OpenQM

2.12-4

Tutorial Guide

Tutorial Guide

Special thanks to:

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

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

Table of Contents

1   Introduction

Welcome to the *QM Tutorial Guide*. This document takes you step by step through construction of a simple application to show many of the features of the QM Multivalue Database. Clearly, a brief tutorial cannot show everything. There is much more for you to discover for yourself once you have mastered the basic principles.

In this tutorial we will create a simple library database, construct reports to view the content of the files and develop a small application program to handle loans and returns.

What Is a Multivalue Database?

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

The application that we are going to build in this tutorial is a database for a library. One of the tables required for this application might be a list of the readers who use the library. We could picture this as below.

Reader id	Name	Address	Loan	Date due
1	A Smith	14 High Street	1737	21 Nov 00
2	R Jones	7 Bank Road	4823	27 Nov 00
3	T Harris	4 George Street		
4	L Williams	5 Earl Street	2543	1 Dec 00

In this simple table, each row represents a reader and each column some data associated with that reader. Reader number 3 currently has no book on loan.

Relational databases are built following a set of rules known as the Laws of Normalisation. One of these, the first law, states that we may not have repeating data. In practical terms this means that we cannot add extra columns to the right of the table to allow a reader to borrow more than one book.

Reader id	Name	Address	Loan	Date due	Loan	Date due
1	A Smith	14 High Street	1737	21 Nov 00	7584	21 Nov 00
2	R Jones	7 Bank Road	4823	27 Nov 00		
3	T Harris	4 George Street				
4	L Williams	5 Earl Street	2543	1 Dec 00	5473	30 Nov 00

There are many reasons why this is not allowed, mostly based on the way in which the data will be stored by the computer system. If we are to observe the First Law of Normalisation, we must reconstruct our data in some way to remove the additional columns. One way would be to split a reader who has multiple books on loan across several rows of our table.

Reader id	Name	Address	Loan	Date due
1	A Smith	14 High Street	1737	21 Nov 00
1-1	A Smith	14 High Street	7584	21 Nov 00
2	R Jones	7 Bank Road	4823	27 Nov 00
3	T Harris	4 George Street		
4	L Williams	5 Earl Street	2543	1 Dec 00
4-1	L Williams	5 Earl Street	5473	30 Nov 00

There are many other ways in which we could redesign our table to comply with the Laws of Normalisation but all of them carry database performance implications.

A multivalue database breaks the First Law of Normalisation by allowing multiple values to be stored in a single cell of the table. Our example table now becomes

Reader id	Name	Address	Loan	Date due
1	A Smith	14 High Street	1737 7584	21 Nov 00 21 Nov 00
2	R Jones	7 Bank Road	4823	27 Nov 00
3	T Harris	4 George Street		
4	L Williams	5 Earl Street	2543 5473	1 Dec 00 30 Nov 00

Note how the values in the loan and date due columns are related together. For any particular reader, the first loan number belongs with the first due date, the second loan number belongs with the second due date and so on.

By adopting this data model instead of using additional columns, the data model imposes no limit to the number of books that a reader may have out at one time.

This extended form of the relational database model is at the heart of the QM database.

The time has come to introduce some terminology. A typical application will have many tables, perhaps hundreds or even thousands (our tutorial database has three!). Each table is stored as a **file**. The rows of our table are known as **records** and the columns as **fields**. The data stored in a field may be made up of multiple **values** and these can be further divided into **subvalues** though our

tutorial database will not use subvalues. The relationship between the values in different fields (e.g. loan and due date above) is referred to as an **association**.

Every record stored in a file must have a unique **record id** by which it can be identified. In the above example, the reader number would be the record id. Although these are often numbers, they can be any sequence of up to 63 characters (this limit can be increased by your system administrator). Although there is a limit to the number of records that a file may hold, it is extremely unlikely to affect any real application.

Every file normally has two parts. The **data part** holds the actual data records. The **dictionary part** holds records that describe the structure of the records in the data part and tell the query processor how to display data from the file. We will discuss dictionaries in more detail later.

2   Installing And Running QM

If you have not already installed QM on your system, now is the time to do it. The installation process varies according to the media on which QM was supplied. In all cases the process starts with execution of the appropriate self-extracting archive file for your operating system.

Opening a QMConsole Session

The QM database can be accessed in a number of ways. In this tutorial guide we will use a **console session**. This is a connection to QM from the system on which it is installed. Other methods allow connection over a network.

On Windows systems, once QM has been successfully installed, the program group chosen during the install (usually QM) will contain an item titled "QM console". Clicking on this item will open a console window. You will see a copyright line and a site specific licence line.

On other systems, login and then type qm at the command prompt.

QM applications are divided into **accounts** which can be used to represent different applications or versions of the same application (e.g. development, test, production). When you enter QM as described above you will be prompted for a QM account name. The only account that always exists is the System Administrator account (QMSYS). Although this tutorial can be completed in this account, it is best to create a new account specifically for this purpose.

When you have entered the name of an existing account you will see a line showing a colon which is the QM command prompt. Whenever you see this prompt, QM is waiting for you to type a command.

To create a test account, type

```
CREATE.ACCOUNT TEST C:\TEST
```

where TEST is the name to be given to the account and C:\TEST is a suitable pathname for your new account. This example shows a Windows style pathname.

Notice how the data that you entered appears in uppercase. For historic reasons concerning compatibility with other multivalue databases, QM commands are usually in uppercase. To avoid the need to use the caps lock key as you move from window to window, keyboard input is normally "case inverted" so that letters entered in lowercase appear in uppercase and vice versa. This can be switched off if you prefer (see the PTERM command for details). In most cases, QM will accept commands in either upper or lower case.

Now that you have created your test account, you can move to it by typing

```
LOGTO TEST
```

When you want to leave the tutorial session, you can leave QM by typing

```
QUIT
```

at the QM command prompt. If you want to return to your test account later, you can enter the new account name when starting the session.

So what can we do at the command prompt? QM has over 150 built-in commands which let us develop and execute applications. This tutorial will introduce some of the more important commands.

There is an extensive Windows based help system which can be accessed by selecting the "QM help" item from the QM program group or by typing HELP at the command prompt.

3   Setting up the Data Files

Our database will have just three files

TITLES	Details of a book by title
BOOKS	Details of an individual copy of a book
READERS	Details of the readers using the library

We store the TITLES and BOOKS information separately as we may have more than one copy of a book.

To create these three files, type the commands

```
CREATE.FILE TITLES
CREATE.FILE BOOKS
CREATE.FILE READERS
```

Notice how QM creates both a data and dictionary part for the file. It also adds a record named @ID to the dictionary to give us a default view of the record id. Although we will set up alternative dictionary items for the record ids, the @ID item should not be removed as some parts of QM rely on it being there.

File Types

QM has two distinct types of file; **directory files** and **dynamic files**. By default, files are created as dynamic. We will create a directory file later.

A **directory file** is represented by an operating system directory and the records within it by operating system files. The record key is the name of the file holding the data for the record except where this would be an invalid name in which case QM performs automatic name mapping. Directory files are generally only used for holding QMBasic programs, COMO (command output) files, and stored select lists.

A **dynamic file** is also represented by an operating system directory but the records within it are stored in a fast access file format in this directory. Users should not place any other files in the directory or make any modifications to the files placed there by QM. Dynamic files are so called because of the dynamic reconfiguration of the file which takes place automatically to compensate for changes in the file's size and record distribution. Users have some control over how this reconfiguration takes place by setting a number of configuration parameters. In most cases, these can be left at their default values.

4   Database Records and Mark Characters

A database record may have any number of fields (table columns). The entire record and the constituent fields are of variable length, there being no restriction applied by QM. A record may exist in the database with no data.

Consider the READERS table example earlier. Ignoring the date due column, which we will store elsewhere in our application, the entry for reader 1 was

Reader id	Name	Address	Loan
1	A Smith	14 High Street	1737 7584

Internally, this is stored as a character string where the boundaries between the individual fields are represented by a special character known as a **field mark**. Similarly, within the loan field, the values are separated by **value mark** characters.

The actual data stored in the database is thus

    A SmithFM14 High StreetFM1737VM7584

The record id is not considered to be part of the data record and is thus not shown above.

This representation of a database record divided into fields, values and (possibly) subvalues using mark characters is known as a **dynamic array**. There are actually five mark characters defined within QM. You can find out about the others from the *OpenQM Reference* manual. By using mark characters in this way, the fields stored in the file are of unrestricted variable length and a field may be divided into any number of values.

In directory files, the internal field mark character is replaced by the ASCII newline character when a record is written to disk so that fields appear as lines when the record is viewed, edited or printed from outside QM. Conversely, ASCII newlines are converted to field marks on reading a record.

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

In our demonstration database, our files will hold the following fields:

TITLES
TITLE.REF	A numeric record id for the title
TITLE	The actual title
AUTHOR	The author name. There could be more than one author so this will be multi-valued.
SUBJECT	A subject code
COPIES	The number of copies of the book

BOOKS
BOOK.REF	The record id for the book details
READER	The reader who has this book on loan. Blank if not on loan.
DATE.OUT	The date the book was borrowed

READERS
READER.REF	A numeric record id for the reader
NAME	The reader's name
ADDRESS	The reader's address. Although a reader only has one address, address fields are usually stored as multi-valued where each value represents one line of the address.
LOANS	A multi-valued list of books on loan

There are inter-relationships between the files. Any record in the BOOKS file must be related back to its corresponding TITLES record. One easy way to do this is to use a **composite key** where the copies of the book described by TITLES record 7, for example, would be given BOOKS file ids of 7-1, 7-2, 7-3, etc. Very often, the component parts of a composite key are padded with leading zeros to make them of fixed length. We will not do this in our example database.

The READER field of the BOOKS file contains the record id of the READERS file entry for the reader who has the book out on loan (if any). The LOANS field of the READERS file provides the reverse link by containing the BOOKS record id of the books on loan.

5 Dictionaries

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

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

D   A D-type entry defines a data field present in the file and specifies its location and how it is to be displayed in a report.

I   An I-type entry defines a value that can be calculated from the data in the file and specifies how it is to be displayed in a report.

PH   A PH-type entry is a phrase which can be substituted into a query processing sentence. There are some reserved phrases which control the default actions of the query processor.

X   An X-type entry is a miscellaneous storage item and may be used for any purpose.

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

Consider our TITLES file. Each data record has four fields holding attributes of the order and each of these fields has a corresponding dictionary record. There is also a dictionary record to describe the record id. The name of the dictionary record is the name by which the query processor will refer to the field.

D-type dictionary records normally consist of 7 fields. The purpose and content of each dictionary field is shown in the example below in which the first column shows the conventional name of the dictionary field, the second column is the dictionary field number and the remaining columns show what the dictionary entry would contain to describe the record id and the four data fields.

The dictionaries for our three files contain the D-type items shown below.

TITLES

		TITLE.REF	TITLE	AUTHOR	SUBJECT	COPIES
Type	1	D	D	D	D	D
Loc	2	0	1	2	3	4
Conv	3					
Name	4	Ref	Title	Author	Subject	Cpy

Format	5	5R	25T	15T	15L	2R
SM	6	S	S	M	S	S
Assoc	7					

BOOKS

		BOOK.REF	READER	DATE.OUT
Type	1	D	D	D
Loc	2	0	1	2
Conv	3			D2DMY[,A3]
Name	4	Ref	Reader	Date out
Format	5	8R	5R	9R
SM	6	S	S	S
Assoc	7			

READERS

		READER.REF	NAME	ADDRESS	LOANS
Type	1	D	D	D	D
Loc	2	0	1	2	3
Conv	3				
Name	4	Ref	Name	Address	Loans
Format	5	5R	20T	20T	8R
SM	6	S	S	M	M
Assoc	7				

The role of each dictionary field is described below:

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

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

Conv   Data is sometimes stored in an internal format. The conversion code describes the conversion to be performed before the data is displayed in a report. Conversion codes are discussed later. In the BOOKS file, the DATE.OUT field has a conversion code to specify how the date is to be displayed.

Name   The name field contains the default column heading to be used in reports.

Format   The Format field specifies the number of columns to be used to show this data and how the data is to be aligned within the given width. Format codes are discussed later.

SM   This flag indicates whether the field is always single valued (S) or may be multi-valued (M). In our TITLES file, for example, the author might be multi-valued.

Assoc   Used only with multi-valued fields, this field shows the relationship between associated multi-valued fields. Any fields that have the same word in this dictionary field are associated together. So far, our dictionaries no not require any associations. This will change later.

We can use any convenient editing tool to create the dictionary items. The MODIFY editor is ideal for this purpose. Work through the following steps to set up the dictionaries.

Type

```
MODIFY DICT TITLES
```

At the "Id:" prompt, enter the name of the dictionary record to be created. The first one from our tables above is TITLE.REF.

MODIFY displays a list of dictionary fields. The numbers alongside the field names are for reference only and are not necessarily the actual field positions.

A prompt is displayed at the bottom of the screen for the Type/Desc value. Enter "D" and press the return key. The entered value is displayed in the top part of the screen and MODIFY moves on to prompt for the next field. Continue entering data until seven dictionary fields have been entered. Do not worry if you make a mistake; you can correct it later.

Once all seven fields have been entered, the action prompt appears. The response to this may be:
- a number identifying a displayed field to be amended
- FI to file the record
- Q to quit from the record, discarding any entered data
- ? for help

For records with more fields than will fit on a single page, two further responses are available:
- N for the next page
- P for the previous page

You can move to the action prompt from data entry by entering Ctrl-X (The Ctrl key is like the Shift key: press and hold the Ctrl key, press X, then release both

keys).

Correct any mistakes in the displayed data and then file the record by typing FI at the action prompt. The "Id:" prompt appears again. Enter the next dictionary record id to be created (TITLE). Continue in this way until all the TITLES file dictionary items have been created. To leave MODIFY, enter a blank response to the "Id:" prompt.

Use MODIFY again to enter the dictionary definitions for the BOOKS and READERS files.

You can view your dictionaries by typing, for example,

```
LIST DICT TITLES
```

Now that you have entered the dictionary definitions, you can use MODIFY to enter some data into the files. Type

```
MODIFY TITLES
```

Again, you receive a record id prompt but this time it will use the file name as the prompt. This default prompt can be changed as described in the MODIFY description in the *QM Reference Manual*.

Enter "1" as the id for the first title record. The field list displayed has three items for the title, subject and copies. The fourth item is shown with a slightly different number prefix as this leads to a second screen for the multi-valued field. Note how the text entered as the field display name in the dictionary is used as the prompt.

Enter a suitable book title and subject and set the copies field to 1. The display then changes to allow entry of multiple authors. Enter the author(s) for the book, using the form "lastname, forename" to fit in with the way our queries will work later.

When all the authors have been entered, the multi-valued field action prompt appears. This allows entry of:
- an item number corresponding to a line to be changed
- I*n* to insert a line above line *n*
- D*n* to delete line *n*
- E to extend (add new items at the end)

For a multi-page list, the prompt also allows
- N to move to the next page
- P to move to the previous page

If no changes are required, just press the return key to return to the first screen.

As before, make any corrections and then enter FI to file the record.

Enter two further book titles. Book number 2 should have two copies, book 3 should have one copy.

Now use MODIFY to enter the corresponding BOOKS records. Remember that these are keyed by the TITLES file key with a suffix of the copy number. Our records will be 1-1, 2-1, 2-2, 3-1.

> Book 1-1 should be on loan to reader 2. Enter a suitable date for the loan.
> Book 2-1 should not be on loan (leave the reader and date out both blank).
> Book 2-2 should be on loan to reader 3.
> Book 3-1 should be on loan to reader 2.

This loan pattern ensures that we have books that are on loan and not on loan and readers who have no books, one book or two books on loan.

Finally, use MODIFY to enter the READERS file items, ensuring that the loans are correct.

> Reader 1 has no books on loan
> Reader 2 has books 1-1 and 3-1 on loan
> Reader 3 has book 2-2 on loan

Enter the names in the form "lastname, initials" with no dots after the initials (e.g. "Smith, A J").

That's it! You now have a data set for experimenting with the QM query processor and as a start for the simple loans application. Feel free to add more data but be careful that the inter-relationship between the records in different files is maintained correctly.

6   Viewing the Data

Type "`LIST TITLES`" to view the TITLES file. This query shows us the ids of the TITLES file records but is not of much use. We could extend the query by adding the names of fields we would like to see:

    `LIST TITLES TITLE AUTHOR SUBJECT COPIES`

This is better but the query processor is showing us the default view of the record id as the first column rather than the one we created with a nice column heading. Try adding our own definition of the record id as the first field in the query:

    `LIST TITLES TITLE.REF TITLE AUTHOR SUBJECT COPIES`

Hint: You can use the cursor up key to walk back to the previous command and then edit it using the cursor left and right keys, delete, backspace, etc.

We now have the column we would like but we still have the default view of the record id. Add the ID.SUP keyword to the query to suppress this default item:

    `LIST TITLES TITLE.REF TITLE AUTHOR SUBJECT COPIES ID.SUP`

This looks like a useful report but we don't want to have to type in the list of fields every time we produce a report. Instead, we can set up a **default listing phrase** in the dictionary to tell the query processor what we want to see if we do not list any display fields in the command.

To do this, use MODIFY on the dictionary of the TITLES file to add an item named @. The Type/Desc field should be PH to show that this is a phrase entry. In a phrase, field 2 contains the expansion of the phrase and the remaining fields are unused. At the F2 prompt, type

    `TITLE.REF TITLE AUTHOR SUBJECT COPIES ID.SUP`

At the F3 prompt type Ctrl-X to skip to the action prompt. File your new phrase.

Now try

    `LIST TITLES`

The query processor uses your phrase to determine the fields to be displayed. Note that if you include any field names for display in the command, the phrase is not used:

    `LIST TITLES AUTHOR`

Set up default listing phrases for the other files:

BOOKS: `BOOK.REF READER DATE.OUT ID.SUP`

READERS: `READER.REF NAME ADDRESS LOANS ID.SUP`

Check that the reports are as you would expect.

7   Conversion and Formatting

Conversion Codes

The date field in our BOOKS file used a conversion code to specify how the data should be displayed. Dates are usually stored as a number of days from 31 December 1967, that being day zero. All later dates are positive numbers, all earlier dates are negative numbers. The conversion code in the dictionary tells MODIFY that the field is a date and, when the date is entered, it is automatically converted to its internal form. When the date is output by MODIFY or by the query processor, the reverse conversion is performed to display a meaningful date. The conversion code specifies the exact form in which the date is to appear and has many possibilities.

Dates are just one data type that is normally stored in some special internal form. Other examples are times (stored as seconds since midnight) and weights and measures (stored scaled to remove the decimal point). There are many other conversion codes and users can add their own. Conversion codes are discussed in detail in the *QM Reference Manual*.

Format Specifications

The format specification in field 5 of the dictionary entry tells the query processor how many columns to allocate across the report for the data and where in this field width the data should be displayed. For example, our TITLES file record id is output using a "5R" format specification which displays the data right justified in a five character wide field. The SUBJECT field format is "15L", left justifying the data in a 15 character wide field. The TITLE uses a format specification of "25T". This is similar to "25L" but will attempt to break between words if the data must be split over multiple lines.

Format specifications contain several other features as described in the *QM Reference Manual*.

8 Dictionary I-type Records

As our dictionaries stand so far, we can construct simple queries to show the data from any one of our three files. We might want to list the BOOKS file, showing the title of each book. This requires access to data from two files.

A query processor command can only directly reference a single file. To bring together data from other files, or for other calculated values, we use I-type dictionary items.

An I-type dictionary record defines a calculation based on data on the data file records. Once an I-type item is defined, it can be referenced in query processor sentences exactly as though it was a real (D-type) data field. I-type items are sometimes known as **virtual attributes**, a term which emphasises the fact that their values are not physically stored in the database.

An I-type dictionary item differs from a D-type item only in that field 1 contains the type code I and field 2 contains the actual calculation to be performed. The remaining fields are as for a D-type entry.

The expression in an I-type dictionary record is actually a little QMBasic program and can use most of the constructs that can appear on the right side of an assignment in the full programming language. In this tutorial we will introduce a few simple I-type expression constructs without going into too much detail of how they work. You will need to read the opening sections of the QMBasic section of the *QM Reference Manual* to gain a better understanding of I-type expressions.

Calculating the Due Date

Our BOOKS file includes a DATE.OUT field holding the date on which a book was borrowed. We might want to know the date it is due back. Assuming a simple three week borrowing period, this can be calculated by adding 21 days to the date out. An I-type expression to do this could be written as

```
DATE.OUT + 21
```

This is not perfect. A book that is not on loan has a blank date out field. In an arithmetic calculation a blank field is treated as zero and hence all books not out on loan would appear to be due back on day 21 (21 January 1968!).

The expression syntax allows us to handle this by adding a conditional element

```
IF DATE.OUT THEN DATE.OUT + 21 ELSE ""
```

Use MODIFY to enter a DATE.DUE item to the BOOKS dictionary. This should read

```
Type/Desc:  I
Loc:        IF DATE.OUT THEN DATE.OUT + 21 ELSE ""
Conv:       D2DMY[,A3]
Name:       Date due
Format:     9R
S/M:        S
Assoc:
```

Modify the @ phrase to include this field. When you select item 2 (the LOC field) of this phrase in MODIFY you can use the cursor left and right keys, etc to move around in the data. You do not need to retype the entire phrase.

Try LIST BOOKS to see your I-type item in action.

Getting the Book Title for a BOOKS File Report

This one requires access to a second file (the TITLES file). Remember that the record ids in our BOOKS file are the TITLES file id with a suffix added to identify the copy.

We will construct two I-type items here. The first (TITLE.REF) will calculate the TITLES file id by extracting the first part of the composite record id. The second (TITLE) will fetch the actual book title, using the TITLE.REF item to identify the record. Note how this implies use of the result of one calculated value in another calculation.

The TITLE.REF I-type dictionary record should read

```
Type/Desc:  I
Loc:        @ID["-", 1, 1]
Conv:
Name:       TitleRef
Format:     5R
S/M:        S
Assoc:
```

The expression in the LOC field extracts the first hyphen separated component of the composite record id. See the description of expression syntax in the *QM Reference Manual* for more details.

You can try this out by adding it to your BOOKS file dictionary and typing

```
LIST BOOKS TITLE.REF
```

The TITLE I-type dictionary record should read

```
Type/Desc:  I
Loc:        TRANS(TITLES,TITLE.REF,TITLE,"X")
Conv:
Name:       Title
Format:     25T
S/M:        S
Assoc:
```

The TRANS() function in the LOC field is used to fetch data from another file. The four items inside the brackets are:

- The name of the file from which the item is to be obtained.

- The name of an item defined in this dictionary which gives the id of the record to be read from the target file.

- The name of a D-type item defined in the dictionary of the target file identifying the data to be returned.

- An error code determining the action if the item cannot be found. In this case "X" specifies that a blank result should be returned.

See the description of the TRANS() function in the *QM Reference Manual* for more details.

Add this new I-type item to the dictionary of your BOOKS file and test it.

It would be useful to be able to get the name of the reader who has a book on loan. See if you can create an I-type item named NAME in the dictionary of the BOOKS file to do this. The solution is on the next page.
The NAME I-type dictionary record should read

```
Type/Desc:  I
Loc:        TRANS(READERS,READER,NAME,"X")
Conv:
Name:       Reader Name
Format:     20T
S/M:        S
Assoc:
```

Add this to your BOOKS dictionary and test it.

We might also find it useful to be able to list the dates on which the books were loaned to a particular reader. We can do this by adding an I-type item named DATE.OUT to the READERS file dictionary such as that shown below.

```
Type/Desc:  I
Loc:        TRANS(BOOKS,LOANS,DATE.OUT,"X")
Conv:       D2DMY[,A3]
Name:       Date out
Format:     9R
S/M:        M
Assoc:
```

This is very similar to the way in which we fetched the reader's name using an I-type item in the BOOKS dictionary. There is, however, one very important feature shown by this expression. A reader may have more than one book on loan. The LOANS field of the READERS file is multi-valued. The TRANS() function in the above expression, therefore, has a multi-valued list of BOOKS file ids as its LOANS item and will process the entire list in one operation, returning a corresponding multi-valued list of dates. Because the LOANS item is multi-valued, the DATE.OUT I-type item must also be multi-valued.

Add this new I-type to your READERS dictionary and test it.

We now have two multi-valued items in the READERS file dictionary where the data has a value by value relationship. The first value in the LOANS field is related to the first value in the DATE.OUT field. The second and subsequent values have the same relationship. This relationship needs to be defined by creating an **association**.

To do this requires two steps. Firstly, we must think up a name for the association and insert it into the Assoc field (field 7) of the dictionary items for LOANS and DATE.OUT. Secondly, we must create a PH (phrase) type entry with the name of the association and with a space separated list of the fields that are members of the association in field 2 (LOC).

For example, we might choose to call this association BOOKS.OUT. We use MODIFY to add this name to the Assoc field of the LOANS and DATE.OUT items. These now become as shown below.

LOANS:
```
Type/Desc:  D
Loc:        3
Conv:
Name:       Loans
Format:     8R
S/M:        M
Assoc:      BOOKS.OUT
```

DATE.OUT:
```
Type/Desc:  I
Loc:        TRANS(BOOKS,LOANS,DATE.OUT,"X")
```

```
Conv:       D2DMY[,A3]
Name:       Date out
Format:     9R
S/M:        M
Assoc:      BOOKS.OUT
```

The phrase defining the association is

BOOKS.OUT:
```
   Type/Desc:  PH
   Loc:        LOANS DATE.OUT
```

Make these changes to the dictionary of your READERS file. When entering the phrase, remember that typing Ctrl-X at the CONV prompt after the LOC field has been entered will jump to the action prompt.

Why do we need to define associations in this way?

The query processor and some other components such as MODIFY need to know about the association so that they can correctly pair up the associated items. Quite often, forgetting to define the association has no effect on the work that we do but sometimes we will get the incorrect results from query reports.

By having a phrase that lists the members of the association we can start from any member, use the Assoc field to find the phrase and hence find all the other associated items. This removes the need to scan the entire dictionary checking for associated items. Clearly, we need to ensure that this two way linking of associated items is correctly maintained if we modify the dictionary.

There is one last I-type expression to add to our dictionaries.

We might also find it useful to be able to list the titles of all books on loan to a particular reader. We can do this by adding an I-type item named TITLE to the READERS file dictionary such as that shown below.

```
   Type/Desc:  I
   Loc:        TRANS(TITLES,FIELDS(LOANS,"-",1),TITLE,"X")
   Conv:
   Name:       Titles on loan
   Format:     25T
   S/M:        M
   Assoc:      BOOKS.OUT
```

The FIELDS() function in the second argument to the TRANS() function takes a multi-valued list of LOANS and extracts just the first hyphen separated

component. Again, you can find out more about this function as well as many other multi-value manipulation functions from the *QM Reference Manual.*

Note how this item is also a member of the BOOKS.OUT association. Don't forget to add it to the list in the BOOKS.OUT phrase.

We now have a sufficiently comprehensive set of dictionary items to handle the rest of this tutorial. Feel free to add new items if you want to explore further possibilities.

QM includes a pair of commands to setup and delete the demonstration files. These will allow you to work with these files again in the future without having to go through the whole file setup process. The commands are:

SETUP.DEMO   Creates or resets the demonstration files

DELETE.DEMO   Deletes the demonstration files

9   Query Processing

In this section we will look at some of the basic features of the QM query processor. There is much more to discover for yourself when you look through the manuals.

The query processor allows us to extract data from the database files for all purposes from simple queries to complex reports. It has many features to specify what data is to be extracted and how that data is to be presented.

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

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

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

where

verb	is the query processor verb to be used.
filename	specifies the file to be processed. A query processor sentence may only reference a single file though the dictionary of that file may include virtual attributes to fetch data from other files. The *filename* may be prefixed by DICT to process the dictionary part of the file.
selection.clause	specifies which records are to be included in the report. QM offers a wide range of methods to select records. If omitted, all record are reported.
sort.clause	specifies the order in which the reported data is to be output. If omitted, the records are reported in the order in which they are found on the file.
display.clause	specifies the fields to be displayed in the report and, optionally, overrides the dictionary details of how they are to be converted, formatted, etc. If omitted, a default set of fields is reported.
options	are various additional options to control the page layout, output destination, etc.

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

and multiple selection or sort clauses will be applied in the order in which they appear.

All command processing in QM is controlled via the VOC (vocabulary) file. This defines the names of verbs, keywords that control their actions, files, etc. The VOC is discussed in detail in the *Introduction to QM* manual.

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

The Selection Clause

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

For example

```
LIST TITLES 1 3
```

The second method to select the records to be processed is by use of the WITH clause. This tests each record against some selection criteria to determine if it should be included in the report.

There are many selection methods available in the WITH clause. The most frequently used is a test of a field value against either a literal value or another field. For example,

```
LIST TITLES WITH COPIES > 1
```

The fields named in a WITH clause must be defined in the dictionary or the VOC as D or I-type items.

The relational operators available are

```
=     EQ
#     NE    <>         ><
<     LT    LESS       BEFORE
>     GT    GREATER    AFTER
<=    LE    =<
>=    GE    =>
```

Where a relational operator is used to compare a field which has a conversion code with a literal value, the literal value is converted to the internal form of the field and subsequent comparison is done using internal values. This means that, for example, a literal date may be entered in any recognisable format rather than

only in the form given by the conversion code.

```
LIST BOOKS WITH DATE.OUT AFTER '1 JUL 00'

LIST TITLES WITH SUBJECT = 'Art'
```

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

```
LIST TITLES WITH AUTHOR = 'Smith, Alan'
```

The above query would show all records in the TITLES file which included the given author, even if the book had more than one author.

We might want to select records where all of the values in a multi-valued field satisfy the selection criteria. This can be done using the WITH EVERY construct. For example,

```
LIST READERS WITH EVERY DATE.OUT > '1 JUL 00'
```

This query shows readers where all books on loan were taken out after 1 July 2000.

Multiple conditions can be included by using the keywords AND or OR in the query sentence. For example,

```
LIST TITLES WITH SUBJECT = 'Art' AND AUTHOR = 'Smith,
Alan'

LIST TITLES WITH SUBJECT = 'Art' OR SUBJECT = 'History'
```

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

```
LIST TITLES WITH SUBJECT = 'Art' WITH AUTHOR = 'Smith,
Alan'
```

This is equivalent to the first of the examples above.

Pattern Matching

Another commonly used WITH clause component is pattern matching to test the general structure of a character string rather than its exact content. The general form of this is

WITH *field* LIKE *template*

where

field   is the field to be compared.

template is the pattern against which *field* is to be compared.

The *template* consists of one or more concatenated items from the following list.

...	Zero or more characters of any type
0X	Zero or more characters of any type
*n*X	Exactly *n* characters of any type
*n-m*X	Between *n* and *m* characters of any type
0A	Zero or more alphabetic characters
*n*A	Exactly *n* alphabetic characters
*n-m*A	Between *n* and *m* alphabetic characters
0N	Zero or more numeric characters
*n*N	Exactly *n* numeric characters
*n-m*N	Between *n* and *m* numeric characters
"*string*"	A literal string which must match exactly. Either single or double quotation marks may be used.

The values *n* and *m* are integers with any number of digits. *m* must be greater than or equal to *n*.

The 0A, *n*A, 0N, *n*N and "*string*" patterns may be preceded by a tilde (~) to invert the match condition. For example, ~4N matches four non-numeric characters such as ABCD (not a string which is not four numeric characters such as 12C4).

Examples

```
LIST READERS WITH NAME LIKE Smith...

LIST TITLES WITH TITLE LIKE ...Concise...
```

The query processor has many short forms. Amongst the more useful, a relational operator or LIKE keyword not immediately preceded by a field name uses the same field as the previous test or the record id if this is the first test.

Examples

```
LIST BOOKS WITH DATE.OUT AFTER '1 APR 00' AND BEFORE '1
MAY 00'

LIST TITLES > 10
```

An operator followed by a list of values tests each in turn in an implied OR relationship.

Examples

```
LIST TITLES WITH SUBJECT = Art History

LIST READERS WITH NAME LIKE Smith... Jones...
```

Note that the first example above reports books with a subject of Art or History. If we wanted to find books on Art history, the query would be

```
LIST TITLES WITH SUBJECT = 'Art history'
```

Note also that the query processor comparison operators are case sensitive. We can work around this using features not discussed in this tutorial.

The Sort Clause

The sort clause specifies the order in which the reported records will appear. In its most commonly used form it is introduced by the BY keyword.

Example

```
LIST BOOKS BY DATE.OUT
```

The BY.DSND keyword performs a descending sort.

Example

```
LIST BOOKS BY.DSND DATE.OUT
```

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

Example

```
LIST TITLES BY SUBJECT BY TITLE
```

The SORT verb is equivalent to LIST with a final BY @ID component. Thus the following two queries produce identical output

```
SORT TITLES BY SUBJECT
LIST TITLES BY SUBJECT BY @ID
```

Do not use SORT unless you actually want to sort by record id.

The Display Clause

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

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

Examples

```
LIST TITLES TITLE SUBJECT

LIST READERS NAME ADDRESS ID.SUP
```

There are various keywords not discussed in this tutorial that may follow a field name to override the dictionary definition for conversion code, format code, column heading, etc.

If a query sentence contains no field names, the query processor looks in the dictionary for a PH-type (phrase) entry named @. If this is found, it is attached to the end of the query sentence. Typically, this phrase contains a default list of fields to be shown but it may also include other query sentence elements. If there is no @ phrase, only the record id will be shown.

Reporting Options

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

The only option discussed in this tutorial is the LPTR keyword.

If the query sentence includes the LPTR keyword, the report is directed to a printer instead of the terminal. Which printer is used depends on the way in which your system is set up.

QM commands and application software send their output to a numbered print unit. The actual destination for each print unit is set using the SETPTR command, often from the LOGIN paragraph or other initialisation script.

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

When the LPTR keyword is used and the query sentence contains no field names, the query processor looks first for a phrase named @LPTR as its default listing phrase and then, if this is not found, for the @ phrase. This allows a different set of default fields to be shown in a printed report from one displayed on the screen.

Select Lists

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

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

Many QM verbs, including all of the query processor verbs, check for an active default select list and, if found, use this to provide a list of items to be processed. Because of the dangers of unwanted effects if a list is left active by accident, the command prompt changes to :: when the default select list is active.

The SELECT verb constructs list 0 by default. An alternative list can be built using the TO clause to specify the target list number. Similarly, all query processor verbs have an optional FROM clause to specify the source list.

Examples

```
SELECT TITLES WITH COPIES > 1

LIST TITLES
```

This pair of commands is equivalent to

```
LIST TITLES WITH COPIES > 1
```

There are times when we might perform a SELECT against one file to generate a list of record ids to process in some other file.

```
SELECT TITLES WITH SUBJECT = Art
DELETE TITLES
```

This pair of commands deletes all the art books from our TITLES file. Don't do it!

10   Building a Report Menu

QM includes an easy to use system for generating menus as part of an application. We will use this to build a simple front end to our reports.

To enter the menu editor, type

```
MED
```

A prompt appears asking for the name of the file to hold the menu. We will store our menu in the VOC file which can be selected simply by pressing the return key to enter a blank response.

The next prompt asks for the menu name. Enter "LIB" as this is our library application. You will be asked to confirm that you want to create a new menu.

The screen now shows five lines on which text can be entered corresponding to the following aspects of the menu:

Title   The title line to appear on the menu

Subr   The name of a subroutine to be used to validate a user's access to each menu option. This is optional and we will leave it blank.

Prompt   The option prompt text to appear at the foot of the menu. We will leave this blank to use the default prompt text.

Exits   A comma separated list of codes that can be used to exit from the menu to whatever started it. We will not use this feature.

Stops   A comma separated list of codes that can be used to exit from the menu to the command prompt. We will not use this feature.

As you work through data entry, a short help message appears at the foot of the screen.

Enter a suitable menu title such as "Library System Menu" then press the return key repeatedly until some further lines appear asking for the details of the first menu item. These details are:

Text   The text of the menu item

Action   The command to be executed when this action is selected. If the command is terminated by a semicolon, the menu processor will issue a "Press return to continue" prompt before returning to the menu. This will be needed on all of our report generation menu items.

Help   An optional single line help message for this menu item

Access   A key passed into the access control subroutine. We are not using this feature so this field can be left blank.

Hide   Determines whether inaccessible options are displayed or hidden.

Since we are not using access control this can be left blank.

Enter three menu items to list our three database files. One simple solution is shown below.

```
Title :Library System Menu
Subr  :
Prompt:
Exits :
Stops :
-------------------------------------------------------------
Text  :Display title information
Action:LIST TITLES;
Help  :Lists the TITLES file
Access:
Hide  :
-------------------------------------------------------------
Text  :Display book copy information
Action:LIST BOOKS;
Help  :Lists the BOOKS file
Access:
Hide  :
-------------------------------------------------------------
Text  :Display reader information
Action:LIST READERS;
Help  :Lists the READERS file
Access:
Hide  :
```

If you make a mistake, you can use the cursor keys to move around in the data and make corrections.

The MED editor uses a subset of the control keys from the SED full screen text editor. You can get extended help for the current field when in MED by pressing the F1 key. Pressing F1 again displays a screen of key bindings.

Once you are happy with your menu, you can save it by typing Ctrl-X followed by S. To exit from MED, type Ctrl-X followed by C.

To test the menu, type its name (LIB) at the command prompt.

11 A Simple Loan Management Application

The final section of this tutorial assembles a very simple QMBasic program to allow readers to take books out on loan or to return them. There is minimal screen handling in this program. A more realistic program would probably use the SCRB screen builder. If you are a programmer you can learn much more about the QMBasic language from the *QM Reference Manual*. You may also want to explore the QMClient API which allows you to write Windows style front ends to QM applications using Visual Basic.

The readable (source language) form of QMBasic programs must be held in a directory file. By convention this is often called BP (Basic Programs). Create a BP file in your test account by typing

```
CREATE.FILE BP DIRECTORY
```

QM provides two text editors; ED which is a simple line based editor very similar to that found on several other multi-value database products and SED which is a screen based editor originally developed for QM but also available from Ladybridge Systems for the UniVerse and Unidata databases.

In this tutorial we will use a few of the basic functions of SED. There are many more for you to discover which make editing much easier.

Our program will be called LIB. To start the editor, type

```
SED BP LIB
```

You will be faced with a blank screen into which program statements can be typed. The cursor keys can be used to move around and the backspace and delete keys can be used to remove text.

This tutorial does not set out to teach the QMBasic programming language, rather we are going to guide you through entry of a very simple program to maintain our demonstration files. Experienced programmers should have no trouble learning QMBasic from the main manual set. We include a discussion of the principles of this program later.

The program that we are going to put together has no error handling. It assumes, for example, that the inter-relationships between the files are correct.

Enter the program text shown over then next two pages. Although we have shown the program in uppercase to follow the conventional style of multi-value database programming languages, QM allows programs to be in lowercase. Only the file names in the OPEN statements must be in uppercase.

```
PROGRAM LIB    * Open the files

   OPEN 'TITLES' TO TTL.F ELSE ABORT 'Cannot open TITLES'
   OPEN 'BOOKS' TO BKS.F ELSE ABORT 'Cannot open BOOKS'
   OPEN 'READERS' TO RDR.F ELSE ABORT 'Cannot open READERS'

   * The main loop of the program prompts for a book id and processes
   * it. In a real application, this book id might come from a barcode
   * reader.

   LOOP
      DISPLAY 'Book id' :
      INPUT BKS.ID

   UNTIL BKS.ID = ''

      * Fetch the details for this book

      READU BKS.REC FROM BKS.F, BKS.ID THEN

         * Read the corresponding TITLES record and display the title

         READ TTL.REC FROM TTL.F, BKS.ID['-',1,1] THEN
            DISPLAY 'Book title : ' : TTL.REC<1>
         END

         * Extract the reader id from the BOOKS record to decide if the
         * book is out on loan or in stock.

         RDR.ID = BKS.REC<1>
         IF RDR.ID # '' THEN    ;* Book is on loan

            * Get the reader's details

            READU RDR.REC FROM RDR.F, RDR.ID THEN
               DISPLAY 'On loan to : ' : RDR.ID : ' (' : RDR.REC<1> : ')'
               DISPLAY 'Date out   : ' : OCONV(BKS.REC<2>, 'D2DMY[,A3]')
               DISPLAY 'Return to stock' :
               INPUT YN
               IF UPCASE(YN) = 'Y' THEN

                  * Remove this book from the reader's loans

                  LOCATE BKS.ID IN RDR.REC<3,1> SETTING POS THEN
                     DEL RDR.REC<3,POS>
                  END
                  WRITE RDR.REC TO RDR.F, RDR.ID

                  * Update the BOOKS file record to show the book as in stock

                  BKS.REC<1> = ''   ;* Clear reader field
                  BKS.REC<2> = ''   ;* Clear date out field
               END ELSE             ;* Not returning, release record lock
                  RELEASE RDR.F, RDR.ID
               END
            END
         END ELSE                 ;* Book is in stock

            * Ask for reader id to generate new loan

            DISPLAY 'In stock'
            DISPLAY 'Enter reader id to loan, blank to ignore'
            INPUT RDR.ID
            IF RDR.ID # '' THEN
               READU RDR.REC FROM RDR.F, RDR.ID THEN

                  * Add this book to the reader's list of loans

                  RDR.REC<3,-1> = BKS.ID
                  WRITE RDR.REC TO RDR.F, RDR.ID
```

```
                    * Update the BOOKS file record to show the book as on loan

                    BKS.REC<1> = RDR.ID    ;* Set reader number
                    BKS.REC<2> = DATE()    ;* Set loan date
                 END ELSE                  ;* Don't have this reader number
                    RELEASE RDR.F, RDR.ID
                    DISPLAY 'Reader not on file'
                 END
              END
           END

        WRITE BKS.REC TO BKS.F, BKS.ID
     END ELSE              ;* Unknown book number entered - Release the lock
        RELEASE BKS.F, BKS.ID
        DISPLAY 'Book is not on file'
     END
     DISPLAY
  REPEAT
END
```

Once you have entered this text, save it by typing Ctrl-X followed by S and exit from SED by typing Ctrl-X followed by C. (The same sequence as for MED).

The program must now be converted to a form that QM can execute. This process is called compilation and is done using the BASIC command:

```
   BASIC LIB
```

If you received any error messages, re-enter SED to correct the errors and try again.

Once your compilation completes without errors, you can execute the program by typing

```
   RUN LIB
```

So, what does this program do? Let's take it apart step by step....

```
PROGRAM LIB
```

The PROGRAM line is optional and gives a name to the program. The name is totally ignored and is for documentation purposes only.

```
   * Open the files

   OPEN 'TITLES' TO TTL.F ELSE ABORT 'Cannot open TITLES'
   OPEN 'BOOKS' TO BKS.F ELSE ABORT 'Cannot open BOOKS'
   OPEN 'READERS' TO RDR.F ELSE ABORT 'Cannot open READERS'
```

The three OPEN statements open the files. Each includes the name of a file variable (e.g. TTL.F) which will be used elsewhere in the program to refer to the file. The ELSE clause of the OPEN is executed if the open fails. In this case an

ABORT statement is used to terminate the program with an error message.

```
* The main loop of the program prompts for a book id and processes
* it. In a real application, this book id might come from a barcode
* reader.

LOOP
```

The LOOP statement marks the top of a loop terminated by a REPEAT statement (almost at the bottom of our program). Every time we arrive at the REPEAT, the program jumps back to the LOOP to start all over again.

```
DISPLAY 'Book id' :
INPUT BKS.ID
```

The DISPLAY statement displays the given text. The trailing colon is a special syntax that causes the cursor to remain at the end of the displayed text instead of moving to the start of the next line.

The INPUT statement prompts the user to enter data by displaying a question mark (this can be changed or suppressed) and then takes data from the keyboard, storing it in a variable named BKS.ID until the return key is pressed. You could use any variable name you liked. We have chosen to adopt a simple convention where our BOOKS file uses file variable BKS.F and the record id is stored in BKS.ID.

```
UNTIL BKS.ID = ''
```

The UNTIL statement is part of the LOOP / REPEAT construct. When the condition is met (a blank book id is entered), the program will jump to the statement following the REPEAT thus exiting from the loop.

```
* Fetch the details for this book

READU BKS.REC FROM BKS.F, BKS.ID THEN
```

The READU statement reads the BOOKS file record for the given book id. Our variable naming convention uses BKS.REC to hold a record from the BOOKS file. The READU statement takes an update lock on the record. This prevents another user accessing the same record while we are performing our update. As this program is written, they would simply wait for us to finish. READU has an option to allow the programmer to take his own action if a read fails because another user has the record locked.

If we succeed in reading the record we execute the THEN clause (all the statement down to the corresponding END. Note how our indentation makes this pairing easy to see). If the read fails because the record does not exist, we will execute the ELSE clause (if present). In this case, the ELSE clause is almost at the end of the program.

```
* Read the corresponding TITLES record and display the title

READ TTL.REC FROM TTL.F, BKS.ID['-',1,1] THEN
   DISPLAY 'Book title : ' : TTL.REC<1>
```

```
    END
```

We also need to read the TITLES record for this book. Here we use READ
instead of READU. The READ statement does not use any locks. Another user
might be working with a different copy of the same book where locking the
TITLES record would cause him to wait. In general, we lock records that we
might change; we do not lock records that we are simply looking at.

The BKS.ID['-',1,1] construct should look familiar as we used this in one of our
dictionary Itype records earlier to extract the TITLES file record id from the
BOOKS id.

The DISPLAY statement displays the fixed text (Book title :) followed by the book
title. In our I-types we could refer to fields by name. QMBasic programs do not
have an automatic way to use the dictionary to find fields by name. Instead we
use the field extraction construct (e.g. TTL.REC<1>) to extract field 1. There are
ways in which programmers can give names to fields but we have not done so
here.

```
    * Extract the reader id from the BOOKS record to decide if the
    * book is out on loan or in stock.

    RDR.ID = BKS.REC<1>
    IF RDR.ID # '' THEN    ;* Book is on loan
```

By examining the READERS field (field 1) of the BOOKS record, we know if the
book is on loan and, if so, to whom.

```
    * Get the reader's details

    READU RDR.REC FROM RDR.F, RDR.ID THEN
        DISPLAY 'On loan to : ' : RDR.ID : ' (' : RDR.REC<1> : ')'
        DISPLAY 'Date out   : ' : OCONV(BKS.REC<2>, 'D2DMY[,A3]')
```

For a book that is on loan, we read the READERS file details, locking it because
we are probably returning the book to stock. We also display the reader id, his
name and the date on which the book as taken out. The OCONV() function
performs output conversion to display the date in a useful format.

```
    DISPLAY 'Return to stock' :
    INPUT YN
    IF UPCASE(YN) = 'Y' THEN
```

We now ask if the book is to be returned to stock. The response should be Y or
N. To allow for entry of either upper or lowercase responses, we use the
UPCASE() function to convert the response to uppercase before testing whether
it was Y. A real program would probably do some better validation of the
response.

```
    * Remove this book from the reader's loans

    LOCATE BKS.ID IN RDR.REC<3,1> SETTING POS THEN
        DEL RDR.REC<3,POS>
    END
```

The reader may have several books on loan. We cannot simply clear out the LOANS field of the READERS record. Instead we use the very powerful LOCATE statement to search for the book id in the list and the DEL statement to delete it.

```
WRITE RDR.REC TO RDR.F, RDR.ID
```

The READERS file record can now be written back to the file. This action will automatically release the record lock, allowing another user access.

```
* Update the BOOKS file record to show the book as in stock

BKS.REC<1> = ''   ;* Clear reader field
BKS.REC<2> = ''   ;* Clear date out field
```

We must also clear the reader id and the date out from the BOOKS record. This record is written back to the file in a path that is common to both loans and returns later.

```
END ELSE              ;* Not returning, release record lock
```

If the user enters N to the prompt asking if the book is to be returned, we must release the lock the we hold on the READERS record so that other users can access it.

```
        RELEASE RDR.F, RDR.ID
      END
    END
END ELSE              ;* Book is in stock
```

If the book is in stock, we are processing a new loan.

```
* Ask for reader id to generate new loan

DISPLAY 'In stock'
DISPLAY 'Enter reader id to loan, blank to ignore'
INPUT RDR.ID
```

Prompt for and input the reader id for this loan.

```
IF RDR.ID # '' THEN
   READU RDR.REC FROM RDR.F, RDR.ID THEN
```

If a reader id was entered, read the READERS file record, locking it as we are going to update it.

```
* Add this book to the reader's list of loans

RDR.REC<3,-1> = BKS.ID
```

This special syntax appends a new value to a list, in this case the list of loans by the reader.

```
WRITE RDR.REC TO RDR.F, RDR.ID
```

We can now write the READERS file record to show the book as on loan to the

reader.

```
    * Update the BOOKS file record to show the book as on loan

    BKS.REC<1> = RDR.ID    ;* Set reader number
    BKS.REC<2> = DATE()    ;* Set loan date
```

The BOOKS file record must be updated to show the reader id and the loan date. The DATE() function returns today's date in internal form.

```
    END ELSE                ;* Don't have this reader number
       RELEASE RDR.F, RDR.ID
       DISPLAY 'Reader not on file'
```

We must cater for the error of entering an invalid reader id. Here we simply release the lock (even though the record did not exist, we have locked the id) and display an error message.

```
            END
        END
    END

    WRITE BKS.REC TO BKS.F, BKS.ID
```

The common path for all updates writes the (possibly) modified BOOKS record back to the file. This also releases the lock, letting other users access the book. If we chose not to return a book that is on loan or entered a blank or invalid reader number for a new loan, we will still perform this write. Although the data in the BOOKS record has not changed, this is an easy way of ensuring that we have also released the lock.

```
    END ELSE          ;* Unknown book number entered - Release the lock
       RELEASE BKS.F, BKS.ID
       DISPLAY 'Book is not on file'
```

For an invalid book number we must also be sure to release the lock.

```
    END
    DISPLAY
```

The processing of each book id ends with output of a blank line to make the display more readable

```
    REPEAT
END
```

And here, at long last, is the bottom of our processing loop and an END to terminate the program.