Have a second look at XML files

Arjen Markus[1]
WL|Delft Hydraulics
PO Box 177
2600 MH Delft
The Netherlands

A few additional points

In the december issue of the Fortran Forum (Markus, 2004) I described the format of XML files and ways to handle them in Fortran. The present paper is meant to elaborate on a few aspects of XML that were not considered then but which are at least as important to appreciate what XML is all about. Most notably: the advantages of using a strict format and internationalisation. I wish to thank Rolf Ade for his valuable comments that led to this second paper.

Not a silver bullet but still …

The first paper stressed that that there is sharp contrast between what XML actually offers and what the hype has had us believe: XML is a file format, no more, no less. You get the syntax for free but you will have to add the meaning yourself.

While that is very true indeed, there are a couple of advantages to using such a predefined and verifiable format:

- Because the rules by which the contents have to be formatted are clearly defined, there can be little argument about typos and other errors in these files. If the file does not conform to the XML standard, any number of *application-independent* tools can check that.
- The contents of an XML file can be described formally by, for instance, a document type definition (DTD). While this technique can not find all errors - especially errors in the data themselves - it can find errors in the *structure*. For instance: a book is characterised by a title, its authors and a few other things. Leaving out one or more of the essential items renders this characterisation invalid. By using a *validating* parser you can catch many such errors without any programming effort - all the logic is in the DTD.

So, in summary, using XML and the related technologies makes it easier to check the data files that you exchange with others.

Internationalisation

Strictly speaking, XML files are *not* ASCII files: it is possible use characters and character encodings beyond the basic 128 defined in the ordinary (7-bits) ASCII table. This has a number of consequences: The files can be used in cultural contexts other than those that mainly use Latin alphabets - that is a great asset in a world where most people are accustomed to letters and characters that have nothing to do with the Western ABC.

It also means that somehow the encoding has to be determined before the file can be read correctly. In general, as remarked in the XML standard document (W3 committee, 2004), this is an impossible task if you do not have additional information: a file is simply a sequence of bytes and the translation into humanly meaningful *characters* is by no means unique. A byte could be a representation of a character in the ASCII table or the EBCDIC table for instance, but more commonly the encoding is used to map characters from, say, Greek into the second half of the byte range, so that all characters are still represented as single bytes.

---

[1] Email: arjen.markus@wldelft.nl

The situation with XML files is not quite that hopeless. First, an XML file must start with the *characters "<?xml"*, this provides one check.

First, if an XML file is not encoded in UTF-8, it must start with the characters "<?xml". (Even if it is encoded in UTF-8, it may have an XML declaration, which is of course found directly at the start of the file.) Secondly, the declaration in the first line may contain the encoding that has been used and otherwise it defaults to UTF-8:

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

Once the encoding is known,[2] the parser can read the file and perform any checks and conversions needed.

As UTF-8 is a commonly used character encoding, perhaps I should explain its basics (*cf.* Kuhn, 1999). UTF-8 is a way to deal with the UNICODE character tables in such a way that as little space is wasted as is practical. The UNICODE tables define the representation of characters from almost all human languages into four bytes, with subsets of commonly used characters that fit into two bytes. As using four byte for each character would waste lots of memory and disk space, UTF-8 tries to find a balance between the full UNICODE tables and the simplicity of the ASCII table.

In UTF-8 a character that falls in the 7-bits ASCII table can still be represented by a single byte. So, if a file contains only characters from the 7-bits ASCII table it can be treated as if its encoding was either ASCII or UTF-8. Characters outside that range are encoding with two or more bytes. The practical consequence is that a program must either convert these byte sequences into sequences of fixed length (two bytes for instance) or be prepared to deal with a variable number of bytes per character.

Here is some code to convert a sequence of bytes, encoded according to UTF-8, into an integer that represents the position in the UNICODE tables:

```
! toUnicode --
!    Turn a UTF-8 encoded character into a Unicode integer
! Arguments:
!    utf8char       UTF-8 encoded character
! Returns:
!    Unicode integer or -1, if the UTF-8 string is invalid
!
! Note:
!    No check for illegal patterns
!
function toUnicode( utf8char )
   character(len=*), intent(in)  :: utf8char

   integer                       :: toUnicode

   integer                       :: utf8byte(4)
   integer                       :: ucbyte(4)

   toUnicode = -1

   !
   ! One character codes
   !
   utf8byte(1) = ichar( utf8char(1:1) )
   if ( utf8byte(1) .lt. 128 ) then
      toUnicode = utf8byte(1)
      return
   endif
```

[2] The cited document describes a detailed algorithm of how to determine the encoding that was used (appendix F).

```
   !
   ! Two character codes
   !
   if ( utf8byte(1) .lt. 224 ) then
      if ( len( utf8char ) .lt. 2 ) return
      utf8byte(2) = ichar( utf8char(2:2) )
      toUnicode   = 64 * ( utf8byte(1)-192 ) + &
                         ( utf8byte(2)-128 )
   else
   !
   ! Three character codes
   !
      if ( len( utf8char ) .lt. 3 ) return
      utf8byte(2) = ichar( utf8char(2:2) )
      utf8byte(3) = ichar( utf8char(3:3) )
      toUnicode   = 4096 * ( utf8byte(1)-224 ) + &
                      64 * ( utf8byte(2)-128 ) + &
                           ( utf8byte(3)-128 )
   endif

end function toUnicode
```

This function returns an integer, as Fortran does not currently know UNICODE characters. A function to go the other way around can be created in a very similar way. (Note that such a function will not be necessary with the new Fortran 2003 standard: it will then be possible to open a file with the encoding set to UTF-8).

The array example again

In the first paper I showed what I consider to be a serious drawback of the XML format: storing data that is organised in a non-hierarchical way, such as large arrays of measurement data or computational results is cumbersome and requires you to force some hierarchical organisation onto these data. Also the current specifications in a DTD do not allow you to link the size of such an array to another element in the XML file – you will have to program that yourself and you will lose at least part of the advantages the XML format brings.

If you were to adhere completely to the XML philosophy, storing a two-dimensional array would indeed be done like this:

```
<matrix>
   <row>
      <value>1.0</value>
      <value>1.1</value>
      …
   </row>
   <row>
      <value>13.0</value>
      <value>25.12</value>
      …
   </row>
   …
</matrix>
```

Reading a file with large arrays stored in this way is more cumbersome than using more conventional Fortran methods, like a read statement with an implied do-loop, not to mention the fact that you need to code an algorithm to assemble all the pieces again into an array that the program can use.

Several alternatives have been proposed, informally known as *binary versions* of XML, that aim to solve these problems (Goldman, 2004). But no single solution has been accepted sofar.

A completely different strategy may be to just store the "metadata" about the array or arrays in the XML file and to store the actual data in a separate file:

```
<matrix>
   <file>matrix.dat</file>
   <rows>20</rows>
   <columns>10</columns>
   <ordering>values-per-row</ordering>
</matrix>
```

This is merely an example, of course, but the meaning ought to be clear:

- The metadata define which data file to read and what sizes this two-dimensional array has
- To make sure that the data are read correctly, an indication of the ordering of data has been added
- As no format has been given, the data should be read via a list-directed READ statement.
- The data file "matrix.dat" can be left open, so that more than one array can be read from the same file.

Such a set of conventions is not much different from any solution using ad hoc formats – except that the structure of the data file is described explicitly and it is possible to define a small set of such conventions that can be used independently of the application.

Conclusion

The purpose of this second paper has been to expose a number of aspects of using XML that were not treated in the first paper. While the final conclusion is still that XML is not a magical cure for all problems regarding data exchange, it should in my view be seriously considered as one of the possible solutions. But please, do not expect miracles: XML provides the means to separate the *parsing* of the input data from the rest of the program. Once that is done, it is up to the programmer to actually *use* the data.

Literature

Oliver Goldman (2004)
Binary XML
Doctor Dobbs' Journal, november 2004

Markus Kuhn (1999)
UTF-8 and Unicode FAQ for Unix/Linux
http://www.cl.cam.ac.uk/~mgk25/unicode.html

Arjen Markus (2004)
Have a look at XML files
Fortran Forum, Volume 23, Number 3, December 2004

W3 committee (2004)
Extensible Markup Language (XML) 1.0 (Third Edition)
http://www.w3.org/TR/2004/REC-xml-20040204/