

A Clarion Magazine Reader

Clarion Language Topics

*This document is exclusively for the use
of Clarion Magazine subscribers*

www.clarionmag.com



Revision History

Oct 13 2014

First release

1. Clarion language	2
1.1 Clarion data type equivalents	2
1.2 Fixing "unresolved external" errors	3
1.3 Handling circular references with #implib	4
1.4 OLE/COM	5
1.5 Returning a Dimensioned Variable from a Procedure	6
1.6 Using classes to handle events	9
1.7 Using the Clarion SAX parser	13
1.8 Variable length strings and arrays	15
1.9 XML	16
1.9.1 Reconstructing Clarion's XML tooling, Part 1	16

Clarion language

- Clarion data type equivalents
- Fixing "unresolved external" errors
- Handling circular references with #implib
- OLE/COM
- Returning a Dimensioned Variable from a Procedure
- Using classes to handle events
- Using the Clarion SAX parser
- Variable length strings and arrays
- XML

Clarion data type equivalents

Most of the C types are from Carl Barnes' excellent Clarion Magazine article [Compiling C with the Clarion IDE, Part 1: It's Easier Than You Think](#).

C type	Clarion type
*	pass by address - <omittable>
&	pass by address - required
[#]	Array, use DIM(#). The name of an array is actually a pointer to the first element of the array. So arrays tend to be passed by address even though you do not see an asterisk.
char, unsigned char, signed char	BYTE, but probably a string type
char *, char[]	CSTRING or STRING (need RAW). If the array has a dimension number it tends to be a fixed length string e.g. char Digest[16] is a STRING(16). A char * could be a *BYTE, but it is rare.

char **	<p>This is a pointer to a pointer. There's an example of this in John Taylor's ClarionMag article Embedding The SQLite Engine In Clarion Applications. The C parameter is <code>char **errmsg</code> and the Clarion parameter is a <code>Long</code>.</p> <p>You need to declare two variables, a CString reference and a Long:</p> <pre>CStringRef &CString CStringRefAddress Long</pre> <p>You pass in the <code>CStringRefAddress</code> variable to the function call. If it comes back as a non-zero value, you obtain the string this way:</p> <pre>CStringRef &= (CStringRefAddress)</pre> <p>If the pointer is to a string array you use a similar technique. From John Christ:</p> <p style="padding-left: 40px;">The value returned by the function is a pointer to the 1st (or in C, 0th) element of an array of pointers to strings.</p> <p style="padding-left: 40px;">You dereference the pointer to to the first element and that is a pointer to the first string.</p> <p style="padding-left: 40px;">You add 4 (in 32 bit code, 8 in 64 bit code which doesn't yet exist in the Clarion world) to the original pointer and you are now pointing to the second pointer.</p> <p style="padding-left: 40px;">Add 4 again, the third pointer, and so on.</p> <p style="padding-left: 40px;">When the dereferenced pointer is zero, you've reached the end of the array.</p>
struct	GROUP (need RAW)
unsigned short	USHORT
signed short, short	SHORT
int, signed int, long, signed long, signed	SIGNED OF LONG - UNSIGNED IS <code>Equate(Long)</code>
unsigned, unsigned long, unsigned int	ULONG - avoid using the ULONG type in Clarion. The object code created for ULONG math uses the decimal library, which is much slower than the code used for a LONG. The UNSIGNED type is equated to a LONG, which is preferable.
float	SREAL
double	REAL
void	Usually appears as a return type, indicating that nothing is returned.
void *	This is a pointer to something. Use a LONG or UNSIGNED.

Fixing "unresolved external" errors

If you work in multi-DLL apps, chances are you'll eventually hit one or more "unresolved external" errors. For that to happen you need to have some code or data with the `,EXTERNAL` attribute. The help has this to say about EXTERNAL:

The EXTERNAL attribute specifies the variable, FILE, QUEUE, GROUP, or CLASS on which it is placed is defined in an external library. Therefore, a variable, FILE, QUEUE, GROUP, or CLASS with the EXTERNAL attribute is declared and may be referenced in the Clarion

code, but is not allocated memory--the memory for the variable, FILE, QUEUE, GROUP, or CLASS is allocated by the external library. This allows the Clarion program access to any variable, FILE, QUEUE, GROUP, or CLASS declared as public in external libraries.

An unresolved external error is a linker error - after compiling your source the linker needs to find your external data declarations in another library, usually a DLL. But the linker doesn't directly associate your code with the DLL itself. Instead it links in a LIB file for that DLL; it's the LIB that contains information on where the variables are located in the DLL.

If you have unresolved external errors, and you don't know which LIB to include in your project, you can always do a text search of all the LIB files you have for any of the missing symbols. But if the symbols are in one of your own apps a search of all of the export (.EXP) files will probably give you a more readable result. (And if the symbol is in the EXP but not in the LIB you need to recompile that DLL!)

Two search utilities I use on a regular basis are [Keystone Source Search \(KSS\)](#) and [Windows Grep](#). KSS is far and away the more powerful option for most Clarion source searching, but as I usually have a DOS window open somewhere I still use grep on a regular basis for quick searches.

Handling circular references with #implib

Circular references are unpleasant things. They happen when a procedure in DLLA calls a procedure in DLLB, which calls a procedure in DLLA.

There's some evidence that circular references contribute to app instability. I don't know whether that's been proven, although there's anecdotal evidence in support of the idea.

But one thing is certain - circular references make the process of compiling and linking apps more complicated. That's because when you compile an app that uses a DLL, you're actually linking in the DLL's corresponding LIB file.

In the above example, DLLA needs the LIB file from DLLB, and DLLB needs the LIB file from DLLA. Which comes first?

Actually it doesn't matter which comes first - just realize that while you'll get a successful compile without a required LIB you won't get a successful link. You have to compile one of the DLLs first (compile succeeds, own LIB is created, link fails), then the other (compile succeeds, own LIB is created, link succeeds), then the first one again (compile succeeds, own LIB is recreated, link succeeds).

That's fine for small systems with just a couple of DLLs and a few circular references, but if you have a lot of DLLs and a lot of circular references (any one of which may involve numerous DLLs) two passes through the compile process may not do the job. I've seen a system where if all the LIB files were deleted it took four or five passes to get a clean compile and link.

The better way

There is a way to create all the LIB files at once, before you begin compiling. The only thing you need to do is make sure you have up to date EXP files, which are created as part of the application generation process. EXP files

You'll need a hand coded project file (with the PRJ extension). Here's one called CreateLibs.prj:

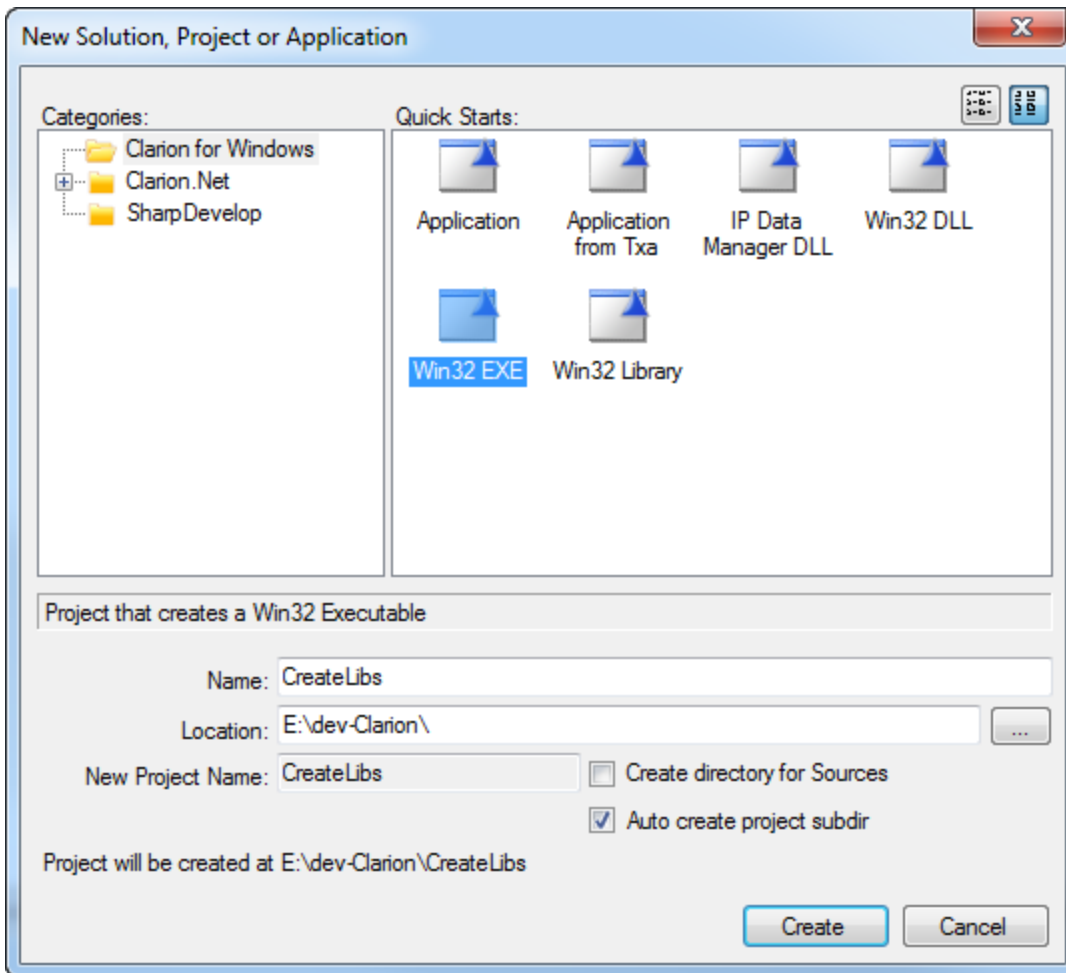
```
#implib DLLA.LIB DLLA.EXP  
#implib DLLB.LIB DLLB.EXP
```

Each line of the prj is simply an #implib statement followed by the name of the LIB to be created and the EXP to use when creating the LIB.

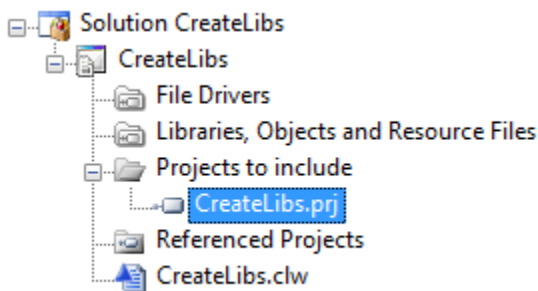



In early versions of C7 #implib was broken, so if it's not working for you it's probably time to update.

Now you need to build that project. Create a new hand coded solution:



The easiest way to do this is as a Win32 EXE. A Win32 DLL works also but you'll need to edit the EXP file as it contains a line of reminder text. In either case you won't need the output of this solution, you just need a way to execute the project (you could include it with one of your existing apps, but I prefer to have it on its own so I control when it gets compiled).



 Make sure you add the .prj (legacy project) file and now the .cwproj file.

Now just compile the app and all the LIB files specified in the #implib statements will be created. With those in hand you can compile all your apps with circular references in one pass.

If you want your implib solution to clean up after itself you can always add a post-build task to delete the resulting EXE.

And if you really want to get rid of those circular references, I strongly recommend [George Lehman's ClarionMag article: Eliminating Circular DLL Calls](#)

OLE/COM

OLE stands for Object Linking and Embedding ([see Wikipedia](#)), and COM stands for Component Object Model ([see Wikipedia](#)).

Type libraries

From Wikipedia:

Automation components are usually provided with *type libraries*, which contain [metadata](#) about classes, interfaces and other features exposed by an object library. Interfaces are described in [Microsoft Interface Definition Language](#). Type libraries can be viewed using various tools, such as the Microsoft OLE/COM Object Viewer (`oleview.exe`, part of the [Microsoft Platform SDK](#)) or the Object Browser in Visual Basic (up to version 6) and [Visual Studio .NET](#). Type libraries are used to generate [Proxy pattern/stub](#) code for interoperating between COM and other platforms, such as [Microsoft .NET](#) and [Java](#).

Here's a link from Arnor Baldvinnson on [creating type libraries using IDL](#) (from the ArcGIS SDK documentation).

Returning a Dimensioned Variable from a Procedure

by [Steve Parker](#)

A recent news group posting asked how to return a dimensioned variable, i.e. an array, from a called procedure. There is no "short answer" to this question ... okay, there is and it is "you can't." It *is* possible to get an array of values back from a called procedure. It just can't be done as a return value.

That is, a procedure prototyped as:

```
CalledProc( ... ),Long
```

where the Long is an array will fail to compile. Whether the array demarcators ("[" and "]") are included or not included, regardless of where they are included or omitted, the procedure will not compile.

But Clarion provides another way of getting data from a called procedure, a callee, back to its caller. Of course, global variables are a possibility, theoretically, but they are neither necessary nor advisable.

As discussed in "Returning Multiple Values," passing a variable by address (i.e., as a "variable parameter") allows a called procedure to change the values in the callee's "copy" of the variable. (It's not really a copy, both procedures work with the same memory location, that is, on the same variables.)

Arrays *can* be passed from one procedure to another. So it seems logical that passing an array by address would fulfill the requirement of getting back a dimensioned variable from a procedure.

The Language Reference explicitly recommends this for passing and retrieving arrays. This is what it has to say on the subject:

To pass an entire array as a parameter, the prototype must declare the array's data type as a Variable-parameter ("passed by address") with an empty subscript list. If the array has more than one dimension, commas (as position holders) must indicate the number of dimensions in the array. The calling statement must pass the entire array to the PROCEDURE, not just one element.

To pass an entire array as a parameter, the prototype must declare the array's data type as a Variable-parameter ("passed by address") with an empty subscript list. If the array has more than one dimension, commas (as position holders) must indicate the number of dimensions in the array. The calling statement must pass the entire array to the PROCEDURE, not just one element.

It gives the following sample prototype:

```
AddCount PROCEDURE(*LONG[,] Total,*LONG[,] Current) !Passing two 2-dimensional arrays
```

and this example code for the procedure:


```

AddCount PROCEDURE(*LONG[,] Total,*LONG[,] Current) !Procedure expects two arrays
CODE
LOOP I# = 1 TO MAXIMUM(Total,1)      !Loop through first subscript
  LOOP J# = 1 TO MAXIMUM(Total,2)    !Loop through second subscript
    Total[I#,J#] += Current[I#,J#]  !increment TotalCount from CurrentCnt
  END
END
CLEAR(Current)                       !Clear CurrentCnt array

```

This sample code highlights not only the “how to” but ways in which arrays are different from other variables.

Prototyping

Procedures receiving arrays are prototyped a little bit differently than typical procedure calls. The entire array must be passed, per the Language Reference, not just a single element. But instead of passing anything in the array delimiters, empty brackets are used. So, for a single dimensional array:

```
MyProc[ ]
```

For two dimensions:

```
MyProc[ , ]
```

three dimensions:

```
MyProc[ , , ]
```

and so on. As the documents state, the subscript list must contain enough commas to define the size of the array but nothing more than that.

In the AppGen, this looks like this:

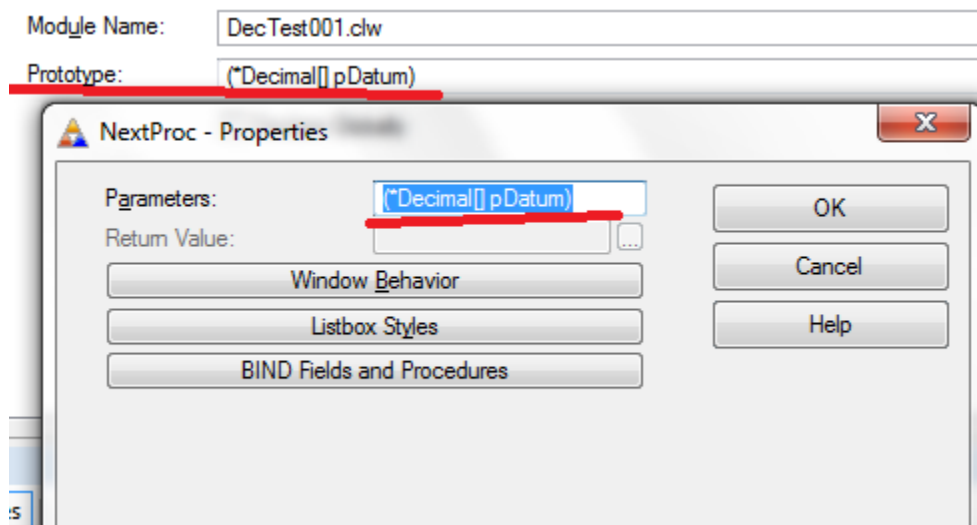


Figure 1: Prototype array-receiving procedure in the AppGen

Calling

In hand code, from the example code, AddCount is called as follows:

```
TotalCount LONG,DIM(10,10)
CurrentCnt LONG,DIM(10,10)
CODE
AddCount(TotalCount,CurrentCnt)
```

The AppGen fully supports the convention for passing arrays:

Figure 2: Button Actions from Window Formatter:

Notice that SV's sample code does not include the bracket delimiters in the call but that I did in my demo app (downloadable at the end of this article – it is built in 8.0.8778, all runtimes are included so that everyone may run it). The application generator is happy with or without the brackets.

Receiving and Using the Array

Where dimensioned variables differ from other received parameters is in how I must handle the variable(s) in the called procedure.

Consider the prototype of the procedure shown in Figure 1, above:

```
NextProc PROCEDURE (*Decimal[] pDatum)
```

"pDatum" is the Label of my "local" variable. If I just want to make a calculation involving pDatum, I can use pDatum just like any other dimensioned variable. This is shown in the help code snippet shown above.

But, if it is necessary to assign the parameter to a (real) local variable, in order, for example, to use the window Formatter to display the incoming values on a window, things change. Arrays are different in that simply assigning the parameter to a local variable:

```
LOC:Datum[] = pDatum[]
```

is illegal. So is:

```
LOC:Datum = pDatum
```

(in other words, the compiler will ... uh, complain).

And while we're on the subject, what if the passed array and the local, receiving, array have different numbers of parameters? Seriously bad things could happen trying to assign a value to an index that doesn't exist.

Clarion provides a function that allows checking the highest index in a DIM declaration, MAXIMUM. So,

```
If Maximum(pDatum,1) > Maximum(LOC:Datum,1)
```

will verify that there are not enough local elements to receive all the elements of the parameter (it seems to me that it would be acceptable for the local variable to support more indices; it is only fewer that will cause trouble).

```
If Maximum(pDatum,1) > Maximum(LOC:Datum,1)
  Message('The local variable has ' & Maximum(LOC:Datum,1) & |
    ' elements. But the incoming parameter has ' & |
    'Maximum(pDatum,1) & '.| |This will ' & |
    'cause an Index Out of Range error.', 'Array ' & |
    'Mismatch', ICON:Hand)
```

in ThisWindow.INIT, therefore, can be used to terminate the call should my local variable not support sufficient subscripts. (I probably don't need these code after initial testing, data declarations don't usually change specifications at run time; but neither does leaving the check hurt anything.)

At the same time, because I am assigning a local version of the array, I can ensure that I don't accidentally make an invalid assignment by determining which of the two arrays is smaller:

```
If Maximum(LOC:Datum,1) >= Maximum(pDatum,1)
  C = Maximum(LOC:Datum,1)
Else
  C = Maximum(pDatum,1)
End
```

and use my "size" variable to control the assignment:

```
Loop NDX = 1 to C
  LOC:Datum[NDX] = pDatum[NDX]
End
```

by not making any assignment where one of the variables could go out of range. This is important. Tracking down out of range array indices can be quite an adventure. Often, an out of range index does not throw an error, the app simply GPFs.

Similar code can be used to safely reload the passed variables to update the values visible to the caller:

```
Loop NDX = 1 to Maximum(pDatum,1)
  pDatum[NDX] = LOC:Datum[NDX]
End
```

Post(EVENT:CloseWindow) and the called procedure closes and the calling procedure will have the correct values.

Conclusion

A good news group question, a quick look at the Language Reference ... and "it can't be done" once again becomes "here's how to achieve the result." Reframing the question – from getting a "return" value to "getting the value back" – as they taught me in Philosopher school also helps.

[Download the source](#)

Using classes to handle events

While reviewing the code for WindowResizerClass (in an attempt to get more agreeable resizing behavior for ClarionTest) I came across a usage of RegisterEvent that I hadn't noticed before:

```
REGISTEREVENT (EVENT:DoResize, ADDRESS(SELF.TakeResize), ADDRESS(SELF))
```

The help says this about Register, which is a synonym for RegisterEvent:

REGISTER registers an event handler PROCEDURE called internally by the currently active ACCEPT loop of the specified window whenever the specified event occurs. This may be a User-defined event, or any other event. User-defined event numbers can be defined as any

integer between 400h and 0FFFh.

RegisterEvent installs a callback procedure for your accept loop, to be called whenever the specified event occurs. You can install event handlers for windows and for individual controls.

The bit I hadn't noticed was the ability to specify a class method as the callback by passing in the address of the method and the address of the class.

Using class methods as callbacks in Clarion is often problematic because internally the first parameter to any method is the class itself. This is why using Omitted(*parameternumber*) is unwise with class methods that have optional parameters - if the first parameter of a method is omissible, you have to specify Omitted(2) rather than Omitted(1). Just use Omitted(*parametername*) - it will save you a lot of trouble.

Similarly, you can't use class methods as WinAPI callbacks because of that hidden first parameter.

But RegisterEvent is part of the Clarion runtime, and it's smart enough to handle a method if that's what you throw at it.

Here's a small program to demonstrate:

```

PROGRAM
MAP
END

EventQueue      queue,type
EventTime       long
end

EventHandler     class
EventQ          &EventQueue
Construct       procedure
Destruct        procedure
TakeEvent       procedure,byte
end

Window          WINDOW('Timer event
callback'),AT(, ,177,129),GRAY,FONT('Microsoft Sans Serif',8), |
TIMER(100)

LIST,AT(3,2,171,124),USE(?LIST1),VSCROLL,FROM(EventHandler.EventQ), |
FORMAT('20L(2)|M@t4@')

END

CODE
open(window)
accept
case event()
of EVENT:OpenWindow

RegisterEvent(event:timer,address(EventHandler.TakeEvent),address(EventHandler))
end
end
UnRegisterEvent(event:timer,address(EventHandler.TakeEvent),address(EventHandler))
close(window)

EventHandler.Construct      procedure
code
self.EventQ &= new EventQueue

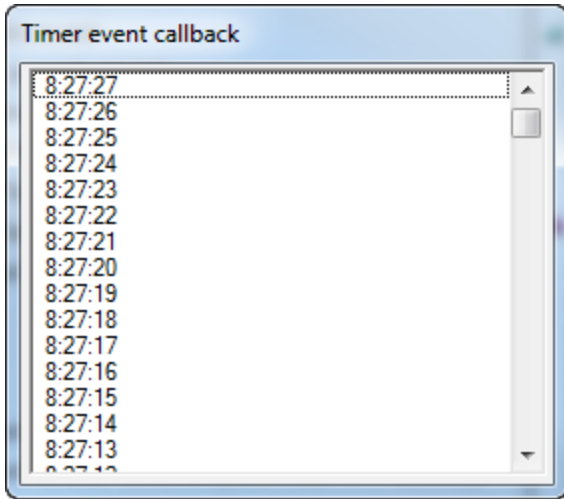
EventHandler.Destruct      procedure
code
free(self.EventQ)
dispose(self.EventQ)

EventHandler.TakeEvent     procedure
code
self.EventQ.EventTime = clock()
add(self.EventQ,-self.EventQ.EventTime)
return level:benign

```

On EVENT:OpenWindow registers the EventHandler.TakeEvent method as a timer event handler. TakeEvent simply adds a record to a queue each time the timer event fires.

Here's the result:



The help has some conflicting information on the event handler prototype:

The handler procedure MUST have 1 parameter: when the handler is called the runtime library is passing the object value (the 3rd parameter in the call to REGISTER) as its parameter.

The handler PROCEDURE must not take any parameters and must return a BYTE containing one of the following EQUATED values (these EQUATES are defined in the ABERROR.INC file):

I'm not completely sure what the first statement is referring to, but the second statement is accurate. The handler procedure does not have any parameters and must return a byte.

The return value can be one of the following:

Level:Benign	Calls any other handlers and the ACCEPT loop, if available.
Level:Notify	Doesn't call other handlers or the ACCEPT loop. This is like executing CYCLE when processing the event in an ACCEPT loop.
Level:Fatal	Doesn't call other handlers or the ACCEPT loop. This is like executing BREAK when processing the event in an ACCEPT loop.

On further experimentation I found that I could reduce the program code to just four lines:

```
open(window)
  RegisterEvent(event:timer,address(EventHandler.TakeEvent),address(EventHandler))
  accept
end
```

As long as RegisterEvent is called after the window is opened it will work - it doesn't have to be called inside the Accept loop. And since all registered handlers are automatically unregistered when the window closes, and the window is closed when the program exits (this also happens at the procedure level if the window is local to the procedure), I don't have to call UnRegister explicitly.

It doesn't end here...

Although this sample program is short, it demonstrates several things that I think are important for the future of Clarion development. For a few years now I've been harping on the need to extract business logic from embeds and into classes, where it can be tested and reused. But that leaves a gap; how do you get that business logic to interact with the user interface? Does that code need to be put in embed points?

This example shows two ways classes can be used to manage user interface interaction. The first is obvious: you can register a class method as event handler. But take a closer look at the window structure, specifically the list statement:

```
LIST,AT(3,2,171,124),USE(?LIST1),VSCROLL,FROM(EventHandler.EventQ),FORMAT('20L(2)|M@t4@')
```

The From attribute points directly to a queue that's a property of the class. That means that not only can use use classes to manage events, you can use classes to supply data that's bound to the user interface. There isn't even any need to issue a Display() after adding a queue record: the

data simply appears on screen as it is added.

It's still important to keep a separation between business logic and UI code, but clearly there's a role for classes in managing UI behavior. More interestingly, you can also apply unit testing to some of that UI behavior. I'll have more to say about that in the near future.

Using the Clarion SAX parser

May Clarion devs may not be aware that Clarion has had XML reading/writing capability since Clarion 6, when SoftVelocity shipped the CenterPoint C++ XML class library along with some wrapper Clarion classes. Unfortunately the open source CenterPoint XML project didn't last, and the original company web site is long gone.

I published several articles on ClarionMag about the CenterPoint classes, including:

- [XML For Clarion Developers](#)
- [Reading XML With The CenterPoint Classes](#)
- [Creating XML Files With The Clarion 6 DOM Parser](#)
- [Creating An XML RSS Web Site Summary With Clarion 6 \(Part 1\)](#)
- [Creating An XML RSS Web Site Summary With Clarion 6 \(Part 2\)](#)

The CenterPoint classes include both SAX and DOM parsers. DOM parsers have to load the entire document into memory before you can use it, but having the entire document at your disposal can make it easier to read and manipulate data. SAX parsers fire callbacks for each element found, and are better for reading massive files where memory usage could be a problem.

I experimented with the DOM parser. One thing I didn't ever try was reading an XML file with the SAX parser. But [Graham Dawson](#) did and he has kindly given me permission to post his example code.

Here's the main program CLW:

```
program
    include('cpxml.inc'),once

                                map
                                end
xmlFileName                    string(128)
mySAXParserClass               class(SAXParserClass)
StartElement                   PROCEDURE(string name),derived
EndElement                     PROCEDURE(string name),derived
Characters                     PROCEDURE(string chars),derived
addID                          byte(false)
addFirstName                   byte(false)
addLastName                    byte(false)
addGender                      byte(false)
                                end
resultQueue                    queue,pre(RST)
id                              long
firstName                      cstring(21)
lastName                       cstring(31)
gender                         cstring(2)
                                end
result                          byte
count                           long

Window                          WINDOW('Result Queue after XML
Parse'),AT(, ,321,193),CENTER,SYSTEM,GRAY,AUTO

LIST,AT(15,11,292,166),USE(?List1),HVSCROLL,VCR,FROM(resultQueue)
                                END

code
xmlFileName = 'people_tags.xml'
free(resultQueue)
result = mySAXParserClass.ParseXMLFile(xmlFileName)
if result <> 1
    message('Error during XML Parsing')
```

```

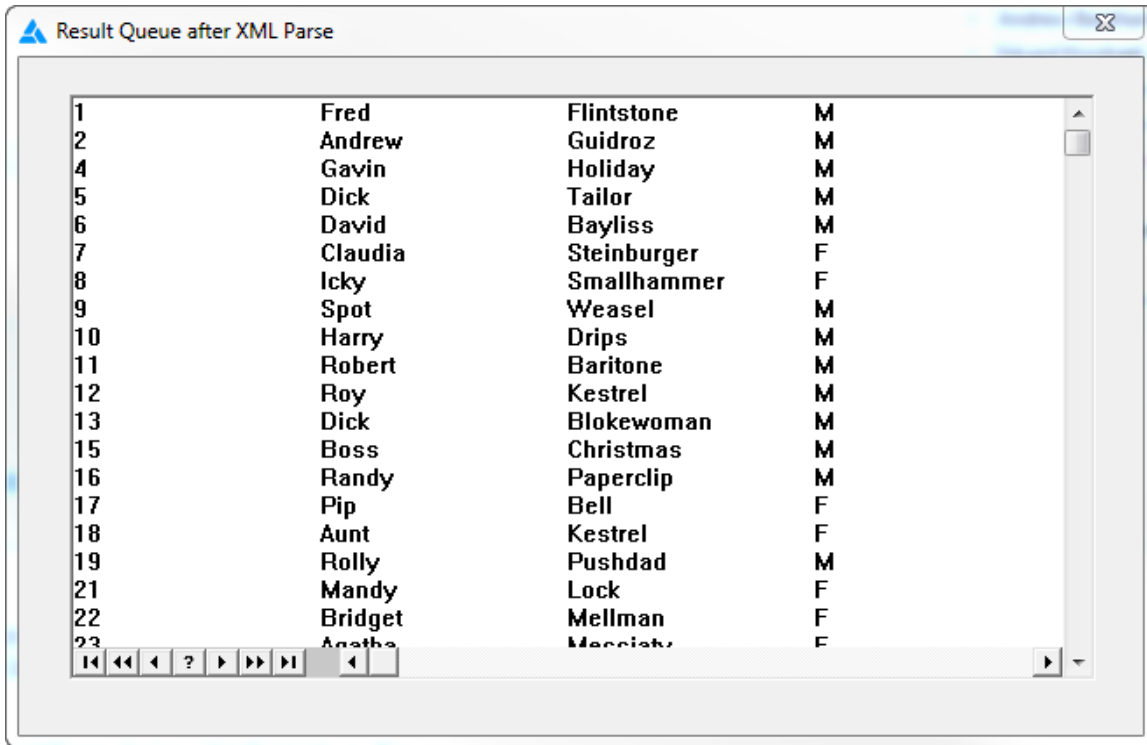
else
    open(Window)
    accept
end
close(Window)
end
mySAXParserClass.StartElement      PROCEDURE(string name)
code
case name
of 'Table'
    clear(resultQueue)
    self.addID = false
    self.addFirstName = false
    self.addLastName = false
    self.addGender = false
of 'id'
    self.addID = true
of 'firstname'
    self.addFirstName = true
of 'lastname'
    self.addLastName = true
of 'gender'
    self.addGender = true
end
mySAXParserClass.EndElement        PROCEDURE(string name)
code
case name
of 'Table'
    add(resultQueue)
of 'id'
    self.addID = false
of 'firstname'
    self.addFirstName = false
of 'lastname'
    self.addLastName = false
of 'gender'
    self.addGender = false
end
mySAXParserClass.Characters        PROCEDURE(string chars)
code
if self.addID = true
    RST:ID = chars
end
if self.addFirstName = true
    RST:FirstName = chars
end
if self.addLastName = true
    RST:LastName = chars
end
end

```



```
if self.addGender = true
  RST:Gender = chars
end
```

And here's the output of the program:



The screenshot shows a window titled "Result Queue after XML Parse" with a scrollable list of data. The data is organized into four columns: an index, a first name, a last name, and a gender. The list contains 23 entries, with the last one partially cut off.

Index	First Name	Last Name	Gender
1	Fred	Flintstone	M
2	Andrew	Guidroz	M
4	Gavin	Holiday	M
5	Dick	Tailor	M
6	David	Bayliss	M
7	Claudia	Steinburger	F
8	Icky	Smallhammer	F
9	Spot	Weasel	M
10	Harry	Drips	M
11	Robert	Baritone	M
12	Roy	Kestrel	M
13	Dick	Blokewoman	M
15	Boss	Christmas	M
16	Randy	Paperclip	M
17	Pip	Bell	F
18	Aunt	Kestrel	F
19	Rolly	Pushdad	M
21	Mandy	Lock	F
22	Bridget	Mellman	F
23	Agatha	Macciato	F

Sometime before the CenterPoint project went offline, I downloaded the source code (see below). I don't know how this version of the source compares with the one shipped with Clarion. If you compile it, let me know how it goes.

Downloads

- [Graham's example](#)
- [CenterPoint 2.01.07 source](#)
- [CenterPoint 2.01.03 documentation](#)

Variable length strings and arrays

While looking through the help for some information on maximum string sizes, I came across this little nugget: you can declare variable length strings and arrays. As in, you pass a number (a dimension) into a procedure, and you can use that dimension in a data declaration.

You can also do this dynamically with New() of course, but this is a much simpler and cleaner approach.

Here's the Help example:

```

PROGRAM

                                MAP
                                TestProc(LONG)
                                END

CODE
TestProc(200) !can also be an initialized variable

TestProc                                PROCEDURE(LONG VarLength)
VarString                                STRING(VarLength)
CODE
VarString = 'String of up to 200 characters'

```

The Help also shows how to handle variable length arrays.

There are restrictions - the variable can only be local to a procedure or routine, and cannot be in a class, group etc.

See the topic Variable Size Declarations for more information.

Rick Martin and Mike Hanson tell me that this syntax will also work.

```

TestProc                                PROCEDURE(string s)
VarString                                STRING(len(s))

```

XML

- [Reconstructing Clarion's XML tooling, Part 1](#)

Reconstructing Clarion's XML tooling, Part 1

If you want to do XML with Clarion you have a few options. A lot of developers opt for one or more of the available third party offerings. These include:

- [Capesoft's XFiles](#) (easy conversion to and from Clarion data structures, with some limitations)
- [ThinkData's xmlFuse](#), a COM wrapper around Microsoft's XML parser
- [Robert Paresi's iQ-XML](#), free and highly regarded but source code is not available

(If I've missed any useful Clarion XML third party products please post a comment below.)

What we sometimes forget is that Clarion itself ships with some extensive XML parsing capabilities. Years ago TopSpeed licensed the CenterPoint XML C++ class library (which is itself, reportedly, built on the [open source Expat XML parser](#)) and incorporated a wrapper for that library into the Clarion product line.

DOM and SAX

The Clarion XML toolset includes both DOM and SAX parsers. A DOM parser reads an XML file and creates a corresponding Document Object Model (DOM) in memory. A SAX parser reads an XML file and fires off events as it encounters the different parts of the document (elements, attributes etc.). SAX parsers are generally viewed as better for really large documents, but as I don't deal with really large XML documents I've only ever used DOM parsers.

The Clarion code

Clarion ships with a bunch of code to support the use of the CenterPoint parsers. You can find this code in the following files:

- cpxml.inc
- cpxml.clw

- cpxmlif.inc

There are some additional source files that add support for translating XML to and from Clarion structures, and I'll get to those at some future date.

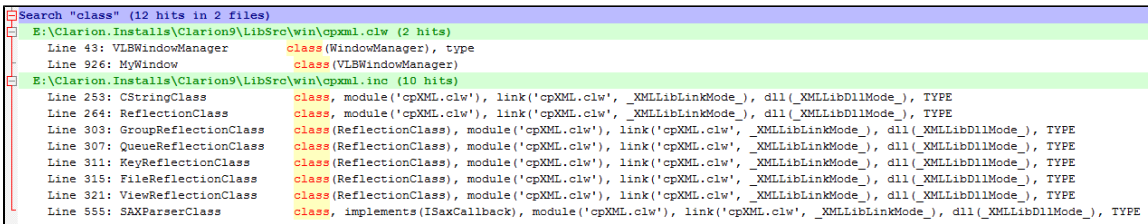
I'm really interested in this core code, however, because it promises me the kind of low level access to XML documents I want and need.

But there's a problem.

Making sense out of the cpxml* files

The cpxml* files are nearly unreadable, for the simple reason that they contain too much code. So I began by looking for specific clues.

Here's what I get when I search all three files for classes (using Notepad++):



```
Search "class" (12 hits in 2 files)
E:\Clarion.Installs\Clarion9\LibSrc\win\cpxml.clw (2 hits)
Line 43: VLBWindowManager      class(WindowManager), type
Line 926: MyWindow              class(VLBWindowManager)
E:\Clarion.Installs\Clarion9\LibSrc\win\cpxml.inc (10 hits)
Line 253: CStringClass          class, module('cpXML.clw'), link('cpXML.clw', _XMLLibLinkMode_), dll(_XMLLibDllMode_), TYPE
Line 264: ReflectionClass       class, module('cpXML.clw'), link('cpXML.clw', _XMLLibLinkMode_), dll(_XMLLibDllMode_), TYPE
Line 303: GroupReflectionClass   class(ReflectionClass), module('cpXML.clw'), link('cpXML.clw', _XMLLibLinkMode_), dll(_XMLLibDllMode_), TYPE
Line 307: QueueReflectionClass   class(ReflectionClass), module('cpXML.clw'), link('cpXML.clw', _XMLLibLinkMode_), dll(_XMLLibDllMode_), TYPE
Line 311: KeyReflectionClass     class(ReflectionClass), module('cpXML.clw'), link('cpXML.clw', _XMLLibLinkMode_), dll(_XMLLibDllMode_), TYPE
Line 315: FileReflectionClass    class(ReflectionClass), module('cpXML.clw'), link('cpXML.clw', _XMLLibLinkMode_), dll(_XMLLibDllMode_), TYPE
Line 321: ViewReflectionClass    class(ReflectionClass), module('cpXML.clw'), link('cpXML.clw', _XMLLibLinkMode_), dll(_XMLLibDllMode_), TYPE
Line 555: SAXParserClass        class, implements(ISaxCallback), module('cpXML.clw'), link('cpXML.clw', _XMLLibLinkMode_), dll(_XMLLibDllMode_), TYPE
```

First of all I'd like to know what a UI component (based on the ABC WindowManager class) is doing in a core XML parser library, but I suppose I can let that slip for now. VLB stands for Virtual List Box so it's probably something to do with displaying XML data, perhaps for debugging purposes.

But what are the other classes for? Apart from SAXParserClass none of them seems to have anything obviously to do with XML.

Searching for interfaces, however, tells a different story:

```

Search "interface" (52 hits in 2 files)
E:\Clarion.Installs\Clarion9\LibSrc\win\cpxml.inc (32 hits)
Line 340: DOMWriter           interface
Line 353: DOMImplementation   interface
Line 360: DOMObject           interface
Line 367: NodeList            interface (DOMObject)
Line 372: NamedNodeMap        interface (DOMObject)
Line 399: Node                 interface (DOMObject)
Line 427: Attr                 interface (Node)
Line 435: CharacterData        interface (Node)
Line 446: Notation             interface (Node)
Line 451: Text                 interface (CharacterData)
Line 455: Comment              interface (CharacterData)
Line 458: CDATASection         interface (Text)
Line 461: EntityReference      interface (Node)
Line 464: Entity               interface (Node)
Line 470: ProcessingInstruction interface (Node)
Line 476: DocumentType         interface (Node)
Line 486: Element              interface (Node)
Line 505: Document             interface (Node)
Line 525: DocumentFragment    interface (Node)
Line 529: ISAXCallback         interface, com
E:\Clarion.Installs\Clarion9\LibSrc\win\cpxmlif.inc (20 hits)
Line 48: DOMWriter           interface
Line 61: DOMImplementation   interface
Line 68: DOMObject           interface
Line 75: NodeList            interface (DOMObject)
Line 80: NamedNodeMap        interface (DOMObject)
Line 107: Node                interface (DOMObject)
Line 135: Attr                 interface (Node)
Line 143: CharacterData        interface (Node)
Line 154: Notation             interface (Node)
Line 159: Text                 interface (CharacterData)
Line 163: Comment              interface (CharacterData)
Line 166: CDATASection         interface (Text)
Line 169: EntityReference      interface (Node)
Line 172: Entity               interface (Node)
Line 178: ProcessingInstruction interface (Node)
Line 184: DocumentType         interface (Node)
Line 194: Element              interface (Node)
Line 213: Document             interface (Node)
Line 233: DocumentFragment    interface (Node)
Line 237: ISAXCallback         interface, com
Search "class" (12 hits in 2 files)

```

I'm not sure (yet) why the interfaces are duplicated between cpxml.inc and cpxmlif.inc, but clearly this is where the good stuff is. To begin with I'll focus my attention on cpxml.inc.



The Clarion help has this definition of an interface:

An INTERFACE is a structure, which contains the methods (PROCEDURES) that define the behavior to be implemented by a CLASS. It cannot contain any property declarations. All methods defined within the INTERFACE are implicitly virtual.

I'll assume for the purposes of this discussion that you already have some understanding of interfaces.

Examining the interfaces

If you look at the DOMWriter interface in the source editor you'll see this:

```

!****Interface to access to a DOMWriter object
DOMWriter          interface
setEncoding        PROCEDURE(*DOMWriter, const *CSTRING encoding)
setEncoding        PROCEDURE(*DOMWriter, const *CSTRING encoding, bool assumeISO88591)
getEncoding        PROCEDURE(*DOMWriter), *CSTRING
getLastEncoding    PROCEDURE(*DOMWriter), *CSTRING
setFormat          PROCEDURE(*DOMWriter, UNSIGNED format)
getFormat          PROCEDURE(*DOMWriter), UNSIGNED
setNewLine         PROCEDURE(*DOMWriter, const *CSTRING newLine)
getNewLine         PROCEDURE(*DOMWriter), *CSTRING
writeNode          PROCEDURE(*DOMWriter, *Node pNode), *CSTRING
writeNode          PROCEDURE(*DOMWriter, const *CSTRING systemId, *Node pNode), cbool
end

```

This looks just like a standard Clarion interface. And when I started working with the XML library I was able to use it just like a standard Clarion user interface, but I noticed something odd: I never got any code completion on the interface methods.

As it turns out there's some jiggery-pokery going on here. Scroll all the way to the *right* in the source editor and you'll see the rest of the story:

```

.;map;module('DOMWriter')
, name('_setEncoding@8'), pascal, raw
, name('_setEncoding2@12'), pascal, raw
, name('_getEncoding@4'), pascal, raw
, name('_getLastEncoding@4'), pascal, raw
, name('_setFormat@8'), pascal, raw
, name('_getFormat@4'), pascal, raw
, name('_setNewLine@8'), pascal, raw
, name('_getNewLine@4'), pascal, raw
, name('_writeNode@8'), pascal, raw
, name('_writeNode2@12'), pascal, raw
;end

```

Get rid of the extra spaces, replace the line continuation characters with line breaks, and format the code according to the Clarion standard and you'll end up with this:

```

DOMWriter          interface
                  END
                  map
                    module('DOMWriter')
setEncoding        PROCEDURE(*DOMWriter, const *CSTRING encoding),
name('_setEncoding@8'), pascal, raw
setEncoding        PROCEDURE(*DOMWriter, const *CSTRING encoding, bool
assumeISO88591), name('_setEncoding2@12'), pascal, raw
getEncoding        PROCEDURE(*DOMWriter), *CSTRING, name('_getEncoding@4'),
pascal, raw
getLastEncoding    PROCEDURE(*DOMWriter), *CSTRING,
name('_getLastEncoding@4'), pascal, raw
setFormat          PROCEDURE(*DOMWriter, UNSIGNED format),
name('_setFormat@8'), pascal, raw
getFormat          PROCEDURE(*DOMWriter), UNSIGNED, name('_getFormat@4'),
pascal, raw
setNewLine         PROCEDURE(*DOMWriter, const *CSTRING newLine),
name('_setNewLine@8'), pascal, raw
getNewLine         PROCEDURE(*DOMWriter), *CSTRING, name('_getNewLine@4'),
pascal, raw
writeNode          PROCEDURE(*DOMWriter, *Node pNode), *CSTRING,
name('_writeNode@8'), pascal, raw
writeNode          PROCEDURE(*DOMWriter, const *CSTRING systemId, *Node
pNode), cbool, name('_writeNode2@12'), pascal, raw
                    end
                  end

```

It turns out the interface itself is an empty structure, which is why I never got any code completion. Instead of interface methods there are procedure declarations which take the interface as the first parameter.

As you may already know the first parameter of any class's (or interface's) method is the class (or interface) itself, which is why these functions behave in exactly the same way as if they were interface methods. Each function is a call into the SoftVelocity wrapper around the CenterPoint C++ XML library.

The problem

This technique of marrying Clarion interfaces to the C++ library is tricky and most interesting, but it raises a concern, at least in my mind. In other languages I'm used to modeling XML documents with a set of classes, but here I don't have any classes at all, just interfaces. How do I work with the XML data?

Luckily, I've been down this path before. Close to a decade ago I [wrote an article in Clarion Magazine](#) on how to create an XML document using the then-C6 DOM parser. Here's the code, which was pretty much a straight port of some of the C++ example code:

```

DOMRss             PROCEDURE                               !
DOMImpl            &DomImplementation
docType            &DocumentType
pDoc               &Document
pText              &Text
pRootElement       &Element
pElement           &Element
pCommentElem       &Comment
pChannelElem       &Element
pItemElement       &Element
writer             &DomWriter
pCDATA             &CDATASection
encoding           cstring('UTF-8')

```

```

s          cciCStringFactory
CODE
DomImpl &= CreateDomImplementation()
DocType &= null
pDoc &= DomImpl.CreateDocument(s.c(''),s.c('rss'),DocType)

pRootElement &= pDoc.GetDocumentElement()
pRootElement.SetAttribute(s.c('version'),s.c('0.91'))
pCommentElem &= pDoc.CreateComment(s.c('ClarionMag DOM RSS example'))
pDoc.InsertBefore(pCommentElem,pRootElement)
pCommentElem.Release()

pChannelElem &= pDoc.CreateElement(s.c('channel'))

pElement &= pDoc.CreateElement(s.c('title'))
pText &= pDoc.CreateTextNode(s.c('Clarion News'))
pElement.appendChild(pText)
pText.Release()
pChannelElem.AppendChild(pElement)
pElement.Release

pElement &= pDoc.CreateElement(s.c('description'))
pText &= pDoc.CreateTextNode(s.c('News, product announcements, |
    & 'and other items of interest to Clarion developers'))
pElement.appendChild(pText)
pText.Release()
pChannelElem.AppendChild(pElement)
pElement.Release

pElement &= pDoc.CreateElement(s.c('language'))
pText &= pDoc.CreateTextNode(s.c('en-us'))
pElement.appendChild(pText)
pText.Release()
pChannelElem.AppendChild(pElement)
pElement.Release

pElement &= pDoc.CreateElement(s.c('link'))
pText &= pDoc.CreateTextNode(s.c('http://www.clarionmag.com'))
pElement.appendChild(pText)
pText.Release()
pChannelElem.AppendChild(pElement)
pElement.Release

pElement &= pDoc.CreateElement(s.c('copyright'))
pText &= pDoc.CreateTextNode(s.c('Copyright 1999-2003 by CoveComm Inc. '))
pElement.appendChild(pText)
pText.Release()
pChannelElem.AppendChild(pElement)
pElement.Release

! Add first <item>

pItemElement &= pDoc.CreateElement(s.c('item'))

pElement &= pDoc.CreateElement(s.c('title'))
pText &= pDoc.CreateTextNode(s.c('RPM Email Survey'))
pElement.appendChild(pText)
pText.Release()
pItemElement.AppendChild(pElement)

```

```
pElement.Release

pElement &= pDoc.CreateElement(s.c('link'))
pText &= pDoc.CreateTextNode(s.c('http://www.cwaddons.com/email/'))
pElement.appendChild(pText)
pText.Release()
pItemElement.AppendChild(pElement)
pElement.Release

pElement &= pDoc.CreateElement(s.c('description'))
pText &= pDoc.CreateTextNode(s.c('Lee White is looking for ' |
    & 'feedback, from current and prospective RPM users, ' |
    & 'about email support in RPM.'))
pElement.appendChild(pText)
pText.Release()
pItemElement.AppendChild(pElement)
pElement.Release

pChannelElem.AppendChild(pItemElement)
pItemElement.Release()

! Add second <item>

pItemElement &= pDoc.CreateElement(s.c('item'))
pElement &= pDoc.CreateElement(s.c('title'))
pText &= pDoc.CreateTextNode(s.c('True Edit-In-Place Template'))
pElement.appendChild(pText)
pText.Release()
pItemElement.AppendChild(pElement)
pElement.Release

pElement &= pDoc.CreateElement(s.c('link'))
pText &= pDoc.CreateTextNode(s.c('http://www.audkus.dk'))
pElement.appendChild(pText)
pText.Release()
pItemElement.AppendChild(pElement)
pElement.Release

pElement &= pDoc.CreateElement(s.c('description'))
pText &= pDoc.CreateTextNode(s.c('This new EIP Template ' |
    & 'adds full template support for the Clarion ' |
    & 'edit-in-place list box. ABC templates only; ' |
    & 'includes source and future updates.'))
pElement.appendChild(pText)
pText.Release()

pCDATA &= pDoc.CreateCDATASection(s.c('This is some text in a CDATA section'))
pElement.AppendChild(pCDATA)
pCDATA.Release()

pItemElement.AppendChild(pElement)
pElement.Release

pChannelElem.AppendChild(pItemElement)
pItemElement.Release()

pRootElement.AppendChild(pChannelElem)
pChannelElem.Release()
```



```
Writer &= CreateDOMWriter()  
Writer.setFormat(format:reformatted)  
if Writer.writeNode(s.c('domrss.xml'),pDoc).
```

```
pDoc.Release()  
  
DestroyDomImplementation(DomImpl)
```

There are a couple of issues with this code. One is that all of my interface references (pText, pElement etc.) have a really ugly "p" prefix. I definitely can't call them Text, Element etc. because those are the actual names of the interfaces and I'll get symbol collisions. Giving the interfaces an appropriate prefix (see below) will solve that problem.

The bigger issue, however, is I'm limited in what I can do with the XML document as it only exists in memory allocated by the C++ library. What if I wanted to validate an XML node, or automatically populate it with some attributes when it was created? I don't have those kind of options.

What I really want is a set of Clarion (not C++) classes that I can use to model an XML document. That way I can add whatever code I need anywhere I need it. Of course I'll still use the existing library to read and write the XML documents by using the existing interfaces and API calls.

Gimme dem classes!

The first thing I do when I start any refactoring project is hit Ctrl-A and Ctrl-I to reformat the code to my liking. It's also something I do regularly while I code, such as after moving or changing a block of code in a way that alters indenting.

The next thing was to come up with a set of actual classes (TYPEd, of course) that model the already-defined interfaces (which themselves are a pretty straightforward model of the DOM specification).

The more classes I write, the more adamant I become about two rules:

1. Each class must be in its own .INC/.CLW file pair, and
2. Each class (and interface) needs a descriptive, prefixed name

There will always be exceptions, but following these two rules will make any class library much easier to understand.

I contacted Bob Zaunere and asked for and received permission to refactor these classes, and they'll eventually be part of SV's future community repository on GitHub. As much as possible I try to follow the example of .NET when it comes to class naming, but since Clarion doesn't support namespaces I use text to mimic .NET naming. And because these are core SV classes, instead of my usual standard I've used the System_XML_ prefix. It's a little presumptuous I suppose, but as I seem to be the first one in the pool...

I renamed selected interfaces to classes as follows:

Old name	New name
Attr	System_XML_Attr
CDATASection	System_XML_CDATASection
CharacterData	System_XML_CharacterData
Comment	System_XML_Comment
Document	System_XML_Document
Element	System_XML_Element
Node	System_XML_Node
NodeList	System_XML_NodeList
Notation	System_XML_Notation
Text	System_XML_Text

I settled on some new class names as well, but for the most part they don't figure into the first round of refactoring. The only one that's radically different is CStringClass, which is a fairly small class used to manufacture CStrings which are needed when communicating with the CenterPoint XML library.

Old name	New name
----------	----------

CStringClass

System_XML_CString

String creation is a capability needed in other places than just XML so originally I wanted to give it a more general name, but to avoid introducing any outside dependencies for now I've simply called it System_XML_CString. That will probably change in the near future.



I have a pet peeve about using "Class" in the name of a class. I suppose this harkens back to naming conventions like Hungarian notation, where it was helpful and necessary to be able to infer the data type from the variable name. There's no need for this from a code safety point of view since classes are strongly typed, and the compiler will complain if you try to use a class as something other than a class. For readability there's code completion. So adding "Class" to a class name is arguably just noise. And in fact within the ABC library the term "Class" is added to class names inconsistently.

Creating the class template

I almost never write a class from scratch anymore. Instead I use John Hickey's excellent ClarionLive Class Creator and create my classes from standard class templates. Those aren't templates in the Clarion sense; they're class files that contain the basic structure of the class I want to create.

When I create a class I'm almost always imagining that at some time in the future (when the code is stable) it will be compiled into, and exported from, a DLL. And that means I need some compiler directives for the LINK and DLL attributes. You've probably seen these before in the form of the `_ABCDLLMode_` and `_ABCLinkMode_` symbols. And if you've ever had to set these manually, for whatever reason, you probably know the disastrous consequences (read: GPF) of getting them wrong.

Here's my class header template file:

```
include( 'System_XML_IncludeInAllClassHeaderFiles.inc' ), once

System_XML_BaseClass
Class, Type, Module( 'System_XML_BaseClass.CLW' ) |

, Link( 'System_XML_BaseClass.CLW', _System_XML_Classes_LinkMode_ ) |
, Dll( _System_XML_Classes_DllMode_ )
Construct Procedure( )
Destruct Procedure( )

End
```

Looks painful to type, right? That's why it's in a template file, so I don't have to keep typing it. But if all I do is have it in the class template I'll still need to make sure it gets into the project somehow, and I don't have to have to type it there either. That's something the application (*.tp?) templates usually do for us, but in this case I don't yet have a template, I just have a bunch of source code. So instead I create a standard header file that sets these symbols. Here's System_XML_IncludeInAllClassHeaderFiles.inc:

```

!-----
! While in the development phase default to classes being compiled. Once
! the code is stable and a DLL is provided the following three lines can
! be removed.
!-----
    omit('***',_Compile_System_XML_Class_Source_)
    _Compile_System_XML_Class_Source_          equate(1)
    ***

    OMIT('***',_Compile_System_XML_Class_Source_)
    _System_XML_Classes_LinkMode_             equate(0)
    _System_XML_Classes_DllMode_             equate(1)
    ***

    COMPILER('***',_Compile_System_XML_Class_Source_)
    _System_XML_Classes_LinkMode_             equate(1)
    _System_XML_Classes_DllMode_             equate(0)
    ***

```

Now instead of two symbols I only need to worry about one symbol: `_Compile_System_XML_Class_Source_`

And I don't even have to worry about that one, since I've added a bit of code at the top to default `_Compile_System_XML_Class_Source_` to 1 so the classes will always be compiled. At some future point when the classes are stable and I'm providing a DLL I can remove the first `Omit` statement and using the classes from a DLL will be the default (which can be overridden by setting `_Compile_System_XML_Class_Source_` to true in the project).

Clear as mud? Good.

Here's the class template for the .CLW file:

```

Member

                                Map
                                End

    Include('System_XML_BaseClass.inc'),Once
    !include('System_Logger.inc'),once
!dbg                                System_Logger
System_XML_BaseClass.Construct      Procedure()
    code

System_XML_BaseClass.Destruct      Procedure()
    code

```

There's no real code here yet, but I always put a constructor and destructor in the template because most of the time I end up needing one or both. And they serve as an example of how methods are implemented.

When I create a class using the Class Creator it will replace `System_XML_BaseClass` with whatever class name I specify.

The XML classes

I'll go into the actual classes and some unit tests next time, but here's the list of source files as it stands now. There are some I haven't yet implemented, and I've also added a few new classes which I'll explain in a future article:

```
System_XML_Attr.clw
System_XML_Attr.inc
System_XML_AttrList.clw
System_XML_AttrList.inc
System_XML_CDataSection.clw
System_XML_CDataSection.inc
System_XML_CenterPointInterfaces.inc
System_XML_CharacterData.clw
System_XML_CharacterData.inc
System_XML_Comment.clw
System_XML_Comment.inc
System_XML_CString.clw
System_XML_CString.inc
System_XML_Document.clw
System_XML_Document.inc
System_XML_Element.clw
System_XML_Element.inc
System_XML_IncludeInAllClassHeaderFiles.inc
System_XML_Node.clw
System_XML_Node.inc
System_XML_NodeBuffer.clw
System_XML_NodeBuffer.inc
System_XML_NodeList.clw
System_XML_NodeList.inc
System_XML_Notation.clw
System_XML_Notation.inc
System_XML_Text.clw
System_XML_Text.inc
System_XML_XPath.inc
System_XML_XPathQuery.clw
System_XML_XPathQuery.inc
```

Next time: some unit tests and an explanation of how to read and write basic XML.

And for those of you who just can't wait, [here's the source](#).