eXML
XML support module

Application Programming Interface

Author: Paul Reuvers
Version 0.42 (08 Jun 2020)
Status: Development

Preliminary Specification

Revision A - 08 Jun 2020

Disclaimer

Please note that the information given in this API is 'as is' and is subject to change without prior notice. Although every effort has been made to ensure that the information in this API is correct and complete, this cannot be guaranteed. Neither the author (Paul Reuvers) nor the company (X-Ample Technology) give any guarantee about the suitability of this information, or the software described herein, for any purpose or application, and can therefore not be held responsible for any damage, direct or indirect, arising from the use or misuse of this information and/or the software.

Paul Reuvers
X-Ample Technology
Elzentlaan 43
5611 LH Eindhoven
The Netherlands
Phone: +31 (0)40 2940297
E-mail: support@xat.nl

Latest release: http://www.xat.nl/riscos/sw/xml/

Contents

1. Introduction

The aim of the eXML module is to add support for the XML file format to RISC OS. Various ports of existing libraries, mainly from the Linux world, have become available in the past, but these have some major drawbacks:

1. The existing libraries are based on C sources written by others,
2. They are optimised for the C language,
3. Use from other popular languages, such as BBC BASIC, isn't straightforward,
4. Some existing solutions have been retracted by their author(s)

For our current range of products (including CableNews Professional), we need support for the XML format as documents and supporting files are increasingly being converted to the XML standard, making it possible to use documents across platforms. As we only need a subset of the XML library specification, only the most essential routines are implemented at this stage. Although it is our intention to make support for XML at complete as possible, this is by no means guaranteed.

1.1 Approach

Most existing libraries rely heavily on callbacks and handlers. The main program registers handlers (functions) that are called by the library's parser when appropriate. Although this approach is quite common when programming in, say, C or C++, it's not suitable for simple languages, such as BBC BASIC.

The eXML module uses a different approach, much along the lines of the RISC OS window manager (WIMP). The main program calls the required SWI (e.g. SWI "eXML_Parse") with the appropriate parameters, and the function exits with a 'return code', indicating the type of return.

Once the return code has been dealt with, the program may call the same SWI again, until it receives the 'Completed' result code, in which case it should resume the flow of the main program.

When handling long XML files, it may be desirable to call 'Wimp_Poll' frequently from within the 'eXML_Parse' loop. However, be careful when doing this, as another program may not be aware of the fact that you are still parsing the document.

1.2 Encodings

Several encodings can be used for the ASCII data in a document. The routines in the eXML module, refer to these encodings by a fixed number. Please note that some (or indeed all bar one) of these encodings may **not have been implemented** in the current release of the module. It is therefore advised, for the time being, that the standard Latin-1 encoding (ISO-8859-1) is used for documents.

The following encodings are known:

0	UTF-8
1	UTF-16
2	**ISO-8859-1**
3	US-ASCII
4	MacRoman
5	Big5

Selecting an unknown encoding (i.e.anything else than '2') will result in an 'UnknownEncoding' result code from a call to SWI eXML_Parse.

Special (i.e. reserved) characters are escaped in both read and write operations. This happens transparently, so that intervention by your code is not required. The following characters are escaped:

Sign	Description	Replaced by
<	Less than	`<`
>	Greater than	`>`
&	Ampersand	`&`
'	Apostrophe	`&apos;`
"	Quote	`&quot;`

UTF-8 Encoding

The ASCII compatible UTF-8 encoding is defined in Unicode, ISO 10646-1, and in RFC 2279.
It is currently not supported by the eXML module.

1.3 Workspace

The workspace for the eXML module must be provided by your application. This way, the module can be swapped for a newer version at any point, without crashing any running applications. Another advantage of this method is that you can have many programs running concurrently, each using their own workspace and hence preferred settings. E.g. a program in C uses '0' as a string terminator, whilst BASIC requires a '13' at the end of a string.

1.4 Stack

The eXML module keeps a stack of open elements in the workspace supplied by the application. The size of this stack is currently fixed to 1024 bytes. Each element that is opened, will be added to the stack as a null-terminated string. Pointers to the current element will be maintained by the module.

This way the module can keep track of the order in which elements are opened and closed, and it may produce an error if it finds any inconsistencies. When writing to an output stream, it may also help by closing any open elements for you, without specifying their names.

1.5 SWI Calls

A set of SWI calls is available to *create* and to *read* (parse) XML files. The advantage of using a generic set of routines in this module is that XML structures are checked for the correct syntax (i.e. opening and closing of elements) automatically. Unlike HTML, XML structures MUST be correct and the module produces an error if this is not the case. The user can therefore be certain that, when an XML file was created with the eXML module, and no error was produced when writing it, the file will be read faultlessly on any XML parser.

1.6 The module name 'eXML'

The module is called 'eXML', pronounced 'ex-em-el', just like the abbreviation 'XML'. The reason for the extra 'e' at the start of the name is that the names SWI calls can't start with the letter 'X'. The letter 'X' is used as a prefix to produce non-error returning SWI calls. In case you want to trap errors when calling one of the eXML SWIs, you should use something like: '`SYS "XeXML_StartElement",...`'.

1.7 Library

A BASIC Library is supplied in the !eXML folder. This folder should be placed on your harddisc at a position where it is seen by the filer at start-up, such as !Boot.Resources.

2. SWI Calls

The eXML module offers a set of SWI calls that can be used by any application to create and read (i.e. parse) XML files. In order to ensure that the module can be replaced dynamically, it uses a workspace buffer supplied by the calling application. A further advantage of this method is that each application (or indeed each routine within an application) can setup its own set of preferences to be followed by the eXML module.

2.1 Procedure

Before the eXML module can be used, the application needs to reserve a block of memory of suitable size. The required size for this memory block can be requested by supplying a null-pointer to the SWI eXML_Init. The workspace, as this memory block is called, will be initialised to its default values by a further call to eXML_Init.

Once this is done, the application may want to set some preferences using the SWI eXML_Options. This way the module can be tailored to each specific application. For example: each textual string returned by the module will be terminated with the ASCII 0x00 (null character) by default. This is required for programs written in C or Assembler. BASIC however, requires strings to be terminated by a character 0x0D (Carriage Return). The call eXML_Options can be used to set the required string terminator.

The available SWI calls can roughly be devided into the following groups:

1. Calls to initialise the XML workspace
2. Calls to read an XML file,
3. Calls to write an XML file
4. Universal calls.

Calls to initialise the XML workspace

SWI eXML_Init
SWI eXML_InitBuffer
SWI eXML_SetOption
SWI eXML_ClearAttributes

Calls to read an XML file

SWI eXML_ParseBuffer
SWI eXML_ParseFile
SWI eXML_Parse
SWI eXML_Context
SWI eXML_DecodeDTD
SWI eXML_Pointer
SWI eXML_DecodeString

Calls to write an XML file

SWI eXML_Open	
SWI eXML_OpenFile	alternative spelling for eXML_Open
SWI eXML_OpenBuffer	send output to a buffer instead of a file
SWI eXML_Close	
SWI eXML_UpdateBuffer	
SWI eXML_UpdateFile	
SWI eXML_Update	
SWI eXML_StartElement	
SWI eXML_EndElement	
SWI eXML_Content	
SWI eXML_Attribute	
SWI eXML_Declaration	

Continued on next page

SWI eXML_Comment
SWI eXML_DTD
SWI eXML_PI
SWI eXML_CDATA
SWI eXML_EncodeString

Universal calls

SWI eXML_GetString
SWI eXML_ColourOp
SWI eXML_DateOp Partially implemented

The rest of this chapter describes each SWI call in detail.

Possible errors

0x0081CD00	FileNotOpen
0x0081CD01	NoElement
0x0081CD02	NoEnd
0x0081CD03	WrongEnd
0x0081CD04	NoEncoding
0x0081CD05	NoDocType
0x0081CD05	Despatch
0x0081CD07	Workspace
0x0081CD08	Buffer
0x0081CD09	BufferSize
0x0081CD10	ColourUnknown
0x0081CD11	ColourSyntax
0x0081CD12	ColourFormat
0x0081CD13	ColourValue

The eXML module needs a bit of memory for use as its workspace.This workspace is used to store pointers, strings, buffers, preferences, etc. and must be supplied by the application. In order to reserve enough workspace, the application should call this SWI twice; once with a null-pointer (to find the required size) and once with a pointer to the workspace in the application's memory.

Entry

r0	-> Workspace in application memory (or 0 to request required size)
r1	Required size for the Content buffer
r5	Flags (0 for now)

Exit

r0	Required size (in bytes)

The following example in BBC BASIC illustrates the use of this call:

```
content_size% = 10*1024
SYS "eXML_InitBuffer",0,content_size%,,,,0 TO size%
DIM xml% size%
SYS "eXML_InitBuffer",xml%,content_size%
```

The first call to eXML_Init is made with a null-pointer in r0. The required size is then returned in r0. Use this size to claim a block of memory and call SWI eXML_Init again, but this time with a pointer to the memory block in r0. Once the SWI is called a 2nd time, the contents of the memory block will be cleared (i.e. filled with 0x00 bytes) and sensible defaults will be setup.

You need to specify the required size of the buffer to hold Content data in r1 on entry to this call. The module will assume a minimum of 1KB if you specify anything less than this (e.g. if you supply a null-pointer).

The pointer to the workspace (`xml%` in the above example) must be supplied in **r0** on entry to all other SWIs in the eXML module.

Remarks

1. This call superceedes the SWI eXML_Init and should be used in preference to that call.

This call is depricated and is supplied for backwards compatibility only.

Initially this call was used to initialise the XML workspace buffer in your application's memory. However, this call is limited as it assumes a fixed size for the buffer to hold the incoming Content (10KB in version 0.18 and 1KB in older versions). The SWI eXML_InitBuffer allows a more flexible initialisation of the XML workspace in your application memory and should be used in preference to this call.

Entry

r0 -> Workspace (in application memory)

Exit

r0 Required size (in bytes)

The following example in BBC BASIC illustrates the use of this call:

```
SYS "eXML_Init" TO size%
DIM xml% size%
FOR dummy% = 0 TO size% STEP 4
   xml%!dummy% = 0
NEXT
SYS "eXML_Init",xml%
```

The first call to eXML_Init is made with a null-pointer in r0. The required size is then returned in r0. Use this size to claim a block of memory and call SWI eXML_Init again, but this time with a pointer to the memory block in r0. Once the SWI is called a 2nd time, the contents of the memory block will be cleared (i.e. filled with 0x00 bytes) and sensible defaults will be setup.

The pointer to the workspace (`xml%` in the above example) must be supplied in **r0** on entry to all other SWIs in the eXML module.

Use the SWI eXML_InitBuffer in preference to this call

Once initialised, the eXML module will use sensible default values for all of its configurable parameters. It may however be neccesary to change some of these settings. The SWI eXML_SetOption allows you to set various configurable parameters as described below.

Entry

r0	-> Workspace
r1	Variable (number)
r2	Value (or pointer to string when appropriate)

The variable number in r1 can be regarded as a Reason Code.

Variable

0	Indent
1	ExtraLF
2	Terminator
3	Element Separator
4	Attribute Separator
5	FileType
6	Temp Path

Value 0 - Indent

In order to produce (user) readable XML output, indenting is used at places where elements are nested. This way, it's easier to see which elements are held by which parents, and mistakes can easily be spotted. By default, the number of indenting spaces is 2, but alternative values may be setup, using this call. Setting the number of indenting spaces to 0, turns off indenting completely.

Some examples in BBC BASIC:

(1)	`SYS "eXML_SetOption",xml%,0,3`	Set indenting to 3 spaces.
(2)	`SYS "eXML_SetOption",xml%,0,0`	Turn off indenting.
(3)	`SYS "eXML_SetOption",xml%,0,2`	Use 2 indenting spaces (default)

The result of example (3) can be this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<page type="Page" category="Test" name="1" fileType="CNDoc">
  <begin>
    <transition name="Dissolve">
      <parameters>
        <speed>Fast</speed>
        <border>False</border>
        <reverse>False</reverse>
        <sound>False</sound>
      </parameters>
    </transition>
  </begin>
  <end>
    <trigger unit="s" duration="20" endEventExist="False"/>
  </end>
  <production state="Active">
    <export active="False" forceExport="False">
    </export>
    <testDate>20041206</testDate>
    <log>
      <a d="19981202 0000" type="created" author="Sys"></a>
    </log>
  </production>
</page>
```

Value 1 - Extra LineFeed

The XML output can be further improved by inserting some extra linefeeds at the start of the higher level nodes (the StartElements of a section). By default, extra linefeeds are inserted at the first *depth* level only, but settings it to 2 or 3 may improve the readibility of the output.

Some examples in BBC BASIC:

```
(1)   SYS "eXML_SetOption",xml%,1,0          Don't insert extra lines.
(2)   SYS "eXML_SetOption",xml%,1,2          Insert extra lines up to depth level 2.
```

The result of example (2) can be this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<page type="Page" category="Test" name="1" fileType="CNDoc">

  <begin>
    <transition name="Dissolve">
      <parameters>
        <speed>Fast</speed>
        <border>False</border>
        <reverse>False</reverse>
        <sound>False</sound>
      </parameters>
    </transition>
  </begin>

  <end>
    <trigger unit="s" duration="20" endEventExist="False"/>
  </end>

  <production state="Active">
    <export active="False" forceExport="False">
    </export>
    <testDate>20041206</testDate>
    <log>
      <a d="19981202 0000" type="created" author="Sys"></a>
    </log>
  </production>
</page>
```

Value 2 - Terminator

By default, all lines of text (i.e. strings) are terminated with a 0x00 byte. This is the default terminating character for programs written in C and Assembler. BASIC programs however, may need a 0xOD byte (Carriage Return) at the end of a string. If you are writing program's in BASIC, this option can be used to change the terminating character appropriately. Please note that this change will only affect all further calls to the eXML module where **r0** contains a pointer to *your* workspace.

For example, to change the terminating character to a CR (Carriage Return), the following BASIC code could be used:

```
SYS "eXML_SetOption",xml%,2,13
```

When using the supplied XML Library, this above looks like:

```
SYS "eXML_SetOption",xml%,eXML_Option_Terminator,13
```

Value 3 - Element Separator

When reading the current context, using `SWI eXML_Context`, a string will be returned, containing all the valid element names in the correct order. The element names are separated by a '/', but another valid ASCII character may be selelected by calling eXML_SetOption,3. R3 on entry to this call may be used to specify the ASCII value of the character, or a text string may be supplied (of which only the first character will be used). Please note that you should not use a character that can appear legally in an element name. Using the default separator, a context string could look like this:

```
/document/chapter/page/frame
```

As an example, the following lines of BASIC all produce the same result; they setup the '$' as separator:

```
SYS "eXML_SetOption",xml%,3,36
SYS "eXML_SetOption",xml%,3,"$"
SYS "eXML_SetOption",xml%,3,"$%£"
```

As a result, the line from the above example of a context string would now look like this:

```
$document$chapter$page$frame
```

Value 4 - Attribute Separator

This separator works much as the Element Separator (described above). It is used to address an Attribute within the *StartElement*-tag of an Element. Consider the following bit of XML:

```
<document>
   <chapter id="1">
      <page id="23" print="True">
         Some text...
      <page>
   <chapter>
<document>
```

If we want to address the highlighted attribute (i.e. 'print') you would set the separator as follows:

```
SYS "eXML_SetOption",xml%,4,"@"
```

and then address the attribute as:

```
/document/chapter/page@print
```

Value 5 - FileType

By default, XML files created with the eXML module, will be of type &FFF (Text). This way, XML files can be easily checked using a standard ASCII text editor, such as !Edit, !StrongEd or !Zap. If you want a different file type to be used, e.g. type XML, or one of your own filetypes, an alternative value can be setup.

For example, to change the filetype to XML, the following line of BASIC could be used:

```
SYS "eXML_SetOption",xml%,5,&F80
```

Value 6 - TempPath

The eXML module may have to create a temporary file, e.g. when updating the contents of an existing XML file.This 'Temp' file can be created at various positions on the main harddisc:

```
SYS "eXML_SetOption",xml%,6,"<MyApp$Dir>.Files.Temp"
```

When undefined (default) the module tries to use the <Wimp$ScrapDir> (in which a sub-directory 'eXML' will be created). If it can't access the <Wimp$ScrapDir> it will try to use the <eXML$Dir>.Scrap directory instead.

There are several ways to read (parse) an XML file. The easiest method is to load the entire file into a buffer in your application's workspace.Of course, this can only be done if the file is small enough to fit in your memory and only if the size of the file is known. Most every-day XML files however, will be sufficiently small, so that they can be loaded in memory entirely. Once the XML file is loaded into memory, it has to be handed over to the eXML module using the SWI eXML_ParseBuffer, before it can be parsed by SWI eXML_Parse.

Entry

r0	-> Workspace
r1	-> Data
r2	Size of data (number of bytes)
r5	Flags (0 for now)

The following example in BBC BASIC illustrates the use of this call:

```
REM --- initialise xml ---

SYS "eXML_InitBuffer" TO size%
DIM xml% size%
SYS "eXML_InitBuffer",xml%

REM --- load the xml file into a buffer ---

SYS "OS_File",17,"adfs::4.$.TestFile" TO type%,,,,xml_len%
IF type% = 0 THEN ERROR 1,"No such file"
DIM xml_buf% xml_len%
OSCLI "LOAD "+xml$+" "+STR$~(xml_buf%)

REM --- present the buffer to the parser ---

SYS "eXML_ParseBuffer",xml%,xml_buf%,xml_len%
```

If the file is too large to fit into memory, or if you don't know the exact size of the file (in case of streaming data), the SWI eXML_ParseFile can be used instead. It will read the file, character-by-character, from file. Again, the SWI eXML_Parse should be called repeatidly, until parsing has finished.

Entry

r0 -> Workspace
r1 -> Filename
r5 Flags (0 for now)

Exit

r0 Handle of opened file (or 0 if not successful)

Flags (bits)

3 NoHeader Accept file without a proper <?xml?> header

Remark

1. In this mode, only the first character of the file is checked and MUST be a '<'.

The following example in BBC BASIC illustrates the use of this call:

```
REM --- initialise xml ---

SYS "eXML_InitBuffer" TO size%
DIM xml% size%
SYS "eXML_InitBuffer",xml%

SYS "eXML_SetOption",xml%,eXML_Option_Terminator,13
SYS "eXML_SetOption",xml%,eXML_Option_Separator,"/"

SYS "eXML_ParseFile",xml%,"adfs::4.$.MyFile" TO in%

REPEAT
   SYS "eXML_Parse",xml%,0 TO r%,element%,attr%,context%
   CASE r% OF
      WHEN eXML_Parser_Start
         element$ = $element%
         PRINT "S/"+$element%
         PROCparse_attr(attr%)

      WHEN eXML_Parser_End
         PRINT "E/"+$element%

      WHEN eXML_Parser_Declaration
         PRINT "D/"+$element%
         PROCparse_attr(attr%)

      WHEN eXML_Parser_ERROR
         PRINT "ERROR !!!"

      WHEN eXML_Parser_Content
         PRINT context$
         PRINT "C/"+$element%

      WHEN eXML_Parser_DTD
         PRINT "D/"+$element%

      WHEN eXML_Parser_Comment
         PRINT "c/"+$element%

   ENDCASE
UNTIL r% = eXML_Parser_Finished OR r% = eXML_Parser_ERROR

SYS "eXML_Close",xml%
END


DEF PROCparse_attr(attr%)
   LOCAL var$, val$
   IF attr% = 0 THEN ENDPROC
   WHILE !attr%
      var$ = $(!attr%)
      val$ = $(attr%!4)
      REM --- do something with var$ and val$ ---
      attr% += 8
   ENDWHILE
ENDPROC
```

Once the XML workspace has been initialised and a suitable XML buffer has been setup by using the SWI eXML_Buffer, the actual parser can be started. Parsing of the file is done by repeated calls to SWI eXML_Parse until it returns a 'Finished' code. The eXML module keeps track of the exact position within the file. If you want to restart parsing at a some point, you need to call SWI eXML_Buffer again in order to reset the pointers.

The call can produce a variety of ReturnCodes. It's up to your application to write an appropriate handler for each of them. Some ReturnCodes may not be important for you, and can be safely ignored, but you **must** respond at least to the **ERROR** and **Finished** codes so that your parsing loop is terminated. Some useful examples are given on the next pages.

Entry

r0	-> Workspace
r1	Mask
r5	Flags (see below)

Exit

r0	Return Code
r1	-> Element name [3, 5, 6]
r2	-> Array of -> Attributes (or 0 if no attributes are used)
r3	-> Context array

Mask

This field is not used at present; it should be 0 for now.

Flags (bits)

0	Context	Return -> Context string in r3
1	NoError	Don't produce an error for badly structured XML (see remark 4 below)
2	NoEscape	Don't translate escaped characters

Return Codes

0	Null	
1	Start	StartElement found
2	End	EndElement found
3	Content	Content found (r1 -> content, r2 = number of bytes to read)
4	Declaration	Declaration found (e.g: '<?xat creator="MyProg" version="4.0"?>')
5	PI	
6	Entity	
7	Comment	
8	CDATA	
9	DTD	DOCTYPE definition found (see also SWI eXML_DecodeDTD)
10	Finished	This code is returned when the parser has finished (r1 = size) [6]
11	ERROR	An error has occurred during parsing (r1 = row, r2 = column)
12	Encoding	

Remarks

1. Self-closing elements (e.g. <colour type="rgb" value="255,255,255" /> always return a StartElement code followed by an EndElement code, so that your code can keep track of the nesting level itself.
2. Return code 0 (Null) is used to allow multitasking to continue whilst parsing large files. Respond to this return code by calling WIMP Poll.
3. Return code 3 (Content) gives a pointer to the content in r1, whilst r2 specifies the number of bytes to be read. The content will always have a suitable terminator at the first byte after the data block, so that simple (i.e. short) strings can be read directly.

Continued on the next page

4. Setting bit 1 of the flags (r5) suppresses most errors when parsing a file. Never set this bit when parsing an

XML file, as the file won't be checked for errors. It may however be useful when parsing simple HTML structures, as it might contain improper element nesting.

5. Whenever an ERROR is returned, r1 and r2 will contain the *row* and *column* of the element that was last encountered before the error occured. It can be used to determine the position in the XML file prior to the error.
6. From version 0.22 onwards, r1 returns the number of bytes read if the Return Code in r0 is 10 (Finished).

Element

On exit, r1 contains a pointer to the current element name (if appropriate). This name is terminated with the default terminator as set by SWI eXML_SetOption,2.

Attributes

On exit, r2 may contain a pointer to an array of double words, each of which holds a pointer to a attribute name and an attribute value respectively. The list is terminated by a null pointer. If no atrributes were found, a 0 is returned in r2 (i.e. a null-pointer). Consider, for example, the following line of XML code:

```
<frame id="1" name="time and date">
```

On exit of the SWI eXML_Parse, r1 will contain a pointer to the element name 'frame', whilst r2 contains a pointer to the following array:

r2	**pointer**
+0	-> 'id'
+4	-> '1'
+8	-> name
+12	-> 'time and date'
+16	0
+20	0

Context array

On exit, r3 contains a pointer to a zero-terminated array of word-pointers. Each of the words contains a pointer to the name of an element, in the same order as the elements have been opened. The list is terminated by a null-pointer. If, for example a new element is opened, whilst we are at level document/chapter/page, r3 (on exit) will point to the following array:

r3	**Pointer**
+0	-> 'document'
+4	-> 'chapter'
+8	-> 'page'
+12	0

Context string

If bit 0 of r5 is set on entry, r3 will contain a pointer to a string on exit, rather than a pointer to an array. This allows direct checking of the current context. Thus, the above example would result in the following string:

```
/document/chapter/page
```

Content

When content is found, r1 contains a pointer to the data, whilst r2 specifies the number of bytes to read. The data may be presented as a single block, but it may also be scattered over a series of blocks, subject to the available buffer space and the transport medium used. Your application should be able to cope with a series of blocks.

DTD (Document Type Declaration)

A valid XML document should contain a DTD (Document Type Declaration) so that it's context and syntax can be verified against a set of rules. Although DTD is gradually being phased out in favour of the XML S*chema*, most files still use DTD to identify the *document-type* (much like the file-type under RISC OS). A DTD can be found at the beginning of an XML file and looks like this:

```
<!DOCTYPE svg PUBLIC "-//W3C//SVG//EN" "http://www/w3c/org/dtd/svg.dtd">
```

However, the !DOCTYPE line can take many forms and is discussed in more detail in the description of the SWI eXML_DecodeDTD. On exit of the call to SWI eXML_Parse, r1 contains a pointer to the undecoded data (the part between the opening and closing brackets). If you want, you can decode this string yourself, but you can also use the SWI eXML_DecodeDTD to filter off the various parts for you.

Example of a simple parser in BBC BASIC:

```
REM --- initialise xml workspace ---
...
SYS "eXML_ParseBuffer",xml%,xml_buf%,xml_len%

REM --- start parser ---

REPEAT
   SYS "eXML_Parse",xml%,0 TO r%,element%,attr%
   CASE r% OF
      WHEN eXML_Parser_Start
         PRINT "S/"+$element%
         PROCparse_attr(attr%)

      WHEN eXML_Parser_End
         PRINT "E/"+$element%

      WHEN eXML_Parser_Declaration
         PRINT "D/"+$element%
         PROCparse_attr(attr%)

      WHEN eXML_Parser_ERROR
         PRINT "ERROR !!!"

      WHEN eXML_Parser_Content
         PRINT "C/"+$element%

      WHEN eXML_Parser_DTD
         PRINT "D/"+$element%

      WHEN eXML_Parser_Comment
         PRINT "c/"+$element%

   ENDCASE
UNTIL r% = eXML_Parser_Finished OR r% = eXML_Parser_ERROR
END

DEF PROCparse_attr(attr%)
   LOCAL var$, val$
   IF attr% = 0 THEN ENDPROC
   WHILE !attr%
      var$ = $(!attr%)
      val$ = $(attr%!4)
      REM --- do something with var$ and val$ ---
      attr% += 8
   ENDWHILE
ENDPROC
```

The above program show a REPEAT-UNTIL loop in which the SWI eXML_Parse returns a series of Return Codes. Each of the Return Codes that we want to see, produces a line of text. Return Codes that are of no importance to use, are simply ignored. The loop finished if it receives an ERROR or Finished code.

By default, a call to eXML_Parse returns a pointer in r3 to a zero-terminated array of pointers. Each of the elements in this array points to a text string containing the name of the element. There may be situations however, where you want to view the current context in a user-readable format (i.e. a single text string). Such a string is created by SWI eXML_Context. Imagine a situation whereby you receive Content from the SWI eXML_Parse and you ask yourself the question: "What series of Elements does this Content belong to?"

Entry

r0 -> Workspace
r1 Separator (or 0 to use default separator '/')

Exit

r0 -> Context string

Using the default settings, the following:

```
REPEAT
   SYS "eXML_Parse",xml% TO r%,element%,attr%
   SYS "eXML_Context",xml%,"/" TO context$
   PRINT context$
UNTIL eXML_Parser_Finished
```

would produce a series of strings, e.g.:

```
/document
/document/chapter
/document/chapter/page
/document/chapter/page/frame
/document/chapter/page
/document/chapter
/document
```

Remarks

1. The string returned in r0 is terminated with the configured terminator. The default terminator is 0x00, as used by C programs, but BASIC requires a 0x0D character to be used instead. Please refer to SWI eXML_SetOption (2) to set the preferred terminator for your session. This won't affect any other instantions of XML, as each session is using it's own workspace (pointer in r0).
2. Note that when parsing a StartElement or EndElement, the name of that element is not included in the context string (simply because at that point it isn't part of the context yet). If you want to see the full path, you should create your own string, e.g.: `context$ = context$ + "/" + $element%`

2.9 SWI eXML_DecodeDTD &576d7

A valid XML document should contain a DTD (Document Type Declaration) somewhere at the start of the file. If it doesn't contain it can still be a well formatted XML document, but it is *not* considered *valid*. In theory, a DTD is used to help an XML parser to varify the syntax of a file, by checking its contents against a set of rules, outlines in a DTD file. Such a DTD file can exist externally, but can also be embedded inside the DTD line (the latter is currently not supported by this module).

Because the syntax of the DTD is so arbitrary, the SWI eXML_DecodeDTD can be used to decode the line into several, more useful, parts.

Entry

r0	-> Workspace
r1	-> Contents of the DTD line (as returned from SWI eXML_Parse)
r5	Flags (should be 0 for now)

Exit

r0	-> Return code (0 if OK, else ERROR)
r1	-> Document-type (text string containing no spaces)
r2	Identifier string
r3	URL
r4	Undefined at present (reserved for future use)
r5	Updated Flags

Flags (r5 on exit)

bit	Name	Description
4	Public	When set, declaration if local (SYSTEM) rather than PUBLIC.

Remarks

1. When creating DTD type data, your are encourages to use the SWI eXML_DTD, as this will ensure the correct syntax. You simply supply pointers to the various text strings and the eXML module does the rest.
2. At present there is no support for embedded (in-line) Data Type Declarations. As these will be phased out in favour of the now more common XML Schemas, it is unlikely that support for this type of DTD will be implemented at all.
3. The URL should be a reference to either a local DTD file (SYSTEM) or an external one (PUBLIC), Please note that the presense of such a DTD file is not commonly checked, so you can put basically anything in there. It is for the sole use of XML syntax checkers and automatic application builders. When specifying a DTD file, it is common to use a filename with a `'.dtd'` extension.

Usage

A DTD line should contain a pointer to a DTD file, which may either be present locally on the computer, in which case it's called **SYSTEM**, or globally on a publicly accessible place, such as the internet, in which case it's called **PUBLIC**. As a bare minimum, the line should contain the DTD Element name '!DOCTYPE' followed by the *document-type* ('svg' in the first example below), the DTD-*type* (PUBLIC or SYSTEM) and a URL (either global or local). DTD lines can take many forms as shown in the following examples:

1. Public DTD (referring to a public URL where the actual DTD file can be fetched:

```
<!DOCTYPE svg PUBLIC "-//W3C//SVG//EN" "http://www.w3c.org/dtd/svg.dtd">
```

2. Local DTD (the URL refers to a local file on the machine running the software):

```
<!DOCTYPE mydoc SYSTEM "files/doctype/mydoc.dtd">
<!DOCTYPE myapp:doc SYSTEM "-//MyApp//Document//EN" "mydoc.dtd">
```

   it may also be spread over several lines, e.g.:

```
<!DOCTYPE svg PUBLIC
   "-//W3C//SVG//EN"
   "http://www.w3c.org/dtd/svg.dtd"
   >
```

3. Embedded DTD (in-line declarations):
   This is currently not supported by the module.

Don't worry too much about the use of the URL and the Identifier. Although the URL is mandatory, neither of the fields are generally checked for validity. The format of the Idendifier string is not defined. The W3C commitee have tried to create a string in a more or less platform independant format, but the exact syntax and use of the string has never really been formalised.

This SWI should be used to open to create a new XML file. Once opened, further SWI calls can be used to write output to the XML file and the eXML module will keep track of the syntax and the correct opening and closing of elements, attributes, content, etc. Once complete, you should use SWI eXML_Close to close the file.

As part of the opening procedure, the SWI eXML_Open will write an appropriate XML header on the first line of the file. The file will therefore have at least **<?xml?>** on the first lines, indicating that it is a genuine XML file. This line is called an XML *declaration*. Attributes can be added to further specify to the XML declaration. As a minimum, it is suggested to add the *version* and *encoding* attributes to this lines, e.g.

```
<?xml version="1.0" encoding="UTF-8"?>
```

The attributes can be specified in the same manner as for a normal element (see SWI eXML_StartElement), but as both the version and encoding attributes are more or less mandatory, they may also be specified directly on entry to the call.

The call returns the handle of the opened file (or 0 if it was not successful), so that your application can keep track of it and close it in case of an error.

Entry

r0	-> Workspace
r1	-> File name
r2	-> Array holding -> Attributes (optional)
r3	-> Version string (optional)
r4	-> Encoding string (optional)
r5	Flags (see below)

Exit

r0	File handle (or 0 of not successful)

Flags (bits)

0	NoXML	Do not write the <?xml?> header

Remarks

1. On entry to the call, r2, r3 and r4 contain pointers to additional data. These pointers are optional and may be omitted. If they are not supplied, these registers should contain 0.
2. Further information about creating an Attribute array, can be found on the page describing SWI eXML_Attribute, further down this manual.
3. If only the version and encoding attributes are needed, pointers to the respective strings can be supplied in r3 and r4 and no further attributes need to be specified (i.e. r2 = 0).

Examples

Some examples in BBC BASIC:

```
SYS "eXML_Open",xml%,"adfs::4.$.TestFile",0,"1.0","UTF-8"
```

will produce this:

```
<?xml version="1.0" encoding="UTF-8"?>
```

But the following sequence produces exactly the same result:

```
SYS "eXML_Attribute",xml%,"version","1.0"
SYS "eXML_Attribute",xml%,"encoding","UTF-8" TO attr%
SYS "eXML_Open",xml%,"adfs::4.$.TestFile",attr%
```

2.11 SWI eXML_OpenFile

Alternative spelling for SWI *e*XML_*Open*.

2.12 SWI eXML_OpenBuffer

This SWI works in the same manner as the previous one (eXML_OpenFile), but causes any further output to be sent to a buffer in memory, rather than to an open file. This can be useful, for example, when creating headers for streaming communication (e.g. internet streaming data) that does not require a file to be saved to disc first. On entry to the call, r1 should contain a pointer to a buffer in memory, the first word of which contains the size of the buffer.

Entry

r0	-> Workspace
r1	-> Buffer for holding the data (first word should contain the size)
r2	-> Array of attributes (or 0 if no attributes are required)
r3	-> Version
r3	-> Version string (optional)
r4	-> Encoding string (optional)
r5	Flags (see below)

Exit

r0	Error code (0 if OK)

Flags (bits)

0	NoXML	Do not write the <?xml?> header

This SWI is used to close an open (i.e. just created) XML file. The SWI closes the current file (if there is one) and clears its pointers. It also sets the filetype of the created file to the currently configured filetype. By default, the file will be typed as "Text" (&FFF), but you might want to change this to another filetype. The latter is only used when creating an XML file. It is ignored when reading an XML file.

Entry

r0 -> Workspace
r1 FileType (or 0 to use configured filetype)

Exit - if OpenBuffer was used to create the XML file (otherwise all registers preserved)

r0 -> Buffer
r1 Number of bytes written
r2 Size of buffer in r0

2.14 SWI eXML_UpdateBuffer &576c8

Not yet implemented.

In some situations you may need to update only part of an existing XML file. This can be the case, for example, when an XML is updated by various applications. The SWI eXML_UpdateFile allows you to open an existing file, scan for the required element, insert your own data, skip the required element, and continue. Once a file is opened, you should call SWI eXML_Update repeatidly until parsing is finised; much in the same way as with the standard SWI eXML_Parse.

Entry

r0 -> Workspace
r1 -> Filename

Exit

r0 Filehandle of input file (or 0 of not successful)
r1 Filehandle of output file (or 0 of not successful)

Related SWIs

eXML_UpdateBuffer
eXML_Update

The following example in BBC BASIC illustrates the use of this call:

```
REM --- initialise xml ---

SYS "eXML_Init" TO size%
DIM xml% size%
SYS "eXML_Init",xml%

SYS "eXML_SetOption",xml%,eXML_Option_Terminator,13
SYS "eXML_SetOption",xml%,eXML_Option_Separator,"/"

SYS "eXML_UpdateFile",xml%,"adfs::4.$.MyFile" TO in%,out%
PRINT "in "+STR$(in%)+", out "+STR$(out%)

claim% = 0
REPEAT
   SYS "eXML_Update",xml%,,,,,claim% TO r%,,,context$
   claim% = 0
   CASE r% OF

   WHEN eXML_Parser_Start
      CASE context$ OF
      WHEN "/document/chapter/page"
         REM --- insert new XML elements here ---
         claim% = 1
      ENDCASE

   WHEN eXML_Parser_ERROR
      PRINT "!!! Parser ERROR !!!"

   ENDCASE
UNTIL r% = eXML_Parser_Finished OR r% = eXML_Parser_ERROR

SYS "eXML_Close",xml%,FileType_XML
```

After opening an XML file for update, using the SWI eXML_UpdateBuffer or eXML_UpdateFile, you should call SWI eXML_Update in a loop until parsing has finished. This mechanism works in the same way as with the SWI eXML_Parse. The first time this SWI is called, bit 0 of r5 has to be clear.

Once you've found an URI (i.e. a path of elements) that matches the requested URI, you should set bit 0 of r5 on your next call to the SWI eXML_Update. This will cause the current element, and all elements within it, to be skipped. No further elements will be reported to you, until the current one has closed.

Entry

r0	-> Workspace
r1	Mask (0 for now)
r5	Flags (see below)

Exit

r0	Return code (see below)
r1	-> name of last element
r2	-> Array of Attributes of last element
r3	Expanded context string (see below)

Flags (bits)

0	Claim	Claim previous element
1	ClaimTag	Claim only the opening tag [1])

Return codes

Unlike the SWI eXML_Parse, this SWI doesn't return all return codes. Instead, it only returns to the caller when it is neccesary. The following return codes can be expected:

1	Start	StartElement found
10	Finished	This code is returned when the parser has finished
11	ERROR	An error has occurred during parsing

Expanded context string

On exit this call returns an expanded context string in r3. This allows quick checking to see if you want to take some action. The string is expanded, which means that it includes the name of the current element (unlike the SWI eXML_Parse).

The example on the previous page illustrates the use of this SWI.

Remarks

1. When bit 1 of the flags (r5) is set on entry to this call, Only the opening tag of this element will be claimed. This allows the opening tag to be replaced by an updated one.

2.17 SWI eXML_StartElement &576cb

This SWI can be used to insert a new element in the output XML file. A StartElement should be followed by (optional) content, (optional) further elements and finally an EndElement. An Element can be made empty (e.g. when it only contain attributes), in which case the EndElement is omitted and a '/' is inserted before the closing '>' bracket. Such empty elements are also known as 'self-closing elements'.

Entry

r0	-> Workspace
r1	-> Element name
r2	-> Array of Attributes (optional)
r3	-> Content string (optional)
r5	Flags

Exit

r0	Result code: 0 if successful, else -> error block

Flags

0	Empty	This is an empty element (i.e. self-closing)
1	NoLF	Don't insert a LF at the end of this tag
2	NoIndent	Don't indent this element (i.e. no spaces before the tag)
3	NewLine	Start each attribute on a new line (improved readability)

Empty elements

You may specify an element to be self-closing, in which case it cannot contain any content. It can contain attributes however. If you mark an element to be empty (i.e. self-closing), a '/' will be inserted before the end of the element tag and you should **not** add an EndElement by calling SWI eXML_EndElement. The resulting XML code will look like this:

```
<colour red="255" green="128" blue="44"/>
```

Note the inserted '/' before the '>' at the end of the element. You may mark an element as being self-closingby:

a) setting bit 0 of r5 on entry to this call, *or*
b) prefix the name of the element (in r1) with a '/', e.g. r1 -> '/colour', *or*
c) provide a string in r3

In these cases you should not add an EndElement, as this will be done for you automatically.

Optional content string

A single (short) line of text can easily be inserted as content by supplying a pointer to it in r3 on entry to the call. As a result, the StartElement will be created, the content will be inserted and the EndElement will also be created. So there is no need, in this case, to add an EndElement. For example, this:

```
SYS "eXML_StartElement",xml%,"comment",attr%,"Just a line of text."
```

will produce this in the output XML file:

```
<comment>Just a line of text.</comment>
```

If more than one line of text is needed, omit the string in r2, and issue the SWI **eXML_Content** between a SWI eXML_ElementStart and SWI eXML_ElementEnd.

This call is complementary to the previous one. It is used to close an element that was opened by a call to SWI eXML_StartElement. Note that each element that was opened with a call to eXML_StartElement, **must** be closed by a call to eXML_EndElement, except when the element was *empty* (see above). Elements must also be closed in the correct order when nested. Overlapping is not possible in the XML standard.

When closing an element, the eXML module verifies the specified element name agains its internal stack and an error will be produced when trying to close the wrong element. Rather than specifying the element name in r1, you may use '0' to close the most recently opened element. This is however not recommended as the eXML module won't be able to verify the correct order of opening and closing elements.

Entry

r0	-> Workspace
r1	-> Element name (or 0 to close the most recently opened element)
r5	Flags

Exit

r0	Result Code (0 if successful, otherwise -> error block)

Flags (bits)

1	NoLF	Don't insert a LF (new line) after this element
2	NoIndent	Don't use indenting (i.e. white spaces before elements)

Possible errors

&0081CD00 - The XML file is not open
&0081CD02 - Trying to close more elements than have been opened
&0081CD03 - Trying to close the wrong element

This example illustrates the correct opening and closing sequence:

```
<document>
   <page id="1">
      <frame>
         <text>Example</text>
      </frame>
   </page>
</document>
```

Whilst this document will produce an error message:

```
<document>
   <page id="1">
      <frame>
         <text>Example</text>
      </page>
   </frame>
</document>
```

This call is used to insert data (a.k.a. content) inside an element. Please note that you should only insert content once an element has been opened. Single lines of text can easily be inserted by supplying them in r3 when calling SWI eXML_StartElement, but when inserting larger blocks of text (or multiple lines) you should use eXML_Content instead. It allows any number of bytes to be inserted. Please note that content may contain NewLine codes (LF) and/or carriage return characters (CR), which *are* taken into account (unlike in HTML).

Entry

r0	-> Workspace
r1	-> Data
r2	Number of bytes to write
r5	Flags

Exit

r0	Result Code (0 if successful, otherwise -> error block)

Flags (bits)

1	NoLF	Don't insert a LF (new line) after the data
2	NoIndent	Don't insert white spaces before the output
6	NoEnc	Don't translate special characters

Possible errors

&0081CD00 - File not open

The following example illustrates the position of *content* inside an XML body:

```
<document>
   <page>
      <frame id="1">This is content</frame>
   </page>
</document>
```

2.20 SWI eXML_Attribute &576ce

When opening an element using the SWI eXML_StartElement, you may specify a pointer to an (optional) array of attributes in r2. The exact syntax of the attribute array is described in the paragraph that deals with the SWI eXML_Parse. In short: the array is a list of double (32-bit) words, terminated by two null-words. Each entry consists of a pointer to the attribute name and a pointer to the value of that attribute.

An attribute takes the form of:

```
name="date and time"
```

In this case, '`name`' is the name of the attribute and '`date and time`' is the value of this attribute. Each element may contain an unlimited number of attributes. In order to take the pain out of creating a suitable attribute array, a SWI eXML_Attribute is available in the eXML module. Each call to eXML_Attribute will add one attribute to the *attibute-stack*. If more that one attribute is required, you need just as many calls to this SWI. The attribute stack may be cleared by calling SWI eXML_ClearAttributes, although this will rarely be used, as the *attribute-stack* is automatically cleared after each new element is created.

Entry

r0 -> Workspace
r1 -> Attribute name
r2 -> Attribute value

Exit

r0 -> Array of attributes (or 0 if unsuccessful)

Consider the following code:

```
...
SYS "eXML_StartElement",xml%,"document"
id$ = "1"
value$ = "date and time"
SYS "eXML_Attribute",xml%,"id",id$ TO attr%
SYS "eXML_Attribute",xml%,"name",value$ TO attr%
SYS "eXML_StartElement",xml%,"field",attr%,"Hello"
SYS "eXML_EndElement",xml%,"document"
```

This will produce the following output:

```
<document>
   <field id="1" name="date and time">Hello</field>
</document>
```

Please note that the element 'field' doesn't need to be closed, as it contains a pointer to content in r3 and is therefore automatically closed.

2.21 SWI eXML_ClearAttributes &576d5

This call can be used to clear the attribute stack. You may do this when you are uncertain about the state of the stack. In most cases it won't be necessary to call this SWI, as the stack will be cleared automatically after the creation of a new element.

Entry

r0 -> Workspace

Exit

No parameters

2.22 SWI eXML_Declaration &576d0

This SWI is used to insert a declaration into an XML document. Declarations are commonly used at the beginning of the document and can be used for a variety of purposes. The contents and processing of a declaration is the responsibility of the XML parser. The most common declaration is the `<?xml?>` declaration at the start of the document. This declaration has to be present for a document to be valid.

Other declarations may be added by your code at will. The SWI eXML_Declaration will produce code similar to the SWI eXML_StartElement, but the line will start with '`<?`' and end with '`?>`' Any attributes are supplied in the attribute-array pointed to by r2. Declarations don't need to be closed and they are not considered part of the document.

Entry

r0 -> Workspace
r1 -> Declaration name
r2 -> Array of Attributes (optional)
r5 Flags

Exit

r0 -> Array of attributes (or 0 if unsuccessful)

Example

```
<?xat doctype="status" version="4.0"?>
```

This information will be presented to the program by the parser, but it has no effect on the further parsing of the document. In the example it is used to identify a document type and a version number to the program.

2.23 SWI eXML_Comment &576cf

This SWI allows a comment to be included in the XML output. Comments always have the following format:

```
<!-- this is a comment -->
```

Entry

r0 -> workspace
r1 -> text string
r2 Length of string in r1 (or 0 to include the entire string (up to a control character).

Exit

r0 Corrupted

Example

```
SYS "eXML_Comment",xml%,"this is a comment"
SYS "eXML_Comment",xml%,"this is a comment",17
```

This is can be used to insert a DOCTYPE line at the start of the document. A DOCTYPE declaration (or DTD) should be inserted *after* the general `<?xml?>` declaration, but *before* the opening of the first element. On entry to the call, you may specify some (optional) pointers to descriptor strings, which will be inserted in the DTD as and when appropriate.

A DTD is basically a description of the structure of your document. It also specifies the possible values for each *attribute*. A DTD can be used by an XML verification tool to check the syntax and validity of your document. The DOCTYPE line is also used by the XML F*ile* D*espatcher* when double clicking an XML file. The *document-type* (r1 on entry) is then used to identify the XML document type and despatch the file to a suitable editor (more on this subject in the chapter 'XML F*ile* D*espatcher*').

Entry

r0	-> Workspace
r1	-> Document type (string)
r2	-> Identifier (string)
r3	-> URL (string)
r5	Flags

Exit

r0	Result Code (0 if successful, otherwise -> error block)

Flags (bits)

2	NoIndent	Don't insert white space before this element
4	Public	DTD is PUBLIC (rather than SYSTEM)
5	Include	Info from the DTD is included in-line

DTD Type

There are three types of DTD:

1 SYSTEM
In this case the DTD is assumed to be stored locally on the machine running the software. A suitable URL (a relative path to the external DTD file) should be specified in r3.

2 PUBLIC
In this case the DTD is assumed to be stored externally at a public address (e.g. the internet). A suitable URL should be supplied in r3 and an Identifier string should be supplied in r2. The Identifier should supply information about the publisher, the file format and the language in which the document is written. A DTD can be made PUBLIC by setting bit 4 of r5 on entry.

3 In-line DTD
The DTD information can also be included in-line, inside the DTD declaration.
At present this method is **not** supported by the eXML module.

Document type

On entry to the call, r1 should point to a string defining the *document-type*. For an SVG document for example, this should be '`svg`'. If your application uses a variety of *document-types*, you could include the application name as well, e.g.: '`artworks:svg`'. This way you can virtually create an unlimited number of *document-types*.

Identifier

The *identifier* is optional. If it *is* supplied it should contain information about the *publisher*, the document *version* and the *language*, e.g.:

```
-//XAT//CableNews Edition 4.0//EN
```

In this example, 'XAT' is the publisher, the document version is 'CableNews Edition 4.0' and the language used in the document is 'EN' (English). The three items are separated by double forward-slashes '//' and the string is preceeded by a '-' (minus sign) to indicate that the publisher is not ISO certified.
If the publisher *is* ISO certified, a '+' (plus sign) should be used.

URL

When using an external DTD file, a suitable URL should be supplied in r3 on entry to the call. Please note that the URL is not used by the XML parser. It is supplied purely for an XML validation tool that wants to check the validity of the document and its syntax.

For a local DTD (SYSTEM) you should supply a relative URL that points to the DTD file, e.g.:

```
"edition.dtd"
"files/edition.dtd"
```

SYSTEM-type DTDs do not have an *identifier* (r2 on entry). If you do supply an identifier, it will be ignored to prevent XML checkers and browsers from producing an error message.

When using a PUBLIC DTD, you should supply an absolute URL that than be accessed by anyone, e.g.:

```
"http://www.xat.nl/dtd/cablenews/edition.dtd"
```

In all cases the filename should have the '`.dtd`' extension. Please note that the path name is in URL format, rather than RISC OS specific format.

Examples

The following BBC BASIC code:

```
SYS "eXML_DTD",xml%,"edition",,"files/edition.dtd"
```

produces this code:

```
<!DOCTYPE edition SYSTEM "files/edition.dtd">
```

whereas this code:

```
id$ = "-//XAT//Edition 4.0//EN"
url$ = "http://www.xat.nl/dtd/edition.dtd"
SYS "eXML_DTD",xml%,"edition",id$,url$,,(1 << 4)
```

produces this:

```
<!DOCTYPE edition PUBLIC "-//XAT//Edition 4.0//EN" "http://www.xat.nl/dtd/edition.dtd">
```

This call is used to insert a Processing Instruction in the output document. Processing instructions my be used to pass information to the XML processor (e.g. the parser inside your application) and are not considered part of the document. As such, their syntax will not be checked. PI lines can take many forms, e.g.:

```
<?xat:impression validation extra?>
```

The only things that defined here, are the opening and closing tags '<?' and '?>' The rest is taken literally from the data supplied in r1 on entry.

Entry

r0	-> Workspace
r1	-> Data
r2	Number of bytes to insert
r5	Flags

Exit

r0	Result Code (0 if successful, otherwise -> error block)

Example

```
...
$data% = "xat:impression validation extra"
size% = LEN($data%) + 1
SYS "eXML_PI",xml%,data%,size%
```

This will produce the following:

```
<?xat:impression validation extra?>
```

The XML standard allows raw data to be inserted in an XML document. Raw data may contain any character, including reserved characters, such as '<' and '>'. The only thing it cannot contain is the *escape sequence* ']]>' that is used to mark the end of the CDATA section.

Entry

r0	-> Workspace
r1	-> Data
r2	Number of bytes to insert
r5	Flags

Exit

r0	Result Code (0 if successful, otherwise -> error block)

The CDATA section will be opened with the following sequence:

```
<![CDATA[
```

and it will be closed with:

```
]]>
```

Please note that *reading* CDATA sections is currently not supported by the eXML module.

2.27 SWI eXML_GetString &576d4

This SWI call isn't really related to XML, but is supplied anyway allow the easy creation of structures (e.g. large menus) in your programs (e.g. BASIC). The call allows a string (terminated by any control character) to be copied into your structure, with an appropriate string terminator for your software.

Entry

r0 -> Original string
r1 -> Destination (somewhere in your workspace)
r2 Required string terminator (e.g. 0 or 13)

Exit

r1 -> Terminating character in destination string
r2 Length of destination string (excluding terminator)

The following example in BBC BASIC illustrates the use of this call. Imagine that we receive a WIMP User_Message (17), of the type Message_XML_DataOpen (&40D59), in which we receive a pointer to a string containing a document-type. (More information on this type of WIMP message in the chapter entitled XML *File Despatcher*.)

```
DIM mes 256
DIM par% 256
...

REM  We've received a User_Message (17)
REM  of the type Message_XMLDataOpen (&40D59)
REM  and we want to extract information from the
REM  data block presented by the WIMP.
...

SYS "eXML_GetString",mes!40,par%,13
doctype$ = $par%
SYS "eXML_GetString",mes+44,par%,13
path$ = $par%
```

There are many ways to express colour values in XML. Some of these are inherited from the HTML standard (used for the creation of web pages), but some new methods have been added as well. Please note that the XML standard is only a method for describing a document, It gives a set of rules, rather than a formal syntax for a number of pre-defined elements. The use of colour in XML documents however *is* standardised and creators of XML documents are advised to obey this 'standard'. It will, at the least, make conversion between programs and platforms much easier.

It is *not* advised to use a colour element in your XML. It would be better to define it as an attribute. E.g.:

```
<background>
   <colour value=rgb(255,255,255)/>
</background>
```

is technically legal, but it would be better to have something like this:

```
<background colour=rgb(255,255,255)/>
```

As you can see in the example above, most programs use the notation rgb(<red>,<green>,<blue>) to define a colour. This is as close as possible to the way the computer stores and uses the colours internally. In a full-colour system (i.e. 24 or 32-bit colour), colours are generally stored as their individual R*ed*, G*reen* and B*lue* components, each with a value between 0 and 255.

There are however other ways of defining a colour, and converting between these 'standards' can be a tedious job, expecially if you want to allow the user to select the required colour model. This is where the SWI eXML_ColourOp comes in. It can be used to convert between standards, read colours and write colours. All calls to this SWI operate directly on a data block in your memory space. This can be a temporary block used to convert between two standards, but it can also be part of a larger structure in which your are storing several other colours or other parameters. On entry to the SWI, r0 should contain the reason code (explained below). The state of the other registers is depending on the reason code in r0.

Data Block

The data block is of a fixed size and format. The size can be requested before allocating the actual datablock and a call is available to initialise the datablock (i.e. put the parameters in a known state). Please note that you should always request the size of the data block first, as it might be extended in a future version of the eXML module. A simple BASIC Library is supplied inside the !eXML resources folder (the BASIC file called 'Colour') that defines the contents of the data block. At present, the data block contains the following information:

+0	Flags	Various bit flags (not used at present)
+4	Model	The original ColourModel used (RGB, CMYK, HSV)
+8	Format	The original Format (syntax) used to define the colour
+12	Value	The actual RISC OS colour (always in RGB + transparency)
+16	Original	Original colour (in the original colour model)
+20	Name	The name of a colour in case of a named colour (32 bytes, including terminator)

All parameters are 4 bytes, except for the N*ame* which is 32 bytes. The ColourModel is stored in the data block for reference only. In your software you may want to present the user with an appropriate default setting for the colour selector, so you may need to know what ColourModel the colour was originally defined in.

Reason Codes

The following reason codes are available (**r0** on entry to the call):

0	Init	Initialise the above structure (or read the required size)
1	Read	Read a colour from a text string into the data structure
2	Write	Read a colour from the data structure and convert it into a text string
3	Get	Convert a colour string into a value
4	Put	Convert a colour value into a string

The reason codes are explained below in more detail.

Colour Model

1	RGB	Red, Green, Blue
2	CMYK	Cyan, Magenta, Yellow, Key (black)
3	HSV	Hue, Saturation, Value
4	YUV	Luminance, B-Y component, R-Y component

The ColourValue is always stored in RISC OS RGB format: &BBGGRRtt. The 'tt' part at the end, is the transparency for the colour (0=fully opaque, 255=fully transparent).

Colour Formats

The official XML standard proposes some colour formats to be used to allow easy conversion between documents and platforms. The most commonly used are rgb() and #RRGGBB (explained below). However, some other formats are allowed as well. The following formats are in the official XML specification and they are fully supported by the eXML module:

rgb(255,128,28)	The recommended colour format, with values in the range 0-255.
rgb(100%,50%,12%)	An alternative to the above. Values are in the range 0%-100%.
#FF7F0A	#RRGGBB. This is the colour format commonly used with HTML pages.
#F71	A rather rare alternative to the above, where one can only define 4096 colours.
hsl(360,100%,100%)	Hue (0-360°), Staturation(0-100%), Lightness (0-100%).
Named HTML *colours*	The standard 17 named colours supported by the HTML 4.0 standard.

In addition to this, the eXML module supports the following formats as well:

&0A7FFF00	RISC OS notation (not officially part of the XML standard).
cmyk(100%,5%,13%,4%)	The printer's model, using Cyan, Magenta, Yellow and Key (black)
cmyk(255,10,28,2)	Same as the above, but using 0-255 values instead of %.
hsv(360,100%,100%)	An equivalent of the hsl() notation, part of the official XML standard.
hsv(360,255,255)	Same as the above, but using values instead of %.

The possible values of the ColourFormat returned in r1+8 are listed here:

#	Name	Description
0	Default	*Use the ColourFormat specified in the Colour Data Block*
1	RISCOS	&FF13D400
2	RGB	rgb(255,40,128)
3	RGB%	rgb(100%,22%,50%)
4	FullHex	#D413FF
5	ShortHex	#D1F
6	CMYK	cmyk(255,100,80,4)
7	CMYK%	cmyk(100%,80%,64%,2%)
8	HSL	hsl(360,100%,100%)
9	HSV	hsv(360,255,255)
10	HSV%	hsv(360,100%,100%)
11	Name	*Named colours, such as*: white, yellow, green, *etc.*
12	YUV	*not yet implemented*
13	YUV%	*not yet implemented*
14	Simple	*Simple* RGB *format without any prefix, e.g.* 255,40,64 *or* 255,40,64,32

The colour formats in more detail

RGB - Red, Green, Blue

This is the most commonly used colour model. It defines colours by specifying the values of the basic colour components **red**, **green** and **blue**. These are the same components as used by a monitor to build the colour image on screen. As the final colour perception is created by adding the *red*, *green* and *blue* light sources together, this system is often referred to as the *additive* colour model. The colour components can be specified as either a value (0-255) or a percentage (0%-100%). The syntax is as follows:

```
rgb(<red>[%],<green>[%],<blue>[%])
```

Some examples:

rgb(255,128,20)
rgb(100%,50%,8%)

HSL - Hue, Saturation, Lightness

This colour model is sometimes referred to as the artist's model. It defines the basic colour, a.k.a. the **hue**, the **saturation** of the colour (how much colour is used) and the **lightness** (or luminance, i.e. the brightness of the colour). The *hue* is always defined as an angle in the standard colour circle between 0° and 360°. The *saturation* and *lightness* are both defined as a percentage (0%-100%). The preferred method in the XML standard is to use %. The syntax is as follows:

```
hsl(<hue>,<saturarion>,<lightness>)
```

Some examples:

hsl(292,78%,82%)

HSV - Hue, Staturation, Value

This is simply an alternative spelling to the above. Most professional drawing and DTP packages refer to this colour model as 'HSV' rather than 'HSL'. HSV stands for **Hue** (the basic colour), **Saturation** (chrominance) and **Value** (luminance). Further details are similar to HSL described above, exept for the fact that both *saturation* and *value* can be defined as either a percentage (0%-100%) or a value (0-255). The latter gives a better resolution and produce less rounding errors when converting between standards. The syntax is as follows:

```
hsv(<hue>[%],<saturation>[%],<value>[%])
```

Some examples:

hsv(292,180,220)
hsv(292,78%,82%)

FullHex notation - #RRGGBB

In order to maintain compatibility with the popular HTML format (used to create internet web pages), colour may also be defined in *full hex* notation. The colour value consists of a '#' sign (hash), followed by six hexadecimal digits. Each RGB colour components is represented by 2 hex digits. The syntax is:

```
#<red><green><blue>
```

Some examples:

#FF034C	equivalent to RISC OS colour `&4C03FF00`
#FFFFFF	represents *white*
#000000	represents *black*

ShortHex notation - #RGB

This form of colour definition is very rarely used, but it is part of the official HTML standard, so it *is* supported here as well. Colours are represented as a 3-digit hex number, allowing 4096 colours to be defined. Like the F*ull* H*ex* notation, the S*hort* H*ex* notation is preceeded by a '#' sign. The syntax is:

```
#<red><green><blue>
```

Some examples:

#F4C	is equivalent to RISC OS colour `&CC44FF00`
#FFF	represents *white*
#000	represents *black*

CMYK - Cyan, Magenta, Yellow, Key (black)

This is not part of the official XML standard, but it may be useful in situations where you want to preserve the full colour definition for printed documents. CMYK is closely related to the printing process, as it specifies the ink colours used to create a *full-colour* document. It is therefore often referred to as *the printer's model*. As colours are created by reflecting white light on a coloured surface, it is also called the *substractive colour model*. A CMYK colour consists of 4 colour components: Cyan, Magenta, Yellow and Key. They Key component is in fact *black*, which is equally subtracted from the other three components. Black is also the only fully opaque ink. The syntax is as follows:

```
cmyk(<cyan[%],<magenta>[%],<yellow>[%],<key>[%])
```

Some examples:

cmyk(100%,45%,23%,5%)	
cmyk(255,120,60,12)	
cmyk(0,0,0,0)	represents *white*
cmyk(0,0,0,255)	represents *black*
cmyk(255,255,255,0)	also represents *black*

Named Colours

The HTML 4.0 standard (used to create internet web pages) allows some colours to be specified by their names, rather than their RGB values. It is a simple set of colours, based on the 16 standard colours from early VGA cards, plus *orange*. The use of named colours is strongly discouraged and you should use RGB values whenever possible. The 17 standard HTML colours are recognised by the eXML module and will be converted into the equivalent RGB values automatically. Colour names are *not* case-sensitive. Some examples:

white	will be converted to &FFFFFF00
yellow	will be converted to &00FFFF00

The full set of named colours:

white, yellow, orange, red, fuchsia, silver, gray, olive, purple, maroon, aqua, lime,
teal, green, blue, navy, black

RISCOS Notation

This is not part of the official XML standard, but it may be useful in situations where you want to define a RISC OS colour directly. The notation is similar to the way colours are defined as hexadecimal numbers in languages such as BBC BASIC. The colour is defined as an 8-digit hexadecimal value, which is preceeded by a '&' sign. The syntax is as follows:

```
&BBGGRRtt
```

One byte is used for each of the RGB colour components. The least significant byte is used for transparency information. When transparency is not used, this byte is set to &00. Some examples:

&0000FF00	represents full *red*
&FFFFFF00	represents *white*
&80808000	represents *mid-grey*

Simple notation

This format is not part of the official XML standard, but it is supported by the module as it is widely used in applications and configuration files. The values for *red*, *green* and *blue* are given in decimal representation [0-255], separated by commas. No prefix and/or brackets are used. Optionally, the *transparecy* value [0-255] may be added to the end of the string. The syntax is:

<red>,<green>,<blue>[,<transparency>]

Some examples:

255,20,64
255,20,64,32

Reason Code 0 - eXML_Colour_Init

Before using a Colour Data Block to call the other reason codes, it may be useful to set the data block to a defined state. This can be done by calling reason code 0. On entry, r1 should contain a pointer to the Colour Data Block. To find the required size of the data block, you should call this reason code with a null pointer in r1. The required size of the block is then returned in r2.

Entry

r0 0 (reason code)
r1 -> Colour Data Block (explained above), or 0 to find the required size
r5 Flags (see below)

Exit

r2 Size of the Colour Data Block

Flags (bits)

0-7 Required string terminator (e.g. 0 or 13)

Reason Code 1 - eXML_Colour_Read

This reson code can be used to read a colour from a string (e.g. an attribute returned from the XML parser) and convert it into a RISC OS colour.

Entry

r0 1 (reason code)
r1 -> Colour Data Block (explained above)
r2 -> Text string containing colour definition
r5 Flags (0 for now)

Exit

Data Block (r1 on entry) will be updated.

Possible errors

&0081CD10 - Unknown colour format
&0081CD11 - Syntax error in colour string

Reason Code 2 - eXML_Colour_Write

This reason code can be used to create a colour format string from a Colour Data Block. It can be used directly to create XML output. You can specify the required ColourFormat, or use the format defined in the Colour Data Block.

Entry

r0 2 (reason code)
r1 -> Colour Data Block
r2 -> Buffer to hold the result
r5 Flags (see below)

Flags (bits)

0-7 Required string terminator (e.g. 0 or 13)
8-15 Required output ColourFormat (see previous desciption of *Colour Formats*)

Possible errors

&0081CD12 - Requested colour format not supported
&0081CD13 - Cannot calculate colour value

The use of the above calls is illustrated in the following example in BBC BASIC:

```
REM --- reserve some memory ---
SYS "eXML_ColourOp",0,0 TO ,,size%
DIM block% size%
DIM string% 256

REM --- initialise colour data block ---
SYS "eXML_ColourOp",0,block%

REM --- write an rgb string ---
SYS "eXML_ColourOp",2,block%,string%,,,13 OR (2 << 8)
PRINT $string%

REM --- read back string ---
SYS "eXML_ColourOp",1,block%,string%
PRINT ~block%!12
```

Reason Code 3 - eXML_Colour_Get

This call can be used to convert a colour string into a value without the need to create a datablock first. It is more or less equivalent to Reason Code 1 - eXML_Colour_Read, but rather than using your applications's workspace, it uses the module's own workspace for the conversion process.

Entry

r0 3 (reason code)
r1 -> formatted colour text string

Exit

r0 Colour Value (in RISC OS format)
r1 -> Datablock in module's workspace

Possible errors

See Reason Code 1 - eXML_Colour_Read

Example in BBC BASIC:

```
SYS "eXML_ColourOp",3,"rgb(255,127,64)" TO colour%
```

Reason Code 4 - eXML_Colour_Put

This call can be used to convert a colour value into a formatted text string in a specified format. It is more or less equivalent to Reason Code 2 - eXML_Colour_Write, but rather than using your applications's workspace, it uses the module's own workspace for the conversion process.

Entry

r0 4 (reason code)
r1 Colour Value (in RISC OS format)
r5 Flags (see Reason Code 1 - eXML_ColourWrite)

Exit

r0 -> Resulting string

Possible errors

See Reason Code 2 - eXML_Colour_Write

Example in BBC BASIC

```
SYS "eXML_ColourOp",4,&4682FF00,,,,&020D TO colour$
PRINT colour$
```

This will produce the following result:

```
rgb(255,130,70)
```

2.29 SWI eXML_DateOp

Partly implemented

There are many ways to express a date in XML, but there is only one official way to do this. Depending on your application and your requirements, you may want to convert from one notation to another. The officlal XML standard dictates that a date should be written in reverse (year-month-date) with the elements separated by dashes. E.g. 31 June 2014, should be written as:

```
2014-06-31
```

Optionally, the time may be added to the string after the date. When present, the time may be separated with an optional <space> followed by the letter 'T', e.g.:

```
2014-06-31 T14:23:01
```

The part after the 'T' represents the time in hours, minutes and seconds, in which each element is separated by a colon (:).

Entry

r0	Flags
r1	-> String

Exit

r0	Error
r1	*Preserved*
r2	Date
r3	Time

Flags

0-7	Format	Date/time format (see below
8-15	*Reserved*	
16	Write [1]	Write date (rather than read)
17	Date	Write date
18	Time	Write time
19	Sec	Write seconds

Format

0	Auto
1	XML
2	EUR
3	USA

Errors

0	No error
1	Wrong format

Remarks

1. This feature has not yet been implemented.

Reading date and time

The SWI can be used to read an existing date (and time) from a text string. It can be forced to read a certain date format by specifying the format in r0 on entry to the call. It can also detect the current format, by selecting 'Auto' (0), in which case it is assumed that in XML and EUR formats, the date parameters are separated by a dash (-), whilst in USA notation they are seperated by a slash "/".

`2016-12-31 T23:59`	Date in XML format
`2016-12-31 T23:59:02`	Date in XML format, time with seconds
`2016-12-31 T23:59.02`	Date in XML format, time with seconds
`31-12-2016 23:59`	Date in EUR format
`31-12-2016 23:59:02`	Date in EUR format, time with seconds
`31-12-2016 23:59.02`	Date in EUR format, time with seconds
`12/31/2016 23:59`	Date in USA format
`12/31/2016 23:59:02`	Date in USA format, time with seconds
`12/31/2016 23:59.02`	Date in USA format, time with seconds

Writing date and time

Not yet implemented.

2.30 SWI eXML_EncodeString

This SWI copies a NULL-terminated string to a destination and translate any reserved characters into an XML escape sequence.

Entry

r0 -> Source string
r1 -> Destination

Exit

r0 Preserved
r1 -> Terminating NULL in Destination buffer

Escaped characters

```
<    &lt;
>    &gt;
&    &amp;
'    &apos;
"    &quot;
```

2.31 SWI eXML_DecodeString

This SWI represents the reverse operation of the SWI eXML_EncodeString, and converts an XML-escaped string into a regular ASCII string with ISO Latin 1 encoding.

Entry

r0 -> Source string
r1 -> Destination

Exit

r0 Preserved
r1 -> Terminating NULL in Destination buffer

Escaped characters

```
<    &lt;
>    &gt;
&    &amp;
'    &apos;
"    &quot;
```

2.32 SWI eXML_Base64

Not implemented

2.33 SWI eXML_Pointer

Added in version 0.36

This SWI reads the current data pointer, which can be used by an application to register the start and end points of content held within and element, allowing parsing and processing of the sub-content at a later moment.

Entry

r0 -> XML workspace

Exit

r1 Offset into buffer holding XML file (or pointer into file)

3. XML File Despatcher

3.1 Introduction

Under RISC OS, most files have been given a filetype, which enables an application (such as an editor) to recognise its files and respond to messages from the Filer when the user double clicks such files. If no suitable application is found, the Filer checks for the existence of a system variable (e.g. Alias$@RunType_FFF) to see if it can launch an application.

With the arrival of the XML document format, many new files and file-formats can be created and an application, such as a database, or document processor, may want to use a variety of file-formats. Although it is perfectly possible and legal to request a RISC OS filetype for each file, you need to be aware that they are in short supply (numbers &000 to &FFF only). Generally, only one filetype will be allocated for each application, and your application has to check the contents of each file to find out how to use it.

3.2 The solution

By default (if no other filetype has been allocated for your file), XML files should have RISC OS filetype &F80. Although every XML file will now look identical (i.e. have the same icon), it may be possible to extract its *document-type* from its contents. A valid XML file will look like this:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE MyDoc SYSTEM "mydoc.dtd">
<document>
   <chapter id="1">
      <page id="1">
         <frame>Hello world...</frame>
      </page>
   </chapter>
</document>
```

The second line in this example (the line starting with !DOCTYPE) contains information about the type of document described in this file. In the example above the *document-type* is 'MyDoc'. in the following description we use **<doctype>** to refer to the *document-type* specified in the !DOCTYPE line. The <doctype> can have any length and may contain any printable character (in XML that is) except for a *space*. Typically, you should only use letters (and perhaps numbers), e.g. 'artworks', svg', 'cablenews', 'config'.

If your application makes extensive use of XML files that each have a different *document-type*, you may consider to use a colon (:) as a separator between the *family-name* and the actual *document-type*, e.g. 'artworks:config', but you should only do this if the file format should be considered *private* (rather than *public*). This way you can create any number of *document-types*, which is not restricted in any way like the system of filetypes under RISC OS.

The eXML module contains a *file-despatcher* which can extract the !DOCTYPE information from a file, read the *document-type* and:

1. Send around a WIMP Message, or
2. Launch a suitable application to edit the file.

For this to work, the XML file must have RISC OS filetype &F80. Double clicking an XML file will be intercepted by the eXML module, which will then despatch it as described above. The despatcher will first broadcast a Message_XMLDataOpen (&40559). Any application that can load XML files in this way, should respond to this message and check whether the *document-type* presented in the message block is one it can handle. If it can process the file, the application should claim the call by sending an acknowledge (see below).

If the message is not claimed by any of the applications, the despatcher checks for the existence of a system variable of the type `'XML$@Run_<doctype>'` (again <doctype> should be the type of the document). If it finds the variable, it executes the command that it finds inside it. Typically, this will be a line to launch a program and pass the path in the command tail.

3.3 System variables

The eXML module uses system variables to determine the action to take when an XML file is launched. These system variables all start with `'XML$@'` and have the following format:

```
XML$@Run_<doctype> <command> <path> %*0
```

Some examples:

```
XML$@Run_svg Run adfs::4.$.MyDisc.Apps.!ArtWorks.!Run %*0
```

The 'Run' variable

An application that wants to be launched automatically when the user double clicks a file of the given <doctype>, can setup a system variable of the type 'XML$@Run_xxx', where 'xxx' should be replaced by the *document-type* of your file. If, for example, ArtWorks wants to be launched when the user double clicks an SVG file that is file-typed as an XML file (&F80), the following line should be added to the !Boot file (and possibly also to the !Run file):

```
Set XML$@Run_svg Run <ArtWorks$Dir>.!Run %%**0
```

Please note the extra '%' sign which is used as an *escape* character to allow the following '%' to be entered. Whenever the user double clicks an XML file (with RISC OS filetype &F80), and ArtWorks isn't already running, it will be launched by the XML despatcher. The pathname of the file will be passed in the tail of the command (i.e. in place of the `%*0` sequence).

All other system variables starting with 'XML$@' are reserved for future use by the eXML module.

3.4 WIMP Messages

The following WIMP messages are used by the eXML module:

1	&40D59	Message_XMLDataOpen
2	&40D5A	Message_XMLDataOpenAck

Whenever a 'Load' variable is not found for the document, the XML despatcher broadcasts a User_Message (18): Message_XMLDataOpen (&40D59), which has a format similar to the standard Message_DataOpen (5) which is broadcast by the Filer whenever a file is double-clicked. The only difference is the value at offset 40 in the returned message block. This now contains a pointer to a string giving the *document-type*, rather than the filetype. The string is terminated with a null (0).

An application responding to the Message_XMLDataOpen, should check the contents of the string to see if this <doctype> is of use to it. If it wants to use the file, it should first claim the call by sending a User_Message 19, Message_XML_DataOpenAck (&40D5A) and then loading the file.

The message block (pointed to by r1) as a result of the broadcast Message_XMLDataOpen contains:

+0	Size of block
+4	Task handle of message sender
+8	my_ref (the sender's reference for this message)
+12	your_ref (0)

+16 &40D59 (Message_XMLDataOpen)
+20 Window handle of directory display containing the file
+24 (unused)
+28 x offset of file icon that was double clicked
+32 y offset of file icon that was double clicked
+36 0
+40 -> string containing the *document-type* (zero-terminated)
+44 Full pathname of the file (zero-terminated)

If an application can load the file, should respond by first claiming the call and then loading the file. This is to prevent the call from being passed to other applications if the load process fails (which would eventually produce another error message).

Claiming the call can be done by replying to this message by sending a User_Message (17) of the type Message_XMLDataOpenAck (&40D5A) to the sender. When doing this, it should also copy the my_ref field to the your_ref field before sending the reply. The following example in BBC BASIC should illustrate the use of the message.

```
DIM mes 256
...

REM --- main polling loop ---
REPEAT
   SYS "Wimp_Poll" 0, mes TO r%
   CASE r% OF
   ...
   WHEN 17, 18, 19 : PROC_message
   ENDCASE
UNTIL quit
END

DEF PROC_message
   CASE mes!16 OF
   ...
   WHEN &40D59 : PROC_xml_dataopen
   ENDCASE
ENDPROC

DEF PROC_xml_dataopen
   doctype$ = FN_read_string(mes!40)
   file$ = FNread_string(mes+44)
   CASE doctype$ OF
   WHEN "svg"
      mes!16 = &40D5A
      mes!12 = mes!8
      SYS "Wimp_SendMessage",17,mes,mes!4
      REM ---> load the file
      ...
   ENDCASE
ENDPROC
```

3.5 Naming conventions

In order to avoid conflicts when loading a document automatically into an application, the DOCTYPE should be chosen carefully. Rather than using something like 'MyDoc' it would be better to use something like 'MyApp:MyDoc' or 'MyCompany:MyDoc'.

Some examples:

```
artworks
xat:database
dbman:users
xat:dbman:database
```