Developer's Guide

Contents

© 2001-2013 Lastic Limited

1. Introduction

The Lastic DTD Project Manager [*L-DPM*] is a GUI development tool for the Mumps [*M*] / Caché environment, designed to ease the development and maintenance of XML generation software by automatically creating export routines based on specified DTDs.

The purpose of this *Developer's Guide* is to describe how the **Routine Generator** function of the L-DPM works, what it does, what it doesn't do, and what is required of the developer. As such, this guide will also be relevant to those who may have purchased the Routine Generator function only, and not the L-DPM itself. For information on using the L-DPM software, refer to the *Software User's Guide*.

2. Enhancing The DTD

The Routine Generator function creates XML export routines by reading and analysing the contents of DTDs. Before the function itself is discussed, it is necessary to examine what a DTD looks like and how it needs to be enhanced for the purposes of the Routine Generator. It is assumed that the reader has knowledge of DTD and XML document syntax.

The following DTD is for a simplified purchase order XML document:

```
<!-- Purchase Order DTD -->
<!-- Created on 05/06/2001 for ABC Retailers Inc -->
<! ELEMENT purchaseOrder (shipTo, shipDate, ItemList)>
<! ATTLIST purchaseOrder
       PONumber     CDATA  #REQUIRED
       customerID   CDATA  #REQUIRED
       customerName CDATA  #IMPLIED
       orderDate    CDATA  #REQUIRED>
<! ELEMENT shipTo  EMPTY>
<! ATTLIST shipTo
       street       CDATA  #REQUIRED
       city         CDATA  #REQUIRED
       state        CDATA  #REQUIRED
       zip          CDATA  #REQUIRED>
<! ELEMENT shipDate (#PCDATA)>
<! ELEMENT ItemList (Item)+>
<! ELEMENT Item EMPTY>
<! ATTLIST Item
       productNo    CDATA  #REQUIRED
       quantity     CDATA  #REQUIRED
       price        CDATA  #REQUIRED >
```

Figure 2.1 purchaseOrder.DTD

A sample XML document [*'purchaseOrder.xml'*], based on this DTD, can be found in Appendix A.

2.1. Data Structure, Data Content and Data Access

The DTD defines what data must be in the XML document and how that data must be structured [*i.e. content and structure*]. Application software used for generating the XML must therefore replicate these DTD rules, in addition to specifying how and from where the data is obtained [*data access*] . The traditional approach in developing such software has been to hardcode the document structure and content rules within the application, along with the relevant data access logic.

However, this approach is flawed for two major reasons:

- Translating DTD rules into program code is both time consuming and error prone, particularly for complex DTDs.
- Replicating DTD rules within program code increases the amount of software maintenance required when DTDs change.

The ideal scenario would be to have the program code generated directly from the DTD. This would enable the DTD document structure and content rules to be quickly and automatically converted into program code, without errors. Any subsequent changes to a DTD's document structure or content will present no major software maintenance overhead, other than having to regenerate the code based on the new DTD.

Indeed, the strictly applied and limited syntax used within DTDs makes this possible. DTD structure and content can be analysed by software and it is possible to create XML export code based upon the rules determined by this analysis. However, one major problem persists. The XML documents created by this code will not contain any data. The DTD defines what the XML data should be and how it should be structured, but not how it is obtained.

The key to automatically generating useful XML export code from DTDs is, therefore, to place the data access logic within the DTD itself. Indeed, if this were possible it would greatly simplify the management and maintenance of XML generation projects, since for any given DTD the data structure, content, and access logic could all be found and modified in one place.

2.2. Adding Data Access Logic

DTDs are used by parsing engines to validate the content of XML documents. We must be careful, therefore, to ensure that any data access logic added to a DTD does not interfere with this process - the parser is only concerned with whether the data is valid, it does not care how and from where that data was obtained. Fortunately, DTD syntax provides a mechanism for 'hiding' information from parsers - the comment tag.

Figure 2.1 illustrated both the appearance and use of comment tags within DTDs. The first two lines of the DTD are repeated in Figure 2.2 below:

```
<!-- Purchase Order DTD -->
<!-- Created on 05/06/2001 by J.Bloggs -->
....
```

Figure 2.2 Comment Tags

As within XML, HTML, and SGML documents, comments are enclosed within opening [***<!--***] and closing [**-->**] tags. These comments are usually used for human reference, such as describing the creation and revision history of the document, or for explaining code within the document.

Whilst parsers tend to ignore comment tags within a DTD, the Routine Generator function certainly does not. Indeed, the use of certain 'specialised' comment tags is necessary in order to add the required data access logic to a DTD.

But what is a 'specialised' comment tag? Well, they are not inherently different from any other comment tag. The only difference is that they contain a 3 character identifier immediately[1] after the opening tag, and the data enclosed within the tags adheres to certain rules. The 3 character identifier is used to indicate to the Routine Generator that data access logic is enclosed within the tag. Three types of 'specialised' comment tags are used. The identifiers, and their purpose, are summarised in Table 2.1 below:

Identifier	Purpose of Tag
%PM	To indicate parameters required to create the XML document
%EL	To specify the data access function name for a given element and the variable reference to be used for holding attribute values
%AF	To specify the executable code for a given data access function

Table 2.1 Summary of 'specialised' comment tag purpose

Each of these will now be examined in greater detail.

[1] Whitespace characters between the opening comment tag and the 3 character identifier is optional. At least one whitespace character must separate the identifier from all subsequent data within the comment tag.

2.3. The %PM Tag

The %PM tag is used to indicate any parameters which may be passed to the generated export routine in order to create the XML document. The tag should only appear once within a single DTD. The tag structure is as follows:

```
'<!--' S? '%PM' (S Param)* S? '-->'

       Param          ::=      Parameter Name
```

The example DTD in Figure 2.1 defines the structure and content of a simplified purchase order XML document used by a fictitious retail company. To generate such a document, an application will need to identify individual purchase orders held within the company's own database . It is likely that, within the retail company, purchase order information will be accessible by purchase order number [*PONumber in the DTD*]. The application would, therefore, pass a specific purchase order number to the export routine. The access functions *(see 2.5.)* used within the routine would then use this parameter to extract all the necessary data to generate the XML document. Figure 2.3 illustrates the inclusion of a %PM tag within the example DTD.

```
<!-- Purchase Order DTD -->
<!-- Created on 05/06/2001 for ABC Retailers Inc -->
<!--%PM poNo -->
<! ELEMENT purchaseOrder (shipTo, shipDate, ItemList)>
<! ATTLIST purchaseOrder
       PONumber      CDATA  #REQUIRED
       customerID    CDATA  #REQUIRED
       customerName  CDATA  #IMPLIED
       orderDate     CDATA  #REQUIRED>
...
```

Figure 2.3 The %PM Tag

Only one %PM tag should be declared within a DTD. When more than one %PM tag is provided, the first declaration is binding and later declarations ignored.

The above example involves only one parameter. However, the only limit to the number of parameters passed is that imposed by your M system.

Parameters should be delimited within the %PM tag by white space.

Parameters will be passed, by value, to the generated routine and coded within the routine header in the same order as they appear in the %PM tag.

If the same parameter is declared more than once in a single %PM tag, only the first declaration is binding and all subsequent declarations ignored.

Parameter names should be in accordance with the parameter naming constraints of your M system.

As an example of multiple parameter passing, suppose a company wished to generate an XML document containing all purchase orders placed within a given time period. In this instance, a 'from date' and 'to date' would be more appropriate parameters than purchase order number. The access functions could then use these dates to filter out any records where the order date does not fall between them. The tag would look as follows:

```
<!--%PM fromDate toDate -->
```

If the XML document was to only contain orders for a specific product item, then product number would be passed as a parameter too:

<!--%PM fromDate toDate prodNo -->

The %PM tag can be placed anywhere within the DTD, although it is usual to place it near the top of the document.

However, the %PM may not appear within a DTD at all. For example, suppose the same company wished to generate an XML document containing all purchase orders held within their system. Since no filtering or access of specific records is required, there is no need to pass any parameters to the export function. The access functions would simply loop through every purchase order record on the database. In this instance, the %PM tag can be left out of the DTD altogether.

2.4. The %EL Tag

The %EL tag is used to specify the data access function name for a given element and the variable reference to be used for holding that element's attribute values and PCDATA content. The tag structure is as follows:

```
'<!--' S? '%EL' S ElName S 'ACCESS=' AFName (S 'VARREF=' VRef)? S? '-->'

       ElName ::=      Element Name
       AFName ::=      Access Function Name
       VRef   ::=      Variable Reference
```

The ACCESS attribute is used to associate an element with an access function *(see 2.5.)* embedded within the DTD. This function will be used to access and format the data to be placed within the attribute values and PCDATA content of the element.

The VARREF attribute specifies the variable name to be used within the associated ACCESS function for storing the required data.

Each element within the DTD must have a %EL tag defined. The %EL tags may appear anywhere in the DTD, although it is usual to place them next to their associated element declarations.

Each %EL tag must have the 'ACCESS' attribute defined. *i.e. Every element must have a data access function specified, even if no attributes or PCDATA content are defined for it.*

Each %EL tag, where the element contains attributes or PCDATA content, must have the 'VARREF' attribute defined. *i.e. Every element which requires populating with data must have a variable reference defined in which that data can be stored by the access functions.*

Only one %EL tag is definable per element. When more than one %EL tag is declared for the same element, the first declaration is binding and later declarations are ignored.

Figure 2.4 illustrates the inclusion of the %EL tags within a portion of the example DTD. The full illustration can be found in Appendix B.

```
<!-- Purchase Order DTD -->
<!-- Created on 05/06/2001 for ABC Retailers Inc -->
<!--%PM poNo -->
<! ELEMENT purchaseOrder (shipTo, shipDate, ItemList)>
<!-- %EL purchaseOrder  ACCESS=getOrder()  VARREF=ord -->
<! ATTLIST purchaseOrder
       PONumber      CDATA  #REQUIRED
       customerID    CDATA  #REQUIRED
       customerName  CDATA  #IMPLIED
       orderDate     CDATA  #REQUIRED>
...
...

<! ELEMENT ItemList (Item)+>
<!-- %EL ItemList  ACCESS=getList() -->
<! ELEMENT Item EMPTY>
<!-- %EL Item  ACCESS=getItem()  VARREF=item -->
...
```

Figure 2.4 The %EL Tag

Access function names and variable references should NOT be enclosed within single or double quotes.

The access function name should match a corresponding %AF tag *(see 2.5.)* defined within the DTD.

Access function names declared in %EL tags must be of the same case as those in the corresponding %AF tags.

The access function name may or may not include a parameter container [*i.e. end with the '( )' characters*]. The Routine Generator will add these automatically, if omitted.

The access function name must not be longer than 8 characters, excluding any closing '( )' characters.

The access function name should not contain any parameter names within the parameter container [*e.g. 'getOrder(poNo)'*]. The Routine Generator will strip out any parameter names found.

The variable reference may be either a local or global variable, and may include subscripts. All local variable references will be automatically 'Newed' at the start of the export routine, whilst all global variables will be 'Killed' at the beginning and end.

The variable reference name should be in accordance with the variable naming constraints of your M system. In order to avoid any potential clash with system variables used by the Routine Generator function, it is recommended that any local variable references do NOT begin with a '%' character.

Variable reference names declared in %EL tags must be of the same case as used in the corresponding access functions.

Different elements should not be given the same variable reference name.

2.5. The %AF Tag

The %AF tag is used to specify the executable code for a given data access function. The tag structure is as follows:

```
'<!--' S? '%AF' S AFName S? '&#13' '&#10' (CodeLn '&#13' '&#10')+ S? '-->'

      AFName          ::=     Access Function Name
      CodeLn          ::=     Executable Line of Code
      '&#13' '&#10'   ::=     Carriage Return / Line Feed
```

The access function name is placed on the same line as the opening tag and '%AF' declaration. The closing tag is also placed on it's own line. All intermediate lines are treated as executable code, as per a standard M routine.

Figure 2.5 illustrates the use of %AF tags within a portion of the example DTD. The full illustration can be found in Appendix B.

```
<!-- Purchase Order DTD -->
<!-- Created on 05/06/2001 for ABC Retailers Inc -->
<!--%PM poNo -->
<! ELEMENT purchaseOrder (shipTo, shipDate, ItemList)>
<!-- %EL purchaseOrder  ACCESS=getOrder()  VARREF=ord -->
<! ATTLIST purchaseOrder
       PONumber      CDATA  #REQUIRED
       customerID    CDATA  #REQUIRED
       customerName  CDATA  #IMPLIED
       orderDate     CDATA  #REQUIRED>
<!-- %AF getOrder()
       n rec k ord
       s rec=$g(^POrders(poNo))
       i rec="" q 0
       s ord("at","PONumber")=poNo
       s ord("at","customerID")=$p(rec,"*",1)
       s ord("at","customerName")=$p(rec,"*",2)
       s ord("at","orderDate")=$p(rec,"*",3)
       q 1
-->
...
```

Figure 2.5 The %AF Tag

An access function must be included within the DTD for every element, even if that element has no attributes or PCDATA content.

The %AF tags may appear anywhere in the DTD, although it is usual to place them next to their associated element declarations.

The access function name may or may not include a parameter container [*i.e. end with the '( )' characters*]. The Routine Generator will add these automatically, if omitted.

The access function name must not be longer than 8 characters, excluding any closing '( )' characters.

The access function name should not contain any parameter names within the parameter container [*e.g. 'getOrder(poNo)'*]. The Routine Generator will strip out any parameter names found.

The Routine Generator does not parse any of the embedded code. It is up to the developer to ensure errors do not occur within this code, or at least that any errors can be trapped within

the application. It is recommended that this code be written using your usual M routine editor, and subsequently copied into the DTD.

Labels, goto commands, block structuring, comments, and external routine/function calls are all allowed. Syntax rules are as per your M system.

In order to avoid any potential clash with system variables it is recommended that any local variables used within the access functions do NOT begin with a '%' character.

Access functions are called as extrinsic functions [*e.g. 'if $$getorder() do...'*] by the export routines. Any QUIT commands within the function must therefore have an argument of 0 or 1 [*or a $TEST equivalent*]. '0' is used to indicate the function has failed in obtaining all the required data, whereas '1' is used to indicate the function has been successful.

2.5.1. Populating The Variable Reference

The purpose of a data access function is to locate and format the required data for a given element, placing it within the variable reference defined by the VARREF attribute of the element's %EL tag *(see 2.4.)*.

An element can contain 2 types of data: attribute values and PCDATA content. These will be discussed in turn.

2.5.1.1. Attributes

Populating attribute values is simple. Once an attribute's data has been accessed and formatted as required, it is stored in an appropriate node of the variable reference as shown below:

```
set variable reference("at",attributeName)=data
```

For a local variable reference [*e.g. ord*], this might look as follows:

```
set ord("at","customerID")=$p(rec,"*",1)
```

And for a global variable reference [*e.g. ^tmp($j,"ord")*]:

```
set ^tmp($j,"ord","at","customerID")=$p(rec,"*",1)
```

The "at" node indicates that the subnodes correspond to attributes of the element. This is necessary to separate the attribute values from any PCDATA content *(see 2.5.1.2.)*.

Ensure that the spelling of attribute names within variable references correspond exactly to their spelling within the ATTLIST declarations of the DTD.

For any attribute values which may exceed the maximum string capacity of your M system, you can break up the string by specifying further sequential sub-nodes within the variable reference:

```
set variable reference("at",attributeName,0..n)=data
```

The export routine will assemble the attribute value by concatenating the sub-nodes onto the attribute's parent node.

It is up to the developer to ensure that all required attributes are populated within the variable reference. However, where default values are defined for an attribute, the generated export routines will automatically populate the XML with these defaults if no other value is specified within the access function code.

2.5.1.2. PCDATA

Populating an element with PCDATA content is also very easy. This time, the data is stored against the "ct" node of the element's variable reference:

```
set variable reference("ct")=data
```

For a local variable reference [*e.g. shipDate*], this might look as follows:

```
set shipDate("ct")=str
```

And for a global variable reference [*e.g. ^tmp($j,"shipDate")*]:

```
set ^tmp($j,"shipDate","ct")=str
```

For any data which may exceed the maximum string capacity of your M system, you can break up the string by specifying further sequential sub-nodes within the variable reference:

```
set variable reference("ct",0..n)=data
```

The export routine will assemble the PCDATA by concatenating the sub-nodes onto the parent node.

2.5.2. Repeatable Elements

The access function illustrated in Figure 2.5. was for a non repeatable element: the root element 'purchaseOrder'. In this instance, data access was simple since it only involved accessing a single node of a database global.

But what about the access logic for repeatable elements, where each instance of the element is represented by a separate node within the database? How do we code the access logic for such elements?

Figure 2.6. illustrates how this is done for the 'Item' element of the sample DTD:

```
<!-- Purchase Order DTD -->
<!-- Created on 05/06/2001 for ABC Retailers Inc -->
<!--%PM poNo -->
<! ELEMENT purchaseOrder (shipTo, shipDate, ItemList)>
...
...
<! ELEMENT ItemList (Item)+>
<!-- %EL ItemList  ACCESS=getList() -->
<!-- %AF getList()
       s prodNo=""
       q 1
-->
<! ELEMENT Item EMPTY>
<! ATTLIST Item
       productNo     CDATA  #REQUIRED
       quantity      CDATA  #REQUIRED
       price         CDATA  #REQUIRED>
<!-- %AF getItem()
       k rec,item
       s prodNo=$o(^POrders(poNo,1,prodNo))
       i prodNo="" q 0
       s rec=$g(^POrders(poNo,1,prodNo))
       s item("at","productNo")=prodNo
       s item("at","quantity")=$p(rec,"*",1)
       s item("at","price")=$p(rec,"*",2)
       q 1
-->
...
```

Figure 2.6 Accessing data for repeatable elements

As Figure 2.6. demonstrates, the access code for repeatable elements is not too dissimilar to that for non-repeatable elements. The key to getting the logic correct is to initialise a looping variable within the access function of the parent element.

In this case, 'Item' is the repeatable element and 'ItemList' it's parent element. The access function of 'ItemList' [*$$getList()*] sets the 'prodNo' variable to null. By analysing the DTD syntax, the Routine Generator function will determine that 'Item' is a repeatable element within ItemList. Consequently, the 'Item' access function [*$$getItem()*] will be repeatedly called until the function returns a $TEST value of 0. Each time it is called, '$$getItem()' uses the 'prodNo' variable to loop to the next node of the database structure. Each time a new node of the database is accessed, an 'Item' element is created and the function returns a $TEST value of 1. When all nodes have been traversed, the function QUITS with a $TEST value of 0. This loop is then broken and the export routine continues through the rest of the document structure.

2.6. Putting It All Together

Together, the %PM, %EL and %AF tags provide a simple and flexible framework for adding data access functionality to a DTD. A full illustration of the example DTD enhanced with these tags can be found in Appendix B.

Using DTDs in this manner has a number of benefits:

- The management and maintenance of XML generation projects is greatly simplified, since for any given DTD the data structure, content, and access logic can all be found and modified in one place.

- The software developer need not be an expert in XML. The Routine Generator software takes care of the DTD syntax and document flow rules to create export routine code which is both consistent with your DTDs and error-free.

- The amount and complexity of program code required to create XML documents is greatly reduced. The developer need not worry about creating the necessary tag structures, attribute lists, or element content. All that is required is that the relevant data is accessed and placed within the specified variable references.

- Consequently, the time spent developing and maintaining your XML generation software is dramatically reduced, enabling you to meet deadlines and control costs.

So how does the Routine Generator function convert these DTDs into XML export code? This is the subject of the next section, where the function itself will be described.

3. The Routine Generator Function

The Routine Generator function creates XML export routines based upon the contents of DTDs enhanced with data access logic *(see Section 2.)*. The function is written in ANSI Standard Mumps [*1990*] and will run on all major Mumps implementations. Although it is an integral component of the Lastic DTD Project Manager [*L-DPM*], the Routine Generator function is also available separately without requiring any Lastic GUI software to be installed on your system.

3.1. Global Structure

The Routine Generator function requires that data-access-enhanced DTDs are already imported into your M database. The L-DPM provides a simple GUI-based facility for importing the files into the required global structure [*see the Software User's Guide*]. However, if the Routine Generator function, only, is installed on your system, the global needs to be populated by the developer. Figure 3.1 shows the required global structure for an imported DTD.

```
^xmlDTD(project_code, dtd_name, 0, 1)    =    DTD line 1
...
^xmlDTD(project_code, dtd_name, 0, n)    =    DTD line n
```

Figure 3.1 Imported DTD global structure

The import global structure for the sample DTD 'purchaseOrder.dtd' is shown in Appendix C.

The global subscript 'project_code' is a maximum 10-character code used to arrange and cluster DTDs into distinct project groups. A business or organisation may be involved in several different XML projects at any one time. It is possible [*though not recommended*] that a single DTD name may be used within more than one project, and that the contents of the DTD is different in each case. Using the 'project_code' subscript ensures that the Routine Generator function can easily accommodate this eventuality. The L-DPM also makes use of the concept of a project to simplify the management and maintenance of your DTDs.

The subscript 'dtd_name' is of unlimited length and is used to identify the individual DTDs within a project. Although not strictly necessary, this name should be the same as the source DTD file, minus the '.dtd' file extension. Thus, the sample 'purchaseOrder.dtd' file would be held with the 'dtd_name' subscript of 'purchaseOrder'. The L-DPM automatically extracts the 'dtd_name' subscript from the source DTD file during the import process.

A DTD name cannot be duplicated within a project. The L-DPM asks for confirmation that any existing data can be overwritten for a given DTD during the import process, but the Routine Generator function itself does not.

The Routine Generator function creates additional nodes of the ^xmlDTD global during the generation process. Since these nodes are only used internally by the function it is not necessary for the developer to understand their content or meaning. However, Appendix F provides detailed information on the global layouts.

The Routine Generator function also makes use of another global [*^xmlRTN*], which contains skeleton code particles used in generating the export routines. The function will automatically generate this global within your M system the first time it is run. Again, it is not necessary for the developer to have any knowledge of the global layouts, but details can be found in Appendix G.

One important point to note is that if the Routine Generator encounters and returns an error during the generation process *(see 3.5.)* global nodes 1-6 [*i.e. ^xmlDTD(project_code, dtd_name, 1..6)*] are deleted, leaving only the import global node 0 [*i.e. ^xmlDTD(project_code, dtd_name, 0)*].

3.2. Calling The Function

The Routine Generator function is called as shown below in Figure 3.2.

```
Set r = $$%RGEN^xmlgen(project_code,dtd_name,root_element,routine_name)
```

Figure 3.2 Routine Generator function call

For details of the 'project_code' and 'dtd_name' parameters see section 3.1. Note that the import global node [*i.e. ^xmlDTD(project_code, dtd_name, 0)*] must exist before this function is called *(see 3.1.)*, otherwise an error will be returned.

The 'root_element' parameter specifies the root element of the selected DTD. The Routine Generator makes use of this parameter in order to analyse the document flow of the DTD from the appropriate 'start point', and to ensure that the generated export code accurately reflects this flow. The specified root element must exist within the DTD, otherwise an error will be returned.

The 'routine_name' parameter specifies the name of the routine in which the generated export code will be saved within your M system. The routine name must begin with one alpha character, and may only contain alpha and numeric characters.

If the function completes successfully, the specified routine will be stored within your M system and a null value returned. If an error is raised by the function a tilde [*"~"*] delimited string containing error code, description and detail will be returned [*For more information on returned error types see section 3.5.*]. All errors are treated as 'fatal' and will terminate the function immediately without saving the routine, deleting any data which may have been generated within global nodes 1-6 of ^xmlDTD [*i.e. ^xmlDTD(project_code, dtd_name, 1..6)*].

3.3. Function Stages

When called, the Routine Generator function initially checks that the 'project_code', 'dtd_name' and 'routine_name' parameters meet the constraints described in sections 3.1. and 3.2, returning an error message and terminating if appropriate.

Otherwise, assuming no other errors are encountered, the Routine Generator function continues through 6 distinct processing stages. These will now be described in turn.

3.3.1. Stage 1 - Restructuring + Normalisation

As stated in section 3.1. the Routine Generator function requires that data-access-enhanced DTDs be stored in a sequential node structure within global [*^xmlDTD(project_code, dtd_name, 0)*]. Appendix C provides an example of this for the sample DTD 'purchaseOrder.dtd'.

Stage 1 of the Routine Generator function restructures these global nodes to ensure that the data is suitable for subsequent analysis. This includes placing the contents of each opening ['<'] and closing ['>'] tag, as well as any external parameter entity references *(see 3.3.2)*, on individual nodes, normalising white space, removing leading and trailing spaces, removing any comments not containing data access logic *(see 2.2.)*, and ensuring each node is structured such that it's contents can subsequently be read and interpreted in latter stages.

At the end of stage 1, global node [*^xmlDTD(project_code, dtd_name, 1)*] has been created, containing the restructured and normalised contents of the imported DTD *(see Appendix F)*.

3.3.2. Stage 2 - Parameter Entity Resolution

Parameter entities are used in DTDs as shortcuts. Internal parameter entities are often used to include element and attribute list declarations as groups which can subsequently be referred to as single entities. External parameter entities are used to reference other DTDs, allowing what would otherwise be extremely complex DTDs to be separated into smaller, reusable and more logical documents, which can then subsequently be 'merged' by XML parsers for validation purposes.

In order for the Routine Generator function to be able to create export routines based upon DTD content, it is essential that all parameter entities are resolved before element and attribute declarations are analysed. Stage 2 performs this task.

Firstly, all **internal** parameter entity references within global node [*^xmlDTD(project_code, dtd_name, 1)*] are resolved.

No validation of internal entity references or declarations is performed. An internal entity reference is simply substituted with the replacement text defined in it's declaration. If a reference exists where there is no corresponding declaration, no substitution takes place and the reference remains within the code. Although an error is not generated, it is likely that this will lead to the generation of erroneous export code since the element or attribute declaration will be incomplete. If a declaration exists where there are no corresponding references, the declaration is simply ignored and will have no impact on the function's subsequent processing.

Secondly, all **external** parameter entity references within global node [*^xmlDTD(project_code, dtd_name, 1)*] are resolved.

The external parameter entity declaration contains a system literal specifying the URI of the referenced DTD. The DTD name is determined from the URI [*URI minus the file path and file extension ::= referenced_dtd_name*] and a check is made to ensure that the DTD has already been imported into the current project. *i.e. that global node [^xmlDTD(project_code, referenced_dtd_name, 0)] exists*. If not, an error is raised and the function terminates. In this instance, the referenced DTD will need to be imported into the appropriate global structure and the function ran again. If the referenced DTD has already been imported, stage 1 processing *(see 3.3.1)* is performed on it with the resulting data 'merged' into the 'parent' DTD's global structure in place of the entity reference and declaration.

There is no limit to the number of external parameter entity references which may appear within a DTD, and also no limit to the number of levels of referencing. *i.e. one DTD may reference another, which may reference another, and so on*. All that is required is that the referenced DTD has already been imported into the global structure of the project containing the 'parent' DTD.

However, external parameter entity references CANNOT be recursive. *i.e. DTD 'A' references DTD 'B' which in turn references DTD 'A'*. If this is the case, an error will be raised and the function terminated.

Note that if a DTD changes, any export routines based upon 'parent' DTDs which reference it will need to be regenerated, since the Routine Generator function only makes a 'copy' of the referenced DTD. It does not maintain pointers to it.

At the end of stage 2, global node [*^xmlDTD(project_code, dtd_name, 1)*] contains the parameter-entity-resolved contents of the imported DTD *(see Appendix F)*.

3.3.3. Stage 3 - Document Content Analysis

The third stage of the Routine Generator function analyses global node [*^xmlDTD(project_code, dtd_name, 1)*] in order to determine the content of the individual ELEMENT, ATTLIST, %PM, %EL and %AF tags within it. This data is read, interpreted and stored within global node [*^xmlDTD(project_code, dtd_name, 2)*].

The content specification of each ELEMENT tag is analysed, and the sequencing, grouping and repetition of it's immediate child elements determined. Any PCDATA content declared for the element is also noted. If more than one ELEMENT tag is declared for the same element name, the first declaration is binding and later declarations ignored.

The ATTLIST declaration for each element is decomposed and the default declaration, attribute type and any defined default values are stored against the element and attribute names. When more than one ATTLIST declaration is provided for a given element, the contents of all those provided are merged. When more than one definition is provided for the same attribute of a given element, the first declaration is binding and later declarations ignored.

Parameters within the %PM tag are stored sequentially in the order in which they are declared. When more than one %PM tag is provided, the first declaration is binding and later declarations ignored. Similarly, if the same parameter name is declared more than once in a single %PM tag, only the first declaration is binding and all subsequent declarations ignored.

The VARREF and ACCESS attribute values of the %EL tag are stored against the relevant element name. Access function names are formatted as described in section 2.4. If more than one %EL tag is declared for the same element name, the first declaration is binding and later declarations ignored.

The data access code defined within the %AF tag is stored against the relevant access function name. Access function names are formatted as described in section 2.5. If more than one %AF tag is declared for the same access function name, the first declaration is binding and later declarations ignored.

See Appendix F for further details of the global [*^xmlDTD(project_code, dtd_name, 2)*] layout.

3.3.4. Stage 4 - Data Access Validation

Stage 4 of the Routine Generator function provides a limited validation of the data access functionality added to the DTD, by analysing the contents of global node [*^xmlDTD(project_code, dtd_name, 2)*].

Each element within the DTD is checked to determine whether it has a data access function declared [*in it's %EL tag*], and that the code for the declared access function is defined [*in a*

%AF tag]. If not, an error is raised and the function terminated. This is true for every element within the DTD, even if no attributes or PCDATA content are defined for it.

Each element with declared attributes or PCDATA content is checked to determine whether it has the 'VARREF' attribute defined [*in it's %EL tag*]. If not, an error is raised and the function terminated.

The Routine Generator function does NOT validate any of the data access code within the %AF tags. It is up to the developer to ensure that this code is error free.

ELEMENT and ATTLIST declarations are also NOT validated. Again, it is up to the developer to ensure the validity of these tags' contents.

3.3.5. Stage 5 - Document Flow Analysis

Stage 3 *(see 3.3.3)* of the Routine Generator function analysed the contents of each individual element within the DTD to determine the sequencing, grouping and repetition of it's immediate child elements. Stage 5 takes this analysis one stage further by merging the data for each of the individual elements such that the document flow can be traced, beginning with the root element and continuing throughout the entire element hierarchy.

Firstly, the specified root element *(see 3.2.)* is checked to ensure that it is present within the DTD. If not, an error is raised and the function terminated.

Subsequently, beginning at the root element, the entire element hierarchy is traversed to determine the element sequencing, grouping and repetition constructs which constitute the document flow. This data is subsequently used within stage 6 *(see 3.3.6)* for generating document flow code which reflects the structural rules of the DTD.

The document flow analysis also checks for the presence of element recursion. *i.e. elements which are [immediate or non-immediate] descendants of themselves.* Although used in some DTDs, element recursion is not supported by the Routine Generator function. If detected, an error is raised and the function terminated.

At the end of stage 5, global node [*^xmlDTD(project_code, dtd_name, 3)*] has been created, containing a document structure representation of the imported DTD *(see Appendix F)*.

3.3.6. Stage 6 - Export Routine Generation

The 6th and final stage of the Routine Generator function uses the data from the document content *(stage 3 - see 3.3.3.)* and flow *(stage 5 - see 3.3.5.)* analyses to generate the export routine and store it within your M system.

The document flow routine reflecting the DTD's element hierarchy is assembled 'piece by piece' using the skeleton code particles in global [*^xmlRTN*] - see Appendix G. Each node within the element hierarchy is matched against a specific code particle based upon the sequencing, grouping and repetition properties of that node. The skeleton code particles are then 'fleshed out' with the necessary block structuring, data access and output function calls, as well as with the embedded skeleton code particles of child nodes. This continues throughout the element hierarchy until all nodes have been traversed.

When complete, the document flow routine is stored in global node [*^xmlDTD(project_code, dtd_name, 5)*] - see Appendix F.

The next step is to create the overall export routine by grouping the generated document flow routine with the access functions embedded within the DTD, as well as creating the routine headers. This data is stored within global node [*^xmlDTD(project_code, dtd_name, 6)*] - see Appendix F.

Finally, the export routine is stored within the M system using a ZSAVE command on the specified routine name *(see 3.2.)*. Appendix D provides an example of a generated export routine for the sample DTD 'purchaseOrder.dtd'.

The Routine Generator function itself will automatically overwrite any existing routine with the specified routine name. The L-DPM, however, will check and ask for confirmation that an existing routine can be overwritten, before it calls the Routine Generator function.

3.4. Function Limitations

Many, but not all, of the limitations of the Routine Generator function have been discussed in previous sections. The purpose of this section, however, is to provide a single point of reference for the developer to discover what the function cannot do.

3.4.1. Recursion

Recursive external parameter entity references [*i.e. DTD 'A' references DTD 'B' which in turn references DTD 'A'*] are not supported *(see 3.3.2.)*. This is the case whether DTD 'A' is an immediate or 'distant' descendant of DTD 'B'. If this situation arises, an error will be raised and the function terminated.

Recursive element hierarchies [*i.e. where an element is an immediate or non-immediate descendant of itself*] are also not supported. This is the case whether the recursive reference occurs within a single DTD or within any descendant DTDs. In this instance, an error will be raised and the function terminated.

3.4.2. String Literal Delimiters

String literals are used within DTDs for specifying the content of internal entities, the values of attributes, and external identifiers. The XML standard allows for either the single quote ['] or double quote ["] character to be used as a delimiter for such string literals. However, the Routine Generator function requires that only the double quote ["] character be used.

3.4.3. Conditional Sections

Conditional sections are portions of a DTD which are included in, or excluded from, the logical structure of the DTD based on the keyword [*i.e. the 'INCLUDE' and 'IGNORE' tags*] which governs them.

The Routine Generator function does not support conditional sections of DTDs.

3.4.4. Well-formedness & Validity

The Routine Generator function cannot guarantee that XML documents created by the generated export routines will be either well-formed or valid, since the data is not parsed during output. The XML data for each element is simply written, to whichever device has been specified *(see 4.2.)*, immediately after the corresponding access function has been called.

It is up to the developer to ensure that all the required data for an XML document is available and/or calculable from the M database prior to calling the export routines, and that the defined access functions extract this data correctly.

3.5. Error Types

Previous sections have described the conditions under which the Routine Generator function will raise errors during processing. The purpose of this section is to provide a list of all the error types which may be encountered.

If an error is raised by the function a tilde ["~"] delimited string containing error code, description and detail will be returned. All errors are treated as 'fatal' and will terminate the function immediately without completing the generation and storage of the export routine within the M system, deleting any data which may have been generated within global nodes 1-6 of ^xmlDTD [*i.e. ^xmlDTD(project_code, dtd_name, 1..6)*].

Table 3.1 lists all error types raised by the Routine Generator function.

Error Code	Description	Detail	Reason for Error
001	Project code must be specified		A null project code has been specified within the function call parameters
002	Project code not recognised	Project code	No such project code exists [*i.e. '$d(^xmlDTD(project_code)*]
003	DTD name must be specified		A null DTD name has been specified
004	Invalid routine name	Routine name	The routine name specified is not valid [*i.e. 1 alpha followed by alphanumerics*]
101	DTD name not recognised	DTD name	No such DTD name exists [*i.e. '$d(^xmlDTD(project_code, dtd_name))*]
201	Referenced DTD not recognised	Referenced DTD name	DTD referenced by external parameter entity has not been imported into project [*i.e. '$d(^xmlDTD(project_code, dtd_name, 0))*]
202	Recursive DTD reference	Recursive DTD name	A DTD has been recursively referenced through external parameter entity declarations. [*e.g. DTD 'A' references DTD 'B' which in turn references DTD 'A'*]
401	Element access function not declared	Element name	An element has no access function [*ACCESS*] declared for it within a %EL tag
402	Element access function not defined	Element name	An element's declared access function has not been defined within a %AF tag
403	Element variable reference not declared	Element name	An element has no variable reference [*VARREF*] declared for it within a %EL tag
501	Root element not recognised	Root element	The root element specified within the function call parameters does not exist within the DTD
502	Recursive element hierarchy	Element name	An element is recursively referenced within the DTD [*i.e. an element is an immediate or non-immediate descendant of itself*]

Table 3.1. Error Types

4. The Export Routines

Section 2 described how DTDs may be enhanced with data access logic using a limited set of 'specialised' comment tags. Section 3 then described how the Routine Generator function creates XML export routines based upon the contents of these enhanced DTDs. The focus of this section is on the generated export routines themselves - what they do and how to use them within your business applications.

4.1. Calling The Routines

The generated document flow routine, along with the access functions defined for a DTD, are saved within the M system using the routine name passed as a parameter to the Routine Generator function *(see 3.2.)*. Appendix D provides an example of a generated export routine ['*xePOrder*'] for the sample DTD 'purchaseOrder.dtd'.

Figure 4.1 illustrates the structure of the document flow routine header within the export routine.

```
%ext(param_list)     ;document flow routine

       param_list    ::=    comma delimited list of
                            parameters as defined
                            within the DTD's %PM tag
```

Figure 4.1 Document flow routine header

The comma-delimited parameter list corresponds to that declared within the %PM tag of the DTD *(see 2.3.)*, with the sequence maintained. If a %PM tag is not declared within the DTD, the resulting parameter list will be null.

The document flow routine header represents the calling point for the export routine. Export routines should be called within applications using a DO command, as illustrated in Figure 4.2.

```
DO %ext^routine_name(param_list)

       routine_name  ::=    name of routine in M system
```

Figure 4.2 Export Routine Call

If no parameters are passed [*i.e. 'param_list' is null*], the parameter containers should still be included in the routine call [*i.e. ' DO %ext^routine_name()'*].

4.2. Initialisation

Before an export routine is called within an application, a number of output 'options' for the generated XML document must be set. This is done using the initialisation routine call shown in Figure 4.3.

```
DO %init^xmllib(device, .indent, .prolog)

       device ::=    Output Device ID
       indent ::=    Indentation/Line Feed Settings/Escaping
       prolog ::=    XML Document Prolog
```

Figure 4.3 Initialisation Routine Call

'xmllib' is a library routine which is shipped along with the Routine Generator function. Not only does it contain the initialisation routine, it also contains an output routine called by all generated export code. 'xmllib' must, therefore, be distributed along with the generated export routines when applications are deployed.

Sections 4.2.1. - 4.2.3. describe the three parameters of the above routine call.

4.2.1. Output Device ID

The generated export routines write data to a specified output device indicated by the device ID provided [*device*]. The device ID is stored within a variable [*'%dv'*] by the initialisation routine and subsequently referenced [*i.e. USE %dv*] by the export routines each time data is written.

Depending upon the Mumps implementation on the target system, the device ID may be a device number or file path, etc. The device does not have to be a file, it could be a TCP/IP port, a serial line, a printer or any other device which accepts serial text data.

The specified device must be open before the initialisation routine is called, and closed once the export routine has completed.

To avoid a conflict of variable names, DO NOT use a '%dv' variable within the calling application.

Upon completion of the export routine the '%dv' variable will still be set.

4.2.2. Indentation/Line Feed Settings/Escaping

By default, the export routines will generate XML documents in which an element and all it's attributes are written on a single line, without any indentation. Whilst this presents no problems for a parser, the XML is not easily read by the human eye.

To produce XML in a more human-readable form, the initialisation routine provides options for indenting subsequent nodes of the document structure. To increase readability still further, each attribute of an element may also be written to a separate line.

Furthermore, if the developer does not want to handle generating escapes for the reserved characters of <’>” or & then, setting the auto-escape parameter will cause the software to generate escapes automatically when writing the XML document.

The second parameter of the initialisation routine [*indent*] is an array, passed by reference, containing the indentation , line feed properties and escape handling requirements of the XML document.

Table 4.1. lists the definable values and what they indicate.

Node	Value	Indicates
indent(1)	'T'	elements to be indented using Tab character
	'S'	elements to be indented using Space character
	null/undefined/any other value	elements not to be indented
indent(2)	'Y'	each attribute of an element to be written to separate line
	null/undefined/any other value	every attribute of an element to be written on same line
indent(3)	“Y”	auto-detect reserved characters and generate escapes
	null/undefined/any other value	Escapes must be handled by the developer

Table 4.1. The Indentation/Line Feed Settings Parameter

Node 1 of the array is stored within a variable [*'%in'*] by the initialisation routine and subsequently referenced by the export routines each time a new element is written. Using tabbed indentation [*'T'*], each child node of the element hierarchy is indented from it's parent by a single ASCII-9 character. Spaced indentation [*'S'*] mimics tabbed indentation, but uses six ASCII-32 characters instead of the single ASCII-9 character per node indentation. Spaced indentation is useful in UNIX systems where filtering has been set on tab characters.

Node 2 of the array is stored within a variable [*'%lf'*] by the initialisation routine and referenced by the export routines each time a new attribute is written. If set to 'Y', individual attributes are written to separate lines, indented to the same position as the element to which they belong.

Node 3 of the array is stored within a variable [%es] by the initialisation routine and referenced by the export routine library whenever data (attributes or PCDATA) is to be written. If set to ‘Y’, all data is checked for the presence of reserved characters (<’>” and &) and, if detected, will be converted to escapes of < ' > " and & respectively.

To avoid a conflict of variable names, DO NOT use '%in' '%lf' or ‘%es’ variables within the calling application. Upon completion of the export routine the '%in' '%lf' and ‘%es’ variables will still be set.

4.2.3. Document Prolog

An XML document's prolog is the section of the document which appears before the root element. This usually includes the 'XML' and 'DOCTYPE' declarations.

Figure 4.4 illustrates the document prolog of a purchase order XML document based upon the sample DTD 'purchaseOrder.dtd'.

```
<?xml version="1.0"?>
<!DOCTYPE purchaseOrder SYSTEM "purchaseOrder.dtd">
...
```

Figure 4.4 The XML Document Prologue

The third parameter of the initialisation routine [*prolog*] is an array, passed by reference, containing the required prolog for the XML document. Each line of the prolog should be stored within a separate node of the array.

The initialisation routine will loop through the array and write it's contents to the specified output device *(see 4.2.1.)*. The export routine, called immediately after the initialisation routine, will then write the data for the root element, and continue through the element hierarchy.

4.3. Putting It All Together

Sections 4.2. and 4.3. described how an export routine, created by the Routine Generator function, is used within an application.

To summarise, the steps are as follows:

1. Create XML document prolog and store in array [*prolog*]
   Store required XML document indentation/line feed settings in array [*indent*]

2. Open output device [*device*]

3. Call initialisation routine [*DO %init^xmllib(device, .indent, .prolog)*]

4. Call export routine [*DO %ext^routine_name(param_list)*]

5. Close output device

Appendix D provides an example of a generated export routine ['*xePOrder*'] for the sample DTD 'purchaseOrder.dtd'. Appendix E demonstrates how this routine would be used within a calling application.

5. Distribution Files

Table 5.1. lists the routines distributed with the Routine Generator function.

Routine Name	Contains
xmlgen	Routine Generator function
xmllib	Library, initialisation and output functions

Table 5.1 Distributed Routines

If the Routine Generator function is purchased independently of the L-DPM, the above routines will be distributed within a Routine Save File.

If the Routine Generator function is purchased as part of the L-DPM software, the above routines will be shipped, along with the L-DPM routines, within a Lastic Distribution File [*see Software User's guide*].

NOTE:
'xmllib' contains library functions used by both the Routine Generator function itself, and the generated export routines. 'xmllib' must, therefore, be distributed along with the export routines when applications are deployed.

Appendix A

Sample XML document ['*purchaseOrder.xml*'] based on purchaseOrder.dtd

```
<?xml version="1.0"?>
<!DOCTYPE purchaseOrder SYSTEM "purchaseOrder.dtd">
<purchaseOrder PONumber="01458694" customerID="132ABC" customerName="ABC Retailers"
               orderDate="21/04/2001" >
      <shipTo street="100 High Street" city="Los Angeles" state="CA" zip="90934"/>
      <shipDate>25/04/2001</shipDate>
      <ItemList>
            <Item productNo="333-333" quantity="10" price="143.95"/>
            <Item productNo="444-444" quantity="6" price="24.95"/>
            <Item productNo="555-555" quantity="25" price="12.50"/>
            <Item productNo="666-666" quantity="1" price="1020.95"/>
      </ItemList>
</purchaseOrder>
```

Appendix B

purchaseOrder.dtd enhanced with data access logic

```
<!-- Purchase Order DTD -->
<!-- Created on 05/06/2001 for ABC Retailers Inc -->
<!--%PM poNo -->
<! ELEMENT purchaseOrder (shipTo, shipDate, ItemList)>
<!-- %EL purchaseOrder  ACCESS=getOrder()  VARREF=ord -->
<! ATTLIST purchaseOrder
        PONumber        CDATA   #REQUIRED
        customerID      CDATA   #REQUIRED
        customerName    CDATA   #IMPLIED
        orderDate       CDATA   #REQUIRED>
<!-- %AF getOrder()
        n rec k ord
        s rec=$g(^POrders(poNo))
        i rec="" q 0
        s ord("at","PONumber")=poNo
        s ord("at","customerID")=$p(rec,"*",1)
        s ord("at","customerName")=$p(rec,"*",2)
        s ord("at","orderDate")=$p(rec,"*",3)
        q 1
-->
<! ELEMENT shipTo EMPTY>
<!-- %EL shipTo  ACCESS=getShip()  VARREF=ship-->
<! ATTLIST shipTo
        street  CDATA   #REQUIRED
        city    CDATA   #REQUIRED
        state   CDATA   #REQUIRED
        zip     CDATA   #REQUIRED>
<!-- %AF getShip()
        n rec k ship
        s rec=$g(^POrders(poNo,0))
        s ship("at","street")=$p(rec,"*",1)
        s ship("at","city")=$p(rec,"*",2)
        s ship("at","state")=$p(rec,"*",3)
        s ship("at","zip")=$p(rec,"*",4)
        q 1
-->
<! ELEMENT shipDate (#PCDATA)>
<!-- %EL shipDate ACCESS=getDate()  VARREF=date-->
<!-- %AF getDate()
        k date
        s date("ct")=$p($g(^POrders(poNo)),"*",4)
        q 1
-->
<! ELEMENT ItemList (Item)+>
<!-- %EL ItemList  ACCESS=getList() -->
<!-- %AF getList()
        s prodNo=""
        q 1
-->
<! ELEMENT Item EMPTY>
<!-- %EL Item  ACCESS=getItem()  VARREF=item -->
<! ATTLIST Item
        productNo       CDATA   #REQUIRED
        quantity        CDATA   #REQUIRED
        price           CDATA   #REQUIRED>
<!-- %AF getItem()
        k rec,item
        s prodNo=$o(^POrders(poNo,1,prodNo))
        i prodNo="" q 0
        s rec=$g(^POrders(poNo,1,prodNo))
        s item("at","productNo")=prodNo
        s item("at","quantity")=$p(rec,"*",1)
        s item("at","price")=$p(rec,"*",2)
        q 1
-->
```

Appendix C

Import global structure for purchaseOrder.dtd

```
^xmlDTD("ordering", "purchaseOrder", 0, 1)      =       "<!-- Purchase Order DTD -->"
                                      , 2)      =       "<!-- Created on 05/06/2001 for ABC Retailers Inc -->"
                                      , 3)      =       "<!--%PM poNo -->"
                                      , 4)      =       "<! ELEMENT purchaseOrder (shipTo, shipDate, ItemList)>"
                                      , 5)      =       "<!-- %EL purchaseOrder  ACCESS=getOrder()  VARREF=ord -->"
                                      , 6)      =       "<! ATTLIST purchaseOrder"
                                      , 7)      =       "               PONumber        CDATA   #REQUIRED"
                                      , 8)      =       "               customerID      CDATA   #REQUIRED"
                                      , 9)      =       "               customerName    CDATA   #IMPLIED"
                                      , 10)     =       "               orderDate       CDATA   #REQUIRED>"
                                      , 11)     =       "<!-- %AF getOrder()"
                                      , 12)     =       "               n rec k ord"
                                      , 13)     =       "               s rec=$g(^POrders(poNo)) "
                                      , 14)     =       "               i rec="" q 0"
                                      , 15)     =       "               s ord("at","PONumber")=poNo"
                                      , 16)     =       "               s ord("at","customerID")=$p(rec,"*",1) "
                                      , 17)     =       "               s ord("at","customerName")=$p(rec,"*",2) "
                                      , 18)     =       "               s ord("at","orderDate")=$p(rec,"*",3) "
                                      , 19)     =       "               q 1"
                                      , 20)     =       "-->"
...
```

Appendix D

Generated export routine ['*xePOrder*'] for purchaseOrder.dtd

```
xePOrder    ;XML Extract Routine;Project=ordering,DTD=purchaseOrder
            ;Copyright 2001 Lastic Ltd
            ;System version 1.00;02 Aug 2001
            ;Routine version 1;02 Aug 2001
            ;
%ext(poNo)  ;document  flow routine
            n %R,item,ord,date,ship
            s %R=0,%R(%R)=1
            i $$getOrder() d
            .d output^xmllib("ordering","purchaseOrder","purchaseOrder","O",0)
            .i $$getShip() d
            ..d output^xmllib("ordering","purchaseOrder","shipTo","O",1)
            ..d output^xmllib("ordering","purchaseOrder","shipTo","C",1)
            ..s %R(%R)=0
            .i $$getDate() d
            ..d output^xmllib("ordering","purchaseOrder","shipDate","O",1)
            ..d output^xmllib("ordering","purchaseOrder","shipDate","C",1)
            ..s %R(%R)=0
            .i $$getList() d
            ..d output^xmllib("ordering","purchaseOrder","ItemList","O",1)
            ..f  q:$$getItem()=0  d
            ...d output^xmllib("ordering","purchaseOrder","Item","O",2)
            ...d output^xmllib("ordering","purchaseOrder","Item","C",2)
            ...s %R(%R)=0
            ..d output^xmllib("ordering","purchaseOrder","ItemList","C",1)
            ..s %R(%R)=0
            .d output^xmllib("ordering","purchaseOrder","purchaseOrder","C",0)
            .s %R(%R)=0
            q
            ;
getDate()   ;
            k date
            s date("ct")=$p($g(^POrders(poNo)),"*",4)
            q 1
            ;
getItem()   ;
            k rec,item
            s prodNo=$o(^POrders(poNo,1,prodNo))
            i prodNo="" q 0
            s rec=$g(^POrders(poNo,1,prodNo))
            s item("at","productNo")=prodNo
            s item("at","quantity")=$p(rec,"*",1)
            s item("at","price")=$p(rec,"*",2)
            q 1
            ;
getList()   ;
            s prodNo=""
            q 1
            ;
getOrder()  ;
            n rec k ord
            s rec=$g(^POrders(poNo))
            i rec="" q 0
            s ord("at","PONumber")=poNo
            s ord("at","customerID")=$p(rec,"*",1)
            s ord("at","customerName")=$p(rec,"*",2)
            s ord("at","orderDate")=$p(rec,"*",3)
            q 1
            ;
getShip()   ;
            n rec k ship
            s rec=$g(^POrders(poNo,0))
            s ship("at","street")=$p(rec,"*",1)
            s ship("at","city")=$p(rec,"*",2)
            s ship("at","state")=$p(rec,"*",3)
            s ship("at","zip")=$p(rec,"*",4)
            q 1
            ;
```

Appendix E

Using the generated export routine ['*xePOrder*'] within an application

```
POGen       ;generating purchaseOrder.xml document
            ;
            …
            ;get last purchaseOrder number
            s poNo=$o(^POrders(""),-1)
            …
            ;set document prolog
            s prolog(1)="<?xml version="_$c(34)_"1.0"_$c(34)_"?>"
            s prolog(2)="<!DOCTYPE purchaseOrder SYSTEM "_$c(34)_"purchaseOrder.dtd"_$c(34)_">"
            ;indentation/line feed settings
            s indent(1)="T",indent(2)="Y"
            ;open output device [ file ]
            s device="C:\XML\export\PO"_poNo_".xml"
            o device:("WNS")
            ;call initialisation routine
            d %init^xmllib(device,.indent,.prolog)
            ;call export routine
            d %ext^xePOrder(poNo)
            ;close output device
            c device
            …
```

Appendix F

Global layouts for ^xmlDTD

Field delimiter = $c(126) ['~']

Project master details

^xmlDTD(project_code)

Field	Description	Format	Comments
S	project_code	X(10)	
1	Project name	X(30)	

This node is set by the L-DPM during project creation.

If the Routine Generator function is being used independently of the L-DPM it is up to the developer whether this node is created or not. To maintain compatibility with the L-DPM [*if it may be used in the future*] it is recommended that this node is created, and that the subscript and field formats are observed.

DTD master details

^xmlDTD(project_code, dtd_name)

Field	Description	Format	Comments
S	dtd_name	X(N)	
1	Date & Time of Last Import	$H	
2	Import File Path/Name	X(N)	
3	Date & Time of Last Generation	$H	
4	Routine Name	X(8)	
5	System Version	9(N)	
6	DTD Root Element	X(N)	

Fields 1 & 2 of this node are set by the L-DPM during DTD import, and field 1 updated if the DTD is edited.

If the Routine Generator function is being used independently of the L-DPM it is up to the developer whether these fields are created or not. To maintain compatibility with the L-DPM [*if it may be used in the future*] it is recommended that these fields are created, and that the field formats are observed.

Fields 3-6 are created and maintained by the Routine Generator function. These fields will be set to null if an error is raised by the function during the generation.

Appendix F

Global layouts for ^xmlDTD

Imported DTD Content

^xmlDTD(project_code, dtd_name, 0, line_no)

Field	Description	Format	Comments
S	line_no	9(N)	
1	DTD source file line	X(N)	

This node contains the source code of the original DTD file, with each CR/LF [*$c(13,10)*] delimited line of the file on a separate *'line_no'*.

This node is created by the L-DPM during DTD import, with CR/LF characters replaced by the $c(127) character.

If the Routine Generator function is being used independently of the L-DPM it is up to the developer to populate this node before the function can be used. To maintain compatibility with the L-DPM [*if it may be used in the future*] it is recommended that CR/LF characters are replaced by the $c(127) character. This will ensure that the editing and exporting facilities of the L-DPM write out the individual lines of the DTD correctly, since the stored $c(127) character is replaced by the CR/LF characters within these functions.

Normalised DTD Content

^xmlDTD(project_code, dtd_name, 1, line_no)

Field	Description	Format	Comments
S	line_no	9(N)	
1	Normlised DTD file line	X(N)	

This, and all subsequent nodes, are created by the Routine Generator function.

The node contains a normalised and restructured copy of the original DTD file. The contents of each opening ['<'] and closing ['>'] tag, as well as any external parameter entity references, are placed on individual *'line_no'* nodes. Contiguous white space is replaced by a single space character. Leading and trailing spaces are removed, as are any comments not containing data access logic. Closing tags ['>'] at the end of each line are removed, and opening comment tags [*'<!--'*] replaced by a normal element closing tag ['>'].

Each line is structured such that it's contents can subsequently be read and interpreted by later stages of the Routine Generation function.

Appendix F

Global layouts for ^xmlDTD

Document Content Analysis

^xmlDTD(project_code, dtd_name, 2)

The %AF Tag

^xmlDTD(project_code, dtd_name, 2, "%AF", access_function_name, line_no)

Field	Description	Format	Comments
S	access_function_name	X(10)	
S	line_no	9(N)	
1	Access function code line	X(N)	

This node contains each access function embedded within the DTD, as it will appear in the final generated export routine. The access_function_name is formatted such that it ends with an empty parameter container [*e.g. getOrder()*]. The first code line [*i.e. line_no = 1*] contains the function header label. All subsequent lines contain the executable code as defined within the DTD.

The %EL Tag

^xmlDTD(project_code, dtd_name, 2, "%EL", element_name)

Field	Description	Format	Comments
S	element_name	X(N)	
1	Access function name declaration	X(10)	
2	Variable reference declaration	X(N)	

This node contains each %EL tag declaration within the DTD. The access function name declaration is formatted such that it ends with an empty parameter container [*e.g. getOrder()*].

The %PM Tag

^xmlDTD(project_code, dtd_name, 2, "%PM", sequence_no)

Field	Description	Format	Comments
S	sequence_no	9(N)	
1	Parameter name	X(10)	

This node contains any parameters declared in the %PM tag within the DTD. The sequence is as per the order in which the parameters are declared within the tag.

Appendix F

Global layouts for ^xmlDTD

The ATTLIST Tag

^xmlDTD(project_code, dtd_name, 2, "ATTLIST", element_name, attribute_name)

Field	Description	Format	Comments
S	element_name	X(N)	
S	attribute_name	X(N)	

1	Attribute Default Declaration Key	X(1)	R = #REQUIRED
			I = #IMPLIED
			F = #FIXED
			NULL = DEFAULT VALUE SPECIFIED

2	Attribute Data Type	X(N)	WHERE THE ATTRIBUTE IS AN ENUMERATED TYPE, FIELDS 2..N CONTAIN THE POSSIBLE VALUES

This node contains an analysis of each attribute's default declaration and data type declared within the ATTLIST tags of the DTD.

^xmlDTD(project_code, dtd_name, 2, "ATTLIST", element_name, attribute_name, "def")

Field	Description	Format	Comments
S	element_name	X(N)	
S	attribute_name	X(N)	
1	Attribute Default Value	X(N)	WHERE Attribute Default Declaration Key = 'F' OR NULL

If the attribute has a default value specified [*i.e. if it's Attribute Default Declaration Key = 'F' OR NULL*], the value is stored in the above node.

The ELEMENT Tag

^xmlDTD(project_code, dtd_name, 2, "ELEMENT", 0, element_name)

Field	Description	Format	Comments
S	element_name	X(N)	
1	Element Content Specification	X(N)	AS PER THE DTD

This node contains the content specification of each element, as declared within the ELEMENT tag of the DTD.

Appendix F

Global layouts for ^xmlDTD

^xmlDTD(project_code, dtd_name, 2, "ELEMENT", 1, element_name, group_no)

Field	Description	Format	Comments
S	element_name	X(N)	
S	group_no	9(N)	
1	Element Content Group	X(N)	

This node is created by separating out any grouping constructs within the content specification of the ELEMENT tag, and placing them on separate *'group_no'* nodes. Pointers are maintained beween nodes such that the content can be re-assembled.

^xmlDTD(project_code, dtd_name, 2, "ELEMENT", 2, element_name)

Field	Description	Format	Comments
S	element_name	X(N)	
1	Child Node Sequence	X(1)	C = CHOICE S = SEQUENCE

This node specifies the primary sequencing pattern of the element [*a sequential or choice group*]

^xmlDTD(project_code, dtd_name, 2, "ELEMENT", 2, element_name,
child_node_structure, ...)

Field	Description	Format	Comments
S	element_name	X(N)	
S	child_node_structure		
1	Child Node Sequence/Repeat Marker	X(2)	CHARACTER 1: C = CHOICE S = SEQUENCE CHARACTER 2: R = CHILD NODE IS REPEATABLE
2	Child Element Repeat Marker	X(1)	R = CHILD ELEMENT IS REPEATABLE
3	Child Element Name	X(N)	

This node contains a representation of the sequencing and grouping constructs of an element's immediate descendant elements, as declared within the content specification of the ELEMENT tag.

For content specifications where there is no sub-grouping of elements [*e.g. <! ELEMENT root (A,B,C)>*], the *'child_node_structure'* subscript is simply a sequence number specifying the child element's position within the declaration.

Where sub-grouping of elements is specified [*e.g. <! ELEMENT root ((A | B)*, C)>*], the *'child_node_structure'* subscript is more complex. In fact, it is not just a single subscript, but any number of them, depending upon the complexity of the element content's sub-grouping. Each sub-group node is represented by a "<G>" subscript. Sequence numbers are also used to specify element positions within the grouping constructs.

Appendix F

Global layouts for ^xmlDTD

Document Flow Analysis

^xmlDTD(project_code, dtd_name, 3, root_element_name)

Field	Description	Format	Comments
S	root_element_name	X(N)	
1	Child Node Sequence	X(1)	C = CHOICE S = SEQUENCE

This node specifies the primary sequencing pattern of the root element [*a sequential or choice group*]

^xmlDTD(project_code, dtd_name, 3, root_element_name, child_node_sequence_no, child_node_name, ...)

Field	Description	Format	Comments
S	child_node_sequence_no	9(N)	
S	child_node_name	X(N)	*'<G>'* = GROUP NODES *element_name* = ELEMENT NODES
1	Child Node Sequence/Repeat Marker	X(2)	CHARACTER 1: C = CHOICE S = SEQUENCE CHARACTER 2: R = CHILD NODE IS REPEATABLE
2	Element Repeat Marker	X(1)	R = ELEMENT IS REPEATABLE

This node contains a representation of the sequencing and grouping constructs of the entire DTD, beginning from the root element. Each node of the DTD is subscripted by it's sequence within it's parent node and it's name. Group nodes are identified by a '<G>' subscript, element nodes by the element name. The *'child_node_sequence_no'* and *'child_node_name'* subscripts are repeated until the entire document has been described.

^xmlDTD(project_code, dtd_name, 4, node_sequence_no)

Field	Description	Format	Comments
S	node_sequence_no	9(N)	
1	Node Nesting Count	X(1)	NUMBER OF NODES FROM THE ROOT ELEMENT TO THE CURRENT NODE
2	Node Name	X(N)	
3	Element Repeat Marker	X(1)	R = ELEMENT IS REPEATABLE
4	Child Node Sequence	X(1)	C = CHOICE S = SEQUENCE
5	Node Sequence	X(1)	C = CHOICE S = SEQUENCE

Appendix F

Global layouts for ^xmlDTD

Document Flow Routine

^xmlDTD(project_code, dtd_name, 5, line_no)

Field	Description	Format	Comments
S	line_no	9(N)	
1	Document Flow Routine Code Line	X(N)	

This node contains the generated docment flow routine, based on the structure rules of the DTD.

Export Routine

^xmlDTD(project_code, dtd_name, 6, line_no)

Field	Description	Format	Comments
S	line_no	9(N)	
1	Export Routine Code Line	X(N)	

This node contains the export routine as it is saved within the M system, including the generated document flow routine, embedded access functions and routine headers.

Appendix G

Global layouts for ^xmlRTN

This global is used to store skeleton code particles used during the generation of the document flow routines.

Choice, Non-Repeating Element

```
^xmlRTN(0,"E","C","NR",    1)  =  i %R(%R) i $$@F d
                           2)  =  .@EO
                           3)  =  .@
                           4)  =  .@EC
                           5)  =  .s %R(%R)=0
```

Choice, Repeating Element

```
^xmlRTN(0,"E","C","R",     1)  =  i %R(%R) f  q:$$@F=0  d
                           2)  =  .@EO
                           3)  =  .@
                           4)  =  .@EC
                           5)  =  .s %R(%R)=0
```

Sequential, Non-Repeating Element

```
^xmlRTN(0,"E","S","NR",    1)  =  i $$@F d
                           2)  =  .@EO
                           3)  =  .@
                           4)  =  .@EC
                           5)  =  .s %R(%R)=0
```

Sequential, Repeating Element

```
^xmlRTN(0,"E","S","R",     1)  =  q:$$@F=0  d
                           2)  =  .@EO
                           3)  =  .@
                           4)  =  .@EC
                           5)  =  .s %R(%R)=0
```

Choice, Non-Repeating Group

```
^xmlRTN(0,"G","C","NR",    1)  =  i %R(%R) d
                           2)  =  .s %R=%R+1,%R(%R)=1
                           3)  =  .@
                           4)  =  .s %R=%R-1,%R(%R)=%R(%R+1)
```

Choice, Repeating Group

```
^xmlRTN(0,"G","C","R",     1)  =  i %R(%R) f  d  q:%R(%R)
                           2)  =  .s %R=%R+1,%R(%R)=1
                           3)  =  .@
                           4)  =  .s %R=%R-1,%R(%R)=%R(%R+1)
```

Sequential, Non-Repeating Group

```
^xmlRTN(0,"G","S","NR",    1)  =  d
                           2)  =  .s %R=%R+1,%R(%R)=1
                           3)  =  .@
                           4)  =  .s %R=%R-1,%R(%R)=%R(%R+1)
```

Sequential, Repeating Group

```
^xmlRTN(0,"G","S","R",     1)  =  f  d  q:%R(%R)
                           2)  =  .s %R=%R+1,%R(%R)=1
                           3)  =  .@
                           4)  =  .s %R=%R-1,%R(%R)=%R(%R+1)
```