

IBM UniVerse Server Developer Self Paced Training

Brian Leach

Paragraphs

When you have several sentences, they make up a paragraph – and thus the Paragraph is the name given to the simplest batch structure in a UniVerse system.

A paragraph contains a number of commands that are normally executed in strict order. Each paragraph can contain several lines with the following format:

Field 1	PA description
Field 2	command
Field 3+	commands

The first field begins with the letters 'PA' and, as you can probably guess by now, this can be optionally followed by a description. The second and following fields onwards hold the commands to run in sequence with each command starting on a new line. As with sentences, long or complex commands can be broken over multiple lines using the continuation character (`\`).

Paragraphs are used to run multiple commands in sequence, and can include in-line prompts as below:

Do this:

Field	VOC CUSTOMER_SALES_PA
1	PA Book sales customer search
2	* Perform a search for sales orders for a customer
3	CS
4	DISPLAY Customer Orders Search:
5	SORT BOOK_SALES WITH SURNAME LIKE "...<<Surname>>..." _
6	OR WITH SURNAME SAID "<<Surname>>" _
7	HEADING "Customer search for <<Surname>>"
8	DISPLAY That's all folks

Paragraphs can include comment lines that begin with an asterisk. There must be a space between the asterisk and the start of the comment text.

To display messages from within a paragraph you can use the DISPLAY command. This is directly followed the text to be displayed, which does not need to be enclosed in quotation marks.

Passing Command Line Parameters

If you are going to run a parameterized command as part of a wider batch of commands, you do not necessarily want the paragraph to stop half way through and ask for details – especially if it means waiting for commands that are going to take a long time to run! Even if you do not mind running interactively, it can be neater to be able to pass a prompted value to a paragraph on the command line, rather than as the response to a prompt on the screen.

An inline prompt is made up of several elements, of which the prompt text is only one part. You can add codes to the prompt that will affect the way in which the prompt is requested, and validation to ensure that only sensible values can be accepted.

The action codes appear before the prompt text, separated from the text and from each other by means of commas. The most commonly used action codes are:

@(CLR)	Clear the screen.
@(c,r)	Place prompt at column c, row r of the screen.
A	Always prompt.
Cn	Take element n from the command line.
F(file)id,fm	take the answer from a field in a record.
In	take element n from the command line or prompt.

The C and I codes and the A code can only be used in the context of a paragraph. The others can be used in paragraphs or in sentences.

Cursor Positioning.

The @(c,r) action positions the cursor at a specified location on screen to ask for a value. This can be used when presenting multiple prompts on screen, though frankly you are better to use a PROC or Basic program if you want to control the screen layout.

Where cursor positioning is required, the @(c,r) action can be added before the prompt text, and positions the cursor using coordinates relative to the top left hand corner of the screen. The top left corner of the screen is @(0,0) whilst the bottom right is probably (79,24) depending on your terminal settings.

Do this:

You can extend the customer sales paragraph to include some simple cursor positioning:

Field	VOC CUSTOMER_SALES_PA
1	PA Book sales customer search
2	* Perform a search for sales orders for a customer
3	CS
4	DISPLAY Customer Orders Search:
5	SORT BOOK_SALES WITH SURNAME LIKE "...<<@(10,5),Surname>>..." _
6	OR WITH SURNAME SAID "<<Surname>>" _
7	HEADING "Customer search for <<Surname>>"
8	DISPLAY That's all folks

Command Parameters.

Of more practical benefit are the `C` and `I` action codes. These both allow you to supply the value for a prompt as a parameter on the command line. The `C` and `I` codes are both followed by a number which gives the element on the command line to be used. The name of the paragraph itself counts as the first element, so if you wanted to call the CUSTOMER_SALES_PA paragraph passing the surname

```
CUSTOMER_SALES_PA SMITH
```

the inline prompt for the Surname would need to specify 2 for the `C` and `I` codes.

Where the `C` and `I` codes differ from one another is in the action they will take when the requested element is missing from the command line. The `C` (Command) code will only look at the command line, and so will substitute an empty string in place of the prompt. The `I` (Interactive) code will look first at the command line, and will then prompt interactively if the element is not found.

Do This:

Change the customer search to pass the surname on the command line or to prompt for the surname if it is not present:

Field	VOC CUSTOMER_SALES_PA
1	PA Book sales customer search
2	* Perform a search for sales orders for a customer
3	CS
4	DISPLAY Customer Orders Search:
5	SORT BOOK_SALES WITH SURNAME LIKE "...<<@(10,5),I2,Surname>>..." _
6	OR WITH SURNAME SAID "<<Surname>>" _
7	HEADING "Customer search for <<Surname>>"
8	DISPLAY That's all folks

Notice that the action code does not have to be repeated for the subsequent uses of the prompt.

The **A** action code is used to ensure that the prompt is repeated each time the command is run. Normally UniVerse will only prompt once for the prompt within a single paragraph.

Lab 5.7

Enhance your AUTHOR_SEARCH so that it will accept an author name passed from the command line. Note that if the author name includes spaces, the name must be passed surrounded by double quotes. These will not be substituted into the prompt.

Opening a UniVerse File

Before you can perform any read or write operations against a UniVerse file from within Basic the file must be explicitly opened. This associates the file with a special type of variable known as a File Variable or File Pointer through which all subsequent actions against that file are performed.

A File Variable contains an internal structure giving details about the underlying file: the location of the file, the file type, modulus and separation and so forth. None of this information is directly visible – you cannot, for example, `CRT` a file variable without being presented with a sulky error message about the data type. The only things you can do with a file variable are legitimate file operations, or assignments to another file variable.

Files are opened using the `OPEN` statement:

```
OPEN filename TO filevariable {THEN|ELSE}
```

For example:

```
Open "VOC" To F.VOC Else STOP "Cannot open VOC"
```

Tip

I use upper case to distinguish special variables, global variables and constants from regular local variables so that I can see immediately that there is something different about these when scanning quickly through code.

As always you should adhere to any local standards at your workplace: one standard convention for example is that files variables are prefixed with an 'F.', as in `F.ORDERS` and `F.CUSTOMERS`.

Statement Branching

The `OPEN` statement is the first statement you have encountered in this course to offer branching.

When you tell UniVerse to open a file there are two possible outcomes: the file exists and can be opened successfully, or the file may not. You may have mistyped the name or you may not have operating system privileges on the file.

When UniVerse Basic encounters a statement that can potentially fail, UniVerse handles these situations by offering the developer two possible branches to the statement: a THEN branch and an ELSE branch. If the statement succeeds, the code will follow the THEN branch. If the statement fails, the ELSE branch is followed. It is a simple and neat form of error trapping that maintains the flow of the program and keeps the errors attached to the statements that have raised them.

The fundamental rule is that for any statement that supports branching, you can specify either a THEN clause, an ELSE clause or both. You **must** specify at least one of these branches or the compiler will reject your code.

There are two forms of syntax for the code that follows a particular branch. If the branch consists of only a single statement you can write this directly after the THEN or ELSE keyword on the same line:

```
Open "VOC" To F.VOC Else ABORT "Cannot open the VOC File"
```

The ABORT statement terminates a program printing an optional message. This is more powerful than the STOP statement and is reserved for fatal error conditions.

Single line branches can be applied to the THEN or ELSE clauses, or both:

```
Open "VOC" To F.VOC Then Crt "VOC opened" Else ABORT "Cannot  
open the VOC"
```

Single line branches are fine for simple operations such as terminating a program. In most cases, you will need to perform a range of activities in each branch. That in turn means that UniVerse needs to know where each branch will end.

Multiple line branches consist of one or more statements starting on the line following the THEN or ELSE clause and terminating with an END statement, as below:

```
Open "VOC" To F.VOC Then  
    Crt "VOC file opened"  
End
```

or:

```
Open "VOC" to F.VOC Else  
    Crt "Cannot open the VOC File"  
    STOP  
End
```

If you want to combine both THEN and ELSE clauses both branches terminate with an END. The END ELSE is usually placed together on the same line:

```
Open "VOC" To F.VOC Then
  Crt "Opened the VOC File"
End Else
  Crt "Could not open the VOC File"
  STOP
End
```

Getting File Information

File variables can be used to obtain information about a UniVerse file.

The FILEINFO() function returns a specific piece of information requested by passing a file variable into the function along with a number that determines what information should be returned.

```
Value = FILEINFO( filevariable, number)
```

The following table lists the FILEINFO() request values:

0	Valid file handle (true or false)
1	VOC name of the file
2	pathname of the file
3	File format: 1 hashed, 3 dynamic, 4 directory, 5 sequential, 7 distributed
4	Hashing algorithm used
5	Current modulus (dynamic file)
6	Minimum modulus (dynamic file)
7	Group size in 1k units
8	Large record size (dynamic file)
9	Merge load parameter (dynamic file)
10	Split load parameter (dynamic file)
11	Current load (dynamic file)
12	Node hosting file (remote file)
13	Secondary indices exist (true or false)
14	Current line number
15	Part number (distributed file)
16	Status (distributed file)
17	Recovery type
18	Recovery id
19	Fixed modulus (true or false)
20	Map used by NLS

Dynamic Array Extraction Operator

Using the `EXTRACT()` function is syntactically correct but quickly becomes painful if you are forced to perform a lot of dynamic array operations – which make up a large part of UniVerse Basic programming! Fortunately there is a shorter way of addressing array elements in Basic: by using the array extraction operator:

```
Value = dynamic_array< element_number >
```

So we can rewrite the example above as the more elegant:

```
Crt "File size is ": FileInfo<6> : " bytes"
```

You have already met this syntax before: in the chapter on descriptor expressions. The dynamic array operators and the `EXTRACT()` function that you used with the `@RECORD` system variable are the same operators used to extract elements from a UniVerse dynamic array.

Why have an `EXTRACT()` function at all when you have the dynamic array operators to hand?

Nowadays, `EXTRACT()` is very seldom used except in dictionary expressions that are compatible the UniVerse sister database, UniData, older versions of which do not support the use of the array extraction operator in dictionary expressions.

Reading Data

Dynamic arrays are important because of the way in which UniVerse reads and writes data.

Read operations take place at a record level. Reads are performed against an open file pointer using the key of the record you wish you read. UniVerse will locate the record in the file, if it exists, and return a copy of that record to the program in the form of a dynamic array.

This operation is handled by the `READ` statement.

```
READ variable FROM filevariable, key_expression {THEN|ELSE}
```

Remember that every record in a UniVerse file is identified by a unique primary key known as the 'item-id' or record key. UniVerse uses this key to identify the candidate record to read. Exactly how you establish the record key, other than knowing them all off the top of your head, is part of the work involved in UniVerse database design.

Notice that `READ` is another example of a branching statement. If the record identified by *key_expression* is found, the `READ` will branch through the `THEN` clause. If no record is found with that key, the `READ` will branch through the `ELSE` clause.

If the record is found, a copy of that record is placed into *variable* in the form of a dynamic array. In fact, the dynamic array structure exactly parallels that of a UniVerse record, allowing you to extract individual fields from the record as dynamic array elements:

Do This:

```
Open "VOC" To F.VOC Then
  Read Rec From F.VOC, "RELLEVEL" Then
    Crt "The release level of this account is ":Rec< 5 >
  End
End
```

The `READ` statement only gives you a copy of the record. The underlying file is not modified in any way, and other users can continue to list the record in the file and to read their own local copies of the same record. Opening files and reading records are shared operations: they do not give you exclusive access to the data or prevent anyone else from using it. What that means in terms of contention management will be covered later in this chapter.

If the `READ` fails, what happens to the variable receiving the record? The answer is - nothing at all. The variable remains in its original state, which for a variable that has never been given a value is an 'Unassigned' state. If you try to access an unassigned variable you will get a run-time error:

```
Variable has not been assigned a value, zero used.
```

For this reason it is normal to set the variable to an empty string before doing a `READ` operation:

```

Open "VOC" To F.VOC Then
  Rec = ""
  Read Rec From F.VOC, "RELLEVEL" Then
    Crt "The release level of this account is ":Rec< 5 >
  End
End

```

Code Warning

UniVerse programs support an implicit file variable that can be used with the OPEN, READ and some other file access statements. Using the implicit file variable is merely a case of omitting the TO and FROM clauses:

```

Open "VOC" Then
  Read Rec From "RELLEVEL" Then
    Crt "The release level of this account is ":Rec< 5 >
  End
End

```

It should go without saying that this is sloppy programming and should be avoided, particularly for larger programs that may manipulate a number of files. The overhead in typing the TO and FROM clauses don't justify this syntax.

Worse, it means that your program will compile if you accidentally forget to specify the record key or file variable in your OPEN or READ statements: but the program will not behave as expected.

So the moral of the tale is: check your OPEN and READ statements and make sure that you have completed the whole syntax for them.

Lab 3.4

Create a program called ReadTitle.

This will open the BOOK_TITLES file and request the key to a record in that file.

The program will attempt to read the corresponding record, and if found will display the short title, ISBN, department, genre, number of units, allocated units and price.

You can find the numbers of those fields in the record by listing the BOOK_TITLES dictionary. Remember that the price requires a conversion code.

```

RUN train.bp ReadTitle
Enter Title Id: ?10
Title      : Hancock a Comedy Genius (BBC Radio Collection)
ISBN      : 0563525452
Dept      : ADULT
Genre     : HUMOUR
Units     : 22
Allocated : 1
Price     : 12.99

```

Pessimistic Model

The form of locking used natively on UniVerse is the Pessimistic Model. This is more secure and can be easier to implement than the optimistic model, though the downside of the model is that it can be less scalable and does not translate easily into session-less applications such as those that are web based or disconnected.

UniVerse enables locks to be set against individual records. Within the normal application model for UniVerse, a process will signal its intent to update a record by applying for a record lock at the time the record is read. This does not prevent other processes from reading the record, but it gives that process the sole right to make changes to the record until such time as that lock is released. The model guarantees that only one process has the right to update the record, and prevents the need for complex merge processing.

This fine-grained approach is extremely secure, but depends upon each process being able to hold the lock for the duration of time that they are keeping the record. This largely rules out any disconnected form of data access, relies on operators to handle updates in a reasonable amount of time and not go off to lunch whilst holding a lock on a valued customer record, and most important of all, it relies upon the programmers to adhere to the update model.

Locking for Update

Concurrency control is established by requesting an Update lock on a record. Once a record has been locked, only the process that owns the lock has the right to write changes to the record or to delete the record from its file. If any process has a lock on any record in a file, the file cannot be cleared or deleted. The lock is a way of saying – hands off, this is mine!

You can request a lock on a record at any time by issuing a `RECORDLOCKU` (record lock for update) statement:

```
RECORDLOCKU filevariable, key [ON ERROR clause][LOCKED clause]
```

For example:

```
RecordLockU F.CUSTOMER, CustId Locked
  Crt "I'm sorry, this record is locked by user ":Status()
End
```

The `RECORDLOCKU` statement requests an Update lock on the record. If this is not successful because another process is already holding the lock, the statement will branch to the `LOCKED` clause if there is one present. If there is no `LOCKED` clause present, the code will simply wait for ever until the lock becomes available.

Record locks are held until they are released, or until the file variable to which they are attached is closed. This can happen either explicitly by issuing a `CLOSE` statement or more usually by the file variable going out of scope, such as when a program terminates. The end of a program closes all local file variables and with that releases any locks held against those file variables.

Record locks are also cleared when a session is closed gracefully, but locks can be left behind if a session terminates abnormally or if a session is not closed correctly, to the frustration of database administrators who then have to search for and release those locks.

Do This

The UniVerse editor intelligently requests a record lock on any record being edited since the normal intent of the editor is to amend the record content.

Open two dynamic connect sessions, and in one session edit a record in the `BOOK_TITLES` file, for example:

```
ED BOOK_TITLES 10
```

If you attempt to edit the same record in your second session, the editor will warn you that the record is locked and will not open the record until the first session releases that lock.

The command `LIST.READU EVERY` can be used to view the record locks in force.

```
LIST.READU EVERY

Active Group Locks:
Group                                     Record Group Group
Device.... Inode..... Netnode Userno   Lmode G-Address.  Locks ...RD ...SH
 504547614 611150963      0  1820   10 IN      7800    1    0    0

Active Record Locks:
Device.... Inode..... Netnode Userno   Lmode      Pid Item-ID.....
 504547614 611150963      0  1820   10 RU      1820 10
```

Record locking does not affect the record itself in any way. Locks are held against the file name and record id in a memory structure called the UniVerse Lock Table.

Locking on Read

Using the `RECORDLOCKU` statement to gain a record lock is very rare. In most cases, you will signal your intent to lock a record at the point when you first read it, since it is from that point forward that you need to ensure that nobody is going to change that record until you have finished with it. The normal pattern for locking records is to use a version of the `READ` statement that both reads and locks as a single action:

```
READU variable FROM filevariable, id [ON ERROR][LOCKED]
{THEN|ELSE}
```

The `READU` statement stands for 'Read for Update' and will attempt to read a record whilst at the same time requesting an update lock against the record key. As with the `RECORDLOCKU` statement above, this has an optional `LOCKED` clause which is fired if the record is already locked by another user. The `LOCKED` clause can offer the operator a choice of trying again or abandoning the update for now (or calling across to that user to release their lock if they have been distracted).

As with the `RECORDLOCKU` statement, if there is no `LOCKED` clause on the `READU` the program will simply wait until the lock becomes available. This is done silently, so can be a cause for support calls in which operators worry that their terminal has hung. It is a good idea to use the `LOCKED` clause in the first instance to tell the operator why the program is waiting:

```
Found = @False
ReadU Rec From FV, Id Locked
  Crt "Waiting for lock..."
  ReadU Rec From FV, Id Then
    Found = @True
  End
End Then
  Found = @True
End
```

Here is the pattern for updating a record extended to include concurrency control:

- Open file to a file variable.
- Determine the record key.
- Read and Lock the record using the key.
- Change the data in the local copy of the record.
- Write the local copy of the record back using the key.
- Release the update lock

Lab 7.1

The WriteTitle program performs an update against the BOOK_TITLES file, and so the title record should be locked for the duration of the program.

Add record locking to the program, and test that the lock is taken and handled successfully by opening the same book title in the editor using a second Dynamic Connect session. Where else should you be locking in that program?

Record Counters

One of the features missing from UniVerse is an auto-incrementing counter that can be used to generate record keys for inserting new records. As a result, most applications will have somewhere within them a set of one or more records that are used for counters.

If counters are used to generate unique identifiers, it is essential that these are always protected using record locks when they are accessed. Here is an example of a counter being called to generate an update:

```
ReadU Counter From FV, CounterName Else
    Counter = 0
End
Counter += 1
Write Counter On FV, CounterName
```

The update lock is taken out against the record key regardless of whether the record exists at that point, so the counter is protected even if the code follows the ELSE clause. This is often required when adding new records to a file, whether using a counter or not: it is just as essential to protect the record id for the new record as to protect an existing record that you intend to update.

Controlling the XML Format

Using Retrieve or SQL to generate XML output is quick and convenient, but does not provide a whole lot of control over the format of the XML that is produced. That is generally fine if you are both the producer and consumer of the XML document, or if you have the authority to dictate what the document format should be. Often though the format may have been defined externally, especially in the case of data passed between different parties or that needs to adhere to some industry or other standard representation.

For complex formats, and particularly for those involving significant levels of nesting, you may find that you need to resort to the programming APIs that we will be covering later in this chapter. Before automatically reaching for these, you may discover that you can handle some of the document formats by using the enquiry language in conjunction with a mapping file. This combination allows you to determine - to a limited extent - the format of the XML that will be produced and to use this either to create a fully formed document or to build a snippet that can be captured and inserted into a document template.

To define the XML format generated by a Retrieve or SQL query you must first create a mapping file. This is a document in XML format that must reside in a directory named "&XML&" in the account in which the enquiry will be run. If your account does not already contain an &XML& file, you will need to create this as a type 19 (directory) file:

```
CREATE.FILE &XML& 1,1 1,1,19
```

The mapping file defines the way in which Retrieve and SQL will treat particular fields and also the names given to the root, record and association elements in the generated document. The map can be attached to more than one enquiry statement.

Before we look at the structure of the mapping file, it is worth running a simple example to see the mapping in effect.

The following example shows a simple mapping record named 'titles_out.map' which defines the output format for a BOOK_TITLES XML listing:

```

<?xml version="1.0" encoding="utf-8" ?>
<U2XML-mapping xmlns:U2XML="http://www.ibm.com/U2-XML">
<U2XML:mapping record="TitleRec" root="TitlesInStock" schema="type" />
  <U2XML:mapping file="BOOK_TITLES" field="ISBN" treated-as="element" />
<U2XML:mapping file="BOOK_TITLES" field="@ID" treated-as="attribute"
  map-to="TitleID" />
  <U2XML:mapping file="BOOK_TITLES" field="SHORT_TITLE" map-to="Title"
    treated-as="element" />
  <U2XML:mapping file="BOOK_TITLES" field="AUTHOR_NAME" type="S"
    map-to="Author"
    treated-as="element" />
</U2XML-mapping>

```

This mapping file can be applied to an XML listing by adding the `XMLMAPPING` keyword followed by the name given to this record in the `&XML&` file.

Do this:

```

SORT BOOK_TITLES ISBN SHORT_TITLE AUTHOR_NAME TOXML XMLMAPPING
"titles_out.map"

```

```

<?xml version="1.0" encoding="UTF-8"?>
<TitlesInStock
  xmlns:U2XML="http://www.ibm.com/U2-XML">
<TitleRec TitleID = "1">
  <ISBN>0563525428</ISBN>
  <Title>Just William: No. 6 (BBC Radio Collection)</Title>
  <Author>Richmal Crompton</Author>
</TitleRec>
<TitleRec TitleID = "2">
  <ISBN>0955064007</ISBN>
  <Title>Just So Stories</Title>
  <Author>Rudyard Kipling</Author>
</TitleRec>
<TitleRec TitleID = "3">
  <ISBN>0001050478</ISBN>
  <Title>The Duchess of Malfi (abridged version)</Title>
  <Author>John Webster</Author>
</TitleRec>
<TitleRec TitleID = "4">
  <ISBN>1901768384</ISBN>
  <Title>Great Expectations</Title>
  <Author>Charles Dickens</Author>
</TitleRec>
<TitleRec TitleID = "5">
  <ISBN>9626343427</ISBN>
  <Title>The Importance of Being Earnest</Title>
  <Author>Oscar Wilde</Author>
</TitleRec>

```

If you compare this example to an XML listing of the BOOK_TITLES created without the mapping file you will notice a number of significant differences:

- The name of the record set is now TitlesInStock in place of the ROOT.
- The name of each title record is now TitleRec not BOOK_TITLES.
- The name of the @ID field is now TitleID not _ID
- The SHORT_TITLE and AUTHOR_NAME fields have been renamed.
- The TitleID has been changed into an attribute.

As you can see from that last change, the mapping file opens up the possibility of creating mixed-mode documents including both attributes and elements.

Notice however that the mapping file only controls how fields and other elements should be processed *if they appear* in the listing. It does not determine in any way what the content of the listing should be: you still get to decide at the point of building the enquiry which of those fields you want to include. The listing does not need to include all of the fields mentioned in the mapping file, and likewise any additional fields that are not in the mapping file will still be included, but will be shown using their default presentation:

```
SORT BOOK_TITLES ISBN SHORT_TITLE AUTHOR_NAME UNITS TOXML XMLMAPPING
"titles_out.map"
```

```
<TitleRec TitleID = "1" UNITS = "3">
  <ISBN>0563525428</ISBN>
  <Title>Just William: No. 6 (BBC Radio Collection)</Title>
  <Author>Richmal Crompton</Author>
</TitleRec>
```

This means that the mapping file is not specific to one listing. If you so wished, you might create a mapping file holding a sensible layout policy for each of the major files you are likely to want to query into XML to ensure that the data is reported consistently.

The mapping file itself is a regular XML document that defines the four levels of the document produced: the document root element, the record level element, association elements and field elements.

The first two of these (the document root and record) are defined using a single entry as below:

```
<U2XML:mapping record="TitleRec" root="TitlesInStock" schema="type" />
```

The `record` attribute specifies the name given to the record level elements in the generated document, which otherwise default to the file name given as part of the enquiry statement.

The `root` attribute specifies the name of the document root, overriding the default name of `ROOT`. You can also omit this element by adding a `Hideroot` attribute, which can be set to 1 to hide the document root or 0 to display it (default) as below:

```
<U2XML:mapping record="TitleRec" Hideroot="1" schema="type" />
```

This removes the document root element and the XML header to leave just the set of record level elements:

```
<TitleRec TitleID = "1" UNITS = "3">
  <ISBN>0563525428</ISBN>
  <Title>Just William: No. 6 (BBC Radio Collection)</Title>
  <Author>Richmal Crompton</Author>
</TitleRec>
<TitleRec TitleID = "2" UNITS = "6">
  <ISBN>0955064007</ISBN>
  <Title>Just So Stories</Title>
  <Author>Rudyard Kipling</Author>
</TitleRec>
```

The result of this is no longer a well-formed XML document, but this can be useful if you want to capture the XML content produced and place it into a pre-existing XML document template as a sub-element. If you need to handle complex nesting, you may be able to do this easily by creating a template document containing one or more substitution markers and then replacing those marker with the content of such a listing. Consider the following very simple example, in which an XML template is used to hold the structure of a document into which data is merged using a Basic program:

Template books.xml:

```
<?xml version="1.0" encoding="utf-8" ?>
<Goods>
  <Catalog>
    <InStock>
      {BOOK_TITLES}
    </InStock>
  </Catalog>
</Goods>
```

Program BuildTitlesXML:

```
Open "templates" to F.TEMPLATES Else
  Crt "Cannot open templates"
  STOP
End

Read Template From F.TEMPLATES, "books.xml" Else
  Crt "template missing"
  STOP
End

Cmd = 'SORT BOOK_TITLES ISBN SHORT_TITLE AUTHOR_NAME TOXML XMLMAPPING
      "titles_out.map"'

Execute Cmd, OUT. > Text
Template = Change(Template, "{BOOK_TITLES}", Text)
```

Rather than executing and capturing the statement output, there is also a helpful Basic function named `XMLExecute()` that will run a Retrieve or SQL command and return the XML for you:

```
Result = XMLExecute(Command, Options, Document, DTD)
```

where `Command` is the enquiry statement, `Options` is a field mark delimited set of XML options such as `XMLMAPPING`, `Document` holds the returned XML and `DTD` is reserved for a returned document type definition.

So the last part of the example above could be rewritten as:

```
Cmd = 'SORT BOOK_TITLES ISBN SHORT_TITLE AUTHOR_NAME'
MapFile = 'titles_out.map'

Ok = XMLExecute(Cmd, 'XMLMAPPING':@VM:MapFile, Text, '')

Template = Change(Template, '{BOOK_TITLES}', Text)
```

The field elements in the mapping file control the treatment of specific fields when they are included in the listing. Any fields that are not explicitly defined in the mapping file will still show up in the results, but will take on their default settings or use any overrides specified using keywords on the command line.

Returning to the mapping file, we now move on to the definition for each field in the listing. The field element has the following format:

```
<U2XML:mapping file="BOOK_TITLES" field="AUTHOR_NAME" type="S"  
  map-to="Author"  
  conv="MCU"  
  treated-as="element" />
```

A field in the listing is identified by the combination of the `file` and `field` attributes. This is required in particular for SQL listings that may include file joins and so needs to disambiguate possibly identical column names across more than one file, but has the unfortunate side effect that the file name in the mapping file must match exactly the file name given in the command. If you issue a command using a synonym of the file – for example using a Q pointer with an abbreviated name – UniVerse does not recognize that the same file is being referenced and so the field definitions in the map will not be applied.

The `map-to` attribute sets the name given to the attribute or element, and works in the same way as the `AS` command line keyword.

The `treated-as` attribute controls whether this field will be applied as an attribute or as an element in the document, regardless of the format specified on the command line.

The `conv` attribute allows you to override the dictionary conversion code for the field in the same way as the `CONV` keyword on the command line.

The `type` attribute controls the processing of the field as single or multivalued. This can be set to one of three values: S for single valued, MV for multivalued and MS for subvalued. If you specify "S" (single) for a multivalued field, only the last value will be reported.

Lab 4.4

Create a mapping file for the `BOOK_SALES` file to control the output of the `FORENAME`, `SURNAME` and `ADDRESS` fields.

The `FORENAME` field will appear as a `FirstName` attribute in title case.

The `SURNAME` field will appear as a `LastName` attribute in title case.

File Operations

PROCs can handle direct file operations.

These use a special series of buffers called File Buffers. These are limited in number with only 10 buffers available, each of which is associated at run time with a particular file. The pattern for file operations using a PROC is:

- A file is opened to a file buffer
- A record is read into the buffer
- The buffer contents are amended
- The buffer contents are written back to file

Real World

I would like to make it really clear at this point that doing file update operations from PROC is a generally bad idea. PROCs are not easily maintained, do not have centralized logic and because of their limited nature are prone to bypass any serious validation.

Only use updates from PROC if a) the nature of the update is truly trivial or b) as part of a bootstrap process where you may not have access to Basic.

File buffers are referenced using an ampersand followed by the buffer number. Individual elements in the buffer, which effectively equate to the fields in a record, are referenced by number following a period:

```
&buffer.element
```

For example, the second field of the record held in the first file buffer:

```
T &1.2
```

You can also use indirect buffer references to the element, though this starts to become messy:

```
T &1.%2
```

Or even:

```
T &1.%%2
```

Opening a File

Files are opened to a file buffer through the `F-OPEN` statement. This associates a file with the buffer, so that all read and write operations using that buffer will then directly affect that file.

```
F-OPEN buffer filename
Error statement
```

The `F-OPEN` statement is a two line statement which simulates the use of an `ELSE` clause. If the statement fails, the statement on the succeeding line is run. If the statement completes without error, that next line is skipped.

The file name is supplied as a literal, which should not be enclosed in quotation marks (or it will expect the quotes as part of the file name):

```
F-OPEN 1 VOC
XCannot open the VOC File
```

In rare instances the name of the file can also be given as a buffer reference, usually if writing a general purpose utility `PROC`:

```
MV %1 "VOC"
F-OPEN 1 %1
XCannot open the file
```

Reading from a File

Once the file has been opened you can read from the file into the associated file buffer, using the `F-READ` statement:

```
F-READ buffer itemname
Error statement.
```

The `F-READ` is another two line statement: if the read fails the next statement is executed, and if the read succeeds that statement is skipped.