Months have variable numbers of days, so it would be awkward to use a `FOR-NEXT` loop
- By terminating the loop when the month changes, we test every valid date within the month. This ensures that all the items on file are read – even if the sequence of item-id's is not complete.

If you test the program now, you should find that it mostly works OK – but when we change the year while displaying daily data, we strike a similar problem to that encountered earlier when changing the selected file – the data in the grid doesn't change. The answer is to add code to clear the grid when we change the year.

6.6 Editing data

The form now reads and displays data correctly. We now want to let the user edit that data, and to enter new data.

6.6.1 Edit buttons

When we designed the form, we placed a number of buttons along the bottom of the form. So far, we have only activated the 'Exit' button. Let's look at the other buttons now, and consider how they are to be used:

- The 'Update' button will be used to refresh the data from the internet. We want this to be active when data is displayed, but not when it is being edited.
- The 'Edit' button will be used to unlock the grid for editing. We want this to be active when data is displayed, but not when it is being edited.
- The 'Cancel' button will be used to cancel changes made during editing. This button will only be active when the data is being edited. Pressing 'Cancel' should restore the data to the state on disk.
- The 'Save' button will be used to save changes made during editing. This button will only be active when data is being edited.
- The 'Exit' button should be disabled while we are editing data.

The above implies that most of these buttons should be inactive until data is displayed. To that end, we need to disable these buttons in the *`initialise`* subroutine, and enable them during the *`load_grid`* subroutine. We also need to disable the buttons when we clear the grid. So, because we need to disable the buttons from multiple places within the application, we'll put that code in its own subroutine:

```
disable_buttons:
*
ctrlid      = 'BTNCANCEL'
property      = GPENABLED    ;    prop.value      = 0
ctrlid<-1> = 'BTNEDIT'
property<-1> = GPENABLED    ;    prop.value<-1> = 0
ctrlid<-1> = 'BTNSAVE'
property<-1> = GPENABLED    ;    prop.value<-1> = 0
ctrlid<-1> = 'BTNUPDATE'
property<-1> = GPENABLED    ;    prop.value<-1> = 0

CALL ATGUISETPROPS(guiapp, guifrm, ctrlid, property, prop.value,
guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

RETURN
```

Now, add a call to this subroutine at the end of the *`initialise`* subroutine.

Put the code to enable some of the buttons in the *`load_grid`* subroutine:

```
ogdata = gdata

GOSUB disable_buttons
ctrlid     = 'BTNEDIT'
property      = GPENABLED    ;   prop.value     = 1
ctrlid<-1> = 'BTNUPDATE'
property<-1> = GPENABLED    ;   prop.value<-1> = 1

CALL ATGUISETPROPS(guiapp, guifrm, ctrlid, property, prop.value,
guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error
```

and, finally, add the following code to the end of the `clear_control` subroutine:

```
IF ctrlid = 'GRDDATA' THEN
  GOSUB disable_buttons
END
```

The first two bits of code are mostly self-explanatory. They build lists of control, properties, and property values which then get passed to the set properties subroutine. Note that we've called the `disable_buttons` subroutine before we selectively enable some of the buttons. This is because users could change the display by selecting a new file, year, or month, and putting this call here ensures the correct state of the buttons as the data is displayed.

However, there is also a line right at the start of this code that takes a copy of the data we loaded into the grid (not shown). We need this copy of the data to ensure we can easily restore the grid to its original state if we press 'Cancel'.

The final bit of code obviously disables the buttons – but why did we put the call inside an IF statement? The reason is that that this subroutine gets called to clear a number of controls on the form – but we only want the buttons disabled if we clear the grid.

6.6.2 Enabling editing in the grid

If you've tried clicking on a cell in the grid so far, you will have noted that *AccuTerm* places a border around the cell and allows you to change the data in the cell (if you can't do this, you probably have the grid style set to 'Protected' in the GUI Designer). However, it doesn't apply any formatting to the data entered, and it doesn't save any changes. We have to add code to do those things.

First, we need to understand the settings that allow editing in the data grid. The key elements are:

- If you want to edit data in the grid, then the grid must be made 'editable' at design time
- If the grid is editable, then you can enable or disable editing by setting the `GPENABLED` property for the grid. Unfortunately, this completely disables the grid, and means that you cannot scroll the grid using the scroll bars (they are disabled)
- Individual columns can be made editable by setting the `GPREADONLY` property for the column
- If you want to add new data to the grid, then you need to set the `GPSTYLE` property to 1 (or `@TRUE`). This auto-extends the grid allowing you to add a new row.

In terms of this application, we want to be able to:

- edit data in the grid
- enable and disable the grid for editing
- be able to scroll the grid even when editing is disabled

- add new rows of data when we are at the end of the data on file.

The approach to achieve this is to:

- set the grid to 'editable' at design time
- make all columns read-only for normal display
- make all columns not read-only for editing
- set the grid to auto-extend when the last screen of data is displayed.

We'll create a new subroutine *disable_editing* so that we call this from more than one place without duplicating code:

```
disable_editing:
*
temp = ''
FOR ii = 1 TO gridcols
  temp<1, ii> = @TRUE
NEXT ii

CALL ATGUISETPROP(guiapp, guifrm, 'GRDDATA', GPREADONLY, 0, 0,
temp, guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error
*
RETURN
```

This uses the variable *gridcols* which was defined in the *read_data_xxx* subroutines. Therefore, we need to call this subroutine after we have read the data to display. The logical thing to do is call this subroutine as we display the data – i.e. from the *load_grid* subroutine.

The above subroutine makes every column read-only.

Now, we need to enable editing when we click on the 'Edit' button. We also need to change the enabled state of the various buttons. Add the following code to the edit button event handler:

```
*-->BEGIN EVENT HANDLER<--*
GUI.XED.FRMMAIN.BTNEDIT.CLICK: *
*
ctrlid      = 'BTNCANCEL'
property      = GPENABLED    ;   prop.value      = @TRUE
ctrlid<-1> = 'BTNEDIT'
property<-1> = GPENABLED    ;   prop.value<-1> = @FALSE
ctrlid<-1> = 'BTNEXIT'
property<-1> = GPENABLED    ;   prop.value<-1> = @FALSE
ctrlid<-1> = 'BTNSAVE'
property<-1> = GPENABLED    ;   prop.value<-1> = @TRUE
ctrlid<-1> = 'BTNUPDATE'
property<-1> = GPENABLED    ;   prop.value<-1> = @FALSE
ctrlid<-1> = 'GRDDATA'
property<-1> = GPSTYLE      ;   prop.value<-1> = 1

temp = ''
FOR ii = 1 TO gridcols
  temp<1, ii> = @FALSE
NEXT ii
ctrlid<-1> = 'GRDDATA'
property<-1> = GPREADONLY   ;   prop.value<-1> = temp

CALL ATGUISETPROPS(guiapp, guifrm, ctrlid, property, prop.value,
guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

RETURN
*
*-->END EVENT HANDLER<--*
```

and this code to the cancel button event handler:

```
*-->BEGIN EVENT HANDLER<--*
GUI.XED.FRMMAIN.BTNCANCEL.CLICK: *
*
ctrlid      = 'BTNCANCEL'
property      = GPENABLED    ;   prop.value      = @FALSE
ctrlid<-1> = 'BTNEDIT
property<-1> = GPENABLED    ;   prop.value<-1> = @TRUE
ctrlid<-1> = 'BTNEXIT'
property<-1> = GPENABLED    ;   prop.value<-1> = @TRUE
ctrlid<-1> = 'BTNSAVE'
property<-1> = GPENABLED    ;   prop.value<-1> = @FALSE
ctrlid<-1> = 'BTNUPDATE'
property<-1> = GPENABLED    ;   prop.value<-1> = @TRUE

CALL ATGUISETPROPS(guiapp, guifrm, ctrlid, property, prop.value,
guierrrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

GOSUB disable_editing

RETURN
*
*-->END EVENT HANDLER<--*
```

Most of the code in both event handlers is simply setting the state of the various buttons. However, the code for the edit button also sets the `GPSTYLE` property which allows new data to be added to the grid, and the `FOR-NEXT` loop creates a variable to set the `GPREADONLY` property for each column to `@FALSE`. In contrast, the handler for the cancel button calls the ***`disable_editing`*** subroutine to reset the columns to read-only.

Now, the above code (in the 'Edit' button event handler) enables the addition of new rows every time you click on the 'Edit' button. But this isn't really what we want – we only want to enable the addition of new data if we are on the last screen.

First of all, update that line to read:

```
IF lastscreen THEN
  ctrlid<-1> = 'GRDDATA'
  property<-1> = GPSTYLE      ;   prop.value<-1> = 1
END ELSE
  ctrlid<-1> = 'GRDDATA'
  property<-1> = GPSTYLE      ;   prop.value<-1> = 0
END
```

This makes the grid style conditional on a variable named *`lastscreen`*. Now we need to set this variable to `@TRUE` or `@FALSE` as necessary. The place to set this variable is in the read routines. This requires two new lines in each of the two read routines. The ***`read_data_daily`*** subroutine is shown below with these two new lines:

```
read_data_daily:
*
gdata = ''
dataid = ICONV('01 ':selmonth:' ':selyear, 'D')
monthno = OCONV(dataid, 'DM')
rowno = 0
lastscreen = @FALSE
gridcols = DCOUNT(xfields<1>, @VM)
LOOP
  READ datarec FROM selfile.data, dataid THEN
    GOSUB load_row
  END
  IF dataid GE xlast THEN lastscreen = @TRUE

  dataid += 1
  chkmonth = OCONV(dataid, 'DM')
  IF chkmonth NE monthno THEN EXIT
REPEAT
```

```
*
RETURN
```

The routine now initialises the *lastscreen* variable to `@FALSE`, and then tests the values being read to determine if we are on the last data screen. The same two lines need to be added to the *read_data_months* subroutine.

This still isn't quite right. When you click on the 'Edit' button, there is no real sign that the grid is the active element. This is because the 'Edit' button still has the focus – so we need to shift the focus to the grid as part of the event handler. This also means that we should set the cell reference within the grid that is active.

Update the *lastscreen* portion of the 'Edit' button event handler as follows:

```
IF lastscreen THEN
  ctrlid<-1> = 'GRDDATA'
  property<-1> = GPSTYLE      ;   prop.value<-1> = 1
  ctrlid<-1> = 'GRDDATA'
  property<-1> = GPROW        ;   prop.value<-1> = gridrows + 1
END ELSE
  ctrlid<-1> = 'GRDDATA'
  property<-1> = GPSTYLE      ;   prop.value<-1> = 0
  ctrlid<-1> = 'GRDDATA'
  property<-1> = GPROW        ;   prop.value<-1> = gridrows
END
ctrlid<-1> = 'GRDDATA'
property<-1> = GPCOLUMN     ;   prop.value<-1> = 1
```

and add the following code at the end of the event handler:

```
CALL ATGUIACTIVATE(guiapp, guifrm, 'GRDDATA', guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error
```

Finally, add the following line to the end of the two *read_data_xxx* subroutines:

```
gridrows = rowno
```

When we run the application now, the *gridrows* variable will be set when we read the data for the grid. When we click on the edit button, the active cell will be set as the last row of the first column – and the last row may either be the last row of actual data, or a new row with no data if we are on the last screen of data.

6.6.3 Grid formatting and validation

If you run the program now, you should be able to click on the 'Edit' button, and then enter or change data in the grid. Clicking on the 'Cancel' button should then reset the grid so that you cannot enter further data.

However, you can enter anything into the cells, and even if you do enter valid data, it isn't formatted correctly. For example, the interest rates are displayed with two decimal places – e.g. 7.64 per cent. If we enter a number with just one decimal place – say 7.6, then this will simply be displayed as we entered it – it won't be reformatted to display 7.60 per cent. Similarly, if we enter a value as 7.6225, this won't be rounded back to 7.62 per cent.

AccuTerm provides some limited data checking on input via the `GPCOLDATATYPE` property. For example, if you attempt to enter a value of 'abc' into a column defined as `GDCURRENCY`, then a pop-up box will appear telling you this is not valid. However, it doesn't appear to restrict a 4 decimal place number from being entered into a `GDCURRENCY` field, or a 6 decimal place number from being entered into a `GDFINANCIAL` field (even though the documentation notes that this data type is for up to 4 decimal places).

Given this limited ability to validate data within the grid, the best option is to enable some of the validation events and write code to validate and reformat the entered data. This has its own set of issues – specifically, reloading the grid with a reformatted set of data can stop some of the validation events firing.

Some notes on the validation events follow:

- The most relevant events are: Change (cell), Validate Cell, Validate Row
    - The 'Change' event fires whenever you finish changing a value in a cell. It is not like the change event in an edit box where the event fires with every keystroke. The difference between this and the 'Validate Cell' event is that you do not need to move off the cell.
    - The 'Validate Cell' event fires when you have changed a cell value AND have moved to a new cell or control on the form.
    - The 'Validate Row' event fires when you move to a new row and have changed a cell on the current row.
- The sequence of these events is: Change, Validate Cell, Validate Row.
- If your 'Enter key behaviour' is set to 'Normal' in the application properties, then you probably need to enable both the 'Change' and 'Validate Cell' properties:
    - This is to ensure that a validation event occurs when you change a cell and press Enter. In this situation, only the 'Change' event fires.
    - However, if you change a cell, and terminate that change by pressing the 'Tab' key, then both the 'Change' and 'Validate Cell' events will fire.
    - If you terminate your cell entry with the 'Up arrow', then the 'Change', 'Validate Cell' and 'Validate Row' events will fire.
- If you:
    - change the contents of a cell; and
    - reformat your entered data in the 'Change' or 'Validate Cell' event handlers; and
    - update the contents of the grid to display the reformatted data; and
    - move to a new cell in the same row, and then move to a different row, then the 'Validate Row' event will NOT fire.
- However, if you do not update the grid contents with reformatted data, then the 'Validate Row' event WILL fire properly when you move off the row.
- Probably, the best way to maintain reasonable data formatting during data entry is to validate and reformat the data as it is entered by using the 'Change' and 'Validate Cell' event handlers – but don't refresh the grid with that reformatted data until the 'Validate Row' event fires.

Now to put all of the above into effect. We'll start with *AccuTerm*'s data type setting, which we may as well use, even if it is of limited use. The grid properties are all set in the ***read_dict*** subroutine, so we need to update that subroutine to set the column data type. The entire subroutine is shown below, as there are changes spread right through the subroutine:

```
read_dict:
*
xdamcs = ''
xdnames = ''
xdconvs = ''
xdfmts = ''
xdtype = ''

colwidths = ''
colaligns = ''
```

```
colsizeables = ''

CONVERT ' ' TO @VM IN xfields
dc = DCOUNT(xfields<1>, @VM)
FOR ii = 1 TO dc
  xdid = xfields<1, ii>
  READ xdrec FROM selfile.dict, xdid ELSE xdrec = ''

  xdamcs<1, ii> = xdrec<2>
  xdconv = xdrec<3>
  xdconvs<1, ii> = xdconv
  xdname = xdrec<4>
  CONVERT @VM TO ' ' IN xdname
  xdnames<1, ii> = TRIM(xdname)
  colfmt = xdrec<5>
  xdfmts<1, ii> = colfmt

  colwidth = OCONV(colfmt, 'MCN')
  temp = LEN(xdname)
  IF temp GT colwidth THEN colwidth = temp
  colwidths<1, ii> = colwidth

  colalign = OCONV(colfmt, 'MCA')
  CONVERT 'LTURC' TO '00012' IN colalign
  colaligns<1, ii> = colalign
  colsizeables<1, ii> = 1

  dp = OCONV(xdconv, 'MCN')
  IF dp GT 9 THEN dp = dp[1,1]
  BEGIN CASE
    CASE dp EQ 2  ;  datatype = GDCURRENCY
    CASE dp AND dp LE 4  ;  datatype = GDFINANCIAL
    CASE dp       ; datatype = GDNUMERIC
    CASE xdconv EQ 'D'  ; datatype = GDDATE
    CASE 1        ;  datatype = GDANY
  END CASE
  xdtype<1, ii> = datatype
NEXT ii

form_width = SUM(RAISE(colwidths)) + 3
do_resize = @TRUE
GOSUB resize_form

ctrlid     = 'GRDDATA'
property     = GPCOLUMNS     ;   prop.value     = dc
ctrlid<-1> = 'GRDDATA'
property<-1> = GPFIXEDCOLS   ;   prop.value<-1> = 1
ctrlid<-1> = 'GRDDATA'
property<-1> = GPCOLWIDTH    ;   prop.value<-1> = colwidths
ctrlid<-1> = 'GRDDATA'
property<-1> = GPCOLALIGN    ;   prop.value<-1> = colaligns
ctrlid<-1> = 'GRDDATA'
property<-1> = GPCOLHEADING  ;   prop.value<-1> = xdnames
ctrlid<-1> = 'GRDDATA'
property<-1> = GPCOLSIZABLE  ;   prop.value<-1> = colsizeables
ctrlid<-1> = 'GRDDATA'
property<-1> = GPCOLDATATYPE ;   prop.value<-1> = xdtype

CALL ATGUISETPROPS(guiapp, guifrm, ctrlid, property, prop.value,
guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error
*
RETURN
```

The largest change in the subroutine is the inclusion of the `CASE` block used to assign a data type to the column based on the number of decimals defined for display in the field's output conversion. This `CASE` block uses the variable *`xdconv`* which needs to be assigned earlier in the subroutine, and finally the dynamic array of column data types *`xdtype`* needs to be included in the list of properties to be set via the `ATGUISETPROPS` subroutine.

Now, open the form in the GUI Designer, select the data grid, and select ‘Properties’. On the ‘Events’ tab, check the ‘Change’, ‘Validate Cell’, and ‘Validate Row’ events, the click on ‘Apply’. Update the code and the event decoder, then save both the code and the updated form design.

Enter the following code in the grid change event handler:

```
*-->BEGIN EVENT HANDLER<--*
GUI.XED.FRMMAIN.GRDDATA.CHANGE: *
*
gridcol = guiargs<1, 1>
gridrow = guiargs<1, 2>
editvalue = guiargs<2>

ctrlid     = 'GRDDATA'  ;  property     = GPCOLUMN
ctrlid<-1> = 'GRDDATA'  ;  property<-1> = GPROW
CALL ATGUIGETPROPS(guiapp, guifrm, ctrlid, property, prop.value,
guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error
thiscolumn = prop.value<1>
thisrow = prop.value<2>

IF (thiscolumn EQ gridcol) AND (thisrow EQ gridrow) THEN
  GOSUB validate_cell
END

RETURN
*
*-->END EVENT HANDLER<--*
```

and the code for the validate cell event handler is:

```
*-->BEGIN EVENT HANDLER<--*
GUI.XED.FRMMAIN.GRDDATA.VALIDATECELL: *
*
gridcol = guiargs<1, 1>
gridrow = guiargs<1, 2>
editvalue = guiargs<2>

GOSUB validate_cell

RETURN
*
*-->END EVENT HANDLER<--*
```

These two event handlers both call the `validate_cell` subroutine:

```
validate_cell:
*
xconv = xdconvs<1, gridcol>
fmtvalue = ICONV(editvalue, xconv)
fmtvalue = OCONV(fmtvalue, xconv)

IF gridcol = 1 THEN
  iderr = @FALSE
  IF xkeystruct = 'D' THEN
    NULL
  END ELSE
    fmtvalue = OCONV(fmtvalue, 'MCN')
    yyyy = INT(fmtvalue / 100)
    mm = MOD(fmtvalue, 100)
    IF mm LT 1 OR mm GT 12 THEN iderr = @TRUE
    IF LEN(yyyy) NE 4 THEN iderr = @TRUE
  END
  IF iderr THEN
    CALL ATGUIMSGBOX(editvalue:' is not a valid ID', 'ID error',
MBIICON, MBOK, '', ok, guierrors, guistate)
    IF guierrors<1> GE 2 THEN GOTO gui.error

    GOSUB grid_pos
  END
```

```
END

gdata<1, gridrow, gridcol> = fmtvalue
*
RETURN
```

which in turn calls the `grid_pos` subroutine:

```
grid_pos:
*
ctrlid      = 'GRDDATA'
property      = GPCOLUMN ;   prop.value      = gridcol
ctrlid<-1> = 'GRDDATA'
property<-1> = GPROW    ;   prop.value<-1> = gridrow

CALL ATGUISETPROPS(guiapp, guifrm, ctrlid, property, prop.value,
guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error
*
RETURN
```

So, what is happening here?

First of all, both the event handlers get the position of the cell that has just been updated and the new cell value from the `guiargs` variable. The ‘Validate Cell’ event handler then calls the `validate_cell` subroutine to process this information. The ‘Change’ event handler also calls the `validate_cell` subroutine – but only if the current cell position matches the position of the cell that has just been changed.

This means that if you press ‘Enter’ after editing a cell, and therefore remain in the cell, the ‘Change’ event handler will call the `validate_cell` subroutine, while the ‘Validate Cell’ event handler will not fire (because you haven’t moved off the cell).

On the other hand, if you press ‘Tab’ or an arrow key to terminate your editing, then the ‘Change’ event handler will deduce that it is no longer in the cell that was edited, and therefore, will not call the `validate_cell` subroutine. However, the ‘Validate Cell’ event handler will fire in this situation, and will call the `validate_cell` subroutine.

So, overall, the `validate_cell` subroutine will be called once – and only once – regardless of how you terminate your data entry. And the real point of the logic in the ‘Change’ event handler was to prevent the `validate_cell` subroutine from being called twice for the same data change.

So, what validation is done in the `validate_cell` subroutine? The main bit of data validation is carried out in the first few lines of this subroutine. This section gets the output conversion that was read from the dictionary, uses this convert the entered value to internal format, and then re-converts it to output format. What is the significance of this?

- In some cases, conversion codes return null values if applied to invalid data. For example if you apply a date conversion to the string ‘abc’, then the conversion will return a null value
- In other cases, the conversion will “correct” the data. For example, applying a date conversion to the date “29/02/2009” returns an internal date value of 15036 – which is the first of March[18]
- Where data is entered at a differing level of decimal precision, the input and output conversions will standardise the data display. For example, a conversion mask of ‘MR2,Z’ will convert a value of 1234.567 to 123457 using `ICONV`, which will then be converted to 1,234.57 by `OCONV`.

18 This behaviour can be changed by turning the `NO.DATE.WRAPPING` option `ON`. If this option is on, then an `ICONV` on an invalid date will return as null.

The next bit of the validation tests the format of the item-id, but only for those files with an key structure of *YYYYMM*. Some simple tests are applied here – the year portion must have 4 characters, while the month must be between 1 and 12. If it fails these tests, then an error message is displayed and the focus is returned to the cell using the *grid_pos* subroutine.

Finally, the reformatted data is loaded into the *gdata* variable. This is the variable we used to load the grid with, but in this case, we aren't updating the grid with the reformatted data – yet. We'll do that once we have validated that the row is OK.

Enter the following code in the validate row event handler:

```
*-->BEGIN EVENT HANDLER<--*
GUI.XED.FRMMAIN.GRDDATA.VALIDATEROW: *
*
gridrow = guiargs<1, 1>
hasdata = @FALSE
hasid = @FALSE
FOR ii = 1 TO gridcols
  datavalue = gdata<1, gridrow, ii>
  IF datavalue THEN
    hasdata = @TRUE
    IF ii = 1 THEN
      hasid = @TRUE
    END
  END
NEXT ii
ok = (hasdata EQ hasid)

IF ok THEN
  CALL ATGUISETPROP(guiapp, guifrm, 'GRDDATA', GPVALUE, 0, 0,
gdata, guierrors, guistate)
  IF guierrors<1> GE 2 THEN GOTO gui.error
END ELSE
  CALL ATGUIMSGBOX('Row must have a valid ID', 'ID error', MBIICON,
MBOK, '', ok, guierrors, guistate)
  IF guierrors<1> GE 2 THEN GOTO gui.error

  gridcol = 1
  GOSUB grid_pos
END

RETURN
*
*-->END EVENT HANDLER<--*
```

This subroutine starts by getting the grid row from the *guiargs* variable, and then loops through the *gdata* variable checking the data in that row. The validation here only concerns itself over the presence of data, and the presence of an item-id. If there is no data, then it is OK to leave the row. Likewise, if there is an item-id, it is OK to leave the row. It is only where we have data but do not have an item-id that there is a problem. The line:

```
ok = (hasdata EQ hasid)
```

determines whether or not it is OK to leave the row. This is a Boolean expression. Such expressions evaluate to a value of either `@TRUE` or `@FALSE`. In this case, the expression is `@TRUE` if the variable *hasdata* has the same value as *hasid*. Technically, this expression is not the same as the expression outlined above – which would be expressed as:

```
IF NOT(hasid) AND hasdata THEN
  ok = @FALSE
END ELSE
  ok = @TRUE
END
```

The difference between these two expressions is best visualised by a 2 x 2 truth table:

OK to leave row?		**hasid**	
		@TRUE	**@FALSE**
hasdata	**@TRUE**	@TRUE	@FALSE
	@FALSE	@TRUE	@TRUE

This table shows that the only `@FALSE` value occurs when *`hasid`* is `@FALSE` and *`hasdata`* is `@TRUE`. However, the Boolean expression outlined above would give two `@FALSE` values in this truth table – these cells have been shaded in the table. So why is this Boolean expression acceptable?

The answer lies in the relationship between *`hasid`* and *`hasdata`*. According to the logic that sets these variables, *`hasid`* can only be `@TRUE` if *`hasdata`* is `@TRUE`. Therefore, the cell with *`hasid`* = `@TRUE` and *`hasdata`* = `@FALSE` is not feasible, and the truth table is actually as shown below:

OK to leave row?		**hasid**	
		@TRUE	**@FALSE**
hasdata	**@TRUE**	@TRUE	@FALSE
	@FALSE		@TRUE

Accordingly, the Boolean expression gives the same outcome as the `IF-THEN-ELSE` expression.

If our expression returns `@TRUE`, then the subroutine loads the reformatted data into the grid control; otherwise, it displays an error message and moves the focus to the first column of the row we are trying to leave. We know the problem is in this cell because the only reason that the row will not validate is if there is no item-id.

The application should be behaving reasonably well at this point. But if we try some odd things, it still isn't quite right. Try clicking on the 'Edit' button when you are on the last screen of data. Leave the ID column blank and enter some data in another column. Try leaving the row – you should get an error message. So far, so good.

Now click on the 'Cancel' button. The 'Validate Row' event will fire and display the error message again. Click on 'OK' to dismiss the error message, and then click on 'Cancel' again. The buttons change their enabled status indicating that we have successfully escaped from editing mode – but the data isn't correct in the grid. Clearly, there is a little bit more to do here.

The key to the problem here is that when we clicked on the 'Cancel' button, the grid lost focus. Even though the 'Validate Row' event handler displayed the message, and moved the active cell to the ID column, it didn't ensure that the focus was restored to the grid. Therefore, we were able to click on 'Cancel' again, and because the grid didn't have the focus, no grid related events occurred.

Update the last portion of the 'Validate Row' event handler as follows:

```
IF ok THEN
  CALL ATGUISETPROP(guiapp, guifrm, 'GRDDATA', GPVALUE, 0, 0,
gdata, guierrors, guistate)
  IF guierrors<1> GE 2 THEN GOTO gui.error
END ELSE
  CALL ATGUIMSGBOX('Row must have a valid ID', 'ID error', MBIICON,
MBOK, '', ok, guierrors, guistate)
  IF guierrors<1> GE 2 THEN GOTO gui.error

  gridcol = 1
  GOSUB grid_pos
  CALL ATGUIACTIVATE(guiapp, guifrm, 'GRDDATA', guierrors,
guistate)
  IF guierrors<1> GE 2 THEN GOTO gui.error
END
```

This simply adds the call to `ATGUIACTIVATE` at the end of the error handler to ensure that the grid retains the focus after an error occurs.

There are still some problems when we escape from the grid:

- Changes that we have made to the data are still displayed, and from the user's perspective have not been cancelled.
- If we are on the last screen of data, blank rows will continue to be displayed.

To fix these issues up, add the following lines to the block of properties being set in the 'Cancel' button event handler:

```
ctrlid<-1> = 'GRDDATA'
property<-1> = GPSTYLE      ;   prop.value<-1> = 0
```

and then add these lines to the end of the 'Cancel' button event handler:

```
gdata = ogdata
GOSUB load_grid
```

The property setting changes the grid style to display only the rows of data in the grid, while the second change restores the grid data to its earlier state and reloads the grid.

6.7 Saving the data

6.7.1 Necessary steps

We can now load and display our choice of data from the disk. We can also edit the data, and add new data. All we need to do now to make the application functional is to save the updated data to disk.

Let's think about what needs to happen when we click on the 'Save' button:

- We need to change the state of the buttons on the form.
- We need to loop through the rows in the grid, and:
    - extract the data from the row, then:
    - see if this has changed from the data that we initially read from the disk.
    - If we have changed (or added) the data, then:
        - we need to check the original data against what is now on the disk, and:
        - if the data on the disk matches the data we originally read from disk then we need to update the data on the disk.
- We need to report any errors to the user.

We'll start with setting the button state – not because this is the most important part of the process, but because it builds on what we have already done.

6.7.2 Changing the button state

If we think about it, the state of the buttons after we finish the 'Save' process should be the same as they would be after the 'Cancel' process. So, cut the lines of code that set the button and grid properties out of the 'Cancel' event handler, and put them in their own subroutine *exit_editing*:

```
exit_editing:
*
ctrlid      = 'BTNCANCEL'
property     = GPENABLED    ;   prop.value     = @FALSE
ctrlid<-1> = 'BTNEDIT'
property<-1> = GPENABLED    ;   prop.value<-1> = @TRUE
ctrlid<-1> = 'BTNEXIT'
property<-1> = GPENABLED    ;   prop.value<-1> = @TRUE
ctrlid<-1> = 'BTNSAVE'
property<-1> = GPENABLED    ;   prop.value<-1> = @FALSE
ctrlid<-1> = 'BTNUPDATE'
property<-1> = GPENABLED    ;   prop.value<-1> = @TRUE
ctrlid<-1> = 'GRDDATA'
property<-1> = GPSTYLE      ;   prop.value<-1> = 0

CALL ATGUISETPROPS(guiapp, guifrm, ctrlid, property, prop.value,
guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

GOSUB disable_editing

RETURN
```

This reduces the 'Cancel' event handler to:

```
*-->BEGIN EVENT HANDLER<--*
GUI.XED.FRMMAIN.BTNCANCEL.CLICK: *
*
GOSUB exit_editing

gdata = ogdata
GOSUB load_grid

RETURN
*
*-->END EVENT HANDLER<--*
```

and we can call the *exit_editing* subroutine from the 'Save' event handler.

6.7.3 The 'Save' event handler

The main action that has to occur when we click the 'Save' button is for the application to loop through the grid, extract the data for each row, and save that data as required. On this basis, we can set up a skeleton event handler as:

```
error_condition = ''
IF gdata NE ogdata THEN
  dcr = DCOUNT(gdata<1>, @VM)
  FOR reccnt = 1 TO dcr
    GOSUB extract_record
    GOSUB save_record
  NEXT reccnt

  IF error_condition THEN
    emsg = 'Error message'
    CALL ATGUIMSGBOX(emsg, 'Save errors', MBXICON, MBOK, '', ok,
guierrors, guistate)
    IF guierrors<1> GE 2 THEN GOTO gui.error

    gridrow = dcr
    gridcol = 1
    GOSUB grid_pos
    CALL ATGUIACTIVATE(guiapp, guifrm, 'GRDDATA', guierrors,
guistate)
    IF guierrors<1> GE 2 THEN GOTO gui.error
  END
END

IF NOT(error_condition) THEN
  GOSUB exit_editing
END
```

There are a couple of place holders in this code – we haven't defined what *error_condition* is, or what the error message should be. However, the rest of it is functional – or at least it will be once the two new subroutines have been created.

The subroutine starts by testing the current grid data (that we have been maintaining in memory as we have edited the data) against the original grid data that we stored when we first loaded the grid. If the data is the same, then no further action is required, and we can reset the state of the buttons.

The loop to extract and save the data is pretty simple, as is the action taken if an error is found. All we need to do now is fill in the two missing subroutines.

Enter the following code in the *extract_record* subroutine:

```
extract_record:
*
orowdata = ''
rowdata = ''
dataid = gdata<1, reccnt, 1>
testid = ''
ii = 0
LOOP
  ii += 1
  testid = ogdata<1, ii, 1>
  IF testid EQ dataid THEN EXIT
  IF ii GE dcr THEN EXIT
REPEAT

FOR colno = 2 TO gridcols
  IF testid THEN
    orowdata<colno> = ICONV(ogdata<1, ii, colno>, xdconvs<1,
colno>)
  END
  rowdata<colno> = ICONV(gdata<1, reccnt, colno>, xdconvs<1,
colno>)
NEXT colno
*
dataid = ICONV(dataid, xdconvs<1, 1>)

RETURN
```

This subroutine uses two variables that were defined in the calling subroutine – *dcr* which is the number of rows in the grid (and the number of times that this subroutine will be called), and *reccnt* which is the row number in the grid currently being extracted.

The subroutine begins by nulling the variables to hold the row data and the original row data, and then gets the record ID from the first column of the grid. [Note that this entire application is written around the assumption that the first column of the grid holds the record ID].

We know that the row number that we are extracting is defined by *reccnt*. However, we don't know which row in the original grid matches the row we are extracting. The first loop in the subroutine tests the ID of the row in the original data until it finds matching value, or runs out of rows.

You may wonder why the row number in the original grid would not match the row number in our modified grid. There are several reasons:

- The row may have been deleted. Although the application does not allow this at present, a future enhancement could add this feature
- We may have added a new row. In this case, `testid` will have a null value for any new rows added
- We may have changed the ID of an existing row in the grid.

Once we have the matching row number in the original grid data, the next loop extracts the individual data elements from each data set and stores them in the *`rowdata`* and *`orowdata`* variables. Note that the data is converted from external format (as it was displayed in the grid) to internal format (as it is stored on the disk) as it is transferred to the holding variables.

Finally, the ID of the record is converted to internal format.

Now we need to save the data. There are a few steps in this process:

- Because *OpenQM* is a multi-user database, we should obtain an exclusive lock on the record before updating it
- When we read the data from the disk, we didn't (necessarily) store all of the data in the record in the grid. Rather, the read process only placed nominated data fields in the grid. Therefore, the application has no way of knowing whether the grid holds the entire data record, or only part of it.
- This means that we cannot simply form a new data record from the grid data and write it to disk. Rather, we need to transfer the grid data to the record we have just read from disk (when we got the exclusive lock), and write that record back to the disk.
- Possible error points in this process include:
  - Someone else may have a lock on one of the records
  - Someone else may have changed the data on disk.

We'll start with these error points. Add the following lines to the start of the 'Save' event handler:

```
list.locked = ''
list.changed = ''
```

These variables will store the ID's of any records which are locked or have been changed. This will then let us report this information to the user if these errors occur.

Create the *`save_record`* subroutine:

```
save_record:
*
tries = 0
datarec = ''
LOOP
  tries += 1
  READU datarec FROM selfile.data, dataid LOCKED
    NAP 10
    readok = @FALSE
  END THEN
    readok = @TRUE
  END ELSE
    readok = @TRUE
  END
UNTIL readok OR (tries GE 3) DO REPEAT

IF readok THEN
  dowrite = @FALSE
  FOR colno = 2 TO gridcols
    amc = xdamcs<1, colno>
    IF orowdata<colno> NE datarec<amc> THEN
      dowrite = @FALSE
      list.changed<-1> = dataid
      EXIT
    END
    IF rowdata<colno> NE datarec<amc> THEN
      datarec<amc> = rowdata<colno>
      dowrite = @TRUE
    END
  NEXT colno
  IF dowrite THEN
    WRITE datarec ON selfile.data, dataid
  END ELSE
    RELEASE selfile.data, dataid
  END
END ELSE
  list.locked<-1> = dataid
END

RETURN
```

The subroutine has three attempts at reading the record from the data file, pausing 10 milliseconds between each attempt. As exclusive locks should only be held while a record is being updated, this pause should be enough to allow other processes to relinquish their locks. If the subroutine cannot obtain an exclusive lock, then the item-id is added to the *list.locked* variable, and the subroutine moves on to the next record.

The subroutine then loops through the data. If the original data in any column does not match what has just been read from the disk, then that record is noted as having changed, and the loop exits. If the updated data in any column does not match what has just been read from the disk, then the data record is updated with the new data, and the record is noted as requiring an update (*dowrite* = @TRUE).

Finally, if the *dowrite* flag has been set, the updated data record is written back to disk; otherwise, the exclusive lock is released.

This brings us back to the 'Save' event handler, where we now need to update our error condition check, and error message. This is the updated event handler:

```
*-->BEGIN EVENT HANDLER<--*
GUI.XED.FRMMAIN.BTNSAVE.CLICK: *
*
list.locked = ''
list.changed = ''
IF gdata NE ogdata THEN
  dcr = DCOUNT(gdata<1>, @VM)
  FOR reccnt = 1 TO dcr
    GOSUB extract_record
    GOSUB save_record
  NEXT reccnt

  IF list.locked OR list.changed THEN
    emsg = 'The following items could not be saved:'
    IF list.changed THEN
      CONVERT @AM TO ',' IN list.changed
      emsg<1, 1, -1> = 'Changed on disk: ':list.changed
    END
    IF list.locked THEN
      CONVERT @AM TO ',' IN list.locked
      emsg<1, 1, -1> = 'Locked by another user: ':list.locked
    END
    CALL ATGUIMSGBOX(emsg, 'Save errors', MBXICON, MBOK, '', ok,
guierrors, guistate)
    IF guierrors<1> GE 2 THEN GOTO gui.error

    gridrow = dcr
    gridcol = 1
    GOSUB grid_pos
    CALL ATGUIACTIVATE(guiapp, guifrm, 'GRDDATA', guierrors,
guistate)
    IF guierrors<1> GE 2 THEN GOTO gui.error
  END
END

IF NOT(list.locked OR list.changed) THEN
  GOSUB exit_editing
END

RETURN
*
*-->END EVENT HANDLER<--*
```

This now checks for the existence of either error condition. If errors occurred, then an appropriate error message is created and displayed.

6.8 Bug Fixing

6.8.1 Application Review

At this point, the application is functional. It doesn't fulfil our original design specification because it doesn't automatically update the data from the internet, but it does allow the reading, display, editing, and saving of data.

It is worthwhile reviewing the application at this point to make sure that all functions work correctly. As usual, there are some things we haven't adequately covered yet:

- When we are editing data, we can still change the file, year, and/or month via the combo boxes. These should be disabled during editing.
- We used a control item to tell us the item-id's of first and last items in each file. However, when we saved the data in the grid, we didn't check to see whether we added any items outside these ranges.
- Row and cell validation still isn't quite right. Consider the following sequence:
    - Enter an invalid year and month ID (e.g. '123') when editing a file with a *YYYYMM* format ID, and terminate your entry with the up arrow key. You will get an error message response, and the focus will return to the cell.
    - Now use the up arrow to move off the cell. There is no error response.
- Likewise, there can be a small problem when we cancel editing:
    - Enter an invalid item-id as above.
    - After you have cleared the error message, click on 'Cancel'. You will get another error as the validation events fire.
    - Clear the error message, and click on 'Cancel' again. You will now escape from editing.
- There should be more validation of the item-id when we are editing an item. In particular, we need to ensure that the item-id is not already in use.

6.8.2 Disabling combo boxes during editing

To fix the first issue, add the following property setting to the 'Edit' button event handler:

```
ctrlid<-1> = 'FRACONTROLS'
property<-1> = GPENABLED    ;   prop.value<-1> = @FALSE
```

and an equivalent property setting re-enabling the control in the *exit_editing* subroutine.

This bit of code doesn't enable or disable the individual combo boxes, but rather, works on the container control for the combo boxes. By disabling this container, all the controls in the container are disabled.

6.8.3 Updating the control item

Fixing the second issue takes a few more steps. The main bit of code goes immediately after the `WRITE` statement in the *save_record* subroutine:

```
    IF dataid LT xfirst THEN
      xfirst = dataid
      change.firstlast = @TRUE
    END
    IF dataid GT xlast THEN
      xlast = dataid
      change.firstlast = @TRUE
    END
```

This introduces a new variable *change.firstlast*, so we need to initialise that in the 'Save' event handler. Put the following line immediately after the `IF` statement at the top of the event handler:

```
  change.firstlast = @FALSE
```

and then after the loop for *extract_record* and *save_record*, add the following code:

```
  IF change.firstlast THEN
    xfiledef<4> = xfirst
    xfiledef<5> = xlast

    xid = selfile
    xrec = xfiledef
  END
```

This sequence of code changes updates the *xfirst* and *xlast* variables if the data item being saved has an item-id outside the current bounds. Then the new *xfirst* and *xlast* variables are added to the control item, and the control item saved.

You may note an inconsistency between the way that data is saved and the way that the control item is saved. When the data is saved, there are checks to make sure that no-one else has updated the record since the application read it from disk; but for the control item, no such check has taken place, and the item could have changed.

Consider this a challenge to rewrite this section of code. Things that should happen here are:

- Re-read the control item using an exclusive lock
- check *xfirst* against the relevant entry in the control item
- check *xlast* against the relevant entry in the control item.
- If either of these variables are outside those in the control item, then update the control item and write it back to disk; otherwise, release the control item.

6.8.4 Row and cell validation

The application review identified an issue where no error was reported when moving off a row, even though it had been identified immediately prior. To debug this, put a message box display in each of the event handlers involved – change cell, validate cell, and validate row. When we run the program, we find that:

- on the first run through with a bad ID, all three event handlers are called
- on the second run through, none of the event handers are called.

So, lets step through what happens:

- We enter the bad data and terminate the entry using the up arrow.
- The change cell event handler is called. This finds that the entry cell no longer has the focus so does nothing.
- The validate cell event hander is called. This calls the *validate_cell* subroutine which detects an error and displays a dialog box with an error message. The cursor is repositioned to the entry cell. Finally, the reformatted data is stored in the *gdata* variable.
- The validate row event handler is called. This checks that the row has an item-id. Having passed this validation, the data in the *gdata* variable is loaded to the grid.
- We use the up arrow to move off the cell.
- None of the event handlers are called because the data in the cell we are leaving matches the data in the grid – i.e. there is no change in the grid data, so the event handlers are not called.

This identifies the problem fairly clearly. Updating the data in the validate row event handler has removed the grid's ability to determine when to fire an event.

Ideally, what we would like to do is to cancel the validate row event if an error is found in the *validate_cell* subroutine. Unfortunately, there does not appear to be any way to cancel pending events.

The alternative is to set a variable to record the status of the cell validation. If the cell does not validate, then we should bypass the row validation.

This doesn't take too many lines of code. At the start of the *validate_cell* subroutine, add the following line:

```
cellvalidated = @TRUE
```

At the end of this subroutine in the `IF-THEN` block that tests the value of *iderr*, add the following line:

```
cellvalidated = @FALSE
```

Now, put the entire validate row event handler code into an `IF-THEN` block that tests the value of *cellvalidated*:

```
*-->BEGIN EVENT HANDLER<--*
GUI.XED.FRMMAIN.GRDDATA.VALIDATEROW: *
*
IF cellvalidated THEN
  gridrow = guiargs<1, 1>

  etc
END

RETURN
*
*-->END EVENT HANDLER<--*
```

The application should now continue to prompt you that the item-id is invalid.

6.8.5 Cancelling editing

The problem identified with cancelling editing is the same one we had at the end of Section 6.6. In that case, the validate row event handler was not being called after pressing the ‘Cancel’ button for a second time because the grid had lost the focus. The answer then was to put a call to the ATGUIACTIVATE subroutine in the validate row event handler to ensure the focus was restored to the grid after a validation error.

This time, the validation error is occurring in the *validate_cell* subroutine, and that is where we need to put the code to restore the focus to the grid:

```
  IF iderr THEN
    CALL ATGUIMSGBOX(editvalue:' is not a valid ID', 'ID error',
MBIICON, MBOK, '', ok, guierrors, guistate)
    IF guierrors<1> GE 2 THEN GOTO gui.error

    GOSUB grid_pos
    cellvalidated = @FALSE
    CALL ATGUIACTIVATE(guiapp, guifrm, 'GRDDATA', guierrors,
guistate)
    IF guierrors<1> GE 2 THEN GOTO gui.error
  END
```

6.8.6 Item-id validation

The final aspect identified as requiring further work was the validation of the item-id during editing. There are two parts to this:

- We need to make sure that the item-id is not already in use in the current screen
- We need to make sure that the item does not already exist on file. If we did not do this, then we could overwrite existing items by typing in replacement data on the current screen.

Our current item-id validation is in the *validate_cell* subroutine. The code to extend the validation as outlined above increases this code significantly. The revised subroutine is shown below:

```
validate_cell:
*
cellvalidated = @TRUE
xconv = xdconvs<1, gridcol>
IF xkeystruct = 'D' AND gridcol = 1 THEN xconv = 'DL'
fmtvalue = ICONV(editvalue, xconv)
fmtvalue = OCONV(fmtvalue, xconv)

IF fmtvalue THEN
  IF gridcol = 1 THEN
    iderr = @FALSE
    IF xkeystruct = 'D' THEN
      NULL
    END ELSE
      fmtvalue = OCONV(fmtvalue, 'MCN')
      yyyy = INT(fmtvalue / 100)
      mm = MOD(fmtvalue, 100)
      IF mm LT 1 OR mm GT 12 THEN iderr = @TRUE
      IF LEN(yyyy) NE 4 THEN iderr = @TRUE
      emsg = editvalue:' is not a valid ID'
    END

    IF NOT(iderr) THEN
      dcr = DCOUNT(gdata<1>, @VM)
      idcnt = 0
      oidcnt = 0
      FOR ii = 1 TO dcr
        IF ii EQ gridrow THEN
          idcnt += 1
        END ELSE
```

```
          IF gdata<1, ii, 1> = fmtvalue THEN idcnt += 1
        END
        IF ogdata<1, ii, 1> = fmtvalue THEN oidcnt += 1
      NEXT ii
      IF gridrow GT dcr THEN idcnt += 1

      IF idcnt GT 1 THEN
        iderr = @TRUE
        emsg = editvalue:' is already in use as an item-id'
      END

      IF idcnt EQ 1 AND oidcnt EQ 0 THEN
        temp = fmtvalue
        temp = ICONV(temp, xconv)
        READ dummy FROM selfile.data, temp THEN
          found = @TRUE
        END ELSE
          found = @FALSE
        END
        IF found THEN
          iderr = @TRUE
          emsg = editvalue:' already exists on disk'
        END
      END
    END

    IF iderr THEN
      CALL ATGUIMSGBOX(emsg, 'ID error', MBIICON, MBOK, '', ok,
guierrors, guistate)
      IF guierrors<1> GE 2 THEN GOTO gui.error

      GOSUB grid_pos
      cellvalidated = @FALSE
      CALL ATGUIACTIVATE(guiapp, guifrm, 'GRDDATA', guierrors,
guistate)
      IF guierrors<1> GE 2 THEN GOTO gui.error
    END
  END
END

gdata<1, gridrow, gridcol> = fmtvalue
*
RETURN
```

This code has an additional change to those already mentioned. It now bypasses the validation if the entered value is null. This makes it possible to delete the item-id without the subroutine generating an error message.

Most of the new code is in the blocks starting with:

```
IF NOT(iderr) THEN
```

This starts by looping through the *gdata* variable testing the first column to see if the cell value matches the item-id we have just entered. It keeps a count of the number of times this value appears, and keeps a similar count from the original grid data. Note that there is an adjustment of this count if the current grid row is greater than the count of rows in the grid. This means we are just starting to enter a new row in the grid.

If it counts more than one reference to this item-id in the current grid, it reports an error. This is an obvious case of a duplicate item-id.

The next case is a little less obvious. If there is only one reference to the item in the current grid (which should be the case), and there is no reference to it in the original grid data, then we have entered a new item-id. We now need to test that this item-id does not already exist in the data file. If it does, then this is an error.

7 Automating Updates

7.1 Steps to automate the update process

The final part of our design specification was to automate the update process. This involves a number of steps:

- Downloading the source spreadsheets from the internet
- Processing the downloaded spreadsheets to a format suitable for importing the data
- Carrying out the data import.

These steps aren't too difficult. However, we need to make a design choice before we start the implementation – specifically, are these actions going to take place on the client or on the server?

7.1.1 Client or Server?

In terms of the system architecture, the *OpenQM* database represents the server, while the *AccuTerm* terminal emulator represents the client. It is possible to have both components on the same physical computer, and this will usually be the case for single user systems. However, for most multi-user systems, the client software will usually be running on networked PC's connecting to the *OpenQM* database on a central server.

So, what are the differences and the implications of each?

- We will be using a third-party utility to download the spreadsheets from the internet. We need to decide whether we initiate this using `QMBasic` from the server, or via a client command from *AccuTerm*:
  - If we execute the program from the server, then we only need one copy of the download utility

    - If we execute the program from the client, then we need a copy of the utility on every machine that will perform the download/update procedure.
    - The programming required for starting the utility is different for each method.
- Manipulating the spreadsheets is best done on the client using *AccuTerm*'s scripting ability. This could alternatively be done on the server using some other scripting interface.
- The data import could occur from either the client or the server:
    - *AccuTerm* has the ability to read a number of different file types (e.g. *Excel*, *Access*, *DBase*, `CSV`), and import the data from those files into multi-value databases. Accordingly, the source data for these imports should either be on the client directly, or available to the client via a network share
    - *OpenQM* can read O/S level files directly, allowing us to write our own import routines. In particular, *OpenQM* has the `READCSV` statement that would allow relatively easy (and fast) data imports from `CSV` files. Such files would need to be located on the server (or accessible by the *OpenQM* database).

For the purposes of this application, we will use client side control of the download utility. While this has the disadvantage of requiring a copy of the utility for every workstation that will use the download functionality, it does simplify the other two steps in the process.

While using the `READCSV` statement to import the data is interesting, it would require more work than using the readily available and generalised tools that are part of *AccuTerm*. Further, the imports would be specific to this particular application – which isn't really desirable. We would prefer to keep tools general so that they can be reused elsewhere.

7.2 Downloading from the Internet

7.2.1 Choice of tools

Neither *OpenQM* nor *AccuTerm* are designed for downloading data from the internet. That is not to say that we can't program them to do this[19], but there is no need for us to do this. There are plenty of utilities available that will download material for us, and what we need to do is program *OpenQM* and *AccuTerm* to control those utilities.

In this case, we are going to use *`curl`* to download the spreadsheets that we want. *`curl`* is available for download from `http://curl.haxx.se` . Download the version appropriate to your operating system – you do not need the SSL version.

You don't need to use *`curl`* if you prefer another utility. *`wget`* is commonly used, and there are many other tools available. The important thing is that whatever tool you choose, it should be able to be run from a command line (or via a script).

Descriptions here will assume that you are using *Windows*. If you are using another platform, you will need to change the instructions accordingly.

Save the executable file to an appropriate location on your system. These instructions assume that it is in a folder named:

```
C:\Program Files\curl
```

The basic syntax of the command we will use is:

```
{curl-path\}curl -o dest-path url
```

19 See for example the `HTTP.CLS` package that can be downloaded using the QM Package Manager. See the comments on object-oriented programming in Section 11.4 for more information.

The source spreadsheet for the exchange rate data used in this application is located at:

```
http://www.rbnz.govt.nz/statistics/exandint/b1/hb1.xls
```

Let's say we want to store the downloaded file as:

```
C:\Temp\hb1.xls
```

Therefore, the entire command we want to run is:

```
C:\Program Files\curl -o C:\Temp\hb1.xls
http://www.rbnz.govt.nz/statistics/exandint/b1/hb1.xls
```

This is all entered as one single command. Open a command prompt window, and enter the command. `curl` should download the requested file and save it in the *`C:\Temp`* folder. Progress statistics will display during the download.

If the command fails:

- check the command is entered correctly
- check that the destination folder exists
- check your network and firewall settings.

Once you can download the spreadsheet from the command line, then we can continue. We now want to get the application to run this command.

7.2.2 Execution on server

If we were running this command on the server, we could use code like:

```
pgm = '"C:\Program Files\curl\curl.exe"'
dest = '-o C:\Temp\hb1.xls'
url = 'http://www.rbnz.govt.nz/statistics/exandint/b1/hb1.xls'
cmd = pgm:' ':dest:' ':url
OS.EXECUTE cmd CAPTURING junk
```

Note that the *`pgm`* line actually has two sets of quotes. The outer single quotes delimit the variable, so the inner double quotes are actually part of the variable. These ensure that the embedded space in the path name is not misinterpreted by the command processor.

If you actually try to run this code fragment, you will find that *OpenQM* session will appear to lock up. It hasn't really – it is just that the download takes about 80 seconds on a standard broadband connection, during which time the system is unresponsive.

This is a good bit of code to execute via a `PHANTOM` process. This means the download can occur in the background while you keep the foreground process responsive to user requests. Typically, the foreground process will monitor the background process, and give the user an option to cancel the download.

It is simple enough to start a `PHANTOM` process, but we really want to have some standardised software to control phantoms. The following subroutine starts down that road:

```
SUBROUTINE RUN.AS.PHANTOM(cmd, userno, logname)
*****************************************************************
*
$CATALOGUE

IF cmd = '' THEN
  userno = ''
  logname = ''
  RETURN
END

EXECUTE "PHANTOM ":cmd SETTING userno CAPTURING junk

IF userno LT 0 THEN
  userno = ''
  logname = ''
  RETURN
END

CALL !PHLOG(logname, userno)

RETURN
*
* ------------------------------------------------------------- *
*
END
```

Essentially, we pass the command we want to execute as a phantom to this subroutine, and get back the user number of the phantom process, and the name of the log that the phantom has created in the `$COMO` file. Note that this deliberately loses some information if the phantom fails – in this case, the user number is returned as a negative error number. However, for our purposes, it is sufficient to know that the phantom has failed.

To test this subroutine, we can use the following the program:

```
PROGRAM TEST
$CATALOGUE

pgm = 'SH "C:\Program Files\curl\curl.exe"'
dest = '-o C:\Temp\hb1.xls'
url = 'http://www.rbnz.govt.nz/statistics/exandint/b1/hb1.xls'
cmd = pgm:' ':dest:' ':url

userno = ''
logname = ''
CALL RUN.AS.PHANTOM(cmd, userno, logname)

CRT 'Command executed as phantom at ':OCONV(TIME(), 'MTS')
CRT 'User number is ':userno
CRT 'Log name is ':logname

IF NOT(userno) THEN STOP

LOOP
  SLEEP 5
  CRT OCONV(TIME(), 'MTS'):'  ':
  IF NOT(CHILD(userno)) THEN
    CRT 'Phantom ended'
    EXIT
  END
  CRT 'Phantom running'
REPEAT

END
```

There is one very important difference between this code and the code initially discussed in this section that used the `OS.EXECUTE` command – the *`pgm`* line now starts with `SH`. This is because we are now using the `EXECUTE` command rather than the `OS.EXECUTE` command.

- `EXECUTE` executes commands that are internal to *OpenQM* (i.e. a command that would normally be executed from the *OpenQM* command prompt)
- `OS.EXECUTE` executes commands at the operating system level
- The `SH` command (or !) lets you execute an operating system level command from the *OpenQM* command line
- Therefore, the following two commands executed from `QMBasic` are effectively the same:

```
EXECUTE 'SH ':cmd
OS.EXECUTE cmd
```

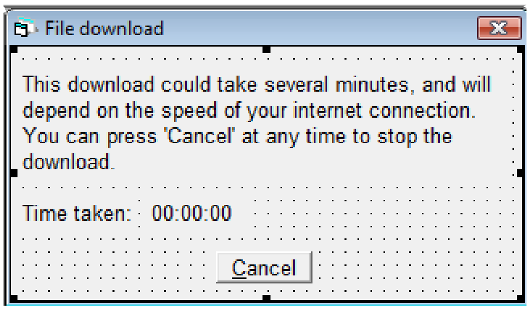
The output from the test program is as follows:

```
TEST
Command executed as phantom at 21:40:41
User number is 16
Log name is PH16_240909_214041
21:40:46  Phantom running
21:40:51  Phantom running
21:40:56  Phantom running
21:41:01  Phantom running
21:41:06  Phantom running
21:41:11  Phantom running
21:41:16  Phantom running
21:41:21  Phantom running
21:41:26  Phantom running
21:41:31  Phantom running
21:41:36  Phantom running
21:41:41  Phantom running
21:41:46  Phantom running
21:41:51  Phantom running
21:41:56  Phantom running
21:42:01  Phantom running
21:42:06  Phantom running
21:42:11  Phantom running
21:42:16  Phantom running
21:42:21  Phantom ended
Phantom 16 : Normal termination.
```

Note: As we intend to run the download from the client, this is a diversion. If you want to try this, copy your existing application and program to a new name - *XED2* is used here. The references to *XED* will need to be updated in both parts of the application.

We want to be able to display progress to the user, and give the user the illusion of being able to stop the download – because once *curl* has been started, there is no way of stopping it short of killing the process at the operating system level. We can kill the phantom process, but this will still leave *curl* running.

To give the user this information, we will use another form. This is named *frmupdate* and is shown below:



File download

This download could take several minutes, and will depend on the speed of your internet connection. You can press 'Cancel' at any time to stop the download.

Time taken: 00:00:00

Cancel

The form contains three labels, and a command button. The click event of the button is enabled by default. The text is entered via the ‘Properties’ form in the GUI Designer.

Use the ‘Update code’ icon in the GUI Designer to add the event handlers to the code. Then enter the following code to the ‘Update’ button event handler:

```
*-->BEGIN EVENT HANDLER<--*
GUI.XED2.FRMMAIN.BTNUPDATE.CLICK: *
*
CALL ATGUISHOW('XED2','FRMUPDATE','','',guierrors,guistate)
IF guierrors<1> >= 2 THEN GOTO gui.error

pgm = 'SH "C:\Program Files\curl\curl.exe"'
dest = '-o C:\Temp\hb1.xls'
url = 'http://www.rbnz.govt.nz/statistics/exandint/b1/hb1.xls'
cmd = pgm:' ':dest:' ':url

userno = ''
logname = ''
time.start = TIME()
```

```
CALL RUN.AS.PHANTOM(cmd, userno, logname)

LOOP
  IF NOT(CHILD(userno)) THEN EXIT
  CALL ATGUICHECKEVENT(1000, guiapp, guifrm, guictl, guievt,
guiargs, guierrors, guistate)
  IF guierrors<1> GE 2 THEN GOTO gui.error
  IF guievt THEN
    EXIT
  END ELSE
    time.taken = OCONV((TIME() - time.start), 'MTS')
    CALL ATGUISETPROP('XED2', 'FRMUPDATE', 'LBLTIMEDISPLAY',
GPVALUE, 0, 0, time.taken, guierrors, guistate)
    IF guierrors<1> GE 2 THEN GOTO gui.error
  END
REPEAT

CALL ATGUIHIDE('XED2','FRMUPDATE','','',guierrors,guistate)
IF guierrors<1> >= 2 THEN GOTO gui.error

RETURN
*
*-->END EVENT HANDLER<--*
```

This is a fairly simple subroutine. It begins by displaying the form that was shown above. It then defines the command that we want to run as a phantom, and calls the subroutine to execute the phantom. Then it loops, waiting for either the phantom to terminate, or for the user to click on the 'Cancel' button. If neither of those events occur, then the time is updated on the form. When the loop exits, it hides the progress form.

Of course, there are some things here that require further comment:

- Firstly, the command that is defined isn't the one that we will always want to run – so we will need to look at ways to get *`curl`* to download different source spreadsheets as we update different files in the database.
- We test whether the phantom process has terminated using the `CHILD` function.
- The `ATGUICHECKEVENT` subroutine waits for an event to occur for a specified period (1000 milliseconds in this case). We don't actually check what the event is as we make the assumption that the 'Cancel' event is the only event that can occur. This isn't quite true[20], but it is near enough for our purposes.

Now, back to the real application.

7.2.3 Execution on client

The previous section identified the command we need to run from the server to download the source spreadsheet. We now want to run that command from the client.

The command itself does not change – except we may change the destination for the downloaded file. What really changes is the way that we run the command.

So, how do we run the command from the client? And what do we mean by running the command from the client?

There are a couple of ways to run the command:

- Use one of *AccuTerm*'s private escape sequences to tell *AccuTerm* to run the command
- Write an *AccuTerm* script, and then get *AccuTerm* to run the script.

20 Because the 'Update' form is a modal form for this application, the only events that this application can create are from the 'Update' form – and the only event enabled there is the click event for the 'Cancel' button. However, the *AccuTerm* GUI does allow more than one application to be running at once. Therefore, it is possible that the event could come from another *AccuTerm* GUI application.

These two options identify what we mean by "running the command from the client". In the previous section, we ran the download from the server. That meant that the `QMBasic` code started the download directly.

Running from the client is a bit more indirect. The server has to tell its agent on the client (*AccuTerm*) to do something. *AccuTerm* can then either run the download directly itself, or indirectly by running a script which runs a download.

The following program shows how to run our download from *AccuTerm*:

```
PROGRAM TEST
$CATALOGUE

EQUATE ESC TO CHAR(27)
EQUATE STX TO CHAR(2)
EQUATE CR  TO CHAR(13)

pgm = "C:\Program Files\curl\curl.exe"
dest = " -o C:\Temp\hb1.xls "
url = "http://www.rbnz.govt.nz/statistics/exandint/b1/hb1.xls"

cmd = DQUOTE(pgm):dest:url

CRT ESC:STX:'>':cmd:CR

END
```

Note that the ***`dest`*** variable now has at each end of the string so that the spacing of the final command is correct. This is a bit easier than adding the spaces during the assembly of the command.

Compile and run this program. The `curl` download window should appear on the screen. *AccuTerm* should display a message of 'Waiting for process to terminate' until the download completes.

Now change the `CRT` line to read:

```
CRT ESC:STX:'<':cmd:CR
```

Re-compile the program, and run it again. Note that this time, *AccuTerm* returns to the command prompt immediately after starting the download.

Now, let's look at how we would run this download via a script:

AccuTerm's scripting language is similar to *Vbscript*. There are a number of ways we can generate scripts for use with *AccuTerm*. The following program creates the script within the `QMBasic` program, and then runs the script using one of *AccuTerm*'s private escape sequences:

```
PROGRAM TEST
$CATALOGUE

EQUATE STX TO CHAR(2)
EQUATE CR TO CHAR(13)
EQUATE EM TO CHAR(25)
EQUATE ESC TO CHAR(27)

script      = 'Dim Shell as Object'
script<-1> = 'Dim cmd as string'
script<-1> = 'Dim pgm as string'
script<-1> = 'Dim dest as string'
script<-1> = 'Dim url as string'

script<-1> = 'Const dq = Chr$(34)'

script<-1> = 'pgm = "C:\Program Files\curl\curl.exe"'
script<-1> = 'dest = " -o C:\Temp\hb1.xls "'
```

```
script<-1> = 'url =
"http://www.rbnz.govt.nz/statistics/exandint/b1/hb1.xls"'

script<-1> = 'Set Shell = CreateObject("WScript.Shell")'
script<-1> = 'cmd = dq & pgm & dq & dest & url'
script<-1> = 'Shell.Run cmd'
script<-1> = 'Set Shell = Nothing'

CONVERT @AM TO EM IN script

CRT ESC:STX:'P':script:CR

END
```

Most of this program simply creates the script in a string variable. This does require some "double-think" when it comes to creating strings within the script. For example, look at the line:

```
script<-1> = 'dest = " -o C:\Temp\hb1.xls "'
```

and:

```
script<-1> = 'cmd = dq & pgm & dq & dest & url'
```

When the script comes to be interpreted, the first of these lines will be seen as:

```
dest = " -o C:\Temp\hb1.xls "
```

while the second will be:

```
cmd = "C:\Program Files\curl\curl.exe" -o C:\Temp\hb1.xls
http://www.rbnz.govt.nz/statistics/exandint/b1/hb1.xls
```

Note the double quote marks need to embedded in the final string, and the must be further quoted to put them in the `QMBasic` string.

If you compile and run this program, you will find that control returns immediately to the *OpenQM* command prompt. So, we now have two ways to run the download where control returns immediately to the user, and one way where *AccuTerm* pauses until completion of the download.

Running the program directly from *AccuTerm* is somewhat simpler than creating and running a script, so we will use that method for this part of the application. However, we will be returning to scripts later.

7.2.4 Control information

Now we know how to run an external program from `QMBasic` (or from *AccuTerm*), and we know which program we want to run. However, there are various things we need to cover before we can get this functioning properly.

- There are several steps in the download / manipulate / update process. We need a control structure allowing the user to run each of these steps, skip a step, or abandon the process
- Each of the data files we want to update has a different data source. We therefore need to be able to specify the data source for each of these files
- Likewise, the processing of the downloaded spreadsheet will differ for each data source, so we need to be able to specify the appropriate processing for each file
- If we want to be complete, we may also want to allow for different client PC's having different configurations – does the client have `curl` installed, and if so, what is the location of the executable?

We also want the application written in such a way that changes can be made to the configuration without requiring the application to be recompiled. Some of these configuration changes could include:

- changing the program we use to run the download
- changing the location of the program on the client PC (or using a different client PC with the download program in a different location)
- changing the URL of the file to be downloaded
- changing the destination of the downloaded file
- changing the script we use to process each file after downloading.

Some of these items are related to the data files, so we can store this information in the control item for each file. Other items vary independently of the data files (location of the download program on different PC's), so we need a different strategy to deal with that.

Let's tabulate the information we want to store:

File:	`XRATES`
Program:	*`curl`*
URL:	http://www.rbnz.govt.nz/statistics/exandint/b1/hb1.xls
Dest:	?
Script:	?

File:	`IRATES`
Program:	*`curl`*
URL:	http://www.rbnz.govt.nz/statistics/exandint/b2/hb2.xls

File:	`FX.DAILY`
Program:	*`curl`*
URL:	http://www.rbnz.govt.nz/statistics/exandint/b4/hb4.xls

The destination and script lines have been left off the last two files because we don't know what they will be yet. Further, we haven't specified anything for the `FX.MONTHLY` file. This is because the data in this file is derived from the `FX.DAILY` file.

Note that we have specified the same download program for each of the files. This raises the question of whether this should be stored here at all – if all files are going to use this program, then perhaps this should be hardcoded into the program, or simply stored elsewhere as the download program to use in all circumstances.

While it is true that all files currently use *`curl`* as the download program, it does not necessarily follow that future files we add will also use *`curl`*. Perhaps they will require a different download program.

The download destination should probably be the same for all downloads. This means that we don't need to store the download location for each individual file – but we do need to store it somewhere.

It is time to add some information to the control file. Use `WED` (or another editor) to add the reference to *`curl`* and the source URL to each of the control items in `XED.VAR`. Given that the structure of the `XED.VAR` file is the same for all items (so far), it would be worthwhile to add some dictionary items to describe and document this file. The program reference should go into field 6 and the URL into field 7. With your newly entered dictionary items, you should be able to list the contents of these fields as shown below:

```
SORT XED.VAR WITH PROGRAM GT "" PROGRAM URL
XED.VAR...   Program   URL...................................................
FX.DAILY     CURL      http://www.rbnz.govt.nz/statistics/exandint/b4/hb4.xls
IRATES       CURL      http://www.rbnz.govt.nz/statistics/exandint/b2/hb2.xls
XRATES       CURL      http://www.rbnz.govt.nz/statistics/exandint/b1/hb1.xls
```

We now need to tell the application where to find *curl*, and what syntax to use. Similarly, we need to define our download destination. We'll put this information into XED.VAR even though it clearly doesn't belong there – this is system control data rather than application control data.

```
LIST XED.VAR 'CURL' 'DEST'  F1 FMT '30L' F2 FMT '20L'
XED.VAR...   F1...........................   F2..................
CURL         C:\Program Files\curl\curl.exe   -o %%DEST%% %%URL%%
DEST          D:\QM\Temp
```

You may recall above that we were going to allow different PC's to have different program locations and/or different download locations. How might we do that?

First of all we need to know the name of the PC. You can find this on the properties page of 'My Computer'. Given this, we can make alternate entries into the XED.VAR file for specific computers. For example, for computer *V2000*:

```
      LIST XED.VAR WITH @ID LIKE "...*V2000" F1 FMT '30L'
      XED.VAR...   F1...........................
      CURL*V2000   C:\curl\curl.exe
      DEST*V2000   C:\Temp
```

Now, to get the program location for a specific computer, we search for the computer's specific control item first, and use the information contained there if the item exists. Otherwise, we use the default control item. Note that the default control item still contains the syntax information.

This begs the question of how do we find the computer name from within QMBasic. If this were the server, we could use the SYSTEM(n) function:

```
PROGRAM TEST
$CATALOGUE

CRT SYSTEM(1015)

END
```

But what about getting the client computer name? We can't do this directly from QMBasic – but we can write a script for *AccuTerm* to run:

```
PROGRAM TEST
$CATALOGUE

EQUATE STX TO CHAR(2)
EQUATE CR TO CHAR(13)
EQUATE EM TO CHAR(25)
EQUATE ESC TO CHAR(27)

script = 'InitSession.Output Environ("COMPUTERNAME") & vbCR'
CRT ESC:STX:'P':script:CR:
INPUT clientname

CRT clientname
STOP
END
```

Note that this script does literally print the computer name on the screen. This can look untidy, so you may want to hide this from the user.

We'll put this code into a subroutine so that we can call it from the main application later:

```
SUBROUTINE GET.COMPUTERNAME(cs, computername)
*********************************************************************
**
* Get's the computername of the client or server
*
* Input : cs - client or server (literal string)
* Output: computername
*
```

```
$CATALOGUE

EQUATE STX TO CHAR(2)
EQUATE CR TO CHAR(13)
EQUATE EM TO CHAR(25)
EQUATE ESC TO CHAR(27)

cs = OCONV(cs[1,1], 'MCU')
computername = ''
BEGIN CASE
  CASE cs = 'C' ; GOSUB client
  CASE cs = 'S' ; GOSUB server
END CASE
*
RETURN
*
* -----------------------------------------------------------------
*
*
client:
*
script = 'InitSession.Output Environ("COMPUTERNAME") & vbCr'
CRT ESC:STX:'P':script:CR:

INPUT computername:
CRT @(0):@(-4):         ;* Delete value off screen
*
RETURN
*
* -----------------------------------------------------------------
*
*
server:
*
computername = SYSTEM(1015)
*
RETURN
*
*
-------------------------------------------------------------------
*
*
END
```

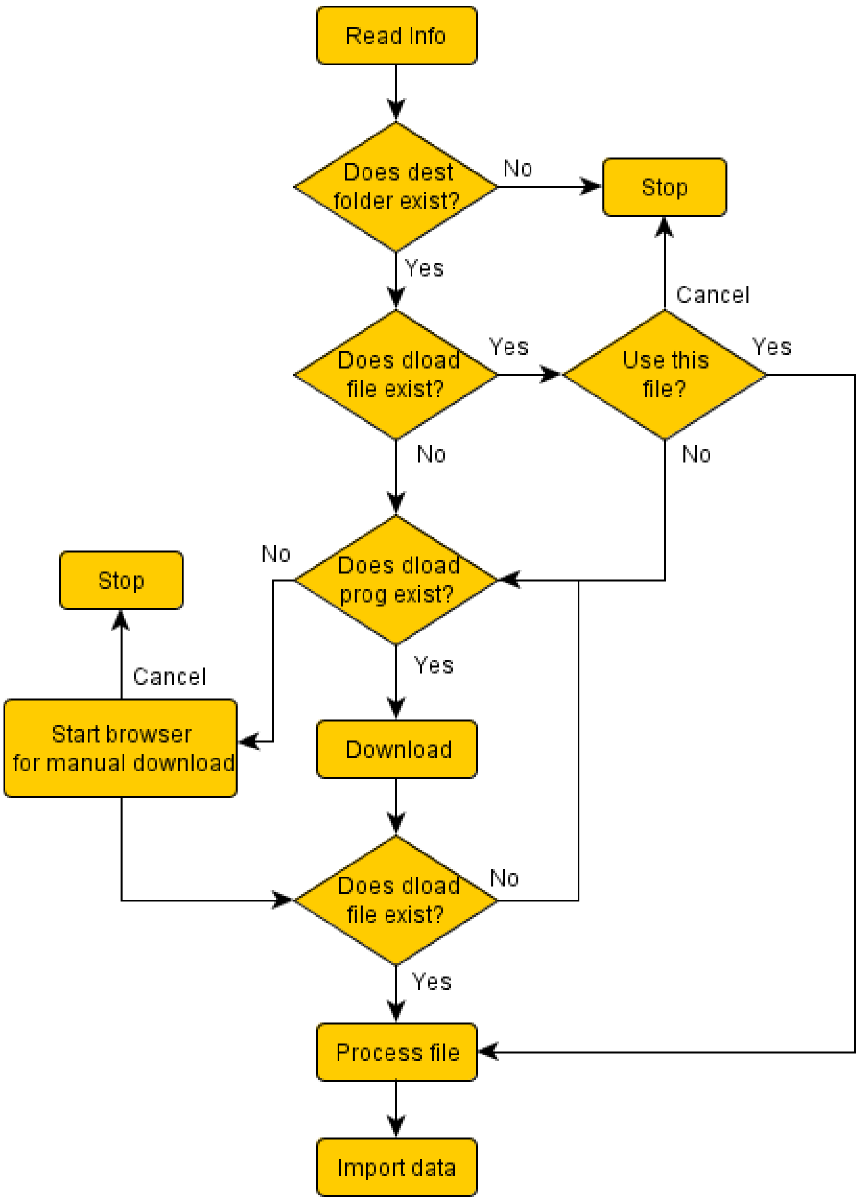
Note the line after `INPUT` statement. This clears the data displayed on the screen by the script by positioning the cursor at the start of the line , and then clearing to the end of the line. This also requires that the `INPUT` statement is terminated by a colon, so that the cursor stays on the input line.

7.2.5 Adding the download to the application

Now, we've got most of the pieces we need to download the source spreadsheets. Let's do a flow chart to work out the logic we'll have to use:

This logic flow requires that we be able to check the existence of files and directories on both the client and the server. As you may have guessed, this requires the use of more scripts on the client side, but we can use `QMBasic` directly for the server side tests. The following subroutine does these tests for us:

Automating Updates



```
SUBROUTINE FILEDIR.EXISTS(cs, path, stat)
*******************************************************************
**
* Test for valid file or directory on client or server
*
* Input
*   cs   - client or server (literal string)
*   path - pass the path of the file or directory to test
* Output
*   stat - return value: 'D' if directory
*                        'F' if file (or 'FH' for hashed file)
*                         null if cannot establish type
*
$CATALOGUE

EQUATE STX TO CHAR(2)
EQUATE CR  TO CHAR(13)
EQUATE EM  TO CHAR(25)
EQUATE ESC TO CHAR(27)

stat = ''
```

```
IF cs = '' THEN RETURN
IF path = '' THEN RETURN

cs = OCONV(cs[1,1], 'MCU')
BEGIN CASE
  CASE cs = 'C'
    GOSUB client
  CASE cs = 'S'
    GOSUB server
END CASE

RETURN
*
* -----------------------------------------------------------------
*
*
client:
*
script = 'On Error Resume Next'
script<-1> = 'Dim X'
script<-1> = 'X = 0'
script<-1> = 'X = GetAttr("':path:'")'
script<-1> = 'InitSession.Output CStr((X And 16) / 16) & vbCr'

script = CHANGE(script, @AM, EM)

CRT ESC:STX:'P':script:CR:
ECHO OFF
INPUT stat:
ECHO ON

IF stat THEN
  stat = 'D'
END ELSE
  script = 'On Error Resume Next'
  script<-1> = 'Dim X'
  script<-1> = 'X = 0'
  script<-1> = 'X = Abs(FileExists("':path:'"))'
  script<-1> = 'InitSession.Output Cstr(X) & vbCr'

  script = CHANGE(script, @AM, EM)

  CRT ESC:STX:'P':script:CR:

  ECHO OFF
  INPUT stat:
  ECHO ON

  IF stat THEN stat = 'F'
END
*
RETURN
*
* -----------------------------------------------------------------
*
server:
*
OPENPATH path TO fp THEN
  pathtype = FILEINFO(fp, 3)
  IF pathtype = 3 THEN stat = 'FH'      ; * Hashed file
  IF pathtype = 4 THEN stat = 'D'       ; * Directory
END ELSE
  OSREAD junk FROM path THEN
    stat = 'F'
  END
END

RETURN
*
* -----------------------------------------------------------------
*
END
```

Automating Updates

The two scripts in this subroutine have been taken from the program samples (see folder *C:\Program Files\Atwin\Samples\PICKBP*) that ship with *AccuTerm*.

The values returned by this subroutine are not entirely consistent. A pathname that resolves to the parent folder of a dynamic file will be evaluated as a file (*stat* = 'FH') by the server subroutine, but as a folder (*stat* = 'D') by the client subroutine. Both are correct – so it is up to you as the developer to be aware of this and to handle the data appropriately.

Now we can start putting the pieces in place. Add the following lines to the application header – just after we open the XED.VAR file:

```
clientname = ''
CALL GET.COMPUTERNAME('C', clientname)
```

From the logic flow diagram we created earlier, we can create the update button event handler as follows:

```
*-->BEGIN EVENT HANDLER<--*
GUI.XED.FRMMAIN.BTNUPDATE.CLICK: *
*
GOSUB get_download_config
GOSUB check_download_dest
IF destok THEN
  GOSUB check_download_file
  IF fileok THEN
    dldcancel = @FALSE
    IF dodownload THEN
      GOSUB check_download_prog
      GOSUB download_control
    END
    IF NOT(dldcancel) THEN
      GOSUB process_download
      GOSUB import_data
    END
  END
END
*
RETURN
*
*-->END EVENT HANDLER<--*
```

Now, we need to create the new subroutines referenced in this event handler. The subroutine to read the download configuration items looks like:

```
get_download_config:
*
id = 'DEST*':clientname
dload.path = ''
READV dload.path FROM xed.var, id, 1 ELSE NULL
IF NOT(dload.path) THEN
  READV dload.path FROM xed.var, 'DEST', 1 ELSE NULL
END

dprog = xfiledef<6>
dprog.path = ''
dprog.syntax = ''
READ temp FROM xed.var, dprog THEN
  dprog.path = temp<1>
  dprog.syntax = temp<2>
END

id = dprog:'*':clientname
READV temp FROM xed.var, id, 1 ELSE temp = ''
IF temp THEN
  dprog.path = temp
END

dload.url = xfiledef<7>
```

```
*
RETURN
```

The following subroutine validates the download destination:

```
check_download_dest:
*
destok = @TRUE
IF dload.path THEN
  stat = ''
  IF dload.path[1] NE '\' THEN dload.path := '\'
  CALL FILEDIR.EXISTS('C', dload.path, stat)
  IF stat NE 'D' THEN destok = @FALSE
END ELSE
  destok = @FALSE
END

IF NOT(destok) THEN
  emsg = 'Either no download destination has been defined'
  emsg := ' or the specified destination does not exist.'
END ELSE
  IF NOT(dload.url) THEN
    emsg = 'No URL has been defined for the download.'
    destok = @FALSE
  END ELSE
    IF dload.url[1,7] NE 'http://' THEN
      emsg = 'The download URL does not appear valid.'
      destok = @FALSE
    END
  END
END

IF NOT(destok) THEN
  CALL ATGUIMSGBOX(emsg, 'Configuration error', MBXICON, MBOK, '',
ok, guierrors, guistate)
  IF guierrors<1> GE 2 THEN GOTO gui.error
END
RETURN
```

while this subroutine checks to see whether the download file already exists:

```
check_download_file:
*
dfilename = FIELD(dload.url, '/', DCOUNT(dload.url, '/'))
dfilepath = dload.path:dfilename
CALL FILEDIR.EXISTS('C', dfilepath, stat)

fileok = @FALSE
dodownload = @FALSE
IF stat = 0 OR stat = 'F' THEN fileok = @TRUE
emsg = ''
IF NOT(fileok) THEN
  emsg = dfilepath:' is not a valid filename.'
  CALL ATGUIMSGBOX(emsg, 'File download', MBXICON, MBOK, '', ok,
guierrors, guistate)
  IF guierrors<1> GE 2 THEN GOTO gui.error
END ELSE
  IF stat = 'F' THEN
    emsg = 'File ':dfilepath:' already exists. '
    emsg := 'Use this file (Yes) or download new file (No)?'
    CALL ATGUIMSGBOX(emsg, 'File download', MBQICON, MBYESNOCANCEL,
'', ok, guierrors, guistate)
    IF guierrors<1> GE 2 THEN GOTO gui.error
    BEGIN CASE
      CASE ok EQ MBANSCANCEL ; fileok = @FALSE
      CASE ok EQ MBANSYES    ; dodownload = @FALSE
      CASE ok EQ MBANSNO     ; dodownload = @TRUE
    END CASE
  END ELSE
    dodownload = @TRUE
  END
END
```

```
*
RETURN
```

Checking the program nominated for the download is relatively simple:

```
check_download_prog:
*
dprogok = @FALSE
CALL FILEDIR.EXISTS('C', dprog.path, stat)
IF stat EQ 'F' THEN dprogok = @TRUE
*
RETURN
```

The download control needs to split the download process into two types – those where the nominated download program is present, and those that use the computer's browser to run the download:

```
download_control:
*
filedownloaded = @FALSE
LOOP
  IF dprogok THEN
    GOSUB download_using_prog
  END ELSE
    GOSUB download_using_browser
  END
  IF dldcancel THEN EXIT

  CALL FILEDIR.EXISTS('C', dfilepath, stat)
  IF stat = 'F' THEN
    filedownloaded = @TRUE
  END ELSE
    emsg = 'File has not been downloaded. '
    emsg := 'Click Retry to run the download again '
    emsg := 'or Cancel to quit.'
    CALL ATGUIMSGBOX(emsg, 'File download', MBEXICON,
MBRETRYCANCEL, '', ok, guierrors, guistate)
    IF guierrors<1> GE 2 THEN GOTO gui.error
    IF ok EQ MBANSCANCEL THEN EXIT
  END
UNTIL filedownloaded DO REPEAT
*
RETURN
```

The subroutine to download using the nominated program looks like:

```
download_using_prog:
*
cmd = dprog.path:' ':dprog.syntax
cmd = CHANGE(cmd, '%%DEST%%', dfilepath)
cmd = CHANGE(cmd, '%%URL%%', dload.url)
CRT ESC:STX:'<':cmd:CR:

emsg = 'Click OK when the download is complete, '
emsg := 'or Cancel to abandon the update.'
CALL ATGUIMSGBOX(emsg, 'File download', MBIICON, MBOKCANCEL, '',
ok, guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

IF ok EQ MBANSCANCEL THEN dldcancel = @TRUE
*
RETURN
```

while the browser download looks like:

```
download_using_browser:
*
emsg = 'This computer does not have the download program installed.
' emsg := 'Click OK to download using your browser, '
emsg := 'or Cancel to abandon the update.'
```

```
CALL ATGUIMSGBOX(emsg, 'File download', MBEXICON, MBOKCANCEL, '',
ok, guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

IF ok EQ MBANSOK THEN
  cmd = 'START ':dload.url
  CRT ESC:STX:'<':cmd:CR:

  emsg = 'Click OK when the download is complete, '
  emsg := 'or Cancel to abandon the update.'
  CALL ATGUIMSGBOX(emsg, 'File download', MBIICON, MBOKCANCEL, '',
ok, guierrors, guistate)
  IF guierrors<1> GE 2 THEN GOTO gui.error

  IF ok EQ MBANSCANCEL THEN dldcancel = @TRUE
END ELSE
  dldcancel = @TRUE
END
*
RETURN
```

Just put a couple of empty subroutines in place for *process_download* and *import_data* and we'll go through the subroutines we've just entered.

The logic of the event handler is straightforward, and closely matches the flow diagram that was created earlier.

The first subroutine called by the event hander is to get the download configuration items. This builds on information gathered earlier – the client computer name is gathered as the program starts, while the download program and download URL are read when we change the selected file in the file combo box.

Note the use of READV several times in this subroutine rather than READ. READV only reads a single field from the item rather than the whole item. In the case of the destination item, you may think this is unnecessary as the item only contains a single field, so a READ should be equivalent to a READV. However, if the item has more than a single field (even an empty field), then a READ will read include the extra field marks and data in the variable. Using a READV ensures that we only get the data we want.

Alternatively, we could have used a read/extract process as is done to get the path to the download program. Here, the item is read from the file using a READ statement, and then the path to the program extracted from the first field, and the syntax to use with the download program from the second field.

Finally in this subroutine, we check whether there is a client-specific location of the download program. This reads the first field of the item, and then tests the item to see if it has data. You could be tempted to use code like:

```
id = dprog:'*':clientname
READV temp FROM xed.var, id, 1 THEN
  dprog.path = temp
END
```

However, this is NOT the same. The THEN clause of the above statement is executed if *OpenQM* reads the item successfully. But a successful read does NOT mean there is data in the field. If the item exists on file, but has nothing in the first field (it could even be a completely empty item), then we would end up assigning a null value to the variable *dprog.path*.

The next subroutine (*check_download_dest*) checks that the download destination actually exists, and does a simple check on the URL. The main check for the existence of the download destination is done using the subroutine written earlier in this section. The check on the URL simply checks for the the string 'http://'. This may need amendment in the future if you need to download data from secure sites (https), or you used the ftp protocol.

Note that this subroutine adds a trailing '\' to the download destination if it does not already have one. This ensures that the variable is in a known state for later manipulation.

The *check_download_file* carries out a similar check to see whether the file to be downloaded already exists on the system. This subroutine begins with the following two lines:

```
dfilename = FIELD(dload.url, '/', DCOUNT(dload.url, '/'))
dfilepath = dload.path:dfilename
```

The first line extracts the filename of the file to be downloaded from the URL that has been provided. It does this by counting the "parts" of the URL that have been delimited by forward slash characters (/), and then extracting the last part of the URL.

The second line appends this filename to the download path that we manipulated and tested in the previous subroutine.

Note that these last two subroutines have made explicit reference to the slash characters (/ and \) as system delimiters. We can only do this if we are certain that the systems we are using actually use those delimiters.

In this case, we can make those assumptions because:

- all internet references use the forward slash as a delimiter
- we know the client is using *Windows* (which uses the backslash as a delimiter) because we are using *AccuTerm* (which only runs on *Windows*).

However, there may be situations where you want to build a path to a filename where either system delimiter could be in use. If this refers to a path on the server, we can use the @DS token in place of a specific delimiter, and it will work for all server operating systems. For example:

```
  IF dload.path[1] NE @DS THEN dload.path := @DS
```

Make sure you only do this for SERVER paths. The above line actually refers to a client path, and if used in our application, could lead to errors.

If the download file already exists, then the *check_download_file* subroutine gets the user's instruction on whether to use this file, or download a new copy (or cancel the update completely). This instruction is passed back to the event handler via the *dodownload* variable. Similarly, the *fileok* variable is set to @FALSE here if the user wants to cancel the download process.

If the user wishes to use the existing download data, the logic in the event handler skips the download process.

This subroutine has introduced a few more tokens defined for use with *AccuTerm* in the ATGUIEQUATES include item. These are the buttons, icons, and responses for use with the ATGUIMSGBOX subroutine. These purpose of these tokens should be self-evident from their names. The key advantage of using these tokens is that we replace numeric values with their mnemonic equivalents. For example, it is easier to remember the token MBQICON than it is to remember that a question mark icon requires a parameter value of 3 in the ATGUIMSGBOX subroutine.

The download process begins by checking that a program file exists at the location our control items suggest. This is our final check before beginning the download, and determines how we do the download.

The *download_control* subroutine is in a loop which will continue until a file has been downloaded onto the client PC, or until we cancel the download process. The main part of this subroutine splits the logic flow into two streams – one to download using the program we have carefully specified so far – the other to fall back to a default download process using the PC's browser. This makes an assumption that the PC has a browser, but that seems reasonable.

The `download_using_prog` subroutine first assembles the download command by appending the syntax portion of the command to the path of the program file. It then substitutes the actual download destination and source URL for the placeholders in the syntax definition. Finally, it uses *AccuTerm*'s private escape sequence for running a program to execute the command. In this case, we've used the variant that returns control to *AccuTerm* immediately following the command execution.

Once *AccuTerm* has regained control and signalled to *OpenQM* to continue, a dialog box is displayed asking the user to indicate when the download is complete. This means the download process requires some manual intervention, but this saves programming a checking routine to determine when the download is complete.

The `download_using_browser` subroutine is only slightly different. We start by informing the user that the specified download program has not been found on this client, and gives them the option of continuing using a browser download or cancelling. Assuming the user continues, the download command is quite simple:

```
cmd = 'START ':dload.url
CRT ESC:STX:'<':cmd:CR:
```

The `START` command simply tells *Windows* to deal with the URL that we pass to it. *Windows* should determine that it is a web reference, and start the default browser. In turn, the browser will determine that the file it is being asked to get is not an html file, and will ask the user whether they want to execute the file (offering the default program associated with xls files) or download it. Once again, the subroutine asks the user to confirm when the download is complete.

At this stage, you should be able to download the source files by clicking on the 'Update' button, with all appropriate error checking occurring. Now we need to do something with the downloaded data.

7.3 Processing the downloaded data

7.3.1 What processing is required?

We now have a spreadsheet on our system that we've downloaded from the internet. What do we do next?

Ultimately, we want to import part of that spreadsheet into our *OpenQM* database. So, we need to work backwards from that point:

- ➢ What file formats does our import process support?
- ➢ How should our data be laid out in the file for the import?
- ➢ What do we need to do to get the downloaded spreadsheet into that format and data layout?

AccuTerm provides a number of ways to import data. Some of these are driven by user responses to *AccuTerm* prompts. These can be incorporated into `QMBasic` programs by "stacking" the responses you want to use in `DATA` statements. However, that is rather clumsy.

We'll use the `FTIMPORT` subroutine which can be called directly from our `QMBasic` program. This subroutine imports data from text files. This largely defines the type of processing that we need to do – we need to transform the Excel spreadsheet into a `CSV` or `TXT` file.

To make things simple for the import, we'll lay out the `CSV` file in the order in which the fields are displayed in the *XED* application.

So, the question becomes, "How do we transform the spreadsheet we have downloaded to a `CSV` file in the layout that we want for the import without starting the application and doing the transformation manually?"

The answer lies in automation. Excel and OpenOffice both allow us to automate their actions via an exposed programming interface. *AccuTerm* provides the "Object Bridge" set of subroutines to access these programming interfaces, and we can also use scripts. In this case we are going to use a script.

7.3.2 Scripts revisited

The way we have written scripts so far has been like:

```
script = 'line 1'
script<-1> = 'next line'
script<-1> = 'next line'

CONVERT @AM TO EM IN script
CRT ESC:STX:'P':script:CR:
```

While this works, it is messy. Surely there is a better way?

Well, how about storing the script as a normal multi-value item in a file, and then doing something like:

```
READ script FROM scripts, scriptname THEN
  CONVERT @AM TO EM IN script
  CRT ESC:STX:'P':script:CR:
END
```

This means the script itself is much more readable. For example:

```
' Script to download the hb1.xls spreadsheet from RBNZ

  Dim Shell as Object
  Dim pgm as string
  Dim dest as string
  Dim URL as string
  Dim cmd as string
  const DQ = Chr$(34)

  Set Shell = CreateObject("WScript.Shell")
  pgm = "C:\Program Files\curl\curl.exe"
  dest = " -o C:\Temp\hb1.xls "
  URL = "http://www.rbnz.govt.nz/statistics/exandint/b1/hb1.xls"
  cmd = DQ & pgm & DQ & dest & URL

  Shell.Run cmd
  Set Shell = Nothing
```

Note the spaces on each end of the *`dest`* string ensuring that the concatenated *`cmd`* is a valid command for use on the command line.

We can create a generalised program to run scripts stored in a file:

```
PROGRAM RUN.SCRIPT
*********************************************************************
* Bp Run.Script - A command line utility to run a saved script.
*
* Author : BSS
* Created: 26 May 2009
* Updated: 26 May 2009
* Version: 1.0.0
*
* Usage: RUN.SCRIPT [filename] scriptname [NO.MSGS]
* If filename is omitted, then 'SCRIPTS' is assumed.
*
* ------------------------------------------------------------------ *
*
$CATALOGUE
PROMPT ''

cmd = @SENTENCE
cmd = OCONV(TRIM(cmd), 'MCU')
CONVERT ' ' TO @AM IN cmd

wordcount = DCOUNT(cmd, @AM)
lastword = cmd<wordcount>
IF lastword EQ 'NO.MSGS' OR lastword EQ 'NO-MSGS' THEN
  msgs = @FALSE
  DEL cmd<wordcount>
  wordcount -= 1
END ELSE
  msgs = @TRUE
END

IF wordcount LT 2 THEN
  IF msgs THEN CRT 'Usage: RUN.SCRIPT [filename] scriptname [NO.MSGS]'
  STOP
END

filename = 'SCRIPTS'
scriptname = cmd<2>
IF wordcount GT 2 THEN
  filename = cmd<2>
  scriptname = cmd<3>
END
*
OPEN filename TO fp ELSE STOP 201, filename
READ script FROM fp, scriptname ELSE
  IF msgs THEN CRT scriptname:' not on file: ':filename
  STOP
END

err = ''
IF msgs THEN CRT 'Running script: ':scriptname
CALL RUN.SCRIPT.SUB(script, err)

STOP
END
```

and the matching subroutine is:

```
SUBROUTINE RUN.SCRIPT.SUB(script, err)
**********************************************************************
* Bp Run.Script.Sub - A subroutine to run a passed script.
*
* Author:  BSS
* Created: 26 May 2009
* Updated: 26 May 2009
* Version: 1.0.0
*
* ------------------------------------------------------------------ *
*
$CATALOGUE

PROMPT ''
EQUATE STX TO CHAR(2)
EQUATE CR TO CHAR(13)
EQUATE EM TO CHAR(25)
EQUATE ESC TO CHAR(27)

err = ''
```

```
IF UNASSIGNED(script) OR script = '' THEN
  err = 1
  RETURN
END

CONVERT @AM TO EM IN script
CRT ESC:STX:'P':script:CR
RETURN

STOP
END
```

Now, let's assume that the script shown above to download the *hb1.xls* spreadsheet is stored in the (directory) file named SCRIPTS with a name of *DLOAD.HB1*. We can run this script by typing the following at the *OpenQM* command prompt:

```
RUN.SCRIPT SCRIPTS DLOAD.HB1
```

or:

```
RUN.SCRIPT DLOAD.HB1
```

This ability to write a script in plain text, and then execute it from the command line makes debugging the script somewhat easier.

You may wonder why the code to run the script has been split from the command line program. This is to allow this subroutine to be called from any program, and thereby simplify the process of running scripts. This may not save many lines of code, but it does make the code much more readable.

Consider the following two lines of code:

```
CRT ESC:STX:'P':script:CR
```

or:

```
CALL RUN.SCRIPT.SUB(script, err)
```

Which line is easier to read and understand?

7.3.3 Script actions

Now we have a way to easily write and run scripts. Let's consider what we want the script to do.

Open the *hb1.xls* spreadsheet using your spreadsheet program. OpenOffice Calc is used here, but most people will probably use Excel. This difference doesn't change the actions, but the script commands will be different for Excel than they are for OpenOffice.

We want the script to carry out the following actions:

- Load the *hb1.xls* spreadsheet
- Make the *monthly* sheet active
- Delete the top two rows of the spreadsheet
- Delete the new rows 2 and 3 (original rows 4 and 5)
- Insert two columns after column A
- Set the value in the cells in the new column B to be the year and month in *YYYYMM* format
- Copy the *TWI* from column I to the new column C
- Delete column A
- Delete all columns past column G

- Update the column headings to the names of the dictionary items we will use during the import
- Write the data that is now displayed out to a CSV file.

The following script carries out these actions:

```
' Script to take the hb1.xls spreadsheet downloaded from RBNZ and
' save it as a CSV file ready for importation into OpenQM

  Dim oSM as Object
  Dim oDesk as Object
  Dim oDoc as Object
  Dim oSheets as Object
  Dim oSheet as Object
  Dim oCell as Object
  Dim args()
  Dim URL as string
  Dim iCnt as long
  Dim iCol as long
  Dim iDate as long
  Dim nTWI as single
  Dim OutLine as string

  const DQ = Chr$(34)
  const Comma = Chr$(44)

  URL = "file:///c:/temp/hb1.xls"

  Set oSM = CreateObject("com.sun.star.ServiceManager")
  Set oDesk = oSM.createInstance("com.sun.star.frame.Desktop")
  Set oDoc = oDesk.loadComponentFromURL(URL, "_blank", 0, args)

  Set oSheets = oDoc.Sheets()
  Set oSheet = oSheets.getByName("monthly")

  oSheet.Rows.removeByIndex(0, 2)
  oSheet.Rows.removeByIndex(1, 2)
  oSheet.Columns.insertByIndex(1, 2)

  iCnt = 0
  Do
    iCnt = iCnt + 1
    Set oCell = oSheet.getCellByPosition(0, iCnt)
    iDate = oCell.value
    If iDate > 0 then
      Set oCell = oSheet.getCellByPosition(1, iCnt)
      Set oCell.value = Year(iDate) * 100 + Month(iDate)
      Set oCell = oSheet.getCellByPosition(8, iCnt)
      nTWI = oCell.value
      Set oCell = oSheet.getCellByPosition(2, iCnt)
      Set oCell.value = nTWI
    else
      Exit Do
    end if
  Loop

  oSheet.Columns.removeByIndex(0, 1)
  oSheet.Columns.removeByIndex(7, 5)

  For iCol = 0 to 6
    Set oCell = oSheet.getCellByPosition(iCol,0)
    Select Case iCol
      Case 0
        Set oCell.string = "@ID"
      Case 1
        Set oCell.string = "TWI"
      Case 2
        Set oCell.string = "USD"
      Case 3
        Set oCell.string = "GBP"
      Case 4
```

```
        Set oCell.string = "AUD"
      Case 5
        Set oCell.string = "JPY"
      Case 6
        Set oCell.string = "EUR"
    End Select
  Next iCol

  Open "c:\temp\hb1.csv" For Output as #1
  iCnt = 0
  Do
    OutLine = ""
    For iCol = 0 to 6
      Set oCell = oSheet.getCellByPosition(iCol, iCnt)
      If iCnt > 0 then
        OutLine = OutLine & oCell.Value & Comma
      else
        OutLine = OutLine & DQ & Trim(oCell.String) & DQ & Comma
      end if
    Next iCol
    Print #1, OutLine
    iCnt = iCnt + 1
    Set oCell = oSheet.getCellByPosition(0, iCnt)
    iDate = oCell.value
    If not(iDate > 0) then
      Exit Do
    end if
  Loop
  Close #1

  oDoc.Close(True)
  Set oDoc = Nothing
End Sub

Sub Dummy
```

Save this script as *`CSV.HB1`* into the `SCRIPTS` file (assuming you have created one).

The method that this script uses to save the manipulated data is fairly crude – it should be possible to simply save the data sheet using the routines built in to OpenOffice and applying the CSV filter.

There are a few points to note from this script:

- The script is specific to the *`hb1.xls`* spreadsheet
- The script is reliant on:
  - the location of the source spreadsheet not changing
  - the format of the spreadsheet not changing
- Likewise, this script will always write to a fixed location
- The script has some `End Sub` and `Sub Dummy` tags at the end.

The first point means that we will need to write equivalent scripts for each of the other imports.

We can overcome the fixed location of the source spreadsheet by replacing the location with a token as we did earlier during the download process, and then swapping the token for the actual location prior to running the script. However, if we are testing the script using the *`RUN.SCRIPT`* program written earlier, we need locations to be hard-coded.

The format of the spreadsheet is out of our control. We simply have to work on the basis that a changed format in a standard spreadsheet such as this will be a rare event.

The `End Sub` and `Sub Dummy` tags are not necessary for this script, because the script contains only a single subroutine. However, if the script called local subroutines, then we need to be aware of the way that *AccuTerm* handles these scripts:

When *AccuTerm* processes scripts, it places an implicit `Sub Main()` at the start of the script, and and `End Sub` at the end. Therefore, if we pass it a script without any subroutine tags, it will effectively end up with a declaration and a terminating statement.

However, if we pass it a script with multiple subroutines, each with appropriate declarations and end statements, then *AccuTerm* will add its own start and end tags, and the script will not compile. There are two solutions to this:

- Leave the declaration off your first subroutine (which must be the *`Main`* subroutine), and leave the `End Sub` statement off the last subroutine
- As we logically view a subroutine without a terminating `End Sub` statement as a programming error, the alternative approach is put the closing statement on the subroutine, but then add a *`Dummy`* subroutine at the end which only has the declaration statement. Once again, you need to leave off the declaration for the first subroutine.

It is probably worthwhile quickly going through the script. In general, the processing in the script closely follows the order of script actions identified earlier in this section. The general structure of the script is as follows:

```
Declare variables
Initialise the OpenOffice environment
Open the spreadsheet and set the active sheet
Remove rows and add columns
Walk down the spreadsheet setting YYYYMM in column B and copying
the TWI into column C
Remove first column, and trailing columns
Set the column headings
Open the output file
Write the data to the output file
Close the output file and the spreadsheet
Release the OpenOffice environment
```

The commands being executed here are a mixture of commands that are part of the *AccuTerm* scripting language, and commands being exposed by the *OpenOffice* API. If you use Excel, you would use the equivalent `VBA` commands for working with spreadsheets.

The step where the column headings are set may seem trivial, but this is quite important. These heading will communicate the dictionary names that will be used by *AccuTerm*'s import program for input conversion of the data. Therefore, a spelling mistake here, or use of a column heading that is not the name of a dictionary item in the target *OpenQM* file, will cause errors during the import.

Now, test the script by using the *`RUN.SCRIPT`* program created earlier:

```
RUN.SCRIPT SCRIPTS CSV.HB1
```

The *`RUN.SCRIPT`* program should respond with 'Running script: CSV.HB1'. You should also see the spreadsheet appear on the screen, and disappear fairly quickly. If you now look in the `C:\Temp` folder, you should find a file named *`hb1.csv`*, and if you open this using a text editor, the data should match that in the spreadsheet. The first few lines are shown below:

```
"@ID","TWI","USD","GBP","AUD","JPY","EUR",
199901,56.9303016662598,0.5392,0.3269,0.8518,60.9925,0.4653,
199902,58.1721992492676,0.5444,0.3341,0.8498,63.3954,0.4851,
199903,57.763801574707,0.5321,0.3283,0.844,63.6335,0.4886,
199904,58.9020004272461,0.5417,0.3368,0.8458,64.8221,0.5058,
```

Note that the *`TWI`* data is shown to an excessive level of precision. We'll leave this as it is, and see if the input conversion in the dictionary handles this during the import process.

Equivalent scripts now need to be created to process the interest rate data from the *`hb2.xls`* spreadsheet, and the (daily) foreign exchange data from the *`hb4.xls`* spreadsheet. Test these scripts against the spreadsheets you have downloaded to make sure they work correctly. Example scripts for processing these spreadsheets are shown in the Appendix.

Note that when we need to manipulate the date in the `hb4.xls` spreadsheet so that we can import it into *OpenQM*. While the spreadsheet is displaying a date, this is actually stored in the spreadsheet as a serial number, and if we write the cell contents out using a script similar to that shown above, we will actually get the serial number in the `CSV` file. If we then use the `DATE` dictionary item in the import process, the date conversion will not produce the correct internal date number for *OpenQM*.

There are a couple of ways around this:

- We could change the script that outputs the serial date number to use the *OpenQM* internal date number, and then import this date number without conversion
- We could get the script to output a date string, and then use the `DATE` dictionary item to import and convert this into the correct serial date number.

The second option has the additional advantage of making date readable within the `CSV` file, thereby making it easy to check the output of the script. The date manipulation within the script could be like:

```
  iCnt = 0
  Do
    iCnt = iCnt + 1
    Set oCell = oSheet.getCellByPosition(0, iCnt)
    iDate = oCell.value
    If iDate > 0 then
      Set oCell = oSheet.getCellByPosition(1, iCnt)
      Set oCell.string = Trim(Str(Day(iDate))) & "/" &
Trim(Str(Month(iDate))) & "/" & Trim(Str(Year(iDate)))
    else
      Exit Do
    end if
  Loop
```

Note that this code fragment creates a date string in the international date format. If you are using a North American date format, you will need to adjust the formula accordingly.

Having a string in the output data also changes the script for writing the data to the file:

```
  Open "c:\temp\hb4.csv" For Output as #1
  iCnt = 0
  Do
    OutLine = ""
    For iCol = 0 to 7
      Set oCell = oSheet.getCellByPosition(iCol, iCnt)
      If iCnt > 0 then
        If iCol > 0 then
          OutLine = OutLine & oCell.Value & Comma
        else
          OutLine = DQ & Trim(oCell.String) & DQ & Comma
        end if
      else
        OutLine = OutLine & DQ & Trim(oCell.String) & DQ & Comma
      end if
    Next iCol
    Print #1, OutLine
    iCnt = iCnt + 1
    Set oCell = oSheet.getCellByPosition(0, iCnt)
    sDate = oCell.string
    If not(sDate > "") then
      Exit Do
    end if
  Loop
  Close #1
```

This gets the string data from the cell's 'String' property and the numeric data from the cell's 'Value' property. Finally, it tests whether to output another line using a string comparison rather than a numeric comparison as in the earlier script.

7.3.4 Linking the scripts to the update process

We now have a set of scripts that will transform the downloaded spreadsheets to `CSV` data files ready for importing into *OpenQM*. What we need to do now is tell our application which script to run for each file.

As with our previous instructions of this type, we'll include the script name in the file definition variable in the `XED.VAR` file. This will be stored in field number 8 in each item. Set up a dictionary item describing this position in the `XED.VAR` file, and enter your script names in the file definition items:

```
SORT XED.VAR WITH SCRIPT > "" SCRIPT
XED.VAR...    Script....
FX.DAILY      CSV.HB4
IRATES        CSV.HB2
XRATES        CSV.HB1
```

Now we need to run the appropriate script from the *`process_download`* subroutine:

```
process_download:
*
xscript = xfiledef<8>
IF xscript THEN
  READ script FROM scripts, xscript THEN
    cfilepath = dfilepath
    dfilepath = CHANGE(dfilepath, '\', '/')
    script = CHANGE(script, '%%SOURCE%%', dfilepath)
    cfilepath = CHANGE(cfilepath, '.xls', '.csv')
    script = CHANGE(script, '%%DEST%%', cfilepath)
    CALL RUN.SCRIPT.SUB(script, err)
  END
END
*
RETURN
```

This code fragment reads from the *`scripts`* file variable – so we need to open the appropriate file and assign this file variable. Include the following line at the start of the program where the other files are opened:

```
OPEN 'SCRIPTS' TO scripts ELSE STOP 201, 'Scripts'
```

The subroutine allows the source and destination paths used by the script to use similar tokens as were used in the download process. If you leave the pathnames hard-coded in the scripts, these substitution lines will effectively be ignored – the scripts will simply run using the hard-coded pathnames.

Note the line that swaps the backslash delimiters for forward slash delimiters in the pathname. This is a requirement for OpenOffice, and is probably not required if you are using Excel.

At this stage, the program should call scripts and create the transfer files when the 'Update' button is pressed. However, there is still an issue we need to deal with here.

Once the script has been passed to *AccuTerm* for processing, normal program flow resumes within the `QMBasic` program – even though the script is only just starting to execute on the client. This creates a timing issue – we don't want to run the import subroutine until the transfer file has been created – but we don't know how long that will take.

Clearly, we need some way for the client process to signal to the server that it has finished processing the spreadsheet.

Add the following lines to the start of each script, immediately following the variable declarations:

```
  Open "%%LOCK%%" For Output as #2
  Print #2, "Locked"
  Close #2
```

and then near the end of each script, after the output file has been closed:

```
  Kill "%%LOCK%%"
```

Now, update the *process_download* subroutine:

```
process_download:
*
xscript = xfiledef<8>
IF xscript THEN
  READ script FROM scripts, xscript THEN
    cfilepath = dfilepath
    dfilepath = CHANGE(dfilepath, '\', '/')
    script = CHANGE(script, '%%SOURCE%%', dfilepath)
    cfilepath = CHANGE(cfilepath, '.xls', '.csv')
    script = CHANGE(script, '%%DEST%%', cfilepath)
    lfilepath = CHANGE(cfilepath, '.csv', '.lck')
    script = CHANGE(script, '%%LOCK%%', lfilepath)
    CALL RUN.SCRIPT.SUB(script, err)

    LOOP
      CALL FILEDIR.EXISTS('C', lfilepath, stat)
    UNTIL NOT(stat) DO
      SLEEP 1
    REPEAT
  END
END
*
RETURN
```

These changes create a “lock” file at the start of the script processing. This lock file is then killed (deleted) once the main script has finished processing. The loop following the call to the *RUN.SCRIPT.SUB* subroutine checks for the existence of this lock file, and continues to loop until the lock file does not exist.

While this correctly delays the passing of control to the next subroutine, it doesn’t work as you might expect from looking at the logic. We can tell this by putting a CRT statement in the loop to see how many times it looks for the lock file. The CRT statement tells us that the loop only checks for the file once, and finds that it is missing on that check – which implies that the subroutine processing has stopped while the script executes.

What is really happening is that FILEDIR.EXISTS subroutine uses a script to check for the file on the client machine. However, that script cannot run until the earlier script has finished processing. Meanwhile, the FILEDIR.EXISTS subroutine waits for a response from its script, which doesn’t come until both scripts have completed.

The loop is therefore redundant. This section of code would work just as well with only the call to the FILEDIR.EXISTS subroutine. Despite this, it may be better to leave the loop in place – because it makes the delaying nature of the code more apparent. Some comments in the code would also be appropriate.

7.4 Importing the CSV data file

It was noted earlier that we will import the data using the *FTIMPORT* subroutine. This subroutine is part of the *AccuTerm* suite of utilities, and the documentation for the subroutine is in the header of the program (in the FTBP file).

Add the following code to the *`import_data`* subroutine:

```
import_data:
*
mode = 'K,N,C,H'
pcfile = cfilepath
hostfile = selfile
attrs = ''
hdrskip = 1
idprefix = ''
idstart = ''
itemcnt = 0
bytecnt = 0
stat = ''

CALL FTIMPORT(mode, pcfile, hostfile, attrs, hdrskip, idprefix,
idstart, itemcnt, bytecnt, stat)
GOSUB get_first_last

IF stat THEN
  emsg = 'Error encountered during update: ':stat
  mbicon = MBXICON
END ELSE
  emsg = 'File ':selfile:' successfully updated'
  mbicon = MBIICON
END

CALL ATGUIMSGBOX(emsg, 'File update', mbicon, MBOK, '', ok,
guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error
*
RETURN
```

The modes specified in this import are:

- Use the Kermit protocol for the data transfer. This is the *AccuTerm* recommendation
- Don't overwrite existing items
- The source file is comma delimited
- The first line of the source file contains the dictionary items to use for import conversions.

The subroutine returns the overall status of the import in *`stat`*, the number of items imported in *`itemcnt`*, and the number of bytes imported in *`bytecnt`*. This would enable a more informative status message to be displayed than is created here.

Once the import has been run, the subroutine calls the internal subroutine to update the first and last item-id's for the file.

Running the program now results in the data files being correctly updated, but a few issues remain:

- The message 'Converting hb1.csv from `CSV` text format for upload... done.' appears on the screen behind the GUI application during the update process
- The combo boxes and grid do not update to reflect the new data that has been loaded
- We don't have a way to update the `FX.MONTHLY` file (as this data was derived from the `FX.DAILY` data).

The message that is output on the screen is generated by the *`FTMODE`* subroutine, which is called from *`FTIMPORT`*. Examination of that program indicates that the message cannot be suppressed by means of any option – but it could be commented out.

Often such messages can be suppressed by hushing the terminal (see the documentation for the HUSH statement). In this case, however, adding a HUSH ON / HUSH OFF pair around the call to *FTIMPORT*' prevents *AccuTerm* from running the file transfer – the error message generated is: '*FTIMPORT* requires *AccuTerm* to run!'

So, the choices are either to comment out the message lines in *FTMODE*, or accept the message appearing on the background screen. Given that *FTMODE* is probably used by a number of other *AccuTerm* programs in a variety of situations, it is probably best to leave this program 'as is' and accept the message appearing on the background screen.

To update the combo boxes and grid, add the following line to the 'Update' event handler after the GOSUB to *import_data*:

```
GOSUB reload_data
```

and create the subroutine:

```
reload_data:
*
GOSUB setup_combo_boxes
IF xkeystruct EQ 'D' THEN
  GOSUB setup_months
  GOSUB read_data_daily
END ELSE
  GOSUB read_data_months
END

ctrlid     = 'CMBYEAR'  ;  property     = GPVALUE    ;   prop.value
= selyear
IF xkeystruct = 'D' THEN
  ctrlid<-1> = 'CMBMONTH' ;  property<-1> = GPVALUE  ;
prop.value<-1> = selmonth
END

CALL ATGUISETPROPS(guiapp, guifrm, ctrlid, property, prop.value,
guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

GOSUB load_grid
*
RETURN
```

This reuses the existing subroutines to reload the combo boxes and the data for the selected file/year/month combination. It then assigns the previously selected year and month values to the combo boxes so these display the correct values, and then calls the subroutine to load the grid. At the end of all this, you should be on the same grid "page" as you were before the start of the update process, but any later data that has been added via the update process will now be visible on the page, or be available for selection via the combo boxes.

Now we need to update the FX.MONTHLY file from the data in the FX.DAILY file. In Part 1 of 'Getting Started in *OpenQM*', the following QMQuery statement was used to create the data in the FX.MONTHLY file:

```
SREFORMAT FX.DAILY BREAK.ON YYYYMM TOTAL ALL CONV "MR" TOTAL USD
CONV "MR" TOTAL GBP CONV "MR" TOTAL AUD CONV "MR" TOTAL JPY CONV
"MR" TOTAL EUR CONV "MR" TOTAL OTHER CONV "MR" DET.SUP COUNT.SUP
```

We need to execute this statement after new data has been imported into the *OpenQM* data files. To make this a self-contained statement, we should also add the optional TO clause, to specify the destination file. This adds the following phrase to the end of the statement:

```
TO FX.MONTHLY
```

We also need to consider how we are going to execute this statement. This doesn't refer to what syntax we need to use, but how do we structure the program to run this statement – or any other type of processing for another data file.

Let's make the general assumption that we want to allow processing of the data following the data import. Clearly, the nature of the processing will vary – in most cases there will be no post-import processing; some imports will require a single `QMQuery` statement to be run to create a new data file (as above); other imports may require multiple statements to be run,, or complex processing in a `QMBasic` program. In a general sense, how do we allow these various types of processing to take place?

One of the general principles that has been used in the development of the *XED* program is that key aspects of the program's behaviour are controlled by parameters stored in the `XED.VAR` control file. This lets us add new files to be displayed and edited by the *XED* program without making any changes to the program itself.

Tracking this forward, this also means that we should allow for post-import processing to take place for files that we have not yet included in our control file – without making changes to the program itself. In other words, we need a generalised way to run the post-import processing, and the obvious way is to include the link to that processing in our control file.

Create a new dictionary item for the *XED.VAR* file for field 9. This will hold the name of an external subroutine to be called after the data has been imported for any given file defined in *XED.VAR*:

```
LIST XED.VAR WITH PISUB GT "" PISUB
XED.VAR...    Post Import Sub.....
FX.DAILY      XED.PISUB.FX.DAILY
```

This shows that we have defined a subroutine named *XED.PISUB.FX.DAILY* to run after the `FX.DAILY` data has been imported.

Create a new subroutine to call the external subroutine that we have just defined:

```
post_import_processing:
*
pisub = xfiledef<9>
IF pisub THEN
  IF CATALOGUED(pisub) THEN
    CALL @pisub
  END
END
*
RETURN
```

and call this subroutine from the 'Update button' event handler:

```
    IF NOT(dldcancel) THEN
      GOSUB process_download
      GOSUB import_data
      GOSUB post_import_processing
      GOSUB reload_data
    END
```

Now, create the external subroutine:

```
SUBROUTINE XED.PISUB.FX.DAILY
**********************************************************************
**
* Bp.Gui Xed.PISub.FX.Daily
*
* Author : BSS
* Date   : 04 Nov 2009
* Updated: 18 Nov 2009
* Version: 1.0.0
*
```

```
* ------------------------------------------------------------ *
*
$CATALOGUE

OPEN 'XED.VAR' TO xed.var ELSE STOP 201, 'Xed.Var'
OPEN 'FX.MONTHLY' TO fx.monthly ELSE STOP 201, 'Fx.Monthly'

stmt = 'SREFORMAT FX.DAILY BREAK.ON YYYYMM TOTAL ALL CONV "MR" '
stmt := 'TOTAL USD CONV "MR" TOTAL GBP CONV "MR" '
stmt := 'TOTAL AUD CONV "MR" TOTAL JPY CONV "MR" '
stmt := 'TOTAL EUR CONV "MR" TOTAL OTHER CONV "MR" '
stmt := 'DET.SUP COUNT.SUP TO FX.MONTHLY'

EXECUTE stmt CAPTURING junk

READ xfiledef FROM xed.var, 'FX.MONTHLY' ELSE xfiledef = ''
xfirst = xfiledef<4>
xlast = xfiledef<5>

SELECT fx.monthly
LOOP
  READNEXT data.id ELSE EXIT
  BEGIN CASE
    CASE xfirst = ''
      xfirst = data.id
      xlast = data.id
    CASE data.id LT xfirst
      xfirst = data.id
    CASE data.id GT xlast
      xlast = data.id
  END CASE
REPEAT

xfiledef<4> = xfirst
xfiledef<5> = xlast

xid = 'FX.MONTHLY'
xrec = xfiledef

writeok = @TRUE
READU dummy FROM xed.var, xid LOCKED
  writeok = @FALSE
END ELSE
  NULL
END

IF writeok THEN
  WRITE xrec ON xed.var, xid
END ELSE
  RELEASE xed.var, xid
END

RETURN
*
* ------------------------------------------------------------ *
*
END
```

Let's look at the external subroutine first. This does the following:

- Executes the `SREFORMAT` statement to update the *`FX.MONTHLY`* file
- Loops through the *`FX.MONTHLY`* file to find the updated first and last item-ids
- Writes the updated values back to the control file.

You should recognise the code fragments used to get the first and last item-ids and carry out the file update, as these were copied from the main *`XED`* program. Note that there is a small possibility that the control file will not be updated – if the `FX.MONTHLY` control item is locked by another process when this subroutine runs, then the subroutine will not write the updated item.

Looking now at the *post_import_processing* subroutine, this firstly retrieves the name of the subroutine to run from the control item. It then carries checks that (a) the variable holds some data; and (b) that the system recognises this data as a catalogued subroutine. Only then does it try calling the subroutine using the line:

```
CALL @pisub
```

The `@` symbol indicates to `QMBasic` that the name of the subroutine to call is held by the variable *pisub* rather, and is not literally named *@pisub*.

We can use parameters in these indirect calls – for example:

```
CALL @pisub(p1, p2, p3)
```

Of course, this would require our subroutine to be defined to use these parameters.

While this may be useful in some situations, it implies any subroutine that is called from this point in the *XED* program must use (or at least be defined with) the same set of parameters. Given that we don't know what other subroutines we might call in the future from this point, it is better to define the subroutine without parameters, and ensure that the subroutine can run as a self-contained unit.

8 Review of the Application

8.1 The good, the bad, and the ugly

At this point, the application is basically complete. Let's consider what is good in the application, and what is not so good:

8.1.1 Good points

What is good with this application?

- It works! We shouldn't overlook this – it is easy to get carried away thinking that the application could be improved, and forget that it actually does what it was designed to do
- The application could be ported to another multi-value database with almost no changes. This isn't important if you only work with *OpenQM*, but if you are a software developer needing to develop applications across multiple platforms, this is vital
- The application is extensible to other files on your system simply by entering those files into the `XED.VAR` control file
- Likewise, the ability to specify different download programs for different files to be updated, and differing locations of the download programs on different client PC's is good
- It has been a great learning exercise for using *OpenQM* and *AccuTerm* to create a GUI application.

8.1.2 Not so good points

No application is perfect. There will always be things that it doesn't do, or could do better. What are some of the things that could be improved in this application?

General issues

- The application does not take full advantage of the features of `QMBasic`. In particular, the application could have been written to use local variables, local subroutines, and/or object-orientated programming. On the other hand, using these features would destroy the portability to other multi-value databases touted as an advantage above.
- The code is not well commented. This will mean that when you go back to modify the code, you may have to spend some time looking through the code to understand exactly why you wrote it that way in the first place.
- The code has not been fully tokenised. In particular, references to items in the `XED.VAR` file have been by field number rather than by a token (or equated value) representing that position. Such tokens could have been created using the `GENERATE` verb outlined in Section 3.2.2, and would replace code like:

  ```
  pisub = xfiledef<9>
  ```

  with:

  ```
  pisub = xfiledef<XV.PISUB>
  ```

  The application developed this way because, when we started, we did not have a clear idea of the contents or structure of `XED.VAR`
- The application doesn't provide any means for adding new files to the application – the control file must be edited manually
- There is no help system
- The application features some code which works by accident rather than by design – see the discussion at the end of Section 7.3. It would be better if all the code worked as expected. Fixing this requires some means by which the server could read the client filesystem (or vice-versa) – which is possible, but is best done with knowledge of the specific network topology in use on the system.

Structural issues

- The application could have been planned better. This would have enabled the control file to be constructed properly and the control file tokens generated before we started on the application code. On the other hand, many applications are written in an iterative fashion, with greater structure being imposed as development continues
- The application is written in a 'monolithic' fashion. The whole application is contained in just one program, with minimal use of external subroutines (other than the *AccuTerm* GUI framework) to carry out specific functions. This minimises the opportunity for code re-use – that is, sharing code modules between applications.

8.1.3 Operational improvements

We could also make some minor changes to the application to improve the user experience:

- Once a user selects a file to view/edit, they currently have to additionally select the year (and month). An improvement to the user experience would be to automatically display the data for the latest year (and month). This reduces the effort required to display the data if it is the latest data that is required. If the user wants to see some other time period, they will be no worse off than they are currently

- Some '+' and '-' buttons (or spin buttons) beside the combo boxes would also be handy for changing the year (and month). These would be relatively simple to add in, with most of the code already in place
- Many users expect a grid to have context-menu functionality. That is, a user expects to be able to right-click on a cell, and be presented with a menu offering valid options for that cell (copy, paste, delete etc). The *AccuTerm* GUI does offer both normal menus (at the top of the *Windows* form) and context menus, which are not difficult to implement. See the documentation for more information.

8.2 Addressing the issues

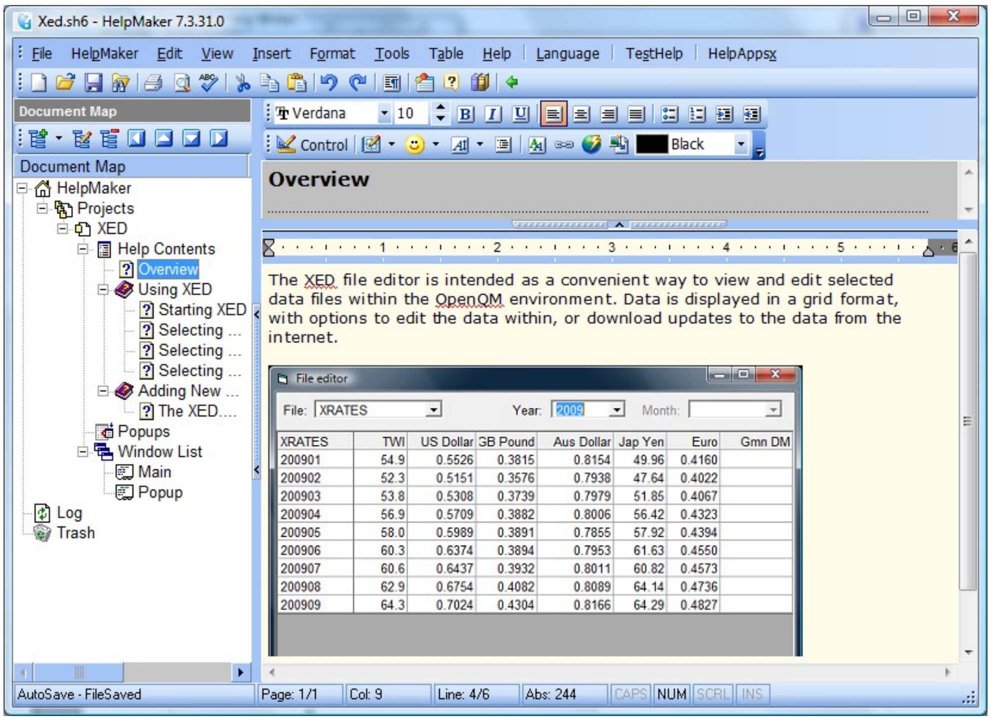
Having identified a number of issues with the application, what are we going to do about them?

For most of the issues identified, the answer (as far as this book is concerned) is nothing – those issues are for you to fix! However, we will cover the addition of help to the application.

Windows help

Windows uses a compiled help file comprising a number of html pages. You can create the pages using any html editor and then use the Help SDK (freely downloadable from Microsoft) to create the compiled help file – but this is far from an easy task.

The easiest way to create a *Windows* help file is to use a dedicated Help utility. Luckily, there is one freely available – with free (in this case) meaning "at no cost". You can download *HelpMaker* from http://www.vizacc.com/



Overview

The XED file editor is intended as a convenient way to view and edit selected data files within the OpenQM environment. Data is displayed in a grid format, with options to edit the data within, or download updates to the data from the internet.

File: XRATES Year: 2009 Month:

XRATES	TWI	US Dollar	GB Pound	Aus Dollar	Jap Yen	Euro	Gmn DM
200901	54.9	0.5526	0.3815	0.8154	49.96	0.4160	
200902	52.3	0.5151	0.3576	0.7938	47.64	0.4022	
200903	53.8	0.5308	0.3739	0.7979	51.85	0.4067	
200904	56.9	0.5709	0.3882	0.8006	56.42	0.4323	
200905	58.0	0.5989	0.3891	0.7855	57.92	0.4394	
200906	60.3	0.6374	0.3894	0.7953	61.63	0.4550	
200907	60.6	0.6437	0.3932	0.8011	60.82	0.4573	
200908	62.9	0.6754	0.4082	0.8089	64.14	0.4736	
200909	64.3	0.7024	0.4304	0.8166	64.29	0.4827	

The screenshot above shows *HelpMaker* with a simple help file created for use with `XED`:

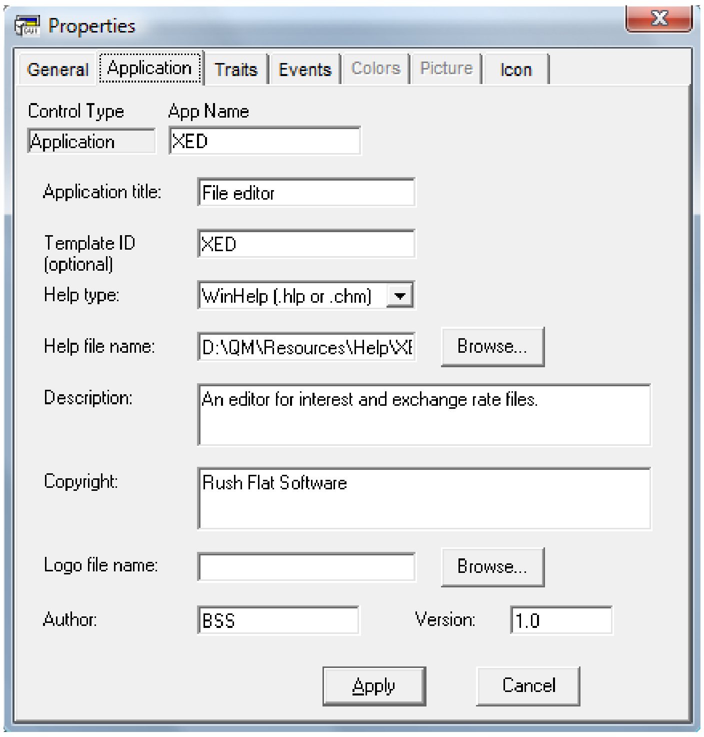
Once you have created the pages for the help file, you can create the compiled help file by clicking on the 'Compile' icon, or selecting 'Tools | Compile' through the menu. This will create a 'chm' file in the 'Project_temphhp' folder beneath your help 'Project' working folder (where 'Project' is the name of your help file). In this case, the help file is named *`XED.chm`*.

Review of the Application

Move this help file to a convenient location. Now, all we need to do is connect our application to the help file.

Firstly, we make a general connection. This means that the help file will open to its default page whenever we press ‘F1’ in the application.

Open the *XED* application in the GED editor. Right-click the application itself in the right-hand pane, and choose ‘Properties’. Select the ‘Application’ tab, click on the ‘Browse’ button beside the ‘Help file name:’ edit box, and select your newly created help file. Click on ‘Apply’, and save the project within GED.



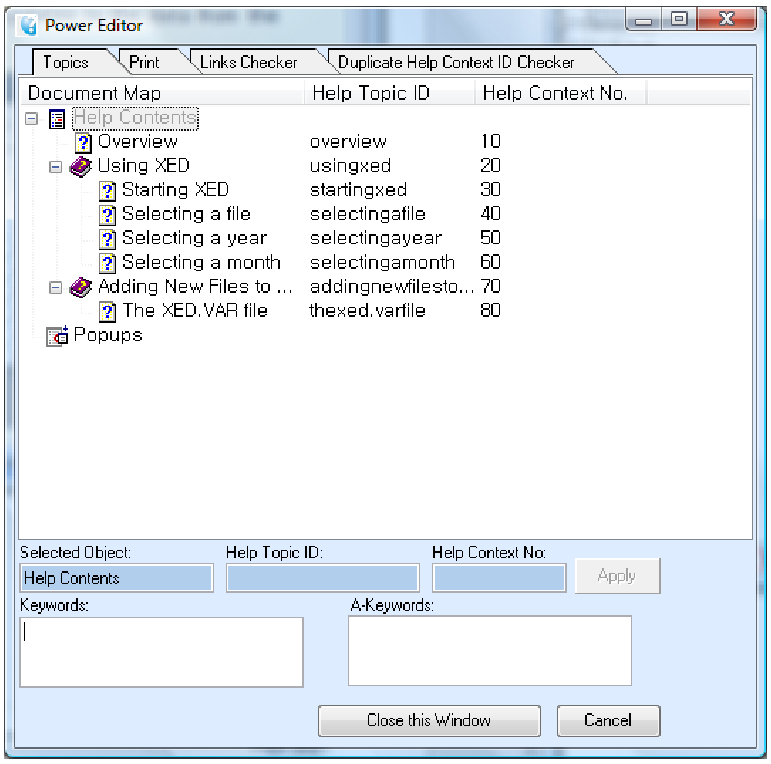
Control Type: Application
App Name: XED
Application title: File editor
Template ID (optional): XED
Help type: WinHelp (.hlp or .chm)
Help file name: D:\QM\Resources\Help\XE
Description: An editor for interest and exchange rate files.
Copyright: Rush Flat Software
Logo file name:
Author: BSS
Version: 1.0

Now run the *XED* application, and press the ‘F1’ key. The help file should open onto its default page:

The next step is to make the help context sensitive. That is, the help that is displayed should be relevant to what you are doing in the application, rather than simply opening at the default page and forcing you to search through the help to find the relevant information.

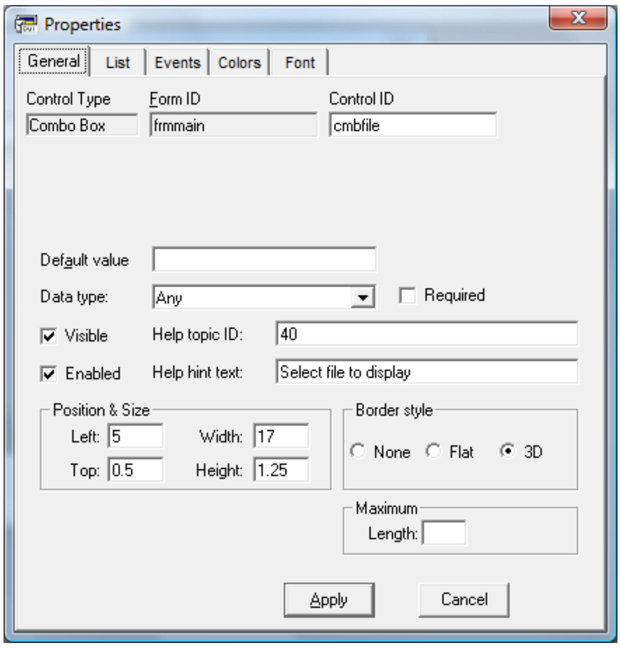
To do this, we need the help context numbers. You can display these in *HelpMaker* by choosing ‘HelpMaker | Power Editor’ from the menu bar. For example, the page ‘Selecting a file’ has a help context number of 40 in the help file. We use this number to open the help file at the right page.

Once you have the list of context numbers, we need to enter them into the application definition. Open the application using GED again, and select the control you wish to apply context sensitive to. Right-click on the control and select ‘Properties’. Enter the help context number in the ‘Help topic ID’ edit box, and enter a hint in the ‘Help hint text’ edit box:



Document Map	Help Topic ID	Help Context No.
Help Contents		
Overview	overview	10
Using XED	usingxed	20
Starting XED	startingxed	30
Selecting a file	selectingafile	40
Selecting a year	selectingayear	50
Selecting a month	selectingamonth	60
Adding New Files to ...	addingnewfilesto...	70
The XED.VAR file	thexed.varfile	80
Popups		

Selected Object: Help Contents
Help Topic ID:
Help Context No:
Keywords:
A-Keywords:



Control Type: Combo Box
Form ID: frmmain
Control ID: cmbfile
Default value:
Data type: Any
Required: ☐
Visible: ☑
Help topic ID: 40
Enabled: ☑
Help hint text: Select file to display
Position & Size — Left: 5 Width: 17 Top: 0.5 Height: 1.25
Border style: None Flat [✓ 3D]
Maximum Length:

Click on ‘Apply’, and then save the changes to the application design. Start the application and press the ‘F1’ key. As the focus is on the ‘File:’ combo box immediately after starting, our new context sensitive help should apply, and the help file will open at the ‘Selecting a file’ page. Similarly, if you hover the mouse pointer over the file combo box, you should see your help hint appear.

8.3 Application Summary

In building the *XED* application, we have covered an enormous amount of material. Here are some of the things you should have learned:

Review of the Application

GUI Designer

- Create a GUI interface for an application
- Add events for the visual controls in the application
- Add context sensitive help to the controls.

Programming

- Break a program into small blocks using subroutines
- Call internal subroutines
- Call an external subroutine
- Call an external subroutine indirectly (using a subroutine name contained in a variable)
- Use LOOP, IF, and CASE statements for program flow control
- Use the DCOUNT and LOCATE statements
- Use the *AccuTerm* GUI subroutines to control a GUI application
- Use the *AccuTerm* file transfer subroutines to import data into *OpenQM*
- Open a file, and read and write data from/to the file
- Use record locking to ensure that multiple users cannot update the data at the same time
- Read the descriptive data about a file from the file's dictionary
- Use the descriptive data from the dictionary to convert the data to an output format for display
- Load the data into the *AccuTerm* grid for display
- Use EXECUTE to run an *OpenQM* command or program
- Use OS.EXECUTE to run an external program on the *OpenQM* server
- Run a phantom (background) program on the *OpenQM* server
- Run scripts on the *AccuTerm* client
- How to start a program on the client using an *AccuTerm* script
- Use information stored in a file to control the way the application operates
- Use tokens to generalise your coding

9 Alternative *AccuTerm*

9.1 Defining The GUI From Within The Program

In Section 5.2, Creating the Initial Form, we used the *AccuTerm* GUI designer to create the GUI interface for the application. This isn't the only way to create a GUI interface with *AccuTerm*.

We can use the *AccuTerm* subroutines to generate a GUI interface from within the application. The following program gives a simple example:

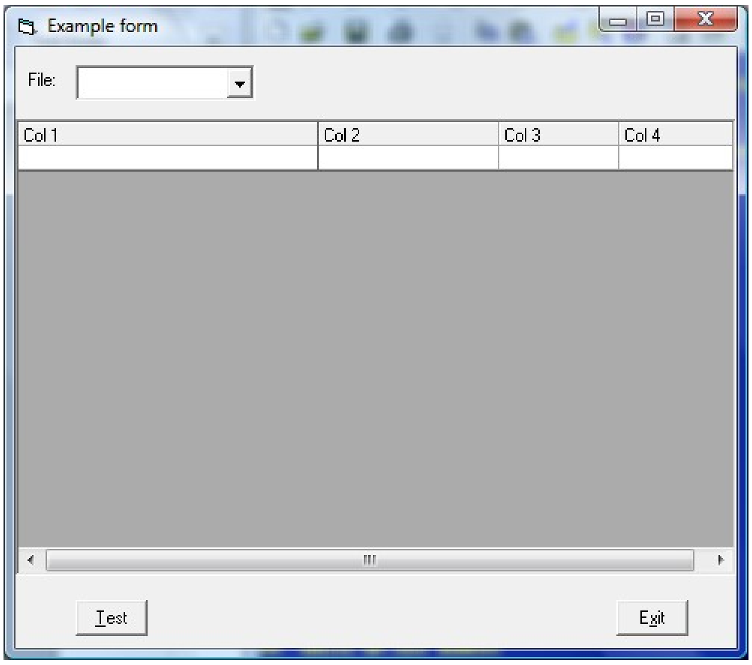
```
PROGRAM ABC
* Bp.Gui ABC - A test program for generating a GUI application.
*
$CATALOGUE
$INCLUDE GUIBP ATGUIEQUATES

PROMPT ''
guiapp = 'ABC'
guifrm = 'FRMMAIN'

CALL ATGUIINIT2(1.3, '', guierrors, guistate)
IF guierrors<1> GE 3 THEN GOTO gui.error

GOSUB buildform

CALL ATGUISHOW(guiapp, guifrm, '', '', guierrors, guistate)
IF guierrors<1> GE 3 THEN GOTO gui.error

LOOP
  CALL ATGUIWAITEVENT(guiapp,guifrm,guictl,guievt,guiargs,
guierrors,guistate)
  IF guierrors<1> GE 2 THEN GOTO gui.error
UNTIL guievt = GEQUIT DO
  guiapp=OCONV(guiapp, 'MCU')
  guifrm=OCONV(guifrm, 'MCU')
  guictl=OCONV(guictl, 'MCU')
  GOSUB gui.decode.event
REPEAT
```

```
CALL ATGUISHUTDOWN

STOP
*
* ----------------------------------------------------------------- *
*
gui.decode.event:
*
IF NUM(guievt) THEN
  GOSUB convertevent
  IF guictl THEN
    subid = guiapp:'.':guifrm:'.':guictl:'.':guievent
  END ELSE
    subid = guiapp:'.':guifrm:'.':guievent
  END
  BEGIN CASE
    CASE subid EQ 'ABC.FRMMAIN.CLOSE'
      GOSUB abc.frmmain.close  ;  guievt = 0
    CASE subid EQ 'ABC.FRMMAIN.BTNEXIT.CLICK'
      GOSUB abc.frmmain.btnexit.click  ;  guievt = 0
  END CASE
  IF guievt THEN
    * Unhandled event - may be dynamic
    GOSUB gui.dynamic.events
  END
END ELSE
  GOSUB gui.custom.events
END

RETURN
*
* ----------------------------------------------------------------- *
*
gui.dynamic.events:
*
* Add any dynamic event handling code here. The guievt, guiapp,
* guifrm, guictl and guiargs variables are valid and available
* for your use.
*
RETURN
*
* ----------------------------------------------------------------- *
*
gui.custom.events:
*
* Add any custom event handling code here. The guievt, guiapp, guifrm,
* guictl and guiargs variables are valid and available for your use.
*
RETURN
*
* ----------------------------------------------------------------- *
*
gui.error:
*
CALL ATGUISHUTDOWN
PRINT 'The following errors have been reported by the GUI system:'
numerrs = DCOUNT(guierrors, CHAR(254))
FOR eacherr = 2 TO numerrs
  PRINT guierrors<eacherr, 6>
NEXT eacherr
*
STOP
*
RETURN
*
* ----------------------------------------------------------------- *
*
abc.frmmain.btnexit.click:
*
GOSUB abc.frmmain.close
*
RETURN
*
```

```
* ----------------------------------------------------------------- *
*
abc.frmmain.close:
*
* Default form close event handler
CALL ATGUIHIDE(guiapp, guifrm, '', '', guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

CALL ATGUIGETPROP(guiapp, '', '', GPSTATUS, 0, 0, NUM.FORMS, guierrors,
guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

IF NUM.FORMS EQ 0 THEN
  CALL ATGUIDELETE(guiapp, '', '', guierrors, guistate)
  IF guierrors<1> GE 3 THEN GOTO gui.error
END

RETURN
*
* ----------------------------------------------------------------- *
*
buildform:

CALL ATGUICREATEAPP(guiapp, GEQUIT, 0, '', 0, 0, 0, 0, 0, 0, 0,
guierrors, guistate)
IF guierrors<1> GE 3 THEN RETURN FROM PROGRAM

eventmask = GECLOSE + GERESIZE
CALL ATGUICREATEFORM(guiapp, guifrm, eventmask, 'Example form', 1, 5, 5,
60, 25, guierrors, guistate)
IF guierrors<1> GE 3 THEN RETURN FROM PROGRAM

CALL ATGUICREATEFRAME(guiapp, guifrm, 'FRATOP', '', 0, '', 0, 0, 60, 3,
guierrors, guistate)
IF guierrors<1> GE 3 THEN RETURN FROM PROGRAM

CALL ATGUICREATELABEL(guiapp, guifrm, 'LBLFILE', 'FRATOP', 0, 'File:',
1, 0.75, 4, 1.5, guierrors, guistate)
IF guierrors<1> GE 3 THEN RETURN FROM PROGRAM

CALL ATGUICREATECOMBO(guiapp, guifrm, 'CMBFILE', 'FRATOP', GECHANGE, 5,
0.75, 15, 1.5, guierrors, guistate)
IF guierrors<1> GE 3 THEN RETURN FROM PROGRAM

CALL ATGUICREATEFRAME(guiapp, guifrm, 'FRABOT', '', 0, '', 0, 22, 60, 3,
guierrors, guistate)
IF guierrors<1> GE 3 THEN RETURN FROM PROGRAM

ctrlids = 'FRATOP':@AM:'FRABOT'
props = GPBORDER:@AM:GPBORDER
propvals = 0:@AM:0
GOSUB setprops

CALL ATGUICREATEBUTTON(guiapp, guifrm, 'BTNTEST', 'FRABOT', GECLICK,
'&Test', 5, 1, 6, 1.5, guierrors, guistate)
IF guierrors<1> GE 3 THEN RETURN FROM PROGRAM

CALL ATGUICREATEBUTTON(guiapp, guifrm, 'BTNEXIT', 'FRABOT', GECLICK,
'E&xit', 50, 1, 6, 1.5, guierrors, guistate)
IF guierrors<1> GE 3 THEN RETURN FROM PROGRAM

eventmask = GECHANGE + GEVALIDATECELL + GEVALIDATEROW
colhdgs = 'Col 1':@VM:'Col 2':@VM:'Col 3':@VM:'Col 4'
CALL ATGUICREATEGRID(guiapp, guifrm, 'GRDMAIN', '', eventmask, 1, 4,
colhdgs, 0, 3, 60, 19, guierrors, guistate)
IF guierrors<1> GE 3 THEN RETURN FROM PROGRAM

ctrlids = 'GRDMAIN':@AM:'GRDMAIN':@AM:'GRDMAIN'
props = GPSTYLE:@AM:GPFIXEDCOLS:@AM:GPCOLWIDTH
colwidths = 25:@VM:15:@VM:10:@VM:10
propvals = 1:@AM:1:@AM:colwidths
GOSUB setprops
RETURN
```

```
*
* ------------------------------------------------------------------ *
*
convertevent:
*
BEGIN CASE
  CASE guievt EQ GECLOSE  ;  guievent = 'CLOSE'
  CASE guievt EQ GECLICK  ;  guievent = 'CLICK'
  CASE 1                  ;  guievent = 'NOTHING'
END CASE
*
RETURN
*
* ----------------------------------------------------------------- *
*
setprops:
*
CALL ATGUISETPROPS(guiapp, guifrm, ctrlids, props, propvals, guierrors,
guistate)
IF guierrors<1> GE 3 THEN GOTO gui.error
*
RETURN
*
* ----------------------------------------------------------------- *
*
END
```

The main program takes up the first 30 lines. This is essentially the program structure defined for *AccuTerm* GUI applications in the ‘GUI Development Environment’ of the *AccuTerm* Programmers Guide.



The key difference between this program and the standard *AccuTerm* GUI skeleton program is that the GUI interface is entirely built within the `buildform` subroutine. This subroutine first defines an application, then a form to go in the application, and then places controls on the form. Like the *XED* application, this form has top and bottom frames, plus a grid taking up the remainder of the form area. The top frame contains a combo box, while the bottom frame contains two buttons. Finally, some control properties are set.

Clearly, this is not a complete application. It is only intended to demonstrate that a GUI can be generated entirely from within the program.

The GUI interface created by this subroutine is shown on the previous page:

The rest of the program responds to the events coming from the form. The `gui.decode.event` subroutine is a variant of that generated by the *AccuTerm* GUI designer, while the `gui.dynamic.events`, `gui.custom.events`, `gui.error`, and form close subroutines are taken directly from the standard *AccuTerm* skeleton program.

The `convertevent` subroutine converts the numeric event identifier (e.g. GECLOSE) to a string describing the event. This string is then used to create the name of the subroutine used to handle the event.

9.2 Creating a Generalised GUI Program

9.2.1 What are the problems?

Note that we strike an issue here regarding how we pass control to the subroutines. The process used is to create a subroutine identifier, and then use a CASE statement to switch control to an internal subroutine. This switching process is somewhat tedious.

Ideally, we would like to something like:

```
subid = guiapp:'.':guifrm:'.':guictl:'.':guievent
GOSUB @subid
```

Unfortunately, this type of indirect call is not available for a local subroutine. We can use this type of indirect call for external subroutines, but that would mean that we would have many external subroutines for each application.

However, we could do something like:

```
subid = guiapp:'*':guifrm:'*':guictl:'*':guievent
evtsub = guiapp:'.EVENTS'
CALL @evtsub(subid, guievt, guiargs, guierrors, guistate)
```

Why might we do this? After all, we will still strike the same problem in the external subroutine where we have to pass control to internal subroutines based on value of the `subid` variable.

There are three reasons why we may want to adopt this type of structure:

- It separates the “business logic” of the program from the building of the user interface
- It means the main program comes down to a relatively standard piece of code
- Most importantly, it breaks the monolithic structure of the *XED* application.

Why is breaking the structure important?

The *XED* application is too large to maintain good control over the number of variables. Whenever we needed a new variable, we simply a created one, and in standard PICK programming (which we’ve used here), all these variables are global in scope. When we use a variable, we aren’t sure whether the value it contains comes from the actions currently under way, or have simply been retained from some previous action.

Forcing all the events to be handled in a separate subroutine provides one solution to this problem. It may not be elegant, but it works.

When we return from the subroutine after processing each event, all the variables in the subroutine are lost. This creates its own set of problems (which we’ll solve later in this section), but we are now certain that the value of any variable that we use while processing any given event is actually related to that event. And if the variable has not been properly defined for that event, then we will get an undefined variable error.

9.2.2 Creating a general calling program for GUI applications

If we strip the form building code and the event handling subroutines out of our previous program, and place those in external subroutines, we are left with a relatively short program that can be used to run any appropriately defined GUI application. This leaves us with a program like:

```
PROGRAM RUNGUI
*****************************************************************
* Bp.Gui RunGui - A generic program for AccuTerm GUI applications.
*
* BSS
* 05 Dec, 2009
*
* Usage: RUNGUI guiapp
*
*   where 'guiapp' must be defined in the 'GUI.VAR' file
*
* ------------------------------------------------------------- *
*
$CATALOGUE
$INCLUDE GUIBP ATGUIEQUATES
$MODE UV.LOCATE

PROMPT ''
OPEN 'GUI.VAR' TO gui.var ELSE STOP 201, 'Gui.Var'

ss = TRIM(@SENTENCE)
CONVERT ' ' TO @AM IN ss
guiprog = ss<2>
READ guidef FROM gui.var, guiprog ELSE
  CRT guiprog:' is not defined in file: GUI.VAR'
  STOP
END

guiapp = guidef<1>
guiapp.name = guidef<2>
guifrm = guidef<3>
guifrmsub = guiapp:'.FORM'
guievtsub = guiapp:'.EVENTS'
guievent = ''

IF NOT(guifrmsub) THEN
  CRT 'The form creation subroutine for application ':guiapp:' has not
been defined'
  STOP
END

IF NOT(CATALOGUED(guifrmsub)) THEN
  CRT 'The form creation subroutine for application ':guiapp:' has not
been catalogued'
  STOP
END

IF NOT(CATALOGUED(guievtsub)) THEN
  CRT 'The event handling subroutine for application ':guiapp:' has not
been catalogued'
  STOP
END

CALL ATGUIINIT2(1.3, '', guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

CALL @guifrmsub(guiapp, guifrm, guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

guifrm = guidef<3>
CALL ATGUISHOW(guiapp, guifrm, '', '', guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

LOOP
```

```
  CALL
ATGUIWAITEVENT(guiapp,guifrm,guictl,guievt,guiargs,guierrors,guistate)
  IF guierrors<1> GE 2 THEN GOTO gui.error
UNTIL guievt = GEQUIT DO
  guiapp=OCONV(guiapp, 'MCU')
  guifrm=OCONV(guifrm, 'MCU')
  guictl=OCONV(guictl, 'MCU')
  GOSUB gui.decode.event
REPEAT

CALL ATGUISHUTDOWN

STOP
*
* -------------------------------------------------------------- *
*
gui.decode.event:
*
CALL EVENTS.CONVERT('O', guievt, guievent)
IF guievent EQ '' THEN guievent = guievt

IF guictl THEN
  subid = guiapp:'*':guifrm:'*':guictl:'*':guievent
END ELSE
  subid = guiapp:'*':guifrm:'*':guievent
END

CALL @guievtsub(subid, guievt, guiargs, guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

IF guievt THEN
  emsg = 'An unhandled event has occurred: ':subid
  CALL ATGUIMSGBOX(emsg, 'Run GUI', MBIICON, MBOK, '', ans, guierrors,
guistate)
  IF guierrors<1> GE 2 THEN GOTO gui.error
END

RETURN
*
* -------------------------------------------------------------- *
*
gui.error:
*
CALL ATGUISHUTDOWN
PRINT 'The following errors have been reported by the GUI system:'
numerrs = DCOUNT(guierrors, CHAR(254))
FOR eacherr = 2 TO numerrs
  PRINT guierrors<eacherr, 6>
NEXT eacherr
*
STOP
*
RETURN
*
* -------------------------------------------------------------- *
*
END
```

and the control item in the `GUI.VAR` file looks like:

```
ABC
ABC Application
FRMMAIN
```

This calls an event conversion subroutine that looks like:

```
SUBROUTINE EVENTS.CONVERT(inout, event, eventvalue)
******************************************************************
* Bp Events.Convert - Convert an to an eventvalue and vice-versa
*
* BSS
* 10 Jan, 2010
*
```

```
* inout      - I/O  I: event      --> eventvalue
*                   O: eventvalue --> event
* event      - event to convert
* eventvalue - value of event
*
* -------------------------------------------------------------- *
*
$CATALOGUE
$MODE UV.LOCATE
$INCLUDE GUIBP ATGUIEQUATES

PROMPT ''

guievts = GECLOSE:' ':GEACTIVATE:' ':GEDEACTIVATE:' ':GECLICK:' '
guievts := GEDBLCLICK:' ':GECHANGE:' ':GEVALIDATE:' ':GELOADLIST:' '
guievts := GEVALIDATECELL:' ':GEVALIDATEROW:' ':GECONTEXT:' '
guievts := GEELLIPSIS:' ':GEACTIVATECELL:' ':GERESIZE:' '
guievts := GECOLCLICK:' ':GEDEACTIVATECELL:' ':GEACTIVATEROW:' '
guievts := GEDEACTIVATEROW:' ':GEHELP:' ':GESTATUS:' ':GEQUIT
CONVERT ' ' TO @VM IN guievts

guievents = 'CLOSE ACTIVATE DEACTIVATE CLICK DBLCLICK CHANGE '
guievents := 'VALIDATE LOADLIST VALIDATECELL VALIDATEROW CONTEXT '
guievents := 'ELLIPSIS ACTIVATECELL RESIZE COLCLICK DEACTIVATECELL '
guievents := 'ACTIVATEROW DEACTIVATEROW HELP STATUS QUIT'
CONVERT ' ' TO @VM IN guievents

inout = OCONV(inout, 'MCU')
BEGIN CASE
  CASE inout = 'I'
    GOSUB getvalue
  CASE inout = 'O'
    GOSUB getevent
  CASE 1
    event = ''
    eventvalue = ''
END CASE

RETURN

STOP
*
* -------------------------------------------------------------- *
*
getevent:
*
event = ''
LOCATE eventvalue IN guievts<1> BY 'AR' SETTING epos THEN
  event = guievents<1, epos>
END
*
RETURN
*
* -------------------------------------------------------------- *
*
getvalue:
*
eventvalue = 0
LOCATE event IN guievents<1> SETTING epos THEN
  eventvalue = guievts<1, epos>
END
*
RETURN
*
* -------------------------------------------------------------- *
*
END
```

Stepping through this program:

- The control file gets opened. Create this as a directory file so that you can use an ordinary text editor to edit the items

- The sentence used to run the program is parsed. The GUI application name is extracted from this sentence
- The control item is read from the control file, and the names of the application, main form, and form building subroutine are read from the item
- Checks are made to ensure the form building subroutine and event handling subroutine are both catalogued
- The form building subroutine is called, and the main form displayed. Note that the *guifrm* variable is redefined here – in case its definition has been changed in the form building subroutine
- The application goes into its main loop
- When an event occurs, we call an external subroutine to convert the event to an event name. We use the name to create an event identifier, and the events subroutine called to handle the event. The event decoder uses the *guievt* variable to determine whether the event has been handled in the external subroutine. This means that the external subroutine should set this variable to zero when an event is handled.
- If the *guievt* variable is still set when control returns to the program, then a message box is displayed indicating that an event has not been handled.

The *ABC.FORM* subroutine consists of the *buildform* and *setproperties* subroutines from the earlier program and is not duplicated here. However, the *ABC.EVENTS* subroutine is shown below:

```
SUBROUTINE ABC.EVENTS(subname, guievt, guiargs, guierrors, guistate)
****************************************************************
* Subroutine to handle events for application: ABC
*
* BSS
* 05 Dec, 2009
*
* ------------------------------------------------------------- *
*
$CATALOGUE
$INCLUDE GUIBP ATGUIEQUATES

guiapp.name = 'ABC Application'
guiapp = FIELD(subname, '*', 1)
guifrm = FIELD(subname, '*', 2)
guictl = FIELD(subname, '*', 3)
guievent = FIELD(subname, '*', 4)
IF NOT(guievent) THEN
  guievent = guictl
  guictl = ''
END

eventhandled = @TRUE
BEGIN CASE
  CASE subname EQ 'ABC*FRMMAIN*CLOSE'
    GOSUB abc.frmmain.close
  CASE subname EQ 'ABC*FRMMAIN*BTNEXIT*CLICK'
    GOSUB abc.frmmain.btnexit.click
  CASE subname EQ 'ABC*FRMMAIN*BTNTEST*CLICK'
    GOSUB abc.frmmain.btntest.click
  CASE 1
    eventhandled = @FALSE
END CASE
IF eventhandled THEN guievt = 0

RETURN
*
* ------------------------------------------------------------ *
*
abc.frmmain.btnexit.click:
```

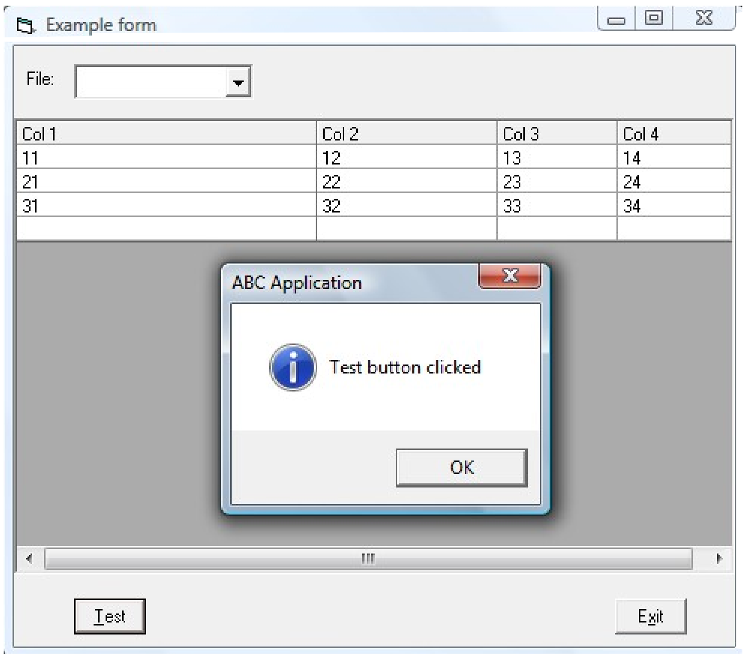
```
*
GOSUB abc.frmmain.close
*
RETURN
*
* ------------------------------------------------------------ *
*
abc.frmmain.btntest.click:
*
grid = ''
grid<1, -1> = 11:@SM:12:@SM:13:@SM:14
grid<1, -1> = 21:@SM:22:@SM:23:@SM:24
grid<1, -1> = 31:@SM:32:@SM:33:@SM:34

CALL ATGUILOADVALUES(guiapp, guifrm, 'GRDMAIN', grid, guierrors,
guistate)
IF guierrors<1> GE 2 THEN RETURN FROM PROGRAM

CALL ATGUIMSGBOX('Test button clicked', guiapp.name, MBIICON, MBOK, '',
ans, guierrors, guistate)
IF guierrors<1> GE 2 THEN RETURN FROM PROGRAM
*
RETURN
*
* ------------------------------------------------------------ *
*
abc.frmmain.close:
*
* Default form close event handler
CALL ATGUIHIDE(guiapp, guifrm, '', '', guierrors, guistate)
IF guierrors<1> GE 2 THEN RETURN FROM PROGRAM

CALL ATGUIGETPROP(guiapp, '', '', GPSTATUS, 0, 0, NUM.FORMS, guierrors,
guistate)
IF guierrors<1> GE 2 THEN RETURN FROM PROGRAM

IF NUM.FORMS EQ 0 THEN
  CALL ATGUIDELETE(guiapp, '', '', guierrors, guistate)
  IF guierrors<1> GE 2 THEN RETURN FROM PROGRAM
END

RETURN
*
* ------------------------------------------------------------ *
*
END
```

This subroutine only handles a limited number of events – two events to close the form, and one to enter some data in the grid and display a message box. Details from this program include:

- The *`guiapp`*, *`guifrm`*, *`guictl`* and *`guievent`* (note this is the string description of the event) are extracted from the passed event identifier for use elsewhere in the subroutine. Note that the *`guictl`* variable may be null, and the code needs to allow for this
- An *`eventhandled`* variable is set up to determine if the event is handled within the subroutine. This variable is initially set to `@TRUE`. However, if control goes through the `CASE 1` clause of the `CASE` statement, then this will be reset to `@FALSE`
- Every handled event must be defined by this `CASE` statement

- If GUI errors are encountered, then the subroutine uses the `RETURN FROM PROGRAM` statement[21] to pass control back to the ***`RUNGUI`*** program where the error is handled.

Does it work? The following screenshot is taken after the 'Test' button has been pressed:



Col 1	Col 2	Col 3	Col 4
11	12	13	14
21	22	23	24
31	32	33	34

Test button clicked

Likewise, the unhandled event trap can be tested by entering some data into the combo box. As no event handler has been defined for this event, the warning message box is displayed.

9.3 Using the GUI Designer Interface

All of the above may be interesting, but most people don't want to design their GUI interfaces in this manner. So, why don't we just use the *AccuTerm* GUI designer to create the GUI interface?

Create a similar interface in the GUI Designer to that of the ***`ABC`*** application. We'll call this ***`DEF`***.

Now we need a ***`DEF.FORM`*** subroutine, and a ***`DEF.EVENTS`*** subroutine. The ***`DEF.FORM`*** subroutine will look like:

```
SUBROUTINE DEF.FORM(guiapp, guifrm, guierrors, guistate)
*****************************************************************
* Bp.Gui DEF.Form - Subroutine to create the GUI for application DEF
*
* BSS
* 06 Dec, 2009
*
* -------------------------------------------------------------- *
*
$CATALOGUE
$INCLUDE GUIBP ATGUIEQUATES

PROMPT ''
OPEN 'APPS' TO apps ELSE STOP 201, 'Apps'
```

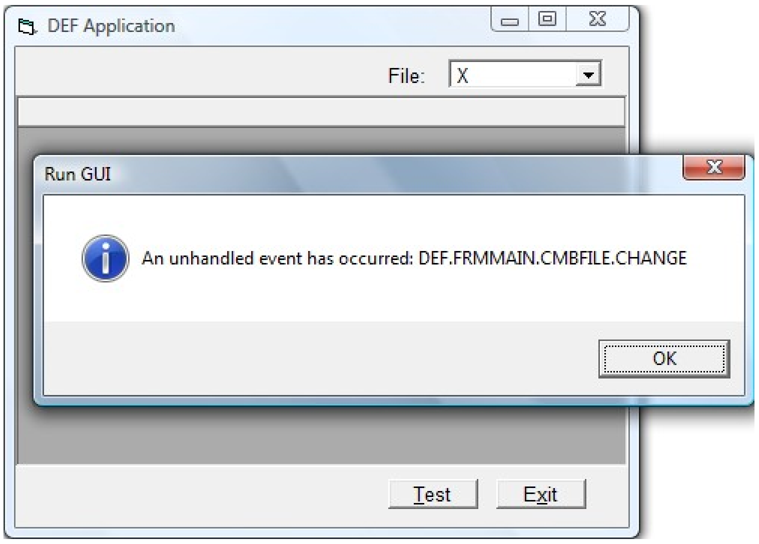
21 Note that this statement is not available in the GPL version of OpenQM. If you are using that version, you should set up a subroutine named `errorhandler:` which contains the statement: `RETURN TO errorhandler`. The `RETURN FROM PROGRAM` statements should the be replaced by "GOTO errorhandler" statements.

```
READ appdef FROM apps, guiapp ELSE appdef = ''

CALL ATGUIRUNMACRO(appdef, '', guierrors, guistate)
IF guierrors<1> GE 2 THEN RETURN FROM PROGRAM

RETURN
*
* -------------------------------------------------------------- *
*
END
```

The *DEF.EVENTS* subroutine will be a copy of the *ABC.EVENTS* subroutine, with all references to 'ABC' replaced with 'DEF'.

Now, we can run the application with:

RUNGUI DEF



An unhandled event has occurred: DEF.FRMMAIN.CMBFILE.CHANGE

The above screenshot shows the application with the unhandled event error message appearing in response to entering an 'X' in the combo box (assuming you have enabled the 'Change' event for the combo box).

9.4 COMMON Variables

9.4.1 Sharing variables

So far, we've shown that we can replace the standard *AccuTerm* skeleton program structure with a standard *RUNGUI* program coupled with standalone subroutines to create the form (which can still use the GUI Designer to create the interface) and handle the events. The key saving in this is that the event decoder does not need to be included in every new application – we can just use the one *RUNGUI* program to run any GUI application.

However, if we try to bring an application like *XED* into this framework, we soon find that we have a problem:

Almost every application depends on some variables being available on a global basis, while others are at least shared between different subroutines. By separating the event handling subroutines from the event decoder, we have broken the ability of the event handling subroutines to share their variables.

This is because the variables in the event handling subroutine will be reinitialised every time the subroutine is called from *RUNGUI*. Therefore, variables such as `xfiledef` which contains the control information about the file being displayed/edited, or *xed.var* which is the file pointer to the XED.VAR file are not maintained between calls.

Clearly, this is inconvenient. We don't want to re-open the *XED.VAR* file and re-read the control information every time that we want to check the control information. So, we need some way to maintain the values of selected variables between calls to the *EVENTS* subroutine.

So, what means have we got of maintaining variable state between calls?

We could call the event handling subroutine with some extra parameters. This could be something like:

```
CALL @guievtsub(subid, guievt, guiargs, files, vars, guierrors,
guistate)
```

These extra variables have no meaning in *RUNGUI*, but they need to be defined in the CALL so that they can be passed back and forth from *RUNGUI* to the event handling subroutine.

File pointers could be stored in the *files* variable, and other variables could be stored in the *vars* variable. However, this would mean stacking the data in specific places within the variables, or we would have to set up identifiers so that we can LOCATE the data we want. Overall, this is awkward.

Alternatively, we could set up the event handling subroutine to use a lot more arguments. This would be something like:

```
SUBROUTINE XYZ.EVENTS(subid, guievt, guiargs, files, vars,
guierrors, guistate, fp1, fp2, var1, var2, var3, var4, var5)
```

and, in the *RUNGUI* program:

```
CALL @guievtsub(subid, guievt, guiapp.name, files, vars, guierrors,
guistate, x1, x2, x3, x4, x5, x6, x7)
```

This means we have more scope to pass data without stacking – but it still isn't a good solution. Firstly, once we have used all the arguments in the call, we will have to implement a stacking system. Secondly, every subroutine we call from the *RUNGUI* program to handle events will need to have this number of arguments – whether we use them or not.

9.4.2 Introducing COMMON

A better solution is the use of COMMON variables. As the name implies, a COMMON variable is one that is common (or shared) between two or more programs. Common variables are defined via a COMMON statement:

```
COMMON {/name/} var1 {,var2 ...}
```

Such COMMON statements must be set up in each program that wishes to share the variable(s) defined by the COMMON.

There are two types of COMMON – named common, and unnamed common. A named common is given a name:

```
COMMON /XED/ xed.var, xed.files, xfiledef
COMMON /XED/ selfile, selyear, selmonth
COMMON /XED/ grid, ogrid
```

while an unnamed common does not have a name:

```
COMMON xed.var, xed.files, xfiledef
```

The "name" adds two characteristics to the use of COMMON variables:

- The variables persist for the duration of your *OpenQM* session rather than simply for the duration of the process that created the variables
- Where you have multiple `COMMON` blocks, it helps prevent confusion over which common variable contains which data item.

The first point is obvious enough. If the `COMMON` statement does not have a name, then once the program that created the variables terminates, the `COMMON` variables will disappear along with all the other variables used by the program.

Note that this does NOT mean that every program or subroutine needs to include the `COMMON` variables. For example, we may have some `COMMON` variables in our called subroutine but not in the calling program. The values of our `COMMON` variables persist between subroutine calls even though the calling program does not share those variables, and even though the subroutine may not currently be in memory. However, once the calling program terminates, these common variables will disappear.

The second point is a little less obvious. Even though `COMMON` variables have names just like other variables, the internal passing process only concerns itself with the position of the data in the passed variables. Therefore, the calling program has a `COMMON` statement of:

```
COMMON alpha, bravo, charlie, delta
```

and the subroutine has a `COMMON` statement of:

```
COMMON echo, foxtrot, golf
```

then value of variable *`alpha`* in the calling program will be passed to variable *`echo`* in the subroutine, *`bravo`* will go to *`foxtrot`*, while *`charlie`* and *`delta`* will BOTH go to *`golf`*.

The above situation is not recommended, but it it possible. Now, we'll confuse things:

Let's say the calling program now has:

```
COMMON alpha, bravo, charlie
COMMON xray, yankee, zulu
```

while the subroutine has:

```
COMMON xray, yankee, zulu
COMMON alpha, bravo, charlie
```

There has now been a total swap of variable names, and you won't have a clue what is going on in the program or subroutine. However, if you used named common statements, then this confusion wouldn't occur:

In this case, the program would be:

```
COMMON /ABC/ alpha, bravo, charlie
COMMON /XYZ/ xray, yankee, zulu
```

and the subroutine would be:

```
COMMON /XYZ/ xray, yankee, zulu
COMMON /ABC/ alpha, bravo, charlie
```

Now, there will be no swapping of variables, and you will be able to trace the program logic between the program and subroutine.

Consider also the situation where the main program has two external subroutines. The first external subroutine has a `COMMON` that declares variables *`alpha`*, *`bravo`*, and *`charlie`*, while the second declares `COMMON` variables *`delta`*, *`echo`*, and *`foxtrot`*. If these variables are declared in an unnamed `COMMON`, then as far as *OpenQM* is concerned, the two sets of variables are the same – i.e. *`alpha`* is the same variable as *`delta`* etc. Once again, using a named `COMMON` ensures that *OpenQM* treats these variables as being different.

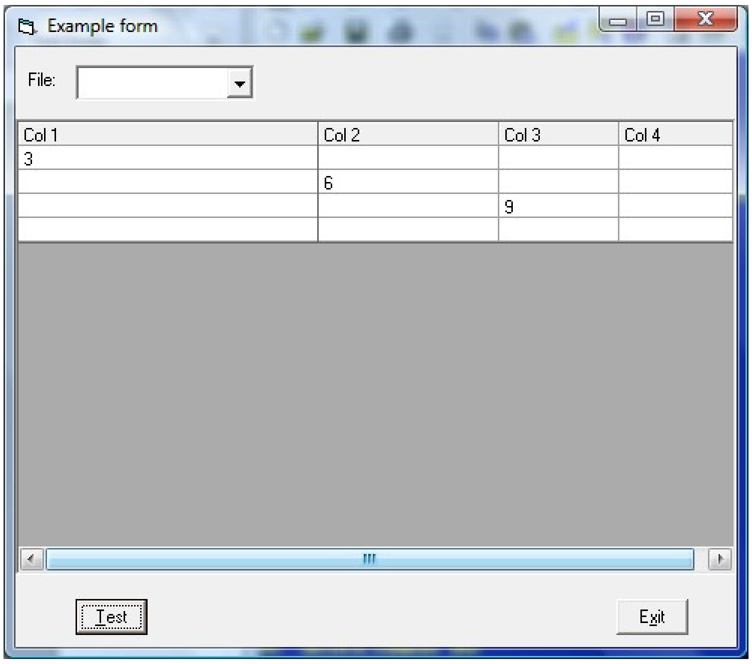
9.4.3 Example usage

Let's check that `COMMON` variables work in our *`ABC`* application. Add the following lines to the top of the *`ABC.EVENTS`* subroutine immediately after the existing compiler directives:

```
$MODE UNASSIGNED.COMMON

COMMON cntr1, cntr2, cntr3
IF UNASSIGNED(cntr1) THEN cntr1 = 0
IF UNASSIGNED(cntr2) THEN cntr2 = 0
IF UNASSIGNED(cntr3) THEN cntr3 = 0
```

and then change the *`abc.frmmain.btntest.click`* subroutine to read:

```
cntr1 += 1
cntr2 += 2
cntr3 += 3
grid = ''
grid<1, 1, 1> = cntr1
grid<1, 2, 2> = cntr2
grid<1, 3, 3> = cntr3

CALL ATGUILOADVALUES(guiapp, guifrm, 'GRDMAIN', grid, guierrors,
guistate)
IF guierrors<1> GE 3 THEN RETURN FROM PROGRAM
```



Col 1	Col 2	Col 3	Col 4
3			
	6		
		9	

Now, compile the subroutine and run the application. Each time you press the 'Test' button, the values in the grid will be incremented. The screenshot above shows their values after three presses:

The fact that these values have successfully been incremented shows that their values have been preserved between the calls to the subroutine.

Now, exit the application and restart it. The values start incrementing from zero again. This means that once the application terminated, the `COMMON` values were lost.

If we now change the `COMMON` statement to read:

```
COMMON /ABC/ cntr1, cntr2, cntr3
```

and then recompile the subroutine, we see no difference on the first time that we run the application. However, if we run the application again, the values start incrementing from their ending values of the previous run. Clearly, the `COMMON` values have persisted beyond the life of the application.

Let’s go back and look at the code a little bit more:

The initial bit of code told the compiler to create any common variables in an unassigned state. If we didn’t include this, common variables would be initialised with a value of zero. So, why didn’t we just leave things at their default setting – particularly when the first thing we did was to initialise the values to zero? The answer is partly to illustrate the use of this mode and the `UNASSIGNED` function, and partly because it is good practice to do things this way.

In your programming so far, you have probably encountered a situation where your program has crashed with an error message like:

```
000000B1: Unassigned variable JJ at line 3 of
D:\QM\QMINTRO\BP.OUT\TEST
```

While this may be inconvenient, it indicates that you have tried to use a variable without assigning it a value. This is a serious problem which you need to address. Otherwise, your code will not work the way you intend it to.

Consider what happens if we automatically assign a value to a variable. Errors such as the one above will no longer occur, which means that we have no indication that we are trying to use a variable that we have not assigned a value to. This is why you should always use the `UNASSIGNED.COMMON` compiler directive – so you don’t hide programming errors.

Having forced our `COMMON` variables to be defined without a value, we now need to ensure that we DO assign those variables a value. This is where we can strike a problem deciding WHEN to assign them a default value.

If the `COMMON` variables are only defined in an external subroutine, we can’t initialise them to a default value when we first enter the subroutine because we don’t know if this is the first time the subroutine has been called. If the subroutine has already been called, the variables should already have a value, and assigning a default value would no longer be appropriate.

So, we need to assign a default value only if the variable has not been assigned. Which brings us to the `UNASSIGNED` (or alternatively the `ASSIGNED)` function. Using these functions, we can decide when to initialise our common variables.

The code in the program tested each variable individually, but we could have used:

```
IF UNASSIGNED(cntr1) THEN
  cntr1 = 0
  cntr2 = 0
  cntr3 = 0
END
```

or:

```
IF NOT(ASSIGNED(cntr1)) THEN
  cntr1 = 0
  cntr2 = 0
  cntr3 = 0
END
```

If you are writing code that is to run on multiple multi-value databases, then check which of these functions is defined on the other databases you are writing for. For example, both *mvBASE* and *UniVerse* support both ASSIGNED and UNASSIGNED, but *UniData* supports only UNASSIGNED.

Use of testing such as this ensures that the variables are only defined once, when you first run the application. Thereafter, they will retain the values from the last time that control passed to the subroutine. If you are dealing with a named common, then you may need to give additional consideration to when you should re-initialise any COMMON variables.

9.4.4 Implementing COMMON variables

There is one final point to note on incorporating COMMON statements into programs.

Typically, COMMON statements are shared between a number of programs and subroutines, or between one main program and multiple subroutines. It is important to make sure that the COMMON statements (or COMMON block) is the same[22] for each of these instances.

This means that any change to the COMMON block in one of these instances should be accompanied by a matching change in other instances of the same COMMON block. Clearly, this could become a little tedious, and there is a risk that you will miss one of the instances.

The usual solution to this problem is to place the COMMON statements into a separate item, and then use the $INCLUDE compiler directive to include the item into the program. Using the common block for the *ABC* application as an example, we could store this in an item named *ABC.COMMON.H*:

```
* Common block for ABC application.
COMMON /ABC/ cntr1, cntr2, cntr3
```

and then in the *ABC.EVENTS* subroutine, we would include this line in place of the COMMON declaration:

```
$INCLUDE ABC.COMMON.H
```

If we make a change to the COMMON block, then we need to recompile all programs that use the common block. The simple way to do this is to use the SEARCH verb to return a list of programs using the COMMON block, and compile the list:

```
SEARCH BP.GUI
String: $INCLUDE ABC.COMMON.H
String:
1 record(s) selected to list 0
::BASIC BP.GUI
Compiling BP.GUI ABC.EVENTS
***********************************************************
0 error(s)
ABC.EVENTS added to private catalogue
Compiled 1 program(s) with no errors
```

The above example only selects one program, but demonstrates the principle.

Note the use of a *.H* suffix on the name of the include file. By default, the QMBasic compiler ignores records with a *.H* suffix when compiling programs. Therefore, if you compile all files in the folder:

```
BASIC BP.GUI *
```

any $INCLUDE records with a *.H* suffix are skipped. This is sensible, as these records are typically only code fragments which are meaningless on their own, and we don't want these fragments to produce object code.

22 Technically, this isn't quite correct. The initial COMMON declaration may contain MORE variables than subsequent declarations. However, in practical terms, the number of variables in each declaration should be the same. See the documentation for more information.

The compiler also automatically skips records with a *.SCR* suffix. You can modify this behaviour by creating a `$BASIC.IGNORE` record in the `VOC` or program file. See the documentation for more information on this feature.

9.5 Creating Skeleton Programs

We now know how to use a generalised calling program for our GUI applications. However, one of the issues we have this approach is that we need to create our event handling program from scratch. If we were using the *AccuTerm* GUI as intended, then `GED` would create the event handling subroutines for us.

So, how do we go about making sure that we have an event handler for every event that we have enabled. Well, we write a program that reads the template file created by `GED`, and creates or updates the *EVENTS* subroutine for us.

But before we can do that, we need to know more about the template created by `GED` that describes the application:

9.5.1 The *AccuTerm* GUI template item

Let’s have a look a the template item that `GED` has created for the XED application. The first few lines are shown below:

```
CT APPS XED
APPS XED
01: TEMPLATE]40140.88323]XED]BP.GUI^XED^51^1
02: 0]1.3]4
03: 2]XED]]1]auto]auto]60]20]1073741825
04: 4]]7]]]4
05: 4]]22]]]File editor
06: 4]]59]]]0
07: 4]]41]]]Rush Flat Software
08: 4]]43]]]An editor for interest and exchange rate files.
09: 4]]42]]]BSS
10: 4]]44]]]1.0
11: 4]]24]]]0
12: 4]]61]]]Processing... please wait
13: 4]]60]]]2
14: 4]]20]]]D:\QM\Resources\Help\XED.chm
15: 2]XED*frmmain]]5]auto]auto]67]20.5]32769
16: 4]]22]]]File editor
17: 4]]48]]]0
18: 4]]9]]]0
19: 4]]16]]]Arial
20: 4]]17]]]9.75
21: 4]]18]]]0
22: 4]]19]]]0
23: 2]XED*frmmain*fracontrols]]26]0]0]67]2.25]0
24: 4]]11]]]0
25: 2]XED*frmmain*lblfile]fracontrols]15]1].5]4]1.25]0
26: 4]]0]]]File:
```

In this listing, any value marks have been replaced by ‘]’ characters, while any sub-value marks have been replaced by ‘^’ characters[23].

By comparing this listing with the information contained in the property settings of the application, we can deduce most of the structure.

- ➢ The first couple of lines relate to general information about the application
  - ○ The application name is *XED*

23 By convention, these are normally replaced by backslash characters, but as this listing contains normal backslashes in the path name to the help file, it was necessary to use another character.

  - The program is located in file BP.GUI in item `XED` and uses custom template 51
  - The *AccuTerm* GUI version is 1.3
- The rest of the lines are structured as a control header line (line starts with 2) followed by a series of properties for that control (lines start with 4)
- The control header lines have the following value-mark delimited structure:
  - 2
  - Control name (e.g. `XED*frmmain*lblfile`)
  - Parent control (e.g. `fracontrols`)
  - Control type identifier (see `ATGUIEQUATES`)
  - Left position
  - Top position
  - Width
  - Height
  - Event mask (a number)
- The property lines have the following value-mark delimited structure:
  - 4
  - empty
  - Property identifier (see `ATGUIEQUATES`)
  - empty
  - empty
  - Property value(s)

The event mask needs a bit of explanation. *AccuTerm* has a value for each event that is enabled on the control. The event mask is the sum of those values. For example, the grid in the `XED` application has the `CHANGE`, `VALIDATECELL`, and `VALIDATEROW` events enabled. Their values (see `ATGUIEQUATES`) are:

Change	128
Validate cell	1024
Validate row	2048
Event mask	**3200**

If you look back in Section 9.1at the code we used to create a grid control from within the program, you will see we went through exactly this process. The relevant line is shown below:

```
eventmask = GECHANGE + GEVALIDATECELL + GEVALIDATEROW
```

This event mask is then used as part of the call to `ATGUICREATEGRID`.

In the template fragment shown above, it can be seen that control `XED*frmmain` has an event mask of *32769*. By referring to the values in `ATGUIEQUATES`, we can find that this corresponds to the `RESIZE` and `CLOSE` events.

9.5.2 From template to program

Clearly, if we are going to use the control item created by `GED` to build a skeleton program, we need some way to decode each event mask into its constituent events. The

EVENTS.CONVERT subroutine we used in Section 9.2.2 does part of this job – but it only converts a single event at a time. However, it isn't too difficult to update this subroutine to generate a list of events from an event mask (or vice-versa):

```
SUBROUTINE EVENTS.CONVERT(inout, events, eventmask)
*****************************************************************
* Bp.Gui Events.Convert - Convert a list of events to an eventmask
*                          and vice-versa*
* BSS
* 10 Jan, 2010
*
* inout     - I/O  I: events    --> eventmask
*                  O: eventmask --> events
* events    - vm delimited list of events
* eventmask - cumulative value of events
*
* -------------------------------------------------------------- *
*
$CATALOGUE
$MODE UV.LOCATE
$MODE UNASSIGNED.COMMON
$INCLUDE GUIBP ATGUIEQUATES

PROMPT ''
COMMON /EVENTS.CONVERT/ GUIEVTS, GUIEVENTS
IF UNASSIGNED(GUIEVTS) THEN
  GOSUB geteventdefs
END

inout = OCONV(inout, 'MCU')
BEGIN CASE
  CASE inout = 'I'
    GOSUB getmask
  CASE inout = 'O'
    GOSUB getevents
  CASE 1
    events = ''
    eventmask = ''
END CASE

RETURN

STOP
*
* -------------------------------------------------------------- *
*
getevents:
*
events = ''
IF eventmask THEN
  LOOP
    LOCATE eventmask IN GUIEVTS<1> BY 'DR' SETTING epos THEN
      thisevent = GUIEVENTS<1, epos>
      LOCATE thisevent IN events<1> SETTING tpos ELSE
        events<1, -1> = thisevent
      END
      EXIT
    END ELSE
      IF eventmask GT 0 THEN
        thisevent = GUIEVENTS<1, epos>
        LOCATE thisevent IN events<1> SETTING tpos ELSE
          events<1, -1> = thisevent
        END
        eventmask -= GUIEVTS<1, epos>
      END ELSE
        EXIT
      END
    END
  REPEAT
END
*
RETURN
```

```
*
* -------------------------------------------------------------- *
*
getmask:
*
eventmask = 0
dc = DCOUNT(events<1>, @VM)
FOR ii = 1 TO dc
  thisevent = OCONV(events<1, ii>, 'MCU')
  LOCATE thisevent IN GUIEVENTS<1> SETTING epos THEN
    eventmask += GUIEVTS<1, epos>
  END
NEXT ii
*
RETURN
*
* -------------------------------------------------------------- *
*
geteventdefs:
*
GUIEVTS = ''
GUIEVENTS = ''
OPEN 'GUIBP' TO guibp ELSE STOP 201, 'Guibp'
READ guiequates FROM guibp, 'ATGUIEQUATES' ELSE RETURN

LOOP
  REMOVE eqln FROM guiequates SETTING delim
  eqln = TRIM(eqln)
  IF (eqln[1, 6] EQ 'EQU GE') THEN
    CONVERT ' ' TO @AM IN eqln
    event = eqln<2>
    event = event[3, LEN(event)]
    evt = eqln<4>
    LOCATE evt IN GUIEVTS<1> BY 'DR' SETTING epos ELSE
      INS evt BEFORE GUIEVTS<1, epos>
      INS event BEFORE GUIEVENTS<1, epos>
    END
  END
WHILE delim DO REPEAT
*
RETURN
*
* -------------------------------------------------------------- *
*
END
```

There are a couple of differences between this and the earlier version of the program. Firstly, the program now gets its list of events and event values direct from the `ATGUIEQUATES` item. Secondly, the conversion subroutines now incorporate loops to assemble or disassemble the event mask. Note also that the lists of events and event values have been incorporated into a named `COMMON` so that the assignment of those values is only done once.

Now we can write a program that will … write a program for us!

```
PROGRAM EVENTS.BUILD
*****************************************************************
* Bp.Gui Events.Build - Build the events subroutine from the Application
*                       Definition item.
*
* Author : BSS
* Created: 10 Jan, 2010
* Updated: 11 Jan, 2010
* Version: 1.0.0
*
* Usage: EVENTS.BUILD guifile guiapp bpfile
*
* -------------------------------------------------------------- *
*
$CATALOGUE
$MODE UV.LOCATE
```

```
$MODE UNASSIGNED.COMMON
$INCLUDE GUIBP ATGUIEQUATES

PROMPT ''
ss = OCONV(TRIM(@SENTENCE), 'MCU')
CONVERT ' ' TO @AM IN ss
thisprog = ss<1>
guifile = ss<2>
guiapp = ss<3>
bpfile = ss<4>

IF guifile = '' OR guiapp = '' OR bpfile = '' THEN
  CRT 'Syntax: ':thisprog:' guifile guiapp bpfile'
  STOP
END

progrec = ''
found = @TRUE

OPEN guifile TO guifile.ptr ELSE STOP 201, guifile
OPEN bpfile TO bpfile.ptr ELSE STOP 201, bpfile
READ guidef FROM guifile.ptr, guiapp ELSE found = @FALSE
IF NOT(found) THEN
  CRT guiapp:' is not defined in file: ':guifile
  STOP
END

bpitem = guiapp:'.EVENTS'
READU bprec FROM bpfile.ptr, bpitem ELSE bprec = ''

guiapp.name = guidef<5, 6>
IF bprec EQ '' THEN
  GOSUB createskeleton
END

switchlist = ''         ;* Event names used in CASE statement
progsublist = ''        ;* Sub names in program
eventsubs = ''          ;* Sub names from application template

GOSUB getsubs           ;* Get sub names from existing program
GOSUB geteventlist      ;* Read template for controls and events
GOSUB addevents         ;* Add missing event handlers
GOSUB removeevents      ;* Remove redundant event handlers

WRITE bprec ON bpfile.ptr, bpitem
EXECUTE \DELETE.COMMON EVENTS.CONVERT\
CRT bpitem:' written to file ':bpfile

STOP
*
* -------------------------------------------------------------- *
*
addevents:
*
FOR controlcnt = 1 TO numcontrols
  control = controls<1, controlcnt>
  eventmask = eventmasks<1, controlcnt>
  events = ''
  CALL EVENTS.CONVERT('O', events, eventmask) ;* Get list of events
  eventcnt = DCOUNT(events<1>, @VM)
  IF eventcnt THEN
    FOR eventno = 1 TO eventcnt
      event = events<1, eventno>
      switchsubname = control:'*':event
      LOCATE switchsubname IN switchlist<1> BY 'AL' SETTING switchsubpos
ELSE
        INS switchsubname BEFORE switchlist<1, switchsubpos>
      END
      subname = switchsubname
      CONVERT '*' TO '.' IN subname
      subname = OCONV(subname, 'MCL')
      LOCATE subname IN progsublist<1> BY 'AL' SETTING subpos ELSE
        GOSUB addsub
        GOSUB addtocase
        INS subname BEFORE progsublist<1, subpos>
      END
      LOCATE subname IN eventsubs<1> BY 'AL' SETTING subpos ELSE
        INS subname BEFORE eventsubs<1, subpos>
      END
    NEXT eventno
  END
NEXT controlcnt
*
RETURN
*
* -------------------------------------------------------------- *
*
addsub:
*
nextsub = progsublist<1, subpos>
GOSUB buildsub

proglines = DCOUNT(bprec, @AM)
IF nextsub THEN
  GOSUB findinsertpoint
  IF NOT(found) THEN
    insertpoint = proglines
  END
END ELSE
  insertpoint = proglines  ;* Assumes last line is "END"
END

prog1 = FIELD(bprec, @AM, 1, insertpoint - 1)
prog2 = FIELD(bprec, @AM, insertpoint, proglines)
bprec = prog1:@AM:sublines:@AM:prog2
*
RETURN
*
* -------------------------------------------------------------- *
*
addtocase:
*
srchstring = ';* ==> Event handler start'
GOSUB findstring
startpoint = foundline
srchstring = ';* <== Event handler end'
GOSUB findstring
endpoint = foundline

IF startpoint AND endpoint THEN
  srchstring = 'GOSUB ':subname
  GOSUB findstring
  IF NOT(foundline) THEN      ;* Call to sub not in program
    srchstring = switchlist<1, switchsubpos + 1>
    IF srchstring THEN
      GOSUB findstring        ;* Look for call to next sub
    END ELSE
      foundline = endpoint
    END
    IF (foundline GT startpoint) AND (foundline LE endpoint) THEN
      sublines = ''
      sublines<-1> = '  CASE subname EQ ':SQUOTE(switchsubname)
      sublines<-1> = '    GOSUB ':subname

      prog1 = FIELD(bprec, @AM, 1, foundline - 1)
      prog2 = FIELD(bprec, @AM, foundline, proglines)
      bprec = prog1:@AM:sublines:@AM:prog2
    END
  END
END
*
RETURN
*
* -------------------------------------------------------------- *
*
buildsub:
*
sublines = ''
```

```
sublines<-1> = subname:':'
sublines<-1> = '*'
IF event = 'CLOSE' THEN
  sublines<-1> = '* Default form close event handler'
  sublines<-1> = "CALL ATGUIHIDE(guiapp, guifrm, '', '', guierrors,
guistate)"
  sublines<-1> = 'IF guierrors<1> GE 2 THEN RETURN FROM PROGRAM'
  sublines<-1> = ''
  sublines<-1> = "CALL ATGUIGETPROP(guiapp, '', '', GPSTATUS, 0, 0,
NUM.FORMS, guierrors, guistate)"
  sublines<-1> = 'IF guierrors<1> GE 2 THEN RETURN FROM PROGRAM'
  sublines<-1> = ' '
  sublines<-1> = 'IF NUM.FORMS EQ 0 THEN'
  sublines<-1> = "  CALL ATGUIDELETE(guiapp, '', '', guierrors, guistate)"
  sublines<-1> = '  IF guierrors<1> GE 3 THEN RETURN FROM PROGRAM'
  sublines<-1> = 'END'
  sublines<-1> = ''
  sublines<-1> = 'EXECUTE "DELETE.COMMON ':bpitem:'" CAPTURING junk'
END
sublines<-1> = '*'
sublines<-1> = 'RETURN'
sublines<-1> = '*'
sublines<-1> = '* ------------------------------------------------ *'
sublines<-1> = '*'
*
RETURN
*
* ------------------------------------------------------------- *
*
createskeleton:
*
bprec = ''
bprec<-1> = 'SUBROUTINE ':bpitem:'(subname, guievt, guiargs, guierrors,
guistate)'
bprec<-1> =
'*****************************************************************'
bprec<-1> = '* Subroutine to handle events for application: ':guiapp
bprec<-1> = '*'
bprec<-1> = '* Author : ':@LOGNAME
bprec<-1> = '* Created: ':OCONV(OCONV(DATE(), 'D'), 'MCT')
bprec<-1> = '* Updated: ':OCONV(OCONV(DATE(), 'D'), 'MCT')
bprec<-1> = '* Version: 0.0.1'
bprec<-1> = '*'
bprec<-1> = '*
------------------------------------------------------------- *'
bprec<-1> = '*'
bprec<-1> = '$CATALOGUE'
bprec<-1> = '$MODE UV.LOCATE'
bprec<-1> = '$MODE UNASSIGNED.COMMON'
bprec<-1> = '$INCLUDE GUIBP ATGUIEQUATES'
bprec<-1> = 'COMMON /':bpitem:'/ CONTROLVALS'
bprec<-1> = ''
bprec<-1> = 'guiapp.name = ':SQUOTE(guiapp.name)
bprec<-1> = "guiapp = FIELD(subname, '*', 1)"
bprec<-1> = "guifrm = FIELD(subname, '*', 2)"
bprec<-1> = "guictl = FIELD(subname, '*', 3)"
bprec<-1> = "guievent = FIELD(subname, '*', 4)"
bprec<-1> = 'IF NOT(guievent) THEN'
bprec<-1> = '  guievent = guictl'
bprec<-1> = "  guictl = ''"
bprec<-1> = 'END'
bprec<-1> = ''
bprec<-1> = 'eventhandled = @TRUE'
bprec<-1> = 'BEGIN CASE  ;* ==> Event handler start'
bprec<-1> = "  CASE subname EQ 'INITIALISE'"
bprec<-1> = '    GOSUB initialise'
bprec<-1> = '  CASE 1    ;* <== Event handler end'
bprec<-1> = '    eventhandled = @FALSE'
bprec<-1> = 'END CASE'
bprec<-1> = 'IF eventhandled THEN guievt = 0'
bprec<-1> = ''
bprec<-1> = 'RETURN'
bprec<-1> = '*'
bprec<-1> = '* ------------------------------------------------ *'
```

```
bprec<-1> = '*'
bprec<-1> = 'initialise:'
bprec<-1> = '*'
bprec<-1> = '*'
bprec<-1> = 'RETURN'
bprec<-1> = '*'
bprec<-1> = '* ------------------------------------------------ *'
bprec<-1> = '*'
bprec<-1> = 'END'
*
RETURN
*
* -------------------------------------------------------------- *
*
findinsertpoint:
*
found = @FALSE
insertpoint = 0
FOR ii = 1 TO proglines
  bpline = TRIM(bprec<ii>)
  colpos = INDEX(bpline, ':', 1)
  IF colpos THEN
    word = FIELD(bpline, ':', 1)
    IF word EQ nextsub THEN
      insertpoint = ii
      found = @TRUE
      EXIT
    END
  END
NEXT ii
*
RETURN
*
* -------------------------------------------------------------- *
*
findstring:
*
ii = 0
found = 0
foundline = 0
proglines = DCOUNT(bprec, @AM)
LOOP
  ii += 1
  thisline = bprec<ii>
  found = INDEX(thisline, srchstring, 1)
  IF found THEN
    foundline = ii
    EXIT
  END
  IF ii GE proglines THEN EXIT
REPEAT
*
RETURN
*
* -------------------------------------------------------------- *
*
geteventlist:
*
controls = ''
eventmasks = ''
controlcnt = 0
numlines = DCOUNT(guidef, @AM)
FOR ii = 1 TO numlines
  thisline = guidef<ii>
  typ = thisline<1, 1>
  IF typ EQ 2 THEN
    controlcnt += 1
    control = OCONV(thisline<1, 2>, 'MCU')
    controls<1, controlcnt> = control
    eventmasks<1, controlcnt> = thisline<1, 9>
  END
NEXT ii
numcontrols = controlcnt
*
```

```
RETURN
*
* ------------------------------------------------------------- *
*
getsubs:
*
goodchars =
'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789.%$_'
proglines = DCOUNT(bprec, @AM)
FOR ii = 1 TO proglines
  bpline = TRIM(bprec<ii>)
  colpos = INDEX(bpline, ':', 1)
  IF colpos THEN
    word = FIELD(bpline, ':', 1)
    testword = word
    CONVERT goodchars TO '' IN testword
    IF testword EQ '' THEN
      LOCATE word IN progsublist<1> BY 'AL' SETTING wpos ELSE
        INS word BEFORE progsublist<1, wpos>
      END
    END
  END
NEXT ii
*
RETURN
*
* ------------------------------------------------------------- *
*
removeevents:
*
subcnt = DCOUNT(progsublist<1>, @VM)
FOR subno = subcnt TO 1 STEP -1
  subname = progsublist<1, subno>
  LOCATE subname IN eventsubs<1> BY 'AL' SETTING subpos ELSE
    dc = DCOUNT(subname, '.')
    IF (dc GE 3) THEN
      event = FIELD(subname, '.', dc)
      CALL R.GUI.EVENTS.CONVERT('I', event, eventmask)
      IF eventmask THEN                 ;* Appears to be an event sub
        GOSUB buildsub                  ;* Build a dummy subroutine
        nextsub = subname
        GOSUB findinsertpoint           ;* Find location in program
        IF found THEN
          dc = DCOUNT(sublines, @AM)
          ok = @TRUE
          FOR ii = 1 TO dc              ;* Loop through the subroutine
            progline = bprec<insertpoint + ii - 1>
            subline = sublines<ii>
            IF progline NE subline THEN;* and test against dummy sub
              ok = @FALSE
              EXIT
            END
          NEXT ii
          IF ok THEN                    ;* All lines match, so delete
            FOR ii = 1 TO dc
              DEL bprec<insertpoint>
            NEXT ii
            GOSUB removefromcase
          END
        END
      END
    END
  END
NEXT subno
*
RETURN
*
* ------------------------------------------------------------- *
*
removefromcase:
*
srchstring = ';* ==> Event handler start'
GOSUB findstring
startpoint = foundline
```

```
srchstring = ';* <== Event handler end'
GOSUB findstring
endpoint = foundline

IF startpoint AND endpoint THEN
  srchstring = 'GOSUB ':subname
  GOSUB findstring
  IF foundline AND (foundline GT startpoint) AND (foundline LT endpoint)
THEN
    DEL bprec<foundline>
    DEL bprec<foundline - 1>
  END
END
*
RETURN
*
* ------------------------------------------------------------- *
*
END
```

That may look a lot, but it is fairly straightforward. The steps are:

- Get the user command, open the specified files, and read the application definition and the program item containing the event subroutines
- If the program does not exist, then create a skeleton program
- Read through the program item and build a list of the existing subroutines
- Read through the application definition item and build lists of the application controls and their event masks
- Add any required new subroutines to the program:
  - Loop through the list of controls and get their list of events from the event mask
  - Loop through the list of events to create the associated subroutine name
  - See if that subroutine is in the program, and add it if it is not present
- Remove any unneeded subroutines from the program:
  - Loop through the list of subroutines in the program
  - See if it is in the list of subroutines defined by the events in the application definition
  - If not, then check the subroutine name to see if it could be an event subroutine
  - Build an empty dummy subroutine, and then test the subroutine in the program against this dummy. If the two match, then delete the subroutine from the program
- Write the updated program back to the specified file.

So, let's try running it:

```
EVENTS.BUILD APPS XED BP.GUI
XED.EVENTS written to file BP.GUI
```

And the output looks like:

```
SUBROUTINE XED.EVENTS(subname, guievt, guiargs, guierrors,
guistate)
*******************************************************************
* Subroutine to handle events for application: XED
*
* Author : BRIAN
* Created: 14 Feb 2010
* Updated: 14 Feb 2010
* Version: 0.0.1
```

```
*
* ------------------------------------------------------------- *
*
$CATALOGUE
$MODE UV.LOCATE
$MODE UNASSIGNED.COMMON
$INCLUDE GUIBP ATGUIEQUATES
COMMON /XED.EVENTS/ CONTROLVALS

guiapp.name = 'File editor'
guiapp = FIELD(subname, '*', 1)
guifrm = FIELD(subname, '*', 2)
guictl = FIELD(subname, '*', 3)
guievent = FIELD(subname, '*', 4)
IF NOT(guievent) THEN
  guievent = guictl
  guictl = ''
END

eventhandled = @TRUE
BEGIN CASE  ;* ==> Event handler start
  CASE subname EQ 'INITIALISE'
    GOSUB initialise
  CASE subname EQ 'XED*CLOSE'
    GOSUB xed.close
  CASE subname EQ 'XED*FRMMAIN*BTNCANCEL*CLICK'
    GOSUB xed.frmmain.btncancel.click
  CASE subname EQ 'XED*FRMMAIN*BTNEDIT*CLICK'
    GOSUB xed.frmmain.btnedit.click
  CASE subname EQ 'XED*FRMMAIN*BTNEXIT*CLICK'
    GOSUB xed.frmmain.btnexit.click
  CASE subname EQ 'XED*FRMMAIN*BTNSAVE*CLICK'
    GOSUB xed.frmmain.btnsave.click
  CASE subname EQ 'XED*FRMMAIN*BTNUPDATE*CLICK'
    GOSUB xed.frmmain.btnupdate.click
  CASE subname EQ 'XED*FRMMAIN*CLOSE'
    GOSUB xed.frmmain.close
  CASE subname EQ 'XED*FRMMAIN*CMBFILE*CHANGE'
    GOSUB xed.frmmain.cmbfile.change
  CASE subname EQ 'XED*FRMMAIN*CMBMONTH*CHANGE'
    GOSUB xed.frmmain.cmbmonth.change
  CASE subname EQ 'XED*FRMMAIN*CMBYEAR*CHANGE'
    GOSUB xed.frmmain.cmbyear.change
  CASE subname EQ 'XED*FRMMAIN*GRDDATA*CHANGE'
    GOSUB xed.frmmain.grddata.change
  CASE subname EQ 'XED*FRMMAIN*GRDDATA*VALIDATECELL'
    GOSUB xed.frmmain.grddata.validatecell
  CASE subname EQ 'XED*FRMMAIN*GRDDATA*VALIDATEROW'
    GOSUB xed.frmmain.grddata.validaterow
  CASE subname EQ 'XED*FRMMAIN*RESIZE'
    GOSUB xed.frmmain.resize
  CASE subname EQ 'XED*QUIT'
    GOSUB xed.quit
  CASE 1    ;* <== Event handler end
    eventhandled = @FALSE
END CASE
IF eventhandled THEN guievt = 0

RETURN
*
* ------------------------------------------------------------ *
*
initialise:
*
*
RETURN
*
* ------------------------------------------------------------ *
*
xed.close:
*
* Default form close event handler
CALL ATGUIHIDE(guiapp, guifrm, '', '', guierrors, guistate)
```

```
IF guierrors<1> GE 2 THEN RETURN FROM PROGRAM

CALL ATGUIGETPROP(guiapp, '', '', GPSTATUS, 0, 0, NUM.FORMS,
guierrors, guistate)
IF guierrors<1> GE 2 THEN RETURN FROM PROGRAM

IF NUM.FORMS EQ 0 THEN
  CALL ATGUIDELETE(guiapp, '', '', guierrors, guistate)
  IF guierrors<1> GE 3 THEN RETURN FROM PROGRAM
END

EXECUTE "DELETE.COMMON XED.EVENTS" CAPTURING junk
*
RETURN
*
* ------------------------------------------------------------ *
*
xed.frmmain.btncancel.click:
*
*
RETURN
*
* ------------------------------------------------------------ *
*
xed.frmmain.btnedit.click:
*
*
RETURN
*
* ------------------------------------------------------------ *
*
xed.frmmain.btnexit.click:
*
*
RETURN
*
* ------------------------------------------------------------ *
*
xed.frmmain.btnsave.click:
*
*
RETURN
*
* ------------------------------------------------------------ *
*
xed.frmmain.btnupdate.click:
*
*
RETURN
*
* ------------------------------------------------------------ *
*
xed.frmmain.close:
*
* Default form close event handler
CALL ATGUIHIDE(guiapp, guifrm, '', '', guierrors, guistate)
IF guierrors<1> GE 2 THEN RETURN FROM PROGRAM

CALL ATGUIGETPROP(guiapp, '', '', GPSTATUS, 0, 0, NUM.FORMS,
guierrors, guistate)
IF guierrors<1> GE 2 THEN RETURN FROM PROGRAM

IF NUM.FORMS EQ 0 THEN
  CALL ATGUIDELETE(guiapp, '', '', guierrors, guistate)
  IF guierrors<1> GE 3 THEN RETURN FROM PROGRAM
END

EXECUTE "DELETE.COMMON XED.EVENTS" CAPTURING junk
*
RETURN
*
* ------------------------------------------------------------ *
*
```

```
xed.frmmain.cmbfile.change:
*
*
RETURN
*
* ------------------------------------------------------------ *
*
xed.frmmain.cmbmonth.change:
*
*
RETURN
*
* ------------------------------------------------------------ *
*
xed.frmmain.cmbyear.change:
*
*
RETURN
*
* ------------------------------------------------------------ *
*
xed.frmmain.grddata.change:
*
*
RETURN
*
* ------------------------------------------------------------ *
*
xed.frmmain.grddata.validatecell:
*
*
RETURN
*
* ------------------------------------------------------------ *
*
xed.frmmain.grddata.validaterow:
*
*
RETURN
*
* ------------------------------------------------------------ *
*
xed.frmmain.resize:
*
*
RETURN
*
* ------------------------------------------------------------ *
*
xed.quit:
*
*
RETURN
*
* ------------------------------------------------------------ *
*
END
```

If you check this skeleton against the one that GED generates, you will find all the necessary event handlers are present. In fact, there is one extra event handler named *xed.quit*.

There are a couple of other minor things to note about this skeleton program:

- A named COMMON has been created for the application. This contains a single variable *CONTROLVALS*. The use of this is up to you, but it is intended that as events occur in the subroutine, the updated values of the controls in the application will be stored in this variable. This will make the last recorded value of the control available to the application without having to query the control for its value.

- A variable named *`guiapp.name`* has been defined for use within the application. This will be needed for the heading on message boxes, and probably any other output from the program.

9.6 Review

At the end of all this, you may wonder why would anyone bother doing all this? In large part, what we have done here is duplicate what `GED` can do for us.

This chapter has been about introducing alternative ways to do things. You don't have to do things this way – just as you don't have to do things the way that `GED` does things for you.

Consider the following points which were covered in this chapter:

- You can build a GUI interface directly using the `ATGUIxxx` subroutines
- You can use the application definition item created by `GED` to extract information about the application
- You can write a program that will generate program code for you. This is a powerful concept, allowing you to build complex systems using rules defined in data items. As your business rules change, you can update the definitions in your data items and regenerate your programs. Obviously, we've only scratched the surface here.

What you do with the techniques outlined in this chapter is now up to you.

10 Other Bits and Pieces

10.1 Cataloguing Options

10.1.1 What is cataloguing?

Cataloguing was briefly covered in Section 3.2.1. The first obvious characteristic is that cataloguing allows a program to be run as:

```
program-name
```

rather than:

```
RUN filename program-name
```

This makes cataloguing programs optional (but convenient). However, if you are going to use external subroutines, then these MUST be catalogued.

Essentially, cataloguing allows the *OpenQM* command processor to find the program or subroutine that you are calling. In the generalised examples shown above, the `RUN` command gives the location of the program to be run as well as the program name. Cataloguing associates this location information with the program name allowing the program to be run simply by typing the name (or calling the subroutine by its name).

10.1.2 Types of cataloguing

There are several ways to catalogue programs in *OpenQM*. These are:

- Local cataloguing
- Private cataloguing
- Global cataloguing

You specify which catalogue to use when you issue the `CATALOGUE` command:

```
CATALOGUE filename program-name {LOCAL | GLOBAL}
```

Private cataloguing is the default, and cannot be explicitly specified. If you do not specify either `LOCAL` or `GLOBAL` cataloguing, then the program will be catalogued in the `PRIVATE` catalogue.

When a program is catalogued into the private catalogue, a copy of the object code is moved to the private catalogue subdirectory. By default, this is a folder named 'cat' located beneath the main account folder. The location of this folder can be changed by using a `VOC` entry named `$PRIVATE.CATALOGUE` – see the online help for more information.

Local cataloguing is similar to the cataloguing system traditionally used in PICK systems. The cataloguing process creates an entry in the account `VOC` that points to the object code in the *`filename`*`.OUT` folder associated with the program file (e.g. `BP.OUT`).

Global cataloguing uses a similar procedure to private cataloguing, but copies the object code to the `gcat` folder underneath the `QMSYS` account folder.

So, to summarise:

- `LOCAL` cataloguing creates a `VOC` entry that points to the object code in the `filename.OUT` folder
- `PRIVATE` cataloguing copies the object code to the account's cat subdirectory
- `GLOBAL` cataloguing copies the object code to the gcat folder in the `QMSYS` account.

10.1.3 Implications

This probably seems very confusing. What does it all mean?

Search order

The key to understanding the implications of the various forms of cataloguing is the order in which *OpenQM* searches these locations to find the pointer or the object code. Essentially, it searches from local to global, and uses the first entry it comes across.

For normal programs[24], *OpenQM* searches in the following order.

- The `VOC` is checked for an entry for the program name that conforms to the format of a catalogue entry
- The private catalogue is checked for a program of that name
- The global catalogue is checked for a program of that name

This means you can create a hierarchy of entries for a given program. A global entry will provide your default procedure for all accounts, while private and/or local entries provide alternative procedures in selected accounts.

The following explanation of the process has been taken from the Google Group for *OpenQM*. Note that this is in reply to a specific question about cataloguing, but most of the response is appropriate here:

There are three catalogue modes:

Global cataloguing makes the same version of the subroutine available to everyone. This does not help in your situation.

Private cataloguing is broadly equivalent to what UniVerse calls normal mode and UniData calls local mode (just to be confusing). The program is copied to a location where it is visible only to the one account. In QM, this is the cat subdirectory of the account but we extend this system by allowing you to use the $PRIVATE.CATALOGUE VOC record to move the private catalogue. This lets you share a single private catalogue

24 Some globally catalogued programs have a prefix character of * or !. If the program being searched for begins with one of these characters, the object code is read directly from the global catalogue.

between multiple accounts so that a program catalogued in one is visible to otehr selected accounts.

Local cataloguing is equivalent to what UniVerse also calls local mode but UniData calls direct mode. This time, the program is not copied at all but a VOC entry is created to point to it. The form of this is

1: V
2: CS
3: pathname

You can set up this type of VOC entry in one account to see the program in another. Unlike use of $PRIVATE.CATALOGUE, you can have an account pointing to programs in more than one alternative location.

By a suitable mix of private and local cataloguing you can achieve anything.

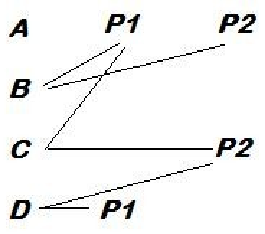
If I have accounts A, B, C and D, and programs P1 and P2....

Account A catalogues both programs using private mode and sees the copy stored in its cat subdirectory.

Account B uses the $PRIVATE.CATALOGUE record to point to the cat subdirectory of account A and hence shares the same programs.

Account C also uses the $PRIVATE.CATALOGUE record to point to the cat subdirectory of account A but uses a locally catalogued version of program P2. Because the search process sees locally catalogued programs before looking in the private catalogue, the local version of P2 is used even though it is also in the relocated private catalogue.

Account D, has its own catalogued version of program P1 using either private or local cataloguing but uses a manually created VOC record to point to the version of P2 in account C.



This is more complicated than what we require at this stage, but it is useful to know that such hierarchies can be created. However, one point from this quote that is worth elaborating on is how to manually create a `VOC` entry for a program in another account:

Say we have a program named ***HANOI*** in a file named `BP` in an account named `GAMES`. To make a reference to the object code for this program, we create an entry in the `VOC` of our current account that looks like:

```
V
CS
D:\QM\GAMES\BP.OUT\HANOI
```

You would need to change the pathname as necessary. Note that the pathname points to the `BP.OUT` file rather than the `BP` file.

Note also that you can use relative pathnames in the `VOC` entry, such as:

```
V
CS
..\GAMES\BP.OUT\HANOI
```

Relative pathnames may help if you move your *OpenQM* accounts at some point in the future.

Object code in use

An implication of cataloguing that is not immediately apparent is that the copy of the object code in the private or global catalogue is NOT automatically refreshed when you recompile a program. This means that you continue to use the version of the code that is in the relevant catalogue until such time as you catalogue the new object code.

This is why the programs shown in this book generally have the line:

```
$CATALOGUE
```

near the top of the program. This tells the compiler to re-catalogue the program after compilation. Other ways of achieving this are discussed later in this chapter under the heading Options and Compiler Directives.

This means that if you use private or global cataloguing, the newly compiled code version will not be put into use until after you re-catalogue the program. If you use local cataloguing, the new object code will be put into use immediately[25] (as the `VOC` entry points directly to the object code rather than to the catalogue copy).

10.1.4 What form of cataloguing should I use?

After reading all of the above, you may well be wondering what difference it makes, and what form of cataloguing to use. Here are some simple rules:

- ➢ If you have some programs that you wish to make available in all accounts, apply `GLOBAL` cataloguing to those programs
- ➢ If you have some programs in remote accounts that you wish to use in the current account, then create a local `VOC` entry for those programs in the current account
- ➢ For other programs – those that are only required in one account – use `PRIVATE` cataloguing.

10.1.5 Removing programs from the catalogues

Undoubtedly, you will end up with some programs catalogued that you no longer wish to be catalogued. This raises two questions:

- ➢ How do I find out what programs are catalogued?
- ➢ How do I remove these programs from the catalogue?

The answer to the first question depends on what type of cataloguing was used for the program. To find programs catalogued using `PRIVATE` or `GLOBAL` cataloguing, use the `MAP` command:

```
MAP
System catalogue map at 19:04:01 on 06 Mar 2010

  Catalogue name.................  Compiled.  Time....  ...Obj  ..Xref  ..Size
* *FIXPATH                         25 Feb 10  12:08:16    602       0     602
* *PATCH                           25 Feb 10  12:08:21   6033       0    6033
* *SET.QUERY                       25 Feb 10  12:08:29    413       0     413
* *SETDBG                          25 Feb 10  12:08:28    577       0     577
  ABC                              05 Dec 09  21:30:21   1937     579    2516
  ABC.EVENTS                       13 Dec 09  17:10:22    708     276     984
  ABC.FORM                         07 Dec 09  20:31:11   1118     287    1405
```

25 This is not technically correct. Each process will continue to use the old object code until the process releases the code. The new object code will not take effect until the old object code has been released by the process. This behaviour can be modified by using the `FORCE.RELOAD` mode of the `OPTION` comand. See the documentation for more information, or search the *OpenQM* Google Group for commentary on this subject.

Items marked with an asterisk have been globally catalogued, while the other programs are located in the local catalogue. Note that the globally catalogued items will include programs that are part of *OpenQM*, and you shouldn't be too worried that you don't recognise all the entries.

Use the `LISTV` command to find locally catalogued programs. This command actually lists verbs available to run from the command line, so it includes many commands that are not catalogued programs. To identify the locally catalogued programs, look for `CS` in the `Dsp` column[26].

Now that you know where the program is catalogued, you can remove it from the catalogue using the `DELETE.CATALOGUE` comand:

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

If you do not specify either `GLOBAL` or `LOCAL`, then `DELETE.CATALOGUE` will delete the program from the `PRIVATE` catalogue.

10.1.6 Removing unwanted object code

The process of deleting a program from the catalogues does not remove the object code for the program from the system. `DELETE.CATALOGUE` only removes the object code from the catalogue (or in the case of local cataloguing, simply removes the `VOC` pointer). This means the program object code still exists in one of the object code files.

Let's prove this:

```
HELLO
Hello World!
DELETE.CATALOGUE HELLO
HELLO deleted from the private catalogue
HELLO
HELLO is not in your VOC
RUN BP HELLO
Hello World!
```

After we deleted the ***`HELLO`*** program from the private catalogue, we could no longer run the program simply by typing its name. However, we could still run it if we used the `RUN` command and referenced the filename containing the program.

Now, if we delete the program, the program will still run …

```
DELETE BP HELLO
1 record(s) deleted
RUN BP HELLO
Hello World!
```

When we deleted the program, we only deleted the source code. The object code still exists, and still runs when called using the `RUN` command.

The object code exists in a file with a `.OUT` extension. For file `BP`, the object code is in file `BP.OUT`. So, we need to delete the object code from this file:

```
DELETE BP.OUT HELLO
1 record(s) deleted
RUN BP HELLO
Program BP.OUT HELLO not found
```

The object code is now deleted, and you can no longer run the program.

26 You could create your own variant of the LISTV command to show only the locally catalogued programs. See Section 11.4 for information on how to do this.

10.2 Options and Compiler Directives

OpenQM has a moderate number of Options and Compiler Directives. In general, these have two functions:

- They allow *OpenQM* to behave in a similar manner to one or more of the other multi-value databases
- They allow new functionality to be added to *OpenQM*. This is done by making the new functionality optional, meaning that existing programs continue to operate as intended.

There are some distinctions between Options and Compiler Directives, although these are not clear to new users.

- An option affects the overall *OpenQM* environment, and is usually set at login.
- An option affects an entire account, an individual user, or the whole system
- A compiler directive affects the way that `QMBasic` operates, and is usually applied at compile time
- A compiler directive applies only to a single program (although they are often applied to all programs in a program file or account).

10.2.1 Options

To see a quick list of options in *OpenQM*, type `OPTION` from the command prompt. This displays the name of each option, its current setting, and a short description of the option's purpose.

Some of these options affect the way that `QMBasic` operates. Examples of these are:

```
DIV.ZERO.WARNING
NON.NUMERIC.WARNING
```

These options can optionally be set during program development to allow a program to continue running after encountering an error rather than crashing. Rather than a fatal error occurring, these options result in an error message being displayed on the terminal (on the *AccuTerm* terminal screen rather than the GUI) alerting you to the presence of the error. If these options aren't set, then the program will crash, and leave you back at the command prompt.

These options should NOT be set for production accounts. If there are errors in the code, these should be debugged before the code makes it to the production account, and errors should not be "hidden" by use of such options.

Options are usually set in the `LOGIN` paragraph of each account. The following is an example of a `LOGIN` paragraph:

```
1: PA
2: OPTION PICK
3: OPTION PICK.IMPLIED.EQ
4: SETPTR 0,90,66,0,2,3, AS NEXT QMPRINT, RAW, PCL, CPI 12,
     LEFT.MARGIN 2, BRIEF
5: SETPTR 1,132,42,0,2,3, AS NEXT QMPRINT, RAW, PCL, CPI 12,
     LANDSCAPE, LEFT.MARGIN 2, BRIEF
6: IF @TTY = 'phantom' THEN STOP
7: REPORT.STYLE DEFAULT.STYLE
```

The options in this `LOGIN` paragraph are shown on lines 2 and 3.

10.2.2 Compiler directives

Compiler directives are instructions to the QMBasic compiler. These may entered into programs directly, or placed in control items that apply to multiple programs.

Compiler directives always start with a '$' symbol. Ones that we have seen so far in this document include:

```
$CATALOGUE
$INCLUDE
$MODE
```

The $MODE compiler directive has quite a number of mode settings. Which mode settings you use is largely a matter of personal preference. However, for compatibility with other multi-value systems, I do recommend use of the UV.LOCATE mode.

In the *XED* application, we added the following two compiler directives:

```
$CATALOGUE
$MODE UV.LOCATE
```

In addition, the *AccuTerm* GUI skeleton program already contained the following compiler directive:

```
$INCLUDE GUIBP ATGUIEQUATES
```

All of these have been included directly in the program (and in most cases, the $INCLUDE compiler directive will always be used in this manner). However, you will note that there is likely to be a small set of these directives that we are likely to use in (nearly) all programs.

OpenQM provides a way to apply compiler directives to a group of programs. This is through use of the $BASIC.OPTIONS control item. An example $BASIC.OPTIONS item is shown below:

```
X
CATALOGUE
MODE UV.LOCATE
```

Note that these entries aren't quite the same as compiler directives listed above. There is no leading '$' symbol, and the entries available in the $BASIC.OPTIONS entry are not the quite the same as compiler directives. Even where the entries do the same thing, the keyword may be different. For example, the keyword DEBUGGING in the $BASIC.OPTIONS item is equivalent to the compiler directive $DEBUG inserted in the program.

Look in the online help to find the available keywords. Compiler directives are under the QMBasic heading, while the $BASIC.OPTIONS keywords are found under the BASIC keyword in QM Commands.

The important thing to understand about the $BASIC.OPTIONS item is that its scope depends on where you create the item:

- If the $BASIC.OPTIONS item is created in the VOC, then the options will apply to all program files contained within the account
- If the $BASIC.OPTIONS item is created in a program file, then the options will apply to all programs within that file.

This makes it relatively easy to apply your standard set of compiler directives (once you have sorted out what they are) to all programs. You can simply enter them into a $BASIC.OPTIONS record in the VOC, and not have to worry about them thereafter. And, therein lies the rub – you can forget about these options because you don't see them after they have been set up.

There is another way to utilise a standard set of compiler directives. That is to use an $INCLUDE record in each program. For example:

In each program:

```
$INCLUDE BP QMMODES.H
```

and in the include file:

```
$IFDEF QM
  $MODE UV.LOCATE
  $MODE FOR.STORE.BEFORE.TEST
  $MODE UNASSIGNED.COMMON
$ENDIF
```

The difference between this approach and the use of a `$BASIC.OPTIONS` record is that there must be explicit reference in each program to the existence of the compiler directives that change the way the program operates. This makes it easier for another programmer to fully understand the subtleties of the operation of the program.

The above `$INCLUDE` item also shows another use of compiler directives – that is, by allowing conditional compilation (or in this case, conditional compiler directives). The compiler directives in this particular include file are only used if program is compiled in *OpenQM*. That may seem an odd thing to do, but it is useful if you wish to write code that is compatible across multiple multi-value databases.

The `$IFDEF` line uses the predefined token *`QM`*. This is automatically defined when the program is compiled in *OpenQM*, as is a corresponding token that identifies the specific operating system version of *OpenQM* (e.g. *`QM.WINDOWS`*). Other tokens to be used by the `$IFDEF` compiler directive must be defined by the `$DEFINE` compiler directive.

Note that `$IFDEF` isn't quite like an `IF` statement within a program – it doesn't test for a value associated with the variable – it tests to see whether the variable has been defined. A more fundamental difference is that the `$IFDEF` conditional statement is evaluated at compile time and impacts on the object code produced. A normal `IF` statement is evaluated at runtime using the object code produced earlier.

10.3 Menus and Security

10.3.1 Building a menu

So far, we've only got a couple of applications, and we can remember how to start them. And so far, we are the only ones with access to our system. But the above conditions won't always hold true.

What we need is some way to start our applications easily, and even some way for users to see the range of applications available. This is usually accomplished by some form of menu system.

OpenQM comes with its own built-in menu system. We'll now build a menu for our current applications:

OpenQM creates menus from menu (M-type) records in the `VOC`. However, the documentation notes that these records are often stored in a separate file, because they can become large. We'll follow this pattern, so we need to create a file to hold our menu records.

```
CREATE.FILE MENUS
```

Now, we need to create a menu record. Once again, *OpenQM* comes with its own menu editor, named `MED`. We invoke this in the standard way:

```
MED MENUS MENU.MAIN
```

This gives us a screen that looks something like the following:

```
Title :
Subr  :
Prompt:
Exits :
Stops :------------------------------------------------




MENUS MENU.MAIN
F1=Help
Title to appear at the head of the menu display
```

Note that most of the blank lines have been stripped out.

We start by filling in the details for the menu description. The essential bits are the title to display on the screen, and the key to press to exit the menu. Note that the exit key is entered in both upper and lower case, with the values separated by commas. This is shown below:

```
Title :M A I N   M E N U
Subr  :
Prompt:
Exits :X,x
Stops :
```

This tells *OpenQM* that using the 'X' key will exit from the menu.

Now, we want to add our applications. Position the cursor on the horizontal line and press 'Enter'. This will create an entry for a menu line:

```
Text 1:
Action:
Help  :
Access:
Hide  :
-------------------------------------------------------
```

The 'Text' is the text to display in the menu. The 'Action' is the command to run within *OpenQM* to start the application. That is all you need to fill in.

When you've completed one entry, position the cursor on the bottom horizontal line, and then press enter. Fill in the details for the next application. The menu should look something like:

```
Title :M A I N   M E N U
Subr  :
Prompt:
Exits :X,x
Stops :
-------------------------------------------------------
Text 1:XED Editor
Action:XED
Help  :Run the Exchange Rate editor
Access:
Hide  :
-------------------------------------------------------
Text 2:ABC Application
Action:RUNGUI ABC
Help  :Run the ABC application
Access:
Hide  :
-------------------------------------------------------
Text 3:Log off
Action:OFF
Help  :
Access:
Hide  :
-------------------------------------------------------
```

Press **`Ctrl-X-S`** to save the menu, and **`Ctrl-X-C`** to exit from `MED`.

Now, *OpenQM* needs an entry in the `VOC` before it can run the menu. So, we now need to create a remote item pointer (R-type) in the `VOC`. Use your favourite editor to create the following entry in the `VOC`, named ‘`MENU.MAIN`’.

```
R Main Menu
MENUS
MENU.MAIN
```

Once you’ve saved this item, type: `MENU.MAIN` from the command prompt, and the menu should appear:

```
                              M A I N   M E N U

  1 = XED Editor
  2 = ABC Application
  3 = Log off

Select option (1 - 3) =
```

Press ‘1’ followed by ‘Enter’ and the *`XED`* editor should start. Exit from that, and press ‘2’ followed by ‘Enter’. The *`ABC`* application should start.

There are two ways to exit from the menu. One is to use the exit keys we defined above (‘X’ or ‘x’). The second is the default ‘Stops’ entry for the menu, which is ‘Q’ (or ‘q’).

Now, it is still inconvenient to type ‘`MENU.MAIN`’ do display the menu, so we should put this into the `LOGIN` item for the account. Edit the `LOGIN` entry in the `VOC`, and put `MENU.MAIN` as the last line in the paragraph:

```
PA
OPTION PICK
SETPTR 0,90,66,0,2,3, AS NEXT QMPRINT, RAW, PCL, CPI 12, LEFT.MARGIN 2, BRIEF
SETPTR 1,132,42,0,2,3, AS NEXT QMPRINT, RAW, PCL, CPI 12, LANDSCAPE, LEFT.MARGIN
  2, BRIEF
IF @TTY = 'phantom' THEN STOP
REPORT.STYLE DEFAULT.STYLE
INIT.USER
MENU.MAIN
```

Next time you log in, the menu should automatically appear. You can test this by typing: `LOGIN` from the command prompt.

10.3.2 Security

A number of the entries in the menu record deal with menu security. You may wonder what this means.

Menu security deals with the following issues:

- Who is allowed to exit from the menu to the command prompt?
- Who is allowed to run any given application?
- Should users be allowed to see menu entries for applications that they are not authorised to run?

The best way to understand the options is to to put some security in place.

We’ll start by defining groups of users. Users will then be given membership of these groups. Users can belong to more than one group.

As this is only a quick overview, we’re not going to formally define these groups by putting them in a file – we’ll just note the following groups “exist”:

`SYSADMIN`
`DEV`

USER
TCL

Most of these groups are self-explanatory. TCL may be a new term to some readers. In the multi-value world, TCL stands for *Terminal Control Language*, and in this usage will define those users who will be able to exit the menu system to the command prompt.

Now, let's create a file which defines the group membership for each user. Create a file named USERS. It is important that this is a dynamic file[27].

CREATE.FILE USERS

Create a dictionary item for the first field named GROUPS. This is a multi-valued field with a format of 10L.

We now need to define the users who can log onto the system in this file. The group membership of each user should be entered as a multi-valued list in field 1. Use the VM button in the WED editor to enter the value marks. The following shows the group membership for 3 users:

SORT USERS GROUPS
```
USERS.....    Groups....
ANDREW        USER
BRIAN         SYSADMIN
              DEV
              USER
              TCL
ROB           USER

3 record(s) listed
```

Andrew and Rob have been assigned membership of only the *USER* group, while Brian is assigned membership of all 4 groups.

Now, we need to create a QMBasic function that queries this file to determine whether a user is a member of a given group:

```
FUNCTION SECURITY.USER.TYPE(username, type)
*******************************************************
* Author : BSS
* Created: 20 Dec 2009
* Updated:
* Version: 1.0.0
*
* Returns true if user is in specified group; false otherwise.
*
$CATALOGUE GLOBAL
$MODE UV.LOCATE

rtnvalue = @FALSE
OPEN 'USERS' TO users ELSE RETURN rtnvalue

username = OCONV(username, 'MCU')
READ userrec FROM users, username ELSE userrec = ''

type = type<1, 1>
LOCATE type IN userrec<1> SETTING upos THEN
  rtnvalue = @TRUE
END

RETURN rtnvalue
*
* ----------------------------------------------- *
*
END
```

27 This is because *OpenQM* cannot enforce security on a directory file. Anyone with appropriate O/S permissions may edit an item within a directory file. Similarly, *OpenQM* cannot apply triggers to directory files. For these reasons, files that involve security MUST be created as dynamic files.

A function must always return a value. In this case, the value returned will always be `@TRUE` (1) if the user is a member of the passed group, or `@FALSE` (0) if they are not.

Now, we want to utilise this function in the menu to determine who gets to execute which programs. The menu requires a subroutine for this purpose:

```
SUBROUTINE SECURITY.MENU(ok, menuname, key)
$CATALOGUE GLOBAL
DEFFUN SECURITY.USER.TYPE(username, type)

ok = SECURITY.USER.TYPE(@LOGNAME, key)

RETURN
END
```

Essentially, this subroutine simply declares the function, and then calls the function using the passed ‘key’ variable. The ‘key’ variable needs to come from the menu definition. So, let’s revisit our menu definition:

```
Title :M A I N   M E N U
Subr  :SECURITY.MENU
Prompt:
Exits :
Stops :
-------------------------------------------------------
Text 1:XED Editor
Action:XED
Help  :Run the Exchange Rate editor
Access:USER
Hide  :1
-------------------------------------------------------
Text 2:ABC Application
Action:RUNGUI ABC
Help  :Run the ABC application
Access:DEV
Hide  :1
-------------------------------------------------------
Text 3:Utilities
Action:MENU.UTILITIES
Help  :
Access:
Hide  :
-------------------------------------------------------
Text 4:Log off
Action:OFF
Help  :
Access:
Hide  :
-------------------------------------------------------
```

This menu calls a second menu for utilities:

```
Title :U T I L I T I E S
Subr  :SECURITY.MENU
Prompt:
Exits :X²x
Stops :
-------------------------------------------------------
Text 1:List users
Action:LISTU;
Help  :
Access:
Hide  :
-------------------------------------------------------
Text 2:List local files
Action:LISTFL;
Help  :
Access:DEV
Hide  :
-------------------------------------------------------
```

```
Text 3:List all files
Action:LISTF;
Help  :
Access:DEV
Hide  :
-------------------------------------------------------
Text 4:Log off
Action:OFF
Help  :
Access:
Hide  :
-------------------------------------------------------
```

This Utilities menu is once again defined in the `MENUS` file, and has an R-type pointer created in the `VOC` so the system can find it.

Now, let's see what is different here:

- Both menus declare the security subroutine that we created above
- The 'Exits' characters have been removed from the Main Menu, but they are still included in the Utilities menu
- A number of options have ***DEV*** entered in the 'Access' field
- Some of the options with 'Access' entries also have a '1' in the 'Hide' field.

What do these changes mean?

When the menu is displayed, each line of the menu is run through the security subroutine. If an 'Access' key is assigned to the menu option, only users belonging to that group will be able to execute that option. Further, if there is a non-zero entry in the 'Hide' field, only users belonging to that group will be able to see the option.

This means that Brian will see this menu:

```
                                  M A I N   M E N U

  1 = XED Editor
  2 = ABC Application
  3 = Utilities
  4 = Log off

Select option (1 - 4) =
```

while Andrew sees this menu:

```
                                  M A I N   M E N U

  1 = XED Editor
  2 = Utilities
  3 = Log off

Select option (1 - 3) =
```

Andrew also has a different perspective on the Utilities menu:

```
                                  U T I L I T I E S

  1 = List users
( 2)= List local files
( 3)= List all files
  4 = Log off

Select option (1 - 4) =
```

The options in parentheses are not available to Andrew, but the menu definition did not hide them.

Entering ‘X’ at the prompt in the Utilities menu returns both users back to the Main menu. However, nothing happens when entering ‘X’ at the Main menu prompt.

At this stage, both users can exit from either menu by entering ‘Q’. This activates the ‘Stop’ option within the menu.

This gives us a little problem. We only want users who are members of the *TCL* group to have access to the command prompt. However, we can’t stop access to these ‘Stop’ codes.

Instead, we catch the result of the ‘Stop’. Internally, the ‘Stop’ action causes an abort event. If we create an `ON.ABORT` paragraph, we can catch the abort, and take action to stop people from getting to the command prompt.

Create the following `ON.ABORT` paragraph in the `VOC` file:

```
PA
IF <<SUBR(SECURITY.USER.TYPE, <<@LOGNAME>>, 'TCL')>> EQ 1 THEN GO tcl

MENU.MAIN

tcl:
```

Now, if you wondered why we created a function to check a user’s group membership above when the menu wanted a subroutine – here is the answer. We created a function so that we could use it in this paragraph (although, curiously, the keyword allowing us to use it is `SUBR`[28]).

This paragraph calls the security function we created earlier, passing the user’s login name and asks whether the user is a member of the *TCL* group. If they are, then control is directed to the *`tcl:`* label; but if they aren’t, then the Main menu is called.

Once this is in place, if Andrew attempts to use ‘Q’ to exit from the menu, then he will simply be redirected back to the Main menu. If he tries to do it again, then he is disconnected.

On the other hand, Brian can still use ‘Q’ to exit from the menu system out to the command prompt.

10.3.3 File and account security

The above measures allow you to control what options are available to users when they are in the menu system. But you need to do more than this before you have a basic secure system. Other measures (may) include:

- forcing users to log in via a specific account
- using security subroutines to restrict access to accounts
- using triggers to restrict who can update certain files
- preventing use of the break key within programs to get to the command prompt.

First of all, let’s consider how someone who knows the structure of the system might circumvent the security we’ve put in place so far. They could:

- log in to another account (such as `QMSYS`)
- create a Q-pointer to the `USERS` file
- give themselves membership of any user group.

At this point, they potentially have total control over the system. So, how do we prevent this?

28 The answer to this curiosity is that a function is actually created as a subroutine with a hidden first parameter that returns the function result. So, while we see a function, *OpenQM* actually sees a subroutine.

Restricting file updates

We'll start by controlling who makes updates to the USERS file. This is done by using a trigger function.

Triggers can be assigned to files to call subroutines before or after certain file actions. Subroutines called before file actions may validate the data before it is written to file, or verify that the user has authority to update the file. Subroutines called after the file action may update other files based on the change made to this file.

A file can have only one trigger subroutine which has to be programmed to handle each of the triggered events. It has a specific calling structure which passes (amongst other things) the file action, and the record.

The following subroutine is used to check the users authority before updating the USERS file:

```
SUBROUTINE TRIGGER.USERS(mode, recid, recdata, onerror, fvar)
$CATALOGUE GLOBAL
$INCLUDE KEYS.H
DEFFUN SECURITY.USER.TYPE(username, type)

@TRIGGER.RETURN.CODE = 0

IF (mode EQ FL$TRG.PRE.WRITE) OR (mode EQ FL$TRG.PRE.DELETE) OR
(mode EQ FL$TRG.PRE.CLEAR) THEN
  usertypes = 'SYSADMIN,DEV'
  CONVERT ',' TO @VM IN usertypes
  GOSUB checkusertypes
  IF NOT(found) THEN @TRIGGER.RETURN.CODE = 1
END

RETURN
*
* -------------------------------------------------------- *
*
checkusertypes:
*
dc = DCOUNT(usertypes<1>, @VM)
FOR ii = 1 TO dc
  usertype = usertypes<1, ii>
  found = SECURITY.USER.TYPE(@LOGNAME, usertype)
  IF found THEN EXIT
NEXT ii
*
RETURN
*
* -------------------------------------------------------- *
*
END
```

Once again, this trigger subroutine uses the function we created earlier to verify the group membership of the user attempting the update on the USERS file. If the user is not a member of either the *SYSADMIN* or *DEV* user groups, then the subroutine will set the @TRIGGER.RETURN.CODE variable to a value of '1'. If this variable is set to any non-zero value, then the file action will fail with an abort.

Before this can happen, we need to apply the trigger. We do this with the SET.TRIGGER command, and we specify which trigger events the system should generate. To apply a trigger, the command format is:

```
SET.TRIGGER filename function-name modes
```

In this case, it would be:

```
SET.TRIGGER USERS TRIGGER.USERS PRE.WRITE PRE.DELETE PRE.CLEAR
```

SET-TRIGGER also reports on the trigger status of a file:

```
SET.TRIGGER USERS
Trigger function: TRIGGER.USERS
Modes: PRE.WRITE, PRE.DELETE, PRE.CLEAR
```

Or, you can use the LIST.TRIGGERS command to show the triggers on all files in the current account:

```
LIST.TRIGGERS
File Name.................. Function................... Pre Post
USERS                       TRIGGER.USERS               WDC
```

Now, if Andrew followed the steps above to circumvent the menu security (by giving himself higher group membership), he should find that he can no longer update the USERS file. The MODIFY editor reports the following error:

```
USERS: ANDREW
Error 3022 writing record
Data validation error: 1
Press return to continue
```

The ERR.H header file (in the SYSCOM file) tells us that error 3022 is a trigger function error.

Of course, the next step for Andrew is to remove the trigger function on the USERS file (he would have to do this from another account). Once this is done, he can update the USERS file. To remove the trigger function, he would use:

```
SET.TRIGGER USERS ""
```

Now, he can modify the USERS file, and give himself membership of other groups, and then go exploring the system.

So, what can we do to prevent this? Well, there are a couple of things:

- We can restrict the login account for a user
- We can run a similar check on changing account as we do for aborts.

Restricting the login account

We can change some settings about users with the ADMIN.USER program. There is a catch, however. You must be an administrative user to use the ADMIN.USER program.

In Linux, this will require starting an interactive root console session before starting an *OpenQM* session in the QMSYS account.

```
sudo -i
qm
ADMIN.USER
```

You can now grant yourself administrative rights. Once you have done this, exit from from your root session, and return to a normal *OpenQM* session.

If you are using the GPL version of *OpenQM*, you may also need to edit the screen definition for the ADMIN.USER screen. Log to the QMSYS account and then:

```
ED $SCREENS $ADMUSER                 invoke the editor
25                                   go to line 25
C/@QMSYS\//                          change the text on the line
FI                                   file the item
```

This modification should not be necessary on recent versions of *OpenQM*. (Try the program first. If it works, it doesn't need the modification).

Start ADMIN.USER and enter in the username of the user you wish to restrict. You can press F2 to get a list of users.

Type in 'A' to Amend the user.

Press ‘Enter’ to move through the user record. At the ‘Force account’ prompt, enter `QMINTRO`. If you can’t enter this (or any other account name), then you need to modify the screen definition as outlined above.

When you get to the bottom of the screen, enter ‘F’ to file the item.

Next time the user logs in, they will not get a prompt asking which account to log in to. They will simply be taken straight into the `QMINTRO` account.

You can see the settings that apply to each account using the `LIST.USERS` command. (This should not be confused with the `LISTU` command which shows users currently logged into *OpenQM*).

```
LIST.USERS
User name............  Login account...  Last login.....  Admin
andrew                 QMINTRO           22 Dec 09 20:50   No
brian                                    22 Dec 09 20:18   Yes
qm                                       15 Mar 09 19:51   No
rob                                      22 Dec 09 20:59   No
root                                     22 Dec 09 20:26   No
```

Restricting account access

Andrew is now forced to log into the `QMINTRO` account. Once there, he can’t get out of the menu system. However, if he does manage to manage to get to the command prompt, he can still log to another account, and start operating from there.

What we now want to do is restrict users from changing accounts. Or more particularly, this document is only going to point you in the right direction, and let you join the dots:

There are two points at which you can intercept the action of changing accounts:

- We can catch the account change before the change occurs using the `ON.LOGTO` paragraph. This is similar to the `ON.ABORT` paragraph that we have already used
- We can intercept the user during the login process by placing appropriate commands in the `LOGIN` paragraph.

While the `ON.LOGTO` paragraph is appealing, we have the difficulty of catching the new account name. It is easier to use the `LOGIN` paragraph as by this time, the account name will be stored in the system variables `@ACCOUNT` and `@WHO`.

So, what we need is to evaluate this new account name against a list of accounts that this particular user is allowed to access. If the user is not permitted to access the account, then the paragraph will log the user off the system.

This is a broadly similar authorisation process to that already used for command prompt access, and file editing access. This suggests we need to:

- store the list of accounts for each user in the `USERS` file along with the user group information
- write an appropriate subroutine to query this information and return a true or false value
- incorporate a call to that subroutine in the `LOGIN` paragraph of EVERY account.

You may want to consider the relationship between a user’s group membership and the account they are logged into. Is their group membership consistent across all accounts, or do they have different memberships depending on the account? This design consideration will impact on the security subroutines that you need to develop (and on the subroutines already covered here).

10.3.4 Other security measures

There are layers of security built into *OpenQM*. You will need to go through the manual to find them all, and then spend some time thinking about your system and how best to implement a security policy.

OpenQM allows you to add security to any of the verbs used from the command prompt. Verbs have a `VOC` entry with a 'V' in field 1. (You can list all the verbs in the account by using the `LISTV` command). This will include locally catalogued programs as well as the general verbs supplied with *OpenQM*.

Verbs are sometimes accessed via remote pointers (R-types). Security can also be added to these remote pointer items.

In both cases, the name of the security subroutine is entered into field 5 of the V-type or R-type item in the `VOC`. The structure of the security subroutine is outlined in the online documentation – see the 'Security Subroutines' topic.

Finally, you should build security into your applications. While that potentially is another large issue, at its heart it is still a matter of checking a user's authority to use the application.

All of these security measures imply some degree of user administration to ensure that users are assigned appropriate rights.

On the other hand, if you have only a single user system, with you being the only user, there may be no need to enable security. Nevertheless, your application may be so good that you others want to use it, and so security could still become an issue. It will be easier to implement security in this instance if it has been thought about from the start – even if it hasn't been turned on.

10.4 Character Interface *AccuTerm*

Almost everything up to this point has dealt with the *AccuTerm* GUI. However, *AccuTerm* can also be used for traditional character user interface (CUI) or "green-screen" applications.

Using *AccuTerm*'s ability to run *VBA* scripts on the client, CUI programs can be given a makeover to make them more like a GUI. It is relatively simple to add message boxes, and pop-up pick-lists, and thereby make an older application more visually appealing.

A number of scripts are included with *AccuTerm* in the `Atwin\Samples\PICKBP` folder (usually beneath the `Program Files` folder) for this purpose. These include:

- ➢ *`ATINPUTTEXT`* – a simple one-line input box
- ➢ *`ATLISTBOX`* – a pop-up pick-list
- ➢ *`ATMSGBOX`* – a message box with configurable buttons and icons

The point of this section is not to cover how to use these subroutines – they are basically similar to the equivalent `ATGUIxxx` subroutines. Rather, it is to warn you NOT to use these subroutines within a GUI application.

There are two particular aspects that make these subroutines troublesome in a GUI environment. The first is that the values for icons and buttons are different from those used by the GUI environment. This isn't a show-stopper as programs can be written to use the correct values depending on which environment you are using.

The second issue is that these subroutines are executed from the main *AccuTerm* window. What this means is that if you call these subroutines from a GUI application, the message box or pick-list may appear underneath the GUI window. This potentially leaves the user wondering what has happened as the GUI window will be unresponsive as it will be waiting for the non-GUI window to terminate. The non-GUI window will stay hidden until the user clicks the *AccuTerm* icon in the task bar.

By all means, use these subroutines within a CUI application, but if you are creating a GUI application, then stick with the `ATGUIxxx` subroutines.

11 Questions and Answers

11.1 Getting Help

Where can I get more help?

See Section 1.4 for sources of help.

Where can I find out more about programming?

There are a huge number of books available about programming. Most of these are tailored to specific languages which, in most cases, are quite different to the versions of `BASIC` in multi-value databases. However, the principles of programming are similar across all languages.

Are there any books available on multi-value programming?

For a programming guide specifically tailored to multi-value databases, see *Pick/BASIC: A Programmer's Guide* by Jonathon E. Sisk. This can be downloaded from: `http://jes.com/pb/index.html`

11.2 Files

How do I manually create a pointer to a file?

Use one of the editors to create an entry in the `VOC`. In general, this should look like:

```
F {File description}
Data file
Dictionary file
```

For example:

```
File for basic programs
BP
BP.DIC
```

The entries for each line may contain operating system paths. For example:

```
F
D:\QM\QMINTRO\BP
D:\QM\QMINTRO\BP.DIC
```

or:

```
F
C:\TEMP
```

Alternatively, the pointer could be a Q-pointer, which has the following format:

```
Q
Account
Filename
```

For example:

```
Q
SYSCTRL
SYSCTRL.SETTINGS
```

A Q-pointer can reference a multi-file using the following format:

```
Q
Account
Dictname,Filename
```

How can I find out whether a file is correctly sized?

In general, dynamic files look after themselves. However, you can check on this by using `ANALYSE.FILE`.

```
ANALYSE.FILE {dict} filename {STATISTICS}
```

You can use the statistics from this report to determine whether the file is overflowed, and whether the group-size is appropriate. If you have fewer than 10 records per group (and the load factor is high), then your group size is probably too small.

You can also use the `FILE.STAT` command to get a quick picture of all the files in the account. Note that the output device (screen or printer) must have at least 132 columns to display the output of this command:

```
FILE.STAT {account-list | ALL} {LPTR}
```

How can I change the configuration of a file?

Use the `CONFIGURE.FILE` command to change the file configuration. For example:

```
CONFIGURE.FILE filename GROUP.SIZE 2
CONFIGURE.FILE filename SPLIT.LOAD 75 MERGE.LOAD 45 IMMEDIATE
CONFIGURE.FILE filename DEFAULT
```

See the online help for more information on the available options for this command.

How can I change the default group-size of new files?

Use the configuration editor (Start | Programs | QM | QM Config Editor), or edit the O/S level `qmconfig` file in the `QMSYS` folder using a text editor. Set the `GRPSIZE` parameter to the desired group size (1, 2, 4, 8, or 16).

Alternatively, place a `CONFIG` command in the account `LOGIN` item:

```
CONFIG GRPSIZE n
```

My GPL version of OpenQM can't read the files from commercial OpenQM.

This should only apply to dynamic files. Directory files should be readable from either version, although you may have to update the line-ends in the items.

Older versions of OpenQM have dynamic files named '~0' and '~1' (and more if you have indices defined). In newer versions of OpenQM, these files have been renamed to '%0' and '%1' respectively. This change was made because some system cleaners mistook the OpenQM files as temporary files left behind by other programs, and deleted them.

To make your newer version files readable on older systems, rename the files by replacing the percentage sign (%)with a tilde (~) for each file component.

11.3 Editing Items

What editors are available within OpenQM?

OpenQM comes with 3 editors built in:

- `ED` is a line editor which can edit any item
- `MODIFY` is a dictionary driven editor that is best suited to editing dictionary items or items that are defined by dictionary items
- `SED` is a full screen editor.

In addition, *AccuTerm* comes bundled with `WED` – a *Windows* based full screen editor.

The syntax for starting any of these editors is:

```
editor-name file-name item-name
```

Can any other editors be used?

Yes. For items in directory files (such as programs), you can use any text editor that has access to the directory.

You can use a program to invoke the text editor. See the `NOTEPAD` program in the `SAMPLES` file of the *`AccuTerm`* account for one example. You can use any other text editor in place of Notepad. The `NOTEPAD` program uses the `DOSSVC` subroutines as an interface between *OpenQM* and the text editor.

Other strategies for using (*Windows*) text editors to edit your items include:

- setting up a mapped drive to the directory. You should then be able to edit the items by right-clicking on them and choosing 'Edit'.
- use a fully qualified pathname (e.g. \\server\share\path\file) to identify the item to be edited as you invoke the editor
- write the item you wish to edit out to a temporary file on your local hard drive. You can then edit the temporary item, and write it back to *OpenQM* when editing is complete. This whole process can be wrapped in an editor "program" (like the `NOTEPAD` example) so it looks like one action.

For an example of several of these strategies being used, download the `SysCtrl` account from www.rushflat.co.nz and try (or look through the code for) the `DE3` editor.

Note that some of these strategies only enable you to edit items in directory files. Others, such as those using `DOSSVC` or writing the item to a temporary file, allow you to edit any item (including those in hashed files).

Another point to watch is that *OpenQM* cannot lock any items in directory files. Therefore, there is potential for multiple users to edit the items simultaneously.

I've created a program to start a Windows editor, but nothing happens.

If you are working on a network, have a look on the server to see if you have started your editor there. If this is the case, then you are executing your commands on the server rather than on the client. *AccuTerm* has private escape sequences that allow you to start Windows programs on the client. See Section 7.2.3 for examples of starting programs on the client.

Otherwise, check to see that the editor is actually installed, and whether there are any permission restrictions on who can start the editor.

11.4 Programming

When I run my program, the changes I have just made are not being used.

This happens if you have catalogued the program, but have not re-catalogued the program after the latest compile.

How do I catalogue a program?

Use the `CATALOGUE` command from the command line. To automatically catalogue the program after each compile, use one of the following two methods:

- Place a `$CATALOGUE` compiler directive in the the program
- Place a `CATALOGUE` command in a `$BASIC.OPTIONS` item in either the basic programs file, or the `VOC`.

How do I create a variant of the LISTV command to only show locally catalogued programs?

Follow the `LISTV` command back from the `VOC`. The `VOC` entry points to the `LISTV` command in the `QM.VOCLIB` file. What we want to do is create a variant of this command. First, copy the existing command and pointer to a new items named `LISTCS`.

```
COPY FROM QM.VOCLIB LISTV,LISTCS
1 record(s) copied.
COPY FROM VOC LISTV,LISTCS
1 record(s) copied.
```

Now, edit the `LISTCS` item in the `QM.VOCLIB` file so that it looks like:

```
Sentence to list locally catalogued programs
SORT VOC @ID COL.HDG "Program" FMT "16L" DESC FMT "28T" DISPATCH PROCESSOR_
WITH TYPE = "V" AND WITH DISPATCH = "CS" ID.SUP HEADING "Locally catalogued
programs defined in the VOC'G'Page 'S'"
```

The changed and added portions of this line have been highlighted.

Now change the `LISTCS` item in the `VOC` so that it points to the `LISTCS` command in `QM.VOCLIB`. Once this is done, you should be able to type in `LISTCS` from the command line and get a list of the locally catalogued programs.

```
LISTCS
Locally catalogued programs defined in the VOC
Program.........   Description................   Dsp   Processor..............
ATCREATEOBJECT     V                             CS    D:\QM\ACCUTERM\OBJBP.OUT
                                                       \ATCREATEOBJECT
ATENDEVENT         V                             CS    D:\QM\ACCUTERM\OBJBP.OUT
                                                       \ATENDEVENT
```

I want to delete the object code for any programs that no longer exist in my programs file. How do I do that?

This is simply a matter of using the `QMQuery` tools for selecting items from files. We want to select items that are in the `.OUT` file, but are not in the base file. If we assume the file is named `BP`, then:

```
SELECT BP.OUT
19 record(s) selected to list 0
NSELECT BP
1 record(s) selected to select list 0
```

At this stage, you should list the items to make sure that these are items that you wish to delete.

```
LIST BP.OUT
BP.OUT......
HELLO

1 record(s) listed
```

If you still want to delete these items, then repeat the selections, and then use a `DELETE` command:

```
SELECT BP.OUT
19 record(s) selected to list 0
NSELECT BP
1 record(s) selected to select list 0
DELETE BP.OUT
Use active select list (First item 'HELLO')? Y
1 record(s) deleted
```

There are a lot of statements and functions in the `QMBasic` language. How do I find out what they do?

Go through the help system. Every statement has a page giving an overview of its functionality and its syntax. Usually, there is an example of its usage.

Look at the page entitled 'QMBasic Statements and Functions by Type'. This groups the statements and gives a one-line description of its purpose. This may help you decide if this is the statement that you are looking for.

Try writing short test programs to see how they operate.

Search the *OpenQM* Google group or `comp.databases.pick` newsgroup for the keyword that you are interested in. There is almost certainly some example usage in those resources.

What about object-oriented programming?

Object-oriented programming is a form of modular programming that groups related functions and subroutines into "objects". To include these functions and subroutines into your code, you instantiate an instance of the object.

See the documentation for more details – see the page entitled 'Object Oriented Programming'. Look also at the class module *`index.cls`* in the `BP` file in the `QMSYS` account.

There are also a number of "packages" available for *OpenQM* which use object-oriented programming. To load packages, you should first be logged into the `QMSYS` account. Then from the command prompt, type:

```
QMPKG INSTALL QMPKG
```

This will download and install the package manager. Then to see a list of packages available, type:

```
QMPKG LIST
```

Download and install a package by:

```
QMPKG INSTALL package-name
```

The packages are generally loaded into the `CLASS.BP` file in the `QMSYS` account. Look at these as examples of object-oriented programming and usage.

Is there any version control system available for OpenQM?

There is no specific version control system for *OpenQM* or any other multi-value system.

However, as *OpenQM* stores its programs in directories, any version control system that operates on directories will be able to be used.

I want to get a data element from an Excel spreadsheet for use within OpenQM.

Section 7.3 outlined using a script to reformat and save the contents of an Excel spreadsheet as a `CSV` file. This may be overkill if you only want a single cell value, or a small range of values.

Look at the Object-Bridge functionality within *AccuTerm*. This lets you control any Windows application on the client machine that is capable of OLE automation. Therefore, you could instantiate an Excel object, open the spreadsheet, and get the cell value(s) that you require.

11.5 AccuTerm

How do I find out the syntax for the AccuTerm GUI subroutines?

Read the documentation – specifically the Programmers Guide.

Alternatively, use the online help system. From the Contents page, choose ‘Features for MultiValue (Pick) Users’, then ‘AccuTerm 2K2 GUI Development’, and ‘The GUI Library’. The subroutines are grouped into categories below this heading.

How do I add a context (right-click) menu to my application?

Open `GED` and select the form that is to have the right-click menu. Click on the ‘popup menu’ icon in the list of controls. This will display the menu creator. Give the menu a proper name (otherwise it will default to ‘Popupmenu1’).

Now add the options to the menu using the ‘New item’ button, or add sub-menus using the ‘New sub-menu’ button. Give each option a ‘Tool ID’ and a Caption. You can add an icon for the menu here too.

Create the menu by clicking on the ‘Create’ button.

You now need to enable an event to call this menu. Let’s say that you want to call the menu by right-clicking on the grid. In `GED`, select the grid and display its properties. Select the ‘Events’ tab, and enable the ‘Context’ event. Apply the change, save the changes, and then update the code using the options accessed from the ‘Update code’ icon. Save the code changes.

At this point, if you compile and run the program, you should be able to right-click on the grid and have a pop-up menu display. However, none of the options will be functional as we haven’t added any code to the event handlers in the program to respond to your choice. (The event handlers were added to the code when you used the ‘Update code’ icon). So, go back to the program and add the necessary code to the event handlers.

I want to add a status bar to my GUI application. How do I do this?

Current versions of *AccuTerm* do not have a status bar component. You probably need to use a panel containing a series of labels to simulate a status bar.

AccuTerm 7 will have a status bar component.

I’m having problems with the AccuTerm GUI. Where can I find help?

Search the forum at www.asent.com to see if anyone else has had a similar problem. If you can’t find the answer there, post a question to the forum.

11.6 Alternative Interfaces

Can I connect any other programming languages to OpenQM?

OpenQM has two API’s that allow connections to *Visual Basic* or *C*. See the documentation for information on `QMClient`.

You could also the multi-value server that is part of *AccuTerm*. See the *AccuTerm* documentation for more information on this approach.

Other third-party products such as *mv.NET* (see: http://www.bluefinity.com/) also allow connection of external programming languages.

Can I create a web interface to OpenQM?

Coyote (http://coyote.easyco.com/) is a web-server that is available for multi-value databases. This used to be bundled with *OpenQM*. However, this arrangement has been discontinued due to the low uptake of *Coyote* by *OpenQM* users.

DesignBais (http://www.designbais.com/) is an application development framework that utilises a web front-end. The resulting application could be used in intranet, extranet, or internet situations.

With some work, you can also drive a web site almost directly from *OpenQM*. For more information on this, go to the *OpenQM* web site (http://www.openqm.com/), click on the ‘About OpenQM’ link, and then ‘Presentations’. Download the presentations from the 2009 International Spectrum Conference.

12 Appendix 1: XED

12.1 Program XED

```
* AccuTerm GUI Application Skeleton
*
* Add your equates and code to open files here...
*
$CATALOGUE
$MODE UV.LOCATE
EQUATE STX TO CHAR(2)
EQUATE CR  TO CHAR(13)
EQUATE EM  TO CHAR(25)
EQUATE ESC TO CHAR(27)

OPEN 'APPS' TO gui.project.file ELSE PRINT "'APPS' is not a file name."; STOP
* Read the GUI template into template
READ template FROM gui.project.file,'XED' ELSE PRINT "'XED' is not on file."; STOP

OPEN 'XED.VAR' TO xed.var ELSE STOP 201, 'Xed.Var'
OPEN 'SCRIPTS' TO scripts ELSE STOP 201, 'Scripts'

clientname = ''
CALL GET.COMPUTERNAME('C', clientname)
*
*
************************************************************
*
*
*-->BEGIN GUI HEADER<--*
$INCLUDE GUIBP ATGUIEQUATES
*
CALL ATGUIINIT2(template<2,2>,'',guierrors,guistate)
IF guierrors<1> >= 3 THEN GOTO gui.error
*
CALL ATGUIRUNMACRO(template,'',guierrors,guistate)
IF guierrors<1> >= 3 THEN GOTO gui.error
*
*-->END GUI HEADER<--*
*
*
************************************************************
*
GOSUB initialise
*
*-->BEGIN GUI STARTUP<--*
```

```
CALL ATGUISHOW('XED','FRMMAIN','','',guierrors,guistate)
IF guierrors<1> >= 2 THEN GOTO gui.error
*
*-->END GUI STARTUP<--*
*
*
************************************************************
*
*
*-->BEGIN EVENT LOOP<--*
LOOP
  CALL ATGUIWAITEVENT(guiapp,guifrm,guictl,guievt,guiargs,guierrors,guistate)
  IF guierrors<1> >= 2 THEN GOTO gui.error
UNTIL guievt = GEQUIT DO
  guiapp=OCONV(guiapp,'MCU')
  guifrm=OCONV(guifrm,'MCU')
  guictl=OCONV(guictl,'MCU')
  GOSUB GUI.DECODE.EVENT
REPEAT
*
*-->END EVENT LOOP<--*
*
*
************************************************************
*
*
*-->BEGIN GUI TRAILER<--*
CALL ATGUISHUTDOWN
STOP
*
*-->END GUI TRAILER<--*
*
*
************************************************************
*
*
*-->BEGIN EVENT DECODER<--*
GUI.DECODE.EVENT: *
*
IF NUM(guievt) THEN
  BEGIN CASE
    CASE guiapp='XED'
      BEGIN CASE
        CASE guifrm='FRMMAIN'
          BEGIN CASE
            CASE guictl=''
              BEGIN CASE
                CASE guievt=GECLOSE
                  GOSUB GUI.XED.FRMMAIN.CLOSE;guievt=0
                CASE guievt=GERESIZE
                  GOSUB GUI.XED.FRMMAIN.RESIZE;guievt=0
              END CASE
            CASE guictl='BTNCANCEL'
              BEGIN CASE
                CASE guievt=GECLICK
                  GOSUB GUI.XED.FRMMAIN.BTNCANCEL.CLICK;guievt=0
              END CASE
            CASE guictl='BTNEDIT'
              BEGIN CASE
                CASE guievt=GECLICK
                  GOSUB GUI.XED.FRMMAIN.BTNEDIT.CLICK;guievt=0
              END CASE
            CASE guictl='BTNEXIT'
              BEGIN CASE
                CASE guievt=GECLICK
                  GOSUB GUI.XED.FRMMAIN.BTNEXIT.CLICK;guievt=0
              END CASE
            CASE guictl='BTNSAVE'
              BEGIN CASE
                CASE guievt=GECLICK
                  GOSUB GUI.XED.FRMMAIN.BTNSAVE.CLICK;guievt=0
              END CASE
            CASE guictl='BTNUPDATE'
              BEGIN CASE
                CASE guievt=GECLICK
                  GOSUB GUI.XED.FRMMAIN.BTNUPDATE.CLICK;guievt=0
              END CASE
            CASE guictl='CMBFILE'
              BEGIN CASE
                CASE guievt=GECHANGE
                  GOSUB GUI.XED.FRMMAIN.CMBFILE.CHANGE;guievt=0
              END CASE
            CASE guictl='CMBMONTH'
              BEGIN CASE
                CASE guievt=GECHANGE
                  GOSUB GUI.XED.FRMMAIN.CMBMONTH.CHANGE;guievt=0
              END CASE
            CASE guictl='CMBYEAR'
              BEGIN CASE
                CASE guievt=GECHANGE
                  GOSUB GUI.XED.FRMMAIN.CMBYEAR.CHANGE;guievt=0
              END CASE
            CASE guictl='GRDDATA'
              BEGIN CASE
                CASE guievt=GECHANGE
                  GOSUB GUI.XED.FRMMAIN.GRDDATA.CHANGE;guievt=0
                CASE guievt=GEVALIDATECELL
                  GOSUB GUI.XED.FRMMAIN.GRDDATA.VALIDATECELL;guievt=0
                CASE guievt=GEVALIDATEROW
                  GOSUB GUI.XED.FRMMAIN.GRDDATA.VALIDATEROW;guievt=0
              END CASE
          END CASE
      END CASE
  END CASE
  IF guievt THEN
    * Unhandled event - may be dynamic
    GOSUB GUI.DYNAMIC.EVENTS
  END
END ELSE
  GOSUB GUI.CUSTOM.EVENTS
END
*
RETURN
*
*-->END EVENT DECODER<--*
*
*
************************************************************
*
*
*-->BEGIN CLOSE EVENT HANDLER<--*
GUI.XED.FRMMAIN.CLOSE: *
*
* Default form close event handler
CALL ATGUIHIDE(guiapp,guifrm,'','',guierrors,guistate)
IF guierrors<1> >= 2 THEN GOTO gui.error
CALL ATGUIGETPROP(guiapp,'','',GPSTATUS,0,0,NUM.FORMS,guierrors,guistate)
IF guierrors<1> >= 2 THEN GOTO gui.error
IF NUM.FORMS = 0 THEN
  CALL ATGUIDELETE(guiapp,'','',guierrors,guistate)
  IF guierrors<1> >= 3 THEN GOTO gui.error
END
*
RETURN
*
*-->END CLOSE EVENT HANDLER<--*
*
*
************************************************************
*
*
*-->BEGIN EVENT HANDLER<--*
GUI.XED.FRMMAIN.BTNCANCEL.CLICK: *
*
GOSUB exit_editing

gdata = ogdata
GOSUB load_grid

RETURN
*
*-->END EVENT HANDLER<--*
*
*
************************************************************
*
*
*-->BEGIN EVENT HANDLER<--*
GUI.XED.FRMMAIN.BTNEDIT.CLICK: *
*
ctrlid     = 'BTNCANCEL'  ;  property     = GPENABLED    ;   prop.value     = @TRUE
ctrlid<-1> = 'BTNEDIT'    ;  property<-1> = GPENABLED    ;   prop.value<-1> = @FALSE
ctrlid<-1> = 'BTNEXIT'    ;  property<-1> = GPENABLED    ;   prop.value<-1> = @FALSE
ctrlid<-1> = 'BTNSAVE'    ;  property<-1> = GPENABLED    ;   prop.value<-1> = @TRUE
```

```
ctrlid<-1> = 'BTNUPDATE'  ;  property<-1> = GPENABLED    ;   prop.value<-1> = @FALSE

IF lastscreen THEN
  ctrlid<-1> = 'GRDDATA'    ;  property<-1> = GPSTYLE      ;   prop.value<-1> = 1
  ctrlid<-1> = 'GRDDATA'    ;  property<-1> = GPROW        ;   prop.value<-1> = gridrows + 1
END ELSE
  ctrlid<-1> = 'GRDDATA'    ;  property<-1> = GPSTYLE      ;   prop.value<-1> = 0
  ctrlid<-1> = 'GRDDATA'    ;  property<-1> = GPROW        ;   prop.value<-1> = gridrows
END
ctrlid<-1> = 'GRDDATA'    ;  property<-1> = GPCOLUMN     ;   prop.value<-1> = 1

temp = ''
FOR ii = 1 TO gridcols
  temp<1, ii> = @FALSE
NEXT ii
ctrlid<-1> = 'GRDDATA'    ;  property<-1> = GPREADONLY   ;   prop.value<-1> = temp
ctrlid<-1> = 'FRACONTROLS' ; property<-1> = GPENABLED    ;   prop.value<-1> = @FALSE

CALL ATGUISETPROPS(guiapp, guifrm, ctrlid, property, prop.value, guierrrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

CALL ATGUIACTIVATE(guiapp, guifrm, 'GRDDATA', guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

RETURN
*
*-->END EVENT HANDLER<--*
*
*
************************************************************
*
*
*-->BEGIN EVENT HANDLER<--*
GUI.XED.FRMMAIN.BTNEXIT.CLICK: *
*
GOSUB GUI.XED.FRMMAIN.CLOSE

RETURN
*
*-->END EVENT HANDLER<--*
*
*
************************************************************
*
*
*-->BEGIN EVENT HANDLER<--*
GUI.XED.FRMMAIN.BTNSAVE.CLICK: *
*
list.locked = ''
list.changed = ''
IF gdata NE ogdata THEN
  change.firstlast = @FALSE
  dcr = DCOUNT(gdata<1>, @VM)
  FOR reccnt = 1 TO dcr
    GOSUB extract_record
    GOSUB save_record
  NEXT reccnt

  IF change.firstlast THEN
    xfiledef<4> = xfirst
    xfiledef<5> = xlast

    xid = selfile
    xrec = xfiledef
    GOSUB update_xed_files
  END

  IF list.locked OR list.changed THEN
    emsg = 'The following items could not be saved:'
    IF list.changed THEN
      CONVERT @AM TO ',' IN list.changed
      emsg<1, 1, -1> = 'Changed on disk: ':list.changed
    END
    IF list.locked THEN
      CONVERT @AM TO ',' IN list.locked
      emsg<1, 1, -1> = 'Locked by another user: ':list.locked
    END
    CALL ATGUIMSGBOX(emsg, 'Save errors', MBXICON, MBOK, '', ok, guierrors, guistate)
    IF guierrors<1> GE 2 THEN GOTO gui.error

    gridrow = dcr
    gridcol = 1
    GOSUB grid_pos

    CALL ATGUIACTIVATE(guiapp, guifrm, 'GRDDATA', guierrors, guistate)
    IF guierrors<1> GE 2 THEN GOTO gui.error
  END
END

IF NOT(list.locked OR list.changed) THEN
  GOSUB exit_editing
END

RETURN
*
*-->END EVENT HANDLER<--*
*
*
************************************************************
*
*
*-->BEGIN EVENT HANDLER<--*
GUI.XED.FRMMAIN.BTNUPDATE.CLICK: *
*
GOSUB get_download_config
GOSUB check_download_dest
IF destok THEN
  GOSUB check_download_file
  IF fileok THEN
    dldcancel = @FALSE
    IF dodownload THEN
      GOSUB check_download_prog
      GOSUB download_control
    END
    IF NOT(dldcancel) THEN
      GOSUB process_download
      GOSUB import_data
      GOSUB post_import_processing
      GOSUB reload_data
    END
  END
END
*
RETURN
*
*-->END EVENT HANDLER<--*
*
*
************************************************************
*
*
*-->BEGIN EVENT HANDLER<--*
GUI.XED.FRMMAIN.CMBFILE.CHANGE: *
*
CALL ATGUIGETPROP(guiapp, guifrm, guictl, GPVALUE, 0, 0, selfile, guierrrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

ok = @TRUE
OPEN 'DICT',selfile TO selfile.dict ELSE ok = @FALSE
IF ok THEN
  OPEN selfile TO selfile.data ELSE ok = @FALSE
END

IF ok THEN
  LOCATE selfile IN xed.files<1> BY 'AL' SETTING fpos THEN
    READ xfiledef FROM xed.var, selfile ELSE xfiledef = ''
    xkeystruct = xfiledef<2>
    xfields = xfiledef<3>
    xfirst = xfiledef<4>
    xlast = xfiledef<5>
  END

  ctrlid = 'GRDDATA'
  GOSUB clear_control
  GOSUB read_dict
  GOSUB setup_combo_boxes
END ELSE
  CALL ATGUIMSGBOX('There was a problem opening file ':selfile, 'File error', MBXICON, MBOK,
'', ok, guierrors, guistate)
  IF guierrors<1> GE 2 THEN GOTO gui.error
END

RETURN
*
```

```
*-->END EVENT HANDLER<--*
*
*
************************************************************
*
*
*-->BEGIN EVENT HANDLER<--*
GUI.XED.FRMMAIN.CMBMONTH.CHANGE: *
*
CALL ATGUIGETPROP(guiapp, guifrm, guictl, GPVALUE, 0, 0, selmonth, guierrrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

GOSUB read_data_daily
GOSUB load_grid

RETURN
*
*-->END EVENT HANDLER<--*
*
*
************************************************************
*
*
*-->BEGIN EVENT HANDLER<--*
GUI.XED.FRMMAIN.CMBYEAR.CHANGE: *
*
CALL ATGUIGETPROP(guiapp, guifrm, guictl, GPVALUE, 0, 0, selyear, guierrrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

IF xkeystruct EQ 'D' THEN
  ctrlid = 'GRDDATA'
  GOSUB clear_control
  GOSUB setup_months
END ELSE
  GOSUB read_data_months
  GOSUB load_grid
END

RETURN
*
*-->END EVENT HANDLER<--*
*
*
************************************************************
*
*
*-->BEGIN EVENT HANDLER<--*
GUI.XED.FRMMAIN.GRDDATA.CHANGE: *
*
gridcol = guiargs<1, 1>
gridrow = guiargs<1, 2>
editvalue = guiargs<2>

ctrlid     = 'GRDDATA'  ;  property     = GPCOLUMN
ctrlid<-1> = 'GRDDATA'  ;  property<-1> = GPROW
CALL ATGUIGETPROPS(guiapp, guifrm, ctrlid, property, prop.value, guierrrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error
thiscolumn = prop.value<1>
thisrow = prop.value<2>

IF (thiscolumn EQ gridcol) AND (thisrow EQ gridrow) THEN
  GOSUB validate_cell
END

RETURN
*
*-->END EVENT HANDLER<--*
*
*
************************************************************
*
*
*-->BEGIN EVENT HANDLER<--*
GUI.XED.FRMMAIN.GRDDATA.VALIDATECELL: *
*
gridcol = guiargs<1, 1>
gridrow = guiargs<1, 2>
editvalue = guiargs<2>

GOSUB validate_cell

RETURN
```

```
*
*-->END EVENT HANDLER<--*
*
*
************************************************************
*
*
*-->BEGIN EVENT HANDLER<--*
GUI.XED.FRMMAIN.GRDDATA.VALIDATEROW: *
*
IF cellvalidated THEN
  gridrow = guiargs<1, 1>
  hasdata = @FALSE
  hasid = @FALSE
  FOR ii = 1 TO gridcols
    datavalue = gdata<1, gridrow, ii>
    IF datavalue THEN
      hasdata = @TRUE
      IF ii = 1 THEN
        hasid = @TRUE
      END
    END
  NEXT ii
  ok = (hasdata EQ hasid)

  IF ok THEN
    CALL ATGUISETPROP(guiapp, guifrm, 'GRDDATA', GPVALUE, 0, 0, gdata, guierrors, guistate)
    IF guierrors<1> GE 2 THEN GOTO gui.error
  END ELSE
    CALL ATGUIMSGBOX('Row must have a valid ID', 'ID error', MBIICON, MBOK, '', ok,
guierrors, guistate)
    IF guierrors<1> GE 2 THEN GOTO gui.error

    gridcol = 1
    GOSUB grid_pos

    CALL ATGUIACTIVATE(guiapp, guifrm, 'GRDDATA', guierrors, guistate)
    IF guierrors<1> GE 2 THEN GOTO gui.error
  END
END

RETURN
*
*-->END EVENT HANDLER<--*
*
*
************************************************************
*
*
*-->BEGIN EVENT HANDLER<--*
GUI.XED.FRMMAIN.RESIZE: *
*
form_width = guiargs<1, 1>
form_height = guiargs<1, 2>
GOSUB resize_form

RETURN
*
*-->END EVENT HANDLER<--*
*
*
************************************************************
*
*
*-->BEGIN DYNAMIC EVENTS<--*
GUI.DYNAMIC.EVENTS: *
*
* Add any dynamic event handling code here. The guievt, guiapp, guifrm,
* guictl and guiargs variables are valid and availble for your use.
*
RETURN
*
*-->END DYNAMIC EVENTS<--*
*
*
************************************************************
*
**-->BEGIN CUSTOM EVENTS<--*
GUI.CUSTOM.EVENTS: *
*
* Add any custom event handling code here. The guievt, guiapp, guifrm,
* guictl and guiargs variables are valid and availble for your use.
```

```
*
RETURN
*
*-->END CUSTOM EVENTS<--*
*
*
************************************************************
*
**-->BEGIN ERROR HANDLER<--*
gui.error: *
*
CALL ATGUISHUTDOWN
PRINT 'The following errors have been reported by the GUI system:'
numerrs=DCOUNT(guierrors,CHAR(254))
FOR eacherr=2 TO numerrs
  PRINT guierrors<eacherr,6>
NEXT eacherr
*
STOP
*
*-->END ERROR HANDLER<--*
*
*
************************************************************
*
check_download_dest:
*
destok = @TRUE
IF dload.path THEN
  stat = ''
  IF dload.path[1] NE '\' THEN dload.path := '\'
  CALL FILEDIR.EXISTS('C', dload.path, stat)
  IF stat NE 'D' THEN destok = @FALSE
END ELSE
  destok = @FALSE
END

IF NOT(destok) THEN
  emsg = 'Either no download destination has been defined'
  emsg := ' or the specified destination does not exist.'
END ELSE
  IF NOT(dload.url) THEN
    emsg = 'No URL has been defined for the download.'
    destok = @FALSE
  END ELSE
    IF dload.url[1,7] NE 'http://' THEN
      emsg = 'The download URL does not appear valid.'
      destok = @FALSE
    END
  END
END

IF NOT(destok) THEN
  CALL ATGUIMSGBOX(emsg, 'Configuration error', MBXICON, MBOK, '', ok, guierrors, guistate)
  IF guierrors<1> GE 2 THEN GOTO gui.error
END

RETURN
*
* -------------------------------------------------------------------- *
*
check_download_file:
*
dfilename = FIELD(dload.url, '/', DCOUNT(dload.url, '/'))
dfilepath = dload.path:dfilename
CALL FILEDIR.EXISTS('C', dfilepath, stat)

fileok = @FALSE
dodownload = @FALSE
IF stat = 0 OR stat = 'F' THEN fileok = @TRUE
emsg = ''
IF NOT(fileok) THEN
  emsg = dfilepath:' is not a valid filename.'
  CALL ATGUIMSGBOX(emsg, 'File download', MBXICON, MBOK, '', ok, guierrors, guistate)
  IF guierrors<1> GE 2 THEN GOTO gui.error
END ELSE
  IF stat = 'F' THEN
    emsg = 'File ':dfilepath:' already exists. Use this file (Yes) or download new file
(No)?'
    CALL ATGUIMSGBOX(emsg, 'File download', MBQICON, MBYESNOCANCEL, '', ok, guierrors,
guistate)
    IF guierrors<1> GE 2 THEN GOTO gui.error
    BEGIN CASE
      CASE ok EQ MBANSCANCEL ; fileok = @FALSE
      CASE ok EQ MBANSYES    ; dodownload = @FALSE
      CASE ok EQ MBANSNO     ; dodownload = @TRUE
    END CASE
  END ELSE
    dodownload = @TRUE
  END
END
*
RETURN
*
* -------------------------------------------------------------------- *
*
check_download_prog:
*
dprogok = @FALSE
CALL FILEDIR.EXISTS('C', dprog.path, stat)
IF stat EQ 'F' THEN dprogok = @TRUE
*
RETURN
*
* -------------------------------------------------------------------- *
*
clear_control:
*
CALL ATGUICLEAR(guiapp, guifrm, ctrlid, guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error
*
IF ctrlid = 'GRDDATA' THEN
  GOSUB disable_buttons
END
*
RETURN
*
* -------------------------------------------------------------------- *
*
debug_msg:
*
CALL ATGUIMSGBOX(debug.txt, 'Debug', MBIICON, MBOK, '', debugok, guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

RETURN
*
* -------------------------------------------------------------------- *
*
disable_buttons:
*
ctrlid     = 'BTNCANCEL'  ;  property     = GPENABLED    ;   prop.value     = @FALSE
ctrlid<-1> = 'BTNEDIT'    ;  property<-1> = GPENABLED    ;   prop.value<-1> = @FALSE
ctrlid<-1> = 'BTNSAVE'    ;  property<-1> = GPENABLED    ;   prop.value<-1> = @FALSE
ctrlid<-1> = 'BTNUPDATE'  ;  property<-1> = GPENABLED    ;   prop.value<-1> = @FALSE

CALL ATGUISETPROPS(guiapp, guifrm, ctrlid, property, prop.value, guierrrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error
*
RETURN
*
* -------------------------------------------------------------------- *
*
disable_editing:
*
temp = ''
FOR ii = 1 TO gridcols
  temp<1, ii> = @TRUE
NEXT ii

CALL ATGUISETPROP(guiapp, guifrm, 'GRDDATA', GPREADONLY, 0, 0, temp, guierrrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error
*
RETURN
*
* -------------------------------------------------------------------- *
*
download_control:
*
filedownloaded = @FALSE
LOOP
  IF dprogok THEN
    GOSUB download_using_prog
  END ELSE
    GOSUB download_using_browser
  END
  IF dldcancel THEN EXIT

  CALL FILEDIR.EXISTS('C', dfilepath, stat)
  IF stat = 'F' THEN
    filedownloaded = @TRUE
  END ELSE
    emsg = 'File has not been downloaded. Click Retry to run the download again '
    emsg := 'or Cancel to quit.'
    CALL ATGUIMSGBOX(emsg, 'File download', MBEXICON, MBRETRYCANCEL, '', ok, guierrors,
guistate)
    IF guierrors<1> GE 2 THEN GOTO gui.error
    IF ok EQ MBANSCANCEL THEN EXIT
  END
UNTIL filedownloaded DO REPEAT
*
RETURN
*
* -------------------------------------------------------------------- *
*
download_using_browser:
*
emsg = 'This computer does not have the download program installed. Click OK to '
emsg := 'download using your browser, or Cancel to abandon the update.'

CALL ATGUIMSGBOX(emsg, 'File download', MBEXICON, MBOKCANCEL, '', ok, guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

IF ok EQ MBANSOK THEN
  cmd = 'START ':dload.url
  CRT ESC:STX:'<':cmd:CR:

  emsg = 'Click OK when the download is complete, or Cancel to abandon the update.'
  CALL ATGUIMSGBOX(emsg, 'File download', MBIICON, MBOKCANCEL, '', ok, guierrors, guistate)
  IF guierrors<1> GE 2 THEN GOTO gui.error

  IF ok EQ MBANSCANCEL THEN dldcancel = @TRUE
END ELSE
  dldcancel = @TRUE
END
*
RETURN
*
* -------------------------------------------------------------------- *
*
download_using_prog:
*
cmd = dprog.path:' ':dprog.syntax
cmd = CHANGE(cmd, '%%DEST%%', dfilepath)
cmd = CHANGE(cmd, '%%URL%%', dload.url)
CRT ESC:STX:'<':cmd:CR:

emsg = 'Click OK when the download is complete, or Cancel to abandon the update.'
CALL ATGUIMSGBOX(emsg, 'File download', MBIICON, MBOKCANCEL, '', ok, guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

IF ok EQ MBANSCANCEL THEN dldcancel = @TRUE
*
RETURN
*
* -------------------------------------------------------------------- *
*
exit_editing:
*
ctrlid      = 'BTNCANCEL'  ;  property     = GPENABLED    ;   prop.value     = @FALSE
ctrlid<-1> = 'BTNEDIT'    ;  property<-1> = GPENABLED    ;   prop.value<-1> = @TRUE
ctrlid<-1> = 'BTNEXIT'    ;  property<-1> = GPENABLED    ;   prop.value<-1> = @TRUE
ctrlid<-1> = 'BTNSAVE'    ;  property<-1> = GPENABLED    ;   prop.value<-1> = @FALSE
ctrlid<-1> = 'BTNUPDATE'  ;  property<-1> = GPENABLED    ;   prop.value<-1> = @TRUE
ctrlid<-1> = 'GRDDATA'    ;  property<-1> = GPSTYLE      ;   prop.value<-1> = 0
ctrlid<-1> = 'FRACONTROLS' ; property<-1> = GPENABLED    ;   prop.value<-1> = @TRUE

CALL ATGUISETPROPS(guiapp, guifrm, ctrlid, property, prop.value, guierrrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

GOSUB disable_editing

RETURN
*
* -------------------------------------------------------------------- *
*
extract_record:
```

```
*
orowdata = ''
rowdata = ''
dataid = gdata<1, reccnt, 1>
testid = ''
ii = 0
LOOP
  ii += 1
  testid = ogdata<1, ii, 1>
  IF testid EQ dataid THEN EXIT
  IF ii GE dcr THEN EXIT
REPEAT

FOR colno = 2 TO gridcols
  IF testid THEN
    orowdata<colno> = ICONV(ogdata<1, reccnt, colno>, xdconvs<1, colno>)
  END
  rowdata<colno> = ICONV(gdata<1, reccnt, colno>, xdconvs<1, colno>)
NEXT colno
*
dataid = ICONV(dataid, xdconvs<1, 1>)

RETURN
*
* -------------------------------------------------------------------- *
*
get_download_config:
*
id = 'DEST*':clientname
dload.path = ''
READV dload.path FROM xed.var, id, 1 ELSE NULL
IF NOT(dload.path) THEN
  READV dload.path FROM xed.var, 'DEST', 1 ELSE NULL
END

dprog = xfiledef<6>
dprog.path = ''
dprog.syntax = ''
READ temp FROM xed.var, dprog THEN
  dprog.path = temp<1>
  dprog.syntax = temp<2>
END

id = dprog:'*':clientname
READV temp FROM xed.var, id, 1 ELSE temp = ''
IF temp THEN
  dprog.path = temp
END

dload.url = xfiledef<7>
*
RETURN
*
* -------------------------------------------------------------------- *
*
get_first_last:
*
SELECT selfile.data
LOOP
  READNEXT data.id ELSE EXIT
  BEGIN CASE
    CASE xfirst = ''
      xfirst = data.id
      xlast = data.id
    CASE data.id LT xfirst
      xfirst = data.id
    CASE data.id GT xlast
      xlast = data.id
  END CASE
REPEAT

xfiledef<4> = xfirst
xfiledef<5> = xlast

xid = selfile
xrec = xfiledef
GOSUB update_xed_files
*
RETURN
*
* -------------------------------------------------------------------- *
*
```

```
grid_pos:
*
ctrlid     = 'GRDDATA'  ;  property     = GPCOLUMN ;   prop.value     = gridcol
ctrlid<-1> = 'GRDDATA'  ;  property<-1> = GPROW    ;   prop.value<-1> = gridrow

CALL ATGUISETPROPS('XED', 'FRMMAIN', ctrlid, property, prop.value, guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error
*
RETURN
*
* -------------------------------------------------------------------- *
*
import_data:
*
mode = 'K,N,C,H'
pcfile = cfilepath
hostfile = selfile
attrs = ''
hdrskip = 1
idprefix = ''
idstart = ''
itemcnt = 0
bytecnt = 0
stat = ''

CALL FTIMPORT(mode, pcfile, hostfile, attrs, hdrskip, idprefix, idstart, itemcnt, bytecnt,
stat)
GOSUB get_first_last

IF stat THEN
  emsg = 'Error encountered during update: ':stat
  mbicon = MBXICON
END ELSE
  emsg = 'File ':selfile:' successfully updated'
  mbicon = MBIICON
END

CALL ATGUIMSGBOX(emsg, 'File update', mbicon, MBOK, '', ok, guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error
*
RETURN
*
* -------------------------------------------------------------------- *
*
initialise:
*
* Get design-time form width and height
*
guiapp = 'XED'
guifrm = 'FRMMAIN'

CALL ATGUIGETPROP(guiapp, guifrm, '', GPWIDTH, 0, 0, form_width_min, guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

CALL ATGUIGETPROP(guiapp, guifrm, '', GPHEIGHT, 0, 0, form_height_min, guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

form_height = form_height_min
form_width = form_width_min
do_resize = @FALSE
*
* Read control item and populate files combo box
*
READ xed.files FROM xed.var, 'XED.FILES' ELSE xed.files = ''
GOSUB sort_files

CALL ATGUISETPROP(guiapp, guifrm, 'CMBFILE', GPITEMS, 0, 0, xed.files, guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error
*
GOSUB disable_buttons
*
RETURN
*
* -------------------------------------------------------------------- *
*
load_grid:
*
CALL ATGUISETPROP(guiapp, guifrm, 'GRDDATA', GPVALUE, 0, 0, gdata, guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

ogdata = gdata
```

```
GOSUB disable_buttons
ctrlid     = 'BTNEDIT'    ;  property     = GPENABLED    ;   prop.value     = 1
ctrlid<-1> = 'BTNUPDATE'  ;  property<-1> = GPENABLED    ;   prop.value<-1> = 1

CALL ATGUISETPROPS(guiapp, guifrm, ctrlid, property, prop.value, guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

GOSUB disable_editing
*
RETURN
*
* --------------------------------------------------------------------- *
*
load_row:
*
rowno += 1
FOR colno = 1 TO gridcols
  amc = xdamcs<1, colno>
  IF amc EQ 0 THEN
    gdata<1, rowno, colno> = OCONV(dataid, xdconvs<1, colno>)
  END ELSE
    gdata<1, rowno, colno> = OCONV(datarec<amc>, xdconvs<1, colno>)
  END
NEXT colno
*
RETURN
*
* --------------------------------------------------------------------- *
*
post_import_processing:
*
pisub = xfiledef<9>
IF pisub THEN
  IF CATALOGUED(pisub) THEN
    CALL @pisub
  END
END
*
RETURN
*
* --------------------------------------------------------------------- *
*
process_download:
*
xscript = xfiledef<8>
IF xscript THEN
  READ script FROM scripts, xscript THEN
    cfilepath = dfilepath
    dfilepath = CHANGE(dfilepath, '\', '/')
    script = CHANGE(script, '%%SOURCE%%', dfilepath)
    cfilepath = CHANGE(cfilepath, '.xls', '.csv')
    script = CHANGE(script, '%%DEST%%', cfilepath)
    lfilepath = CHANGE(cfilepath, '.csv', '.lck')
    script = CHANGE(script, '%%LOCK%%', lfilepath)
    CALL RUN.SCRIPT.SUB(script, err)

    LOOP
      CALL FILEDIR.EXISTS('C', lfilepath, stat)
    UNTIL NOT(stat) DO
      SLEEP 1
    REPEAT
  END
END
*
RETURN
*
* --------------------------------------------------------------------- *
*
read_data_daily:
*
gdata = ''
dataid = ICONV('01 ':selmonth:' ':selyear, 'D')
monthno = OCONV(dataid, 'DM')
rowno = 0
lastscreen = @FALSE
gridcols = DCOUNT(xfields<1>, @VM)
LOOP
  READ datarec FROM selfile.data, dataid THEN
    GOSUB load_row
  END
  IF dataid GE xlast THEN lastscreen = @TRUE
  dataid += 1
  chkmonth = OCONV(dataid, 'DM')
  IF chkmonth NE monthno THEN EXIT
REPEAT
gridrows = rowno
*
RETURN
*
* -------------------------------------------------------------------- *
*
read_data_months:
*
gdata = ''
rowno = 0
lastscreen = @FALSE
gridcols = DCOUNT(xfields<1>, @VM)
FOR mth = 1 TO 12
  dataid = selyear * 100 + mth
  READ datarec FROM selfile.data, dataid THEN
    GOSUB load_row
  END
  IF dataid GE xlast THEN lastscreen = @TRUE
NEXT mth
gridrows = rowno
*
RETURN
*
* -------------------------------------------------------------------- *
*
read_dict:
*
xdamcs = ''
xdnames = ''
xdconvs = ''
xdfmts = ''
xdtype = ''

colwidths = ''
colaligns = ''
colsizeables = ''

CONVERT ' ' TO @VM IN xfields
dc = DCOUNT(xfields<1>, @VM)
FOR ii = 1 TO dc
  xdid = xfields<1, ii>
  READ xdrec FROM selfile.dict, xdid ELSE xdrec = ''

  xdamcs<1, ii> = xdrec<2>
  xdconv = xdrec<3>
  xdconvs<1, ii> = xdconv
  xdname = xdrec<4>
  CONVERT @VM TO ' ' IN xdname
  xdnames<1, ii> = TRIM(xdname)
  colfmt = xdrec<5>
  xdfmts<1, ii> = colfmt

  colwidth = OCONV(colfmt, 'MCN')
  temp = LEN(xdname)
  IF temp GT colwidth THEN colwidth = temp
  colwidths<1, ii> = colwidth

  colalign = OCONV(colfmt, 'MCA')
  CONVERT 'LTURC' TO '00012' IN colalign
  colaligns<1, ii> = colalign
  colsizeables<1, ii> = 1

  dp = OCONV(xdconv, 'MCN')
  IF dp GT 9 THEN dp = dp[1,1]
  BEGIN CASE
    CASE dp EQ 2   ;  datatype = GDCURRENCY
    CASE dp AND dp LE 4   ;  datatype = GDFINANCIAL
    CASE dp        ; datatype = GDNUMERIC
    CASE xdconv EQ 'D'  ; datatype = GDDATE
    CASE 1         ;  datatype = GDANY
  END CASE
  xdtype<1, ii> = datatype
NEXT ii

form_width = SUM(RAISE(colwidths)) + 3
do_resize = @TRUE
GOSUB resize_form
```

```
ctrlid     = 'GRDDATA'  ;  property     = GPCOLUMNS     ;   prop.value     = dc
ctrlid<-1> = 'GRDDATA'  ;  property<-1> = GPFIXEDCOLS   ;   prop.value<-1> = 1
ctrlid<-1> = 'GRDDATA'  ;  property<-1> = GPCOLWIDTH    ;   prop.value<-1> = colwidths
ctrlid<-1> = 'GRDDATA'  ;  property<-1> = GPCOLALIGN    ;   prop.value<-1> = colaligns
ctrlid<-1> = 'GRDDATA'  ;  property<-1> = GPCOLHEADING  ;   prop.value<-1> = xdnames
ctrlid<-1> = 'GRDDATA'  ;  property<-1> = GPCOLSIZABLE  ;   prop.value<-1> = colsizeables
ctrlid<-1> = 'GRDDATA'  ;  property<-1> = GPCOLDATATYPE ;   prop.value<-1> = xdtype

CALL ATGUISETPROPS(guiapp, guifrm, ctrlid, property, prop.value, guierrrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error
*
RETURN
*
* -------------------------------------------------------------------- *
*
reload_data:
*
GOSUB setup_combo_boxes
IF xkeystruct EQ 'D' THEN
  GOSUB setup_months
  GOSUB read_data_daily
END ELSE
  GOSUB read_data_months
END

ctrlid     = 'CMBYEAR'  ;  property     = GPVALUE    ;   prop.value     = selyear
IF xkeystruct = 'D' THEN
  ctrlid<-1> = 'CMBMONTH' ;  property<-1> = GPVALUE  ;   prop.value<-1> = selmonth
END

CALL ATGUISETPROPS(guiapp, guifrm, ctrlid, property, prop.value, guierrrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error

GOSUB load_grid
*
RETURN
*
* -------------------------------------------------------------------- *
*
resize_form:
*
IF (form_width LT form_width_min) OR (form_height LT form_height_min) OR do_resize THEN
  IF form_width LT form_width_min THEN form_width = form_width_min
  IF form_height LT form_height_min THEN form_height = form_height_min
  CALL ATGUIMOVE(guiapp, guifrm, '', '', '', form_width, form_height, guierrors, guistate)
  IF guierrors<1> GE 2 THEN GOTO gui.error
  do_resize = @FALSE
END

ctrlid = 'FRACONTROLS'  ;  property = GPWIDTH       ;   prop.value = form_width
ctrlid<-1> = 'FRABTNS'  ;  property<-1> = GPWIDTH   ;   prop.value<-1> = form_width
ctrlid<-1> = 'FRABTNS'  ;  property<-1> = GPTOP     ;   prop.value<-1> = form_height - 2.5
ctrlid<-1> = 'BTNEXIT'  ;  property<-1> = GPLEFT    ;   prop.value<-1> = form_width - 9
ctrlid<-1> = 'BTNSAVE'  ;  property<-1> = GPLEFT    ;   prop.value<-1> = form_width - 17
ctrlid<-1> = 'GRDDATA'  ;  property<-1> = GPWIDTH   ;   prop.value<-1> = form_width
ctrlid<-1> = 'GRDDATA'  ;  property<-1> = GPHEIGHT  ;   prop.value<-1> = form_height - 5

CALL ATGUISETPROPS(guiapp, guifrm, ctrlid, property, prop.value, guierrrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error
*
RETURN
*
* -------------------------------------------------------------------- *
*
save_record:
*
tries = 0
datarec = ''
LOOP
  tries += 1
  READU datarec FROM selfile.data, dataid LOCKED
    NAP 10
    readok = @FALSE
  END THEN
    readok = @TRUE
  END ELSE
    readok = @TRUE
  END
UNTIL readok OR (tries GE 3) DO REPEAT

IF readok THEN
  dowrite = @FALSE
  FOR colno = 2 TO gridcols
    amc = xdamcs<1, colno>
    IF orowdata<colno> NE datarec<amc> THEN
      dowrite = @FALSE
      list.changed<-1> = dataid
      EXIT
    END
    IF rowdata<colno> NE datarec<amc> THEN
      datarec<amc> = rowdata<colno>
      dowrite = @TRUE
    END
  NEXT colno
  IF dowrite THEN
    WRITE datarec ON selfile.data, dataid
    IF dataid LT xfirst THEN
      xfirst = dataid
      change.firstlast = @TRUE
    END
    IF dataid GT xlast THEN
      xlast = dataid
      change.firstlast = @TRUE
    END
  END ELSE
    RELEASE selfile.data, dataid
  END
END ELSE
  list.locked<-1> = dataid
END

RETURN
*
* -------------------------------------------------------------------- *
*
setup_combo_boxes:
*
IF xfirst = '' THEN
  GOSUB get_first_last
END

IF xkeystruct = 'D' THEN
  firstyear = OCONV(xfirst, 'DY')
  lastyear = OCONV(xlast, 'DY')
  enabled = @TRUE
END ELSE
  firstyear = xfirst[1, 4]
  lastyear = xlast[1, 4]
  enabled = @FALSE
END

years = ''
FOR thisyear = firstyear TO lastyear
  years<1, -1> = thisyear
NEXT thisyear

ctrlid = 'CMBYEAR'
GOSUB clear_control

ctrlid = 'CMBMONTH'
GOSUB clear_control

ctrlid     = 'CMBYEAR'  ;  property     = GPITEMS    ;   prop.value     = years
ctrlid<-1> = 'LBLMONTH' ;  property<-1> = GPENABLED  ;   prop.value<-1> = enabled
ctrlid<-1> = 'CMBMONTH' ;  property<-1> = GPENABLED  ;   prop.value<-1> = enabled

CALL ATGUISETPROPS(guiapp, guifrm, ctrlid, property, prop.value, guierrors, guistate)
IF guierrors<1> GE 2 THEN GOTO gui.error
*
RETURN
*
* -------------------------------------------------------------------- *
*
setup_months:
*
start_yyyymm = OCONV(xfirst, 'DY') * 100 + OCONV(xfirst, 'DM')
last_yyyymm = OCONV(xlast, 'DY') * 100 + OCONV(xlast, 'DM')

months = ''
FOR mth = 1 TO 12
  this_yyyymm = selyear * 100 + mth
  IF (this_yyyymm GE start_yyyymm) AND (this_yyyymm LE last_yyyymm) THEN
    months<1, -1> = OCONV(ICONV('15/':mth:'/':selyear, 'D'), 'DMAL')
  END
```

```
NEXT mth

ctrlid = 'CMBMONTH'
GOSUB clear_control

IF months THEN
  CALL ATGUISETPROP(guiapp, guifrm, 'CMBMONTH', GPITEMS, 0, 0, months, guierrrors, guistate)
  IF guierrors<1> GE 2 THEN GOTO gui.error
END
*
RETURN
*
* ------------------------------------------------------------------- *
*
sort_files:
*
ofiles = xed.files

xed.files = ''
xed.filedescs = ''
dc = DCOUNT(ofiles<1>, @VM)
FOR ii = 1 TO dc
  thisfile = ofiles<1, ii>
  LOCATE thisfile IN xed.files<1> BY 'AL' SETTING fpos ELSE
    READV filedesc FROM xed.var, thisfile, 1 THEN
      INS thisfile BEFORE xed.files<1, fpos>
      INS filedesc BEFORE xed.filedescs<1, fpos>
    END
  END
NEXT ii

IF xed.files NE ofiles THEN
  xid = 'XED.FILES'
  xrec = xed.files
  GOSUB update_xed_files
  ofiles = xed.files
END
*
RETURN
*
* ------------------------------------------------------------------- *
*
update_xed_files:
*
writeok = @TRUE
READU dummy FROM xed.var, xid LOCKED
  writeok = @FALSE
END ELSE
  NULL
END

IF writeok THEN
  WRITE xrec ON xed.var, xid
END ELSE
  RELEASE xed.var, xid
END
*
RETURN
*
* ------------------------------------------------------------------- *
*
validate_cell:
*
cellvalidated = @TRUE
xconv = xdconvs<1, gridcol>
fmtvalue = ICONV(editvalue, xconv)
fmtvalue = OCONV(fmtvalue, xconv)

IF fmtvalue THEN
  IF gridcol = 1 THEN
    iderr = @FALSE
    IF xkeystruct = 'D' THEN
      NULL
    END ELSE
      fmtvalue = OCONV(fmtvalue, 'MCN')
      yyyy = INT(fmtvalue / 100)
      mm = MOD(fmtvalue, 100)
      IF mm LT 1 OR mm GT 12 THEN iderr = @TRUE
      IF LEN(yyyy) NE 4 THEN iderr = @TRUE
      emsg = editvalue:' is not a valid ID'
    END
    IF NOT(iderr) THEN
      dcr = DCOUNT(gdata<1>, @VM)
      idcnt = 0
      oidcnt = 0
      FOR ii = 1 TO dcr
        IF ii EQ gridrow THEN
          idcnt += 1
        END ELSE
          IF gdata<1, ii, 1> = fmtvalue THEN idcnt += 1
        END
        IF ogdata<1, ii, 1> = fmtvalue THEN oidcnt += 1
      NEXT ii
      IF gridrow GT dcr THEN idcnt += 1

      IF idcnt GT 1 THEN
        iderr = @TRUE
        emsg = editvalue:' is already in use as an item-id'
      END

      IF idcnt EQ 1 AND oidcnt EQ 0 THEN
        temp = fmtvalue
        temp = ICONV(temp, xconv)
        READ dummy FROM selfile.data, temp THEN
          found = @TRUE
        END ELSE
          found = @FALSE
        END
        IF found THEN
          iderr = @TRUE
          emsg = editvalue:' already exists on disk'
        END
      END
    END

    IF iderr THEN
      CALL ATGUIMSGBOX(emsg, 'ID error', MBIICON, MBOK, '', ok, guierrors, guistate)
      IF guierrors<1> GE 2 THEN GOTO gui.error

      GOSUB grid_pos
      cellvalidated = @FALSE
      CALL ATGUIACTIVATE(guiapp, guifrm, 'GRDDATA', guierrors, guistate)
      IF guierrors<1> GE 2 THEN GOTO gui.error
    END
  END
END

gdata<1, gridrow, gridcol> = fmtvalue
*
RETURN
*
* -------------------------------------------------------------------- *
*
END
```

12.2 XED.VAR Control File

SORT DICT XED.VAR

```
@ID.........     TYPE    LOC...........    CONV..    NAME........    FORMAT    S/M
ASSOC...
@ID              D       0                           XED.VAR         10L       S
DESC             D       1                           Description     15L       S
KEYSTRUCT        D       2                           Key Struct      7L        S
FIELDS           D       3                           Fields          25L       S
FIRST            D       4                           First ID        7L        S
LAST             D       5                           Last ID         7L        S
PROGRAM          D       6                           Program         7L        S
URL              D       7                           URL             60L       S
SCRIPT           D       8                           Script          10L       S
PISUB            D       9                           Post Import     20L       S
                                                     Sub
```

CT XED.VAR *

```
XED.VAR CURL
1: C:\Program Files\curl\curl.exe
2: -o %%DEST%% %%URL%%

XED.VAR CURL*V2000
1: C:\curl\curl.exe

XED.VAR DEST
1: C:\Temp

XED.VAR DEST*V2000
1: C:\Temp

XED.VAR FX.DAILY
1: Foreign Exchange (daily)
2: D
3: DATE ALL USD GBP AUD JPY EUR OTHER
4: 13332
5: 15279
6: CURL
7: http://www.rbnz.govt.nz/statistics/exandint/b4/hb4.xls
8: CSV.HB4
9: XED.PISUB.FX.DAILY

XED.VAR FX.MONTHLY
1: Foreign Exchange (monthly)
2: YYYYMM
3: @ID ALL USD GBP AUD JPY EUR OTHER
4: 200407
5: 200910

XED.VAR IRATES
1: Interest rates (monthly avg)
2: YYYYMM
3: @ID INTERBANK 30DAY 60DAY 90DAY 1YR 5YR 10YR
4: 198501
5: 200909
6: CURL
7: http://www.rbnz.govt.nz/statistics/exandint/b2/hb2.xls
8: CSV.HB2

XED.VAR XED.FILES
1: FX.DAILY²FX.MONTHLY²IRATES²XRATES

XED.VAR XRATES
1: Exchange rates (month avg)
2: YYYYMM
3: @ID TWI USD GBP AUD JPY EUR GDM
4: 198501
5: 200909
6: CURL
7: http://www.rbnz.govt.nz/statistics/exandint/b1/hb1.xls
8: CSV.HB1
```

12.3 Scripts

12.3.1 CSV.HB1

```
' Script to take the hb1.xls spreadsheet downloaded from RBNZ and save it
' as a CSV file ready for importation into OpenQM

  Dim oSM as Object
  Dim oDesk as Object
  Dim oDoc as Object
  Dim oSheets as Object
  Dim oSheet as Object
  Dim oCell as Object
  Dim args()
```

Appendix 1: XED

```
Dim URL as string
Dim iCnt as long
Dim iCol as long
Dim iDate as long
Dim nTWI as single
Dim OutLine as string

const DQ = Chr$(34)
const Comma = Chr$(44)

Open "%%LOCK%%" For Output as #2
Print #2, "Locked"
Close #2

URL = "file:///%%SOURCE%%"

Set oSM = CreateObject("com.sun.star.ServiceManager")
Set oDesk = oSM.createInstance("com.sun.star.frame.Desktop")
Set oDoc = oDesk.loadComponentFromURL(URL, "_blank", 0, args)

Set oSheets = oDoc.Sheets()
Set oSheet = oSheets.getByName("monthly")

oSheet.Rows.removeByIndex(0, 2)
oSheet.Rows.removeByIndex(1, 2)
oSheet.Columns.insertByIndex(1, 2)

iCnt = 0
Do
  iCnt = iCnt + 1
  Set oCell = oSheet.getCellByPosition(0, iCnt)
  iDate = oCell.value
  If iDate > 0 then
    Set oCell = oSheet.getCellByPosition(1, iCnt)
    Set oCell.value = Year(iDate) * 100 + Month(iDate)
    Set oCell = oSheet.getCellByPosition(8, iCnt)
    nTWI = oCell.value
    Set oCell = oSheet.getCellByPosition(2, iCnt)
    Set oCell.value = nTWI
  else
    Exit Do
  end if
Loop

oSheet.Columns.removeByIndex(0, 1)
oSheet.Columns.removeByIndex(7, 5)

For iCol = 0 to 6
  Set oCell = oSheet.getCellByPosition(iCol,0)
  Select Case iCol
    Case 0
      Set oCell.string = "@ID"
    Case 1
      Set oCell.string = "TWI"
    Case 2
      Set oCell.string = "USD"
    Case 3
      Set oCell.string = "GBP"
    Case 4
      Set oCell.string = "AUD"
    Case 5
      Set oCell.string = "JPY"
    Case 6
      Set oCell.string = "EUR"
  End Select
Next iCol

Open "%%DEST%%" For Output as #1

iCnt = 0
Do
  OutLine = ""
  For iCol = 0 to 6
    Set oCell = oSheet.getCellByPosition(iCol, iCnt)
    If iCnt > 0 then
      OutLine = OutLine & oCell.Value & Comma
    else
      OutLine = OutLine & DQ & Trim(oCell.String) & DQ & Comma
    end if
  Next iCol
  Print #1, OutLine
  iCnt = iCnt + 1
```

```
    Set oCell = oSheet.getCellByPosition(0, iCnt)
    iDate = oCell.value
    If not(iDate > 0) then
      Exit Do
    end if
  Loop
  Close #1
  Kill "%%LOCK%%"

  oDoc.Close(True)
  Set oDoc = Nothing
End Sub

Sub Dummy
```

12.3.2 CSV.HB2

```
' Script to take the hb2.xls spreadsheet downloaded from RBNZ and save it
' as a CSV file ready for importation into OpenQM

  Dim oSM as Object
  Dim oDesk as Object
  Dim oDoc as Object
  Dim oSheets as Object
  Dim oSheet as Object
  Dim oCell as Object
  Dim args()
  Dim URL as string
  Dim iCnt as long
  Dim iCol as long
  Dim iDate as long
  Dim OutLine as string

  const DQ = Chr$(34)
  const Comma = Chr$(44)

  Open "%%LOCK%%" For Output as #2
  Print #2, "Locked"
  Close #2

  URL = "file:///%%SOURCE%%"

  Set oSM = CreateObject("com.sun.star.ServiceManager")
  Set oDesk = oSM.createInstance("com.sun.star.frame.Desktop")
  Set oDoc = oDesk.loadComponentFromURL(URL, "_blank", 0, args)

  Set oSheets = oDoc.Sheets()
  Set oSheet = oSheets.getByName("monthly")

  oSheet.Rows.removeByIndex(0, 3)
  oSheet.Rows.removeByIndex(1, 1)
  oSheet.Columns.insertByIndex(1, 1)

  iCnt = 0
  Do
    iCnt = iCnt + 1
    Set oCell = oSheet.getCellByPosition(0, iCnt)
    iDate = oCell.value
    If iDate > 0 then
      Set oCell = oSheet.getCellByPosition(1, iCnt)
      Set oCell.value = Year(iDate) * 100 + Month(iDate)
    else
      Exit Do
    end if
  Loop

  oSheet.Columns.removeByIndex(0, 1)
  oSheet.Columns.removeByIndex(9, 2)

  For iCol = 0 to 8
    Set oCell = oSheet.getCellByPosition(iCol,0)
    Select Case iCol
      Case 0
        Set oCell.string = "@ID"
      Case 1
        Set oCell.string = "INTERBANK"
      Case 2
        Set oCell.string = "30DAY"
      Case 3
```

```
        Set oCell.string = "60DAY"
      Case 4
        Set oCell.string = "90DAY"
      Case 5
        Set oCell.string = "1YR"
      Case 6
        Set oCell.string = "2YR"
      Case 7
        Set oCell.string = "5YR"
      Case 8
        Set oCell.string = "10YR"
    End Select
  Next iCol

  Open "%%DEST%%" For Output as #1
  iCnt = 0
  Do
    OutLine = ""
    For iCol = 0 to 8
      Set oCell = oSheet.getCellByPosition(iCol, iCnt)
      If iCnt > 0 then
        OutLine = OutLine & oCell.Value & Comma
      else
        OutLine = OutLine & DQ & Trim(oCell.String) & DQ & Comma
      end if
    Next iCol
    Print #1, OutLine
    iCnt = iCnt + 1
    Set oCell = oSheet.getCellByPosition(0, iCnt)
    iDate = oCell.value
    If not(iDate > 0) then
      Exit Do
    end if
  Loop
  Close #1
  Kill "%%LOCK%%"

  oDoc.Close(True)
  Set oDoc = Nothing
End Sub

Sub Dummy
```

12.3.3 CSV.HB4

```
' Script to take the hb1.xls spreadsheet downloaded from RBNZ and save it
' as a CSV file ready for importation into OpenQM

  Dim oSM as Object
  Dim oDesk as Object
  Dim oDoc as Object
  Dim oSheets as Object
  Dim oSheet as Object
  Dim oCell as Object
  Dim args()
  Dim URL as string
  Dim iCnt as long
  Dim iCol as long
  Dim iDate as long
  Dim nALL as single
  Dim OutLine as string
  DIM sDate as string

  const DQ = Chr$(34)
  const Comma = Chr$(44)

  Open "%%LOCK%%" For Output as #2
  Print #2, "Locked"
  Close #2

  URL = "file:///%%SOURCE%%"

  Set oSM = CreateObject("com.sun.star.ServiceManager")
  Set oDesk = oSM.createInstance("com.sun.star.frame.Desktop")
  Set oDoc = oDesk.loadComponentFromURL(URL, "_blank", 0, args)

  Set oSheets = oDoc.Sheets()
  Set oSheet = oSheets.getByName("spot")
  oSheet.Rows.removeByIndex(0, 3)
  oSheet.Rows.removeByIndex(1, 1)
  oSheet.Columns.insertByIndex(1, 2)

  iCnt = 0
  Do
    iCnt = iCnt + 1
    Set oCell = oSheet.getCellByPosition(0, iCnt)
    iDate = oCell.value
    If iDate > 0 then
      Set oCell = oSheet.getCellByPosition(1, iCnt)
      Set oCell.string = Trim(Str(Day(iDate))) & "/" & Trim(Str(Month(iDate))) & "/" &
Trim(Str(Year(iDate)))
      Set oCell = oSheet.getCellByPosition(9, iCnt)
      nALL = oCell.value
      Set oCell = oSheet.getCellByPosition(2, iCnt)
      Set oCell.value = nALL
    else
      Exit Do
    end if
  Loop

  oSheet.Columns.removeByIndex(0, 1)
  oSheet.Columns.removeByIndex(8, 15)

  For iCol = 0 to 7
    Set oCell = oSheet.getCellByPosition(iCol,0)
    Select Case iCol
      Case 0
        Set oCell.string = "DATE"
      Case 1
        Set oCell.string = "ALL"
      Case 2
        Set oCell.string = "USD"
      Case 3
        Set oCell.string = "GBP"
      Case 4
        Set oCell.string = "AUD"
      Case 5
        Set oCell.string = "JPY"
      Case 6
        Set oCell.string = "EUR"
      Case 7
        Set oCell.string = "OTHER"
    End Select
  Next iCol

  Open "%%DEST%%" For Output as #1
  iCnt = 0
  Do
    OutLine = ""
    For iCol = 0 to 7
      Set oCell = oSheet.getCellByPosition(iCol, iCnt)
      If iCnt > 0 then
        If iCol > 0 then
          OutLine = OutLine & oCell.Value & Comma
        else
          OutLine = DQ & Trim(oCell.String) & DQ & Comma
        end if
      else
        OutLine = OutLine & DQ & Trim(oCell.String) & DQ & Comma
      end if
    Next iCol
    Print #1, OutLine
    iCnt = iCnt + 1
    Set oCell = oSheet.getCellByPosition(0, iCnt)
    sDate = oCell.string
    If not(sDate > "") then
      Exit Do
    end if
  Loop
  Close #1
  Kill "%%LOCK%%"

  oDoc.Close(True)
  Set oDoc = Nothing
End Sub

Sub Dummy
```

Index

Index

Index