Reborn Free CLARION online

published by CoveComm Inc.

Clarion MAGAZINE

Search

Go

Home

COL Archives

Subscribe
New Subs
Renewals

Info
Log In
FAQ
Privacy Policy
Contact Us

Downloads
PDFs
Freebies
Open Source

Site Index

Call for Articles

The Clarion Advisor: Procedure Prototypes

Compiler changes, unless they implement bug fixes, don’t often get a lot of press in the Clarion community. A few years ago the compiler began handling an alternate form of procedure prototype, which can make your programming life just a tiny bit easier.

Posted Wednesday, February 28, 2001

The Clarion Challenge: Useless Tab Text

I was taking a tour of the Language Reference Manual when I came across a curious and, as near as I can tell, completely useless feature: inverted tab text. I've written a small program to demonstrate this feature, but the program has a bug. The challenge: fix the bug and/or find a better way to write this program.

Posted Wednesday, February 28, 2001

Introduction To SQL: Part 1

In the first installment of this new series, Dave Harms compares TPS and SQL databases and explains why SQL is important to Clarion developers.

Posted Wednesday, February 28, 2001

The Cranky Programmer: Got Them Bloated ABC Blues

Cranky's back, and he's not at all pleased with the way ABC packs ABC-compliant classes into data DLLs.

Posted Thursday, February 22, 2001

COM: Getting Easier By The Minute (Part 3 of 3)

COM? Easy? Well, easier, at least. In this three-parter, Jim Kane, Clarion Magazine's resident guru of Microsoft component object technology, shows how to call COM

RSBackUp 1.2 Released

MessageEx Updated

Free Outlook Style Template

HTML Designer Version 1 Release Imminent

VariView FAQ Page

Freeware Template Handles Capitalization

Solace VariView Demo Available

MenuTree Template Includes TXAs

Registry Function Library 1.5 Available

Intuit To Provide QuickBooks Interface

Logic Central Adds Remote Reporter

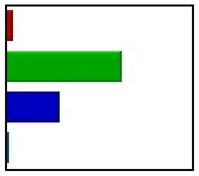
FrameText Adds Text To Application Frame

CLARION.COM Sold For $30,000

Clarion ABC Templates In Spanish

SURVEY

Which browse & form control method do you use?



🟥 Toolbar
🟩 Button
🟦 Both
🟦 Neither

Votes: 71

objects with the new COM-compatible INTERFACE in Clarion 5.5. As Jim says, if VB programmers can use COM, so can you!

Posted Thursday, February 22, 2001

The Clarion Advisor: Locating Records With PROP:SQLFilter

If you're working with SQL tables, you can muster the full power of the database server to quickly locate the records you want, using PROP:SQLFilter. Just add an entry field to your browse window, type the filter, and away you go. Until you type an incorrect filter, and away the browse goes instead. Here's one way to solve the problem.

Posted Tuesday, February 13, 2001

Information Systems Planning

This Whitemarsh Information Systems paper presents an information systems model for information systems managers. Includes comparisons to IBM's Business System Plan, James Martin's Strategic Data Planning, and Clive Finklestein's Strategic Management Plan.

Posted Tuesday, February 13, 2001

COM: Getting Easier By The Minute (Part 2 of 3)

Easy? Well, easier, at least. In this three-parter, Jim Kane, Clarion Magazine's resident guru of Microsoft component object technology, shows how to call COM objects with the new COM-compatible INTERFACE in Clarion 5.5. As Jim says, if VB programmers can use COM, so can you!

Posted Tuesday, February 13, 2001

COM: Getting Easier By The Minute (Part 1 of 3)

COM? Easy? Well, easier, at least. In this three-parter, Jim Kane, Clarion Magazine's resident guru of Microsoft component object technology, shows how to call COM objects with the new COM-compatible INTERFACE in Clarion 5.5. As Jim says, if VB programmers can use COM, so can you!

Posted Tuesday, February 06, 2001

The Five Minute Developer: Understanding Interfaces

GREGPlus Update Available

This week Jim Kane begins a three-parter on calling COM using Clarion Interfaces. But what are Interfaces anyway, and why should you care? Read on...

Posted Tuesday, February 06, 2001

February 2001 News

Clarion news for February, 2001.

Posted Thursday, February 01, 2001

January 2001 PDF

This PDF contains all of the January 2001 articles. You need access rights to all the January 2001 issues to dowload this file.

Posted Thursday, February 01, 2001

Clarion MAGAZINE

The Clarion Advisor: Procedure Prototypes

by Dave Harms

Published 2001-02-28

Compiler changes, unless they implement bug fixes, don’t often get a lot of press in the Clarion community. A few years ago, I believe in Clarion version 4, the compiler began handling an alternate form of procedure prototype. I’m pretty sure this slipped past a lot of developers, because I still often see procedures prototyped the "old" way.

In Clarion, a procedure has to be declared twice, first as a prototype (in a map or class), and second in the actual procedure code. Consider a small program, contained in a single source file:

```
program

   map
      testproc
   end

   code
   testproc()

testproc    procedure
   code
   message('Hi!')
```

This program shows a `testproc` procedure declared in the map, and defined at the end of the listing. The code statement following the map is the beginning of program execution. All this program does is call the `testproc` procedure, which displays a message box. Now suppose you want to pass the message to display to `testproc`. In the old days, you’d declare the data type in the prototype, and the variable name in the procedure, like this:

```
program

   map
      testproc(string)
   end

   code
   testproc('Hi!')
   testproc('Bye!')

testproc    procedure(msg)
   code
   message(msg)
```

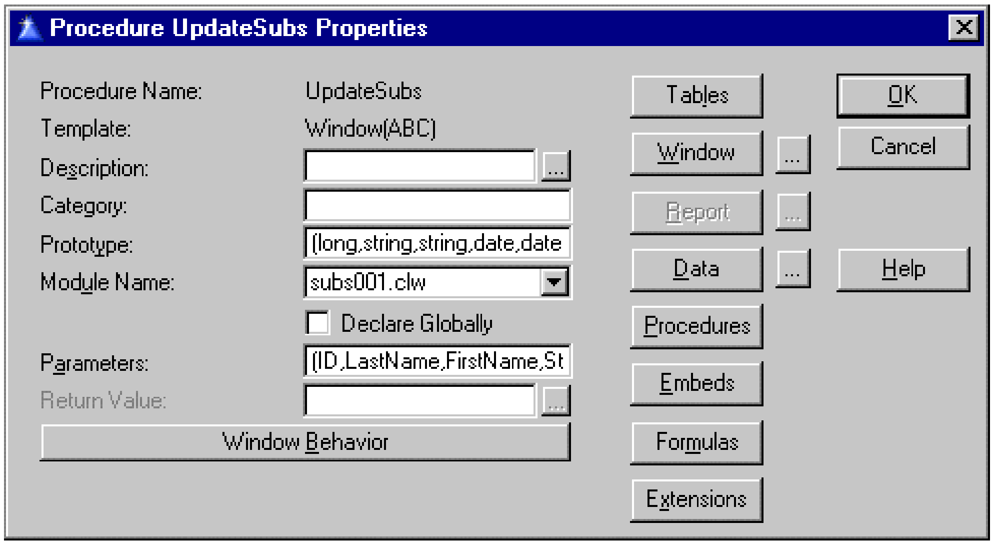
Guess what? This still works in Clarion 5.5. And you're probably wondering why it wouldn't work. But take a look at the same program, written to take advantage of the compiler feature introduced in Clarion 4:

```
program

   map
      testproc(string msg)
   end

   code
   testproc('Hi!')
   testproc('Bye!')

testproc    procedure(string msg)
   code
   message(msg)
```

This is the same program, except that now the parameter list in the prototype and in the procedure declaration are the same:

```
(string msg)
```

There are two advantages to using this newer syntax when declaring procedures with parameters. One is that you can always tell the parameter data type from the procedure definition; sometimes it's very helpful to know that you've passed, say, a string, not a group, or whether something is passed by value or address. With this syntax you don't have to chase down the prototype. The other benefit is that there's less chance of mixing up parameter names and data types in lengthy parameter lists, particularly in AppGen. Figure 1 shows an AppGen procedure properties window


Procedure UpdateSubs Properties

Procedure Name: UpdateSubs
Template: Window(ABC)
Description:
Category:
Prototype: (long,string,string,date,date
Module Name: subs001.clw
Declare Globally
Parameters: (ID,LastName,FirstName,St
Return Value:

Figure 1. Entering prototypes and parameters the old way.

The data types in the prototype are as follows (you won't have the line break character in your code):

```
(long,string,string,date,date,↵
   string,string,short)
```

and the variables in the parameters list are:

```
(ID,LastName,FirstName,StartDate,↵
   EndDate,Flag1,Flag2,Count)
```

Unfortunately, if you're typing the parameter list to match the prototype, the entry field isn't big enough to see everything at once, and it's easy to mix up or miss parameters. With the newer syntax, you only have to type either the prototype or the parameter list, and since the two should be identical you just copy what you've typed to the other field:

```
(long ID,string LastName,string FirstName,↵
   date StartDate,date EndDate,string Flag1,↵
   string Flag2,short Count)
```

The only exception to the rule of identical prototype and parameter list occurs when a procedure returns a value. In this case you append a return data type (such as ,long) to the prototype only. If you accidentally append the return data type to the parameter list, you will get a compiler error.

David Harms is an independent software developer and the co-author with Ross Santos of Developing Clarion for Windows Applications, published by SAMS (1995). He is also the editor and publisher of Clarion Magazine.

Reader Comments

Add a comment

Reborn Free CLARION online published by CoveComm Inc.

Clarion MAGAZINE

The Clarion Challenge: Useless Tab Text

by Dave Harms

Published 2001-02-28

During a recent tour of the Language Reference Manual I came across a curious and, as near as I can tell, completely useless feature. As you probably know, you can place tabs on any side of a sheet, left, right, top, bottom. For left and right side tabs you can force the text to run top to bottom or vice versa, assuming you use a TrueType font. You can also make the tab text appear upside down. I suppose this is to accommodate users who hang from gravity boots during working hours. See Figure 1.

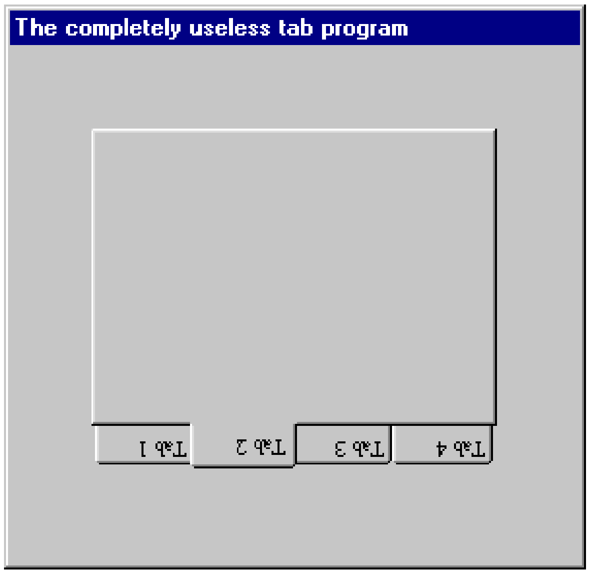


Figure 1. Inverted text on tabs

I was hard pressed to find a use for inverted tab text, but I finally settled on a little application that sends the selected tab scurrying around all four sides of a sheet. My intention was to have the text showing normally at the top of the sheet, then running top to bottom along the right side, inverted on the bottom, bottom to top on the left side, in an endless square loop. Listing 1 shows the source for an application that does this. You can also view a very short MPEG video of this stunningly pointless application.

The problem is that while this program demonstrates a use for inverted tab text, it also has a major bug. After I changed the orientation using the property syntax, I was unable to get the tab text back on its feet. As a result I had to make the text on the top of the sheet inverted as well! So challenge number one is to find a way of making the text appear right side up on the top of the sheet while the application is running.

The second challenge, of course, is to find another application for inverted tab text, and the third is to find a very compact way of rewriting this program. Three challenges for the price of one! Good luck!

Send your answers to the challenge to editor@clarionmag.com, and I'll post the results in a few weeks.

Download the source

Listing 1. The completely useless tab program

```
   program

   map
      testproc
   end

   code
   testproc()

testproc    procedure

Window WINDOW('The completely useless tab program')|
   ,AT(,,196,152),FONT('Times New Roman',,,,CHARSET:ANSI)|
   ,TIMER(10),GRAY
       SHEET,AT(28,24,139,100),USE(?Sheet),SPREAD
         TAB('Tab 1'),USE(?Tab1)
         END
         TAB('Tab 2'),USE(?Tab2)
         END
         TAB('Tab 3'),USE(?Tab3)
         END
         TAB('Tab 4'),USE(?Tab4)
         END
       END
     END

count byte(1)
side  byte(1)
tab   short(1)

   code
   open(window)
   display()
   accept
      case event()
      of event:timer
         case count
         of 1
```

```
              side = 1
              tab = 1
              ?sheet{prop:above} = 1
              ?sheet{prop:default} = 1
              ?sheet{prop:upsidedown} = 0
           of 5
              side = 2
              tab = 1
              ?sheet{prop:right} = 1
              ?sheet{prop:down} = 1
           of 9
              side = 3
              tab = 4
              ?sheet{prop:upsidedown} = 1
              ?sheet{prop:below} = 1
           of 13
              tab = 4
              side = 4
              ?sheet{prop:left} = 1
              ?sheet{prop:up} = 1
           end
           execute (tab)
              select(?tab1)
              select(?tab2)
              select(?tab3)
              select(?tab4)
           end
           if side < 3
              tab += 1
           else
              tab -= 1
           end
           if count = 16
              count = 1
           else
              count +=1
           end
           display(?sheet)
        end
     end
     close(window)
```

David Harms is an independent software developer and the co-author with Ross Santos of Developing Clarion for Windows Applications, published by SAMS (1995). He is also the editor and publisher of Clarion Magazine.

Reader Comments

Add a comment

Introduction To SQL: Part 1

by Dave Harms

Published 2001-02-28

In a recent Clarion Magazine poll I asked developers how much of their development effort was directed at SQL databases. Out of 193 responses, 46% did no SQL development, 20% did half or less development for SQL, and 34% did most of their development for SQL.

Interest in SQL is at an all-time high in the Clarion development community, but many are still uncertain about whether to go to SQL, or how to make the switch. In this article I'll compare SQL databases to TPS databases, and suggest some reasons and strategies for moving to SQL.

What is SQL?

SQL stands for Structured Query Language, and can be pronounced either "sequel" or "ess-queue-ell" depending on which side of the religious war you prefer. SQL was originally developed by IBM, and inspired by IBM researcher E.F. Codd. As the name suggests, SQL is an English-like language that lets you selectively retrieve data from a database. For instance, the statement:

```
SELECT FirstName, LastName FROM Names WHERE Country= 'Canada'
```

will retrieve the first and last names of all the Canadians from a table called Names. It could be that your application created that `SELECT` statement, or perhaps you typed it in yourself using a special database client. In the former case, your application will then display the names (perhaps in a browse); in the latter, the database client will display a list of names.

Actually SQL can do a whole lot more than just retrieve data. You can use SQL to create and modify databases, tables, and keys as well as update, insert, and delete data. You can also use SQL to enforce relationships between tables, creating what's called a relational database. (For more on relational databases, see Tom Ruby's series of articles on database normalization.) A relational database doesn't have to be a SQL database, but in most cases you'll find that relational and SQL go hand in hand. That's because SQL isn't just a standard query language, it's also a standard language for defining tables and their relationships.

SQL is a bit like the standard Clarion language grammar for accessing databases. The Clarion language has `CREATE`, `SET`, `NEXT`, `PREVIOUS`, `ADD`, and `DELETE` statements which you can use on a variety of different file formats; all you need is a file driver appropriate for the data files you're accessing. SQL has `CREATE`, `SELECT`, `INSERT`, `DELETE`, and `UPDATE` statements which will work on any SQL database; all you need is a way of presenting those statements to the database.

If SQL and Clarion take similar approaches to handling data, why would you bother with SQL? Part of the answer lies in the differences between relational and flat file databases.

Relational vs. flat file databases

A lot of people think of Clarion as a relational database development tool. In fact, Clarion is really a database-agnostic fourth generation language, or 4GL. You can use Clarion with flat file and relational databases. And for those of you who think of TPS files as a relational database, nope, that isn't the case. TPS files are flat files.

A flat file database simply contains data files; there is no code involved in processing the data, rather, the application does all the work. If, for instance, you have an order entry application, you'll probably have something like an `Order` header table, which contains one record for each order, and an `OrderDetail` table, which contains one record for each item purchased as part of a given order. If you delete an `Order` record, and there are related `OrderDetail` records, you could leave behind orphaned records. To prevent this you'll want to either delete those related records, or prevent the deletion of the Order record. In a flat file database your application has to manage the relationships between tables; in a relational database, this is the server's job.

> **NOTE:** The term *database server* has two common meanings: one is the software that manages the database, and the other is the physical computer that holds the database and the database server software. In most cases, database server software is installed on a dedicated machine, so that other processes don't slow database handling.

Clarion is quite good at handling relationships between tables in a flat file database. You can define those relationships in the data dictionary, and by clicking a few options you can tell Clarion to generate appropriate code to cascade or restrict deletes, and so forth. What Clarion can't do, however, is stop any other program from violating the rules you've so carefully defined in the dictionary. Your data is just sitting out there in a no-brain TPS file, waiting to be trashed by any program that can read the file.

Client/server and relational databases

Another term frequently used in database application development is *client/server*. If you're using a flat file (i.e. TPS database) on local computer, you're not doing client/server. If you place that flat file database on a server, so more than one program can work with the data at a time, you're still not doing client/server. In a client/server environment, both the client and the server have some intelligence. Relational SQL databases are a common example of client/server processing.

The database server's intelligence provides a number of key advantages over flat file databases, including speed, data integrity, compatibility with other products, ease of administration, and scalability.

The need for speed

A SQL database can provide dramatic speed improvements over flat file databases in a network environment,(from this point on, I'll use the term "SQL database" to mean "relational SQL database," but keep in mind that not all SQL databases are fully relational). If you're using flat files such as TPS files, every time you request a record from a table, all of the fields in that record have to travel across the network. If your table has fifteen fields with a total of 500 bytes per row, and you only want to retrieve one 25 byte field, then you're moving 20 times more data across the network than you need! In most cases you'll want more than just five percent of the row's data, but if you even use only half of the available data, you've doubled your bandwidth requirements by using a flat file.

And since the flat file database contains no intelligence, it doesn't know anything about how tables are related. If your browse makes use of multiple tables, your application will have to retrieve full rows of data from the related tables as well, and match those results with the first table.

If you're using a SQL database, your application sends a `SELECT` statement to the database server, which sends only the requested fields across the network. Your application can also tell the database server to return fields in related tables; it's up to the server to decide how it locates this related data, and it only sends the requested fields back to the application. This also means the client computer doesn't have to expend processor cycles matching the related records.

> **NOTE:** Reducing the client computer's processing load may or may not speed the application. Much will depend on the speed of the network, and the load on the database server.

Data integrity

I've already mentioned data integrity in the case of orphaned records. Although you can easily create a Clarion application that manages related table data, this puts the burden entirely on the application. In a shared database environment you often have multiple applications working with the same data. Even if you've created all these applications with Clarion, you'll need to ensure that you use the same dictionary, or maintain the same relationship settings across multiple dictionaries. And if someone wants to work with the database using, say, Excel, or a Visual Basic application, all bets are off.

In a heterogeneous environment, you're far better off putting core database integrity rules in the database itself. These rules can embody the sort of basic relational integrity (RI) options you see in the Clarion data dictionary (restrict or cascade changes and deletes), and they can specify default and allowed values for individual fields. In most SQL databases you can also create triggers, which execute SQL code when a certain action (like a delete or insert) happens. Triggers can call stored procedures, which are functions written in SQL and stored on the server.

You can build a SQL database with sufficient RI and other rules so that it's virtually impossible for any application (or any individual executing SQL statements by hand) to corrupt that database. I'll discuss this subject in more detail in upcoming articles.

Compatibility

Although the major database vendors each have their own flavor of SQL, there is enough adherence to the 1992 ANSI SQL standard that you can readily port most applications from one server to the next. And because SQL is a relatively standardized language, there are many tools and utilities available, for everything from database design to syntax checking to reporting. One place to look for SQL tools is the WinFiles web site.

Ease of administration

As I indicated earlier, SQL isn't just for querying data. You also use SQL to create and alter databases, tables, and indexes. It's easy to add a field to or remove a field from an existing table, or to change a field's data type or default value. In a TPS file, changing the table definition doesn't change the physical data; you still have to create a conversion program, or use the data dictionary's table conversion feature.

You can also easily do mass updates in SQL by applying UPDATE or DELETE statements to a selected set of records. While mass updates are inherently dangerous (you can easily wipe out an entire table), they're also amazingly useful.

Scalability

Applications are not only a lot bigger than they used to be, but they typically deal with a lot more data. TPS files can store a lot more data than the old Clarion DAT files, but in general flat file databases hit storage limits and performance walls a lot sooner than SQL databases. The history of each kind of database suggests this is likely to happen: Flat file databases derive from small, single user systems, and SQL databases started out on the big iron. You wouldn't want to store a terabyte of data in a TPS file, even if it were possible (it isn't – the maximum size of a TPS file is two gigabytes).

SQL databases are designed to hold massive amounts of data, and to work with that data efficiently. SQL server performance scales with the hardware, while the same isn't generally true of flat file databases. At the same time, SQL servers are invading the personal computer data space, and some of the more popular SQL databases are even available for handheld computers.

Clarion and SQL

Clarion's support for SQL databases has improved considerably in recent years. Prior to ABC, your only real option was the Cowboy

Computing Solutions SQL templates. ABC provides acceptable SQL features and performance, although for serious work you should still take a look at the CCS SQL templates.

If you already have an SQL database, and you have a suitable SQL or ODBC driver for that database, all you probably need to do is import the tables into a dictionary and you can start creating your application in the usual manner.

If you're porting a TPS database to SQL, I strongly suggest you read the free articles on this subject by Stephen Mull and Scott Ferrett. There are a number of important data type and table design requirements in SQL that don't exist in TPS files, and there's a good chance that you'll need to make at least a few minor changes. If you don't yet have a SQL database to play with, take a look at Tom Hebenstreit's suggestions for getting into SQL on the cheap. One free SQL database increasingly popular with Clarion developers is MySQL, and you can read more about this database in my earlier articles on using MySQL with Clarion.

I hope I've whet your appetite for SQL databases. That was the appetizer; next week I'll serve up a first course of database and table creation.

David Harms is an independent software developer and the co-author with Ross Santos of Developing Clarion for Windows Applications, published by SAMS (1995). He is also the editor and publisher of Clarion Magazine.

Reader Comments

Add a comment

Reborn Free
CLARION online
published by CoveComm Inc.
Clarion MAGAZINE

The Cranky Programmer: Got Them Bloated ABC Blues

by Cranky

Published 2001-02-22

Got Them Bloated ABC Blues

Do you know what a gnat is?

It’s a teeny-tiny-itsy-bitsy flying insect that thinks you make a pretty tasty snack. (Of course, Hannibal Lecter might think the same thing about you, but I digress.)

In ones or twos, you probably never even notice gnats; they’re just a petty annoyance, quickly forgotten. The problem is, they tend to descend in vast swarms, and all those minor annoyances soon become a major pain-in-the... well, pick your favorite body part (or maybe that should be your *least* favorite body part).

That’s what I’ve been going through lately, only it’s nits instead of gnats – a swarm of annoying little things that have finally added up to a major attack of Cranky-itis.

That’s right. Just like Hannibal, I’m baaaaaaaaaaaack. And this is only the beginning.

Today’s tasty topic is...

ABC gives you that little something extra

I was setting up a new ABC multi-DLL application the other day in Clarion 5.5, and in the process of creating the initial data DLL I encountered some annoying link errors. After a few moments thought (more on that later), I was able to fix the errors, compile the app, and proceed with making a couple other applications for related DLLs and the EXE.

Each of these applications was basically a stub, i.e. just an entry point without much of anything else. The EXE contained just a small menu, and the entire application only had 10 or 15 files defined. Nothing special yet.

In the course of poking around in the folder for my new application, I noticed that the EXE and most of the DLLs were around 20-25k in size. Cool, that's what I'd expect.

Then I took a look at the data DLL and fell out of my chair.

Well, actually, I *almost* fell out of my chair. I got my new Super Executive Hyper Swivel Mega Roller Ultra Lounger with arm rests for just that reason – I was tired of hitting the floor every time I ran into a new "undocumented feature'.

Where was I? Oh, yeah, the data DLL. It weighed in at a hefty 1.5 megs, and that was *without* debug turned on. Yikes! What was going on?

Time for a bit of "sleuthing,", otherwise known as crying and whining on the newsgroups.

Can I pack that for you, sir?

Did you know that when you create an ABC multi-DLL application, *every* "ABC compliant" class is included in your data DLL? I don't mean just every SoftVelocity ABC class; I mean every single compliant class from every single template set and/or tool that you have installed for that version of Clarion.

What constitutes an ABC compliant class? There are several requirements, but the main one is, to quote the online help:

> The header file (.INC) containing the CLASS declaration must contain the following comment before compilable code begins:
>
> `!ABCIncludeFile`

If a class is in your \LibSrc folder and has that line in it, it is probably ABC compliant. And if it is compliant, then it is a) loaded when you open your first ABC app of the day (the ever popular "Please wait... Reading ABC header files" message), and b) included in every data DLL you create (i.e., a DLL that doesn't have the "Generate template globals and ABC's as EXTERNAL" flag turned on).

So it doesn't matter whether you are using the class or not, it gets compiled in. That's why even for fairly small and simple ABC applications, the size of the data DLL can be all out of proportion to the number of files you have defined or the classes actually being used.

It is also the cause of a fairly common cry for help on the newsgroups that goes something like this:

"When I try to compile my data DLL, I'm getting link errors for a bunch of stuff that starts with "vsaftp.' I don't even know what that is!" (Umm, remember those link errors I mentioned earlier?)

The culprit in the above case is Vince Sorensen's ABC Free templates, a *very* useful (and highly recommended) collection of free templates and tools. The package includes a wrapper class for the Catalyst SocketTools ftp library (another third party product), and guess what -- the wrapper is ABC compliant. Thus, the ABC Free classes get sucked into your data DLL and compiled along with everything else.

The problem, of course, is that if the hapless recipient of the link errors doesn't happen to own SocketTools, there is nothing there to link to and now they are stuck.

So what's the solution?

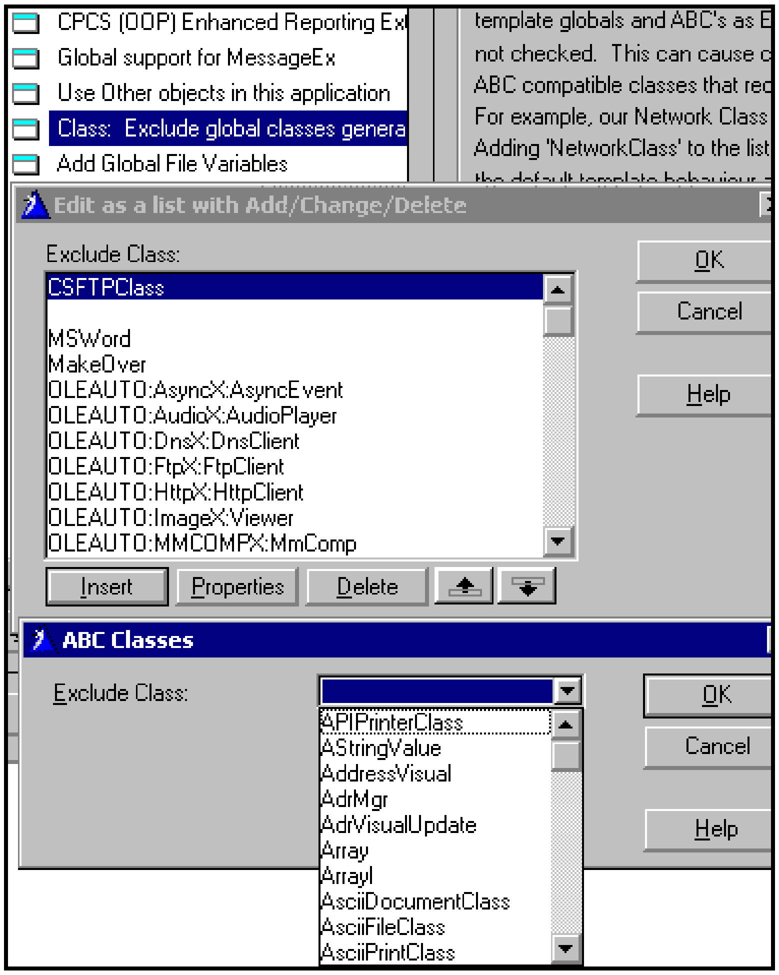
You really have two basic options. The first is to not use some of these free tools. After all, if they aren't there, they can't cause problems. The major drawback to this, of course, is that you are severely limiting your options and not "working smarter" by taking advantage of the time and effort savings that these kinds of no-cost templates and tools can provide. The other option is to selectively remove the classes you don't need.

I vastly prefer the second option, and there's even a free template that does this in... you guessed it, Vince's ABC Free templates.

Fixes *and* cleans for one low, low price!

One of the little gems Vince provides is a template to exclude generation of global classes, i.e., not include them in your data DLL. This not only eliminates errors like the ones caused by the ftp wrapper, it has the cardinal virtue of letting you trim a lot of that extra baggage in your data DLL from classes that you simply aren't using.

To use the template, you add it to your data DLL application under Global Extensions and then just choose the classes you want to exclude by picking them from a convenient list. It looks something like this:



CPCS (OOP) Enhanced Reporting Ex
Global support for MessageEx
Use Other objects in this application
Class: Exclude global classes genera
Add Global File Variables

template globals and ABC's as E
not checked. This can cause c
ABC compatible classes that rec
For example, our Network Class
Adding 'NetworkClass' to the list
the default template behaviour

Exclude Class:

CSFTPClass

MSWord
MakeOver
OLEAUTO:AsyncX:AsyncEvent
OLEAUTO:AudioX:AudioPlayer
OLEAUTO:DnsX:DnsClient
OLEAUTO:FtpX:FtpClient
OLEAUTO:HttpX:HttpClient
OLEAUTO:ImageX:Viewer
OLEAUTO:MMCOMPX:MmComp

Exclude Class:

APIPrinterClass
AStringValue
AddressVisual
AdrMgr
AdrVisualUpdate
Array
ArrayI
AsciiDocumentClass
AsciiFileClass
AsciiPrintClass

Figure 1. The "Exclude classes" template in action

As Figure 1 shows, you click on Insert and then just drop down the list of ABC compliant classes. Select a class, say OK and –poof-, it's gone from your app. If you look at the first one on my list, you'll see it is "CSFTPClass", the class wrapper that causes the link errors.

Remember that application I was talking about at the start of this column? The one with a data DLL that weighed in at around 1.5 megs? After some judicious class pruning using this template, I not only eliminated the link errors but also reduced the size to under 500K. Not bad for a few minutes "work", and the application showed another benefit in a faster load time.

So, don't let ABC turn your programs into disk pigs – slim down those applications!

Tidying up

By the way, with this template you can exclude *any* ABC class, including the base SoftVelocity ABC classes. If you want to be ruthless in trying to exclude absolutely everything that your app isn't using, feel free to add whatever ABC classes you don't think you need to the exclude list. If you excluded something that you shouldn't have (like maybe the BrowseClass), you'll just get a million compile errors. Take that item back off the list and everything should be fine again.

Note: All of this only applies if you are creating a multi-DLL application. If you are compiling local (i.e., one big EXE with no DLL files), only the classes you actually use will be included.

Is something annoying you? Even better, have you figured out a way around some other Clarion annoyance? Share it with Cranky, and I'll share it with the world!

I'm waiting...

Reader Comments

Add a comment

Reborn Free CLARION online published by CoveComm Inc.

Clarion MAGAZINE

COM: Getting Easier By The Minute (Part 3 of 3)

by Jim Kane

Published 2001-02-22

Part 3 of 3

In the first and second installments in this series I showed how Clarion COM is really pretty easy, using Clarion Interfaces that correspond to COM Interfaces. The approach I outlined uses early binding, where you know the names of the Interface methods at compile time.

You can also locate and call COM methods at runtime, using late binding. This is what the Clarion OLE control does, and you can do it too in your code. I'll touch on this briefly, and then I'll go back to early binding to finish up the IXMLSaxReader wrapper classes.

Late Binding

There are two basic types of interfaces used by COM objects: IUnknown and IDispatch. IDispatch is derived from IUnknown, and it adds a few additional methods used to call COM methods without first prototyping them. This is also known as late binding, and it's what the Clarion OLE control uses. The IDispatch interface looks like this:

```
IDISPATCH       INTERFACE(IUNKNOWN),COM
GetTypeInfocount  Procedure(*Unsigned pctinfo),hresult
GetTypeInfo     Procedure(unsigned itinfo, ↵
                  Unsigned lcid,*long pptinfo),hresult
GetIdsOfNames   Procedure(long riid, long rgsznames, ↵
                  unsigned cnames,unsigned lcid, ↵
                  *long rgdispid),hresult
Invoke          Procedure(long dispidmember, long riid,↵
                  unsigned lcid,ushort flags, ↵
                  long pdispparams, *long pvarresult, ↵
                  *long pexceptinfo,*unsigned puArgErr)↵
                  ,hresult
             END
```

Of the additional methods in IDispatch, Invoke is the most noteworthy. Lets say you have an IDispatch interface that is known to support a method called DoIt. Here is the prototype:

```
ITestcom Interface(IDispatch),COM
End
```

Notice DoIt is not among the methods in the interface. The reason for this is all method calls go through Invoke when you are using IDispatch. You simply pass the string 'DoIt' to Invoke, along with any parameters you need to pass. Invoke parses the function name and parameters, and then calls the appropriate class method for you. In this scheme there is no way to call DoIt directly. Invoke is very convenient because you don't have to do any prototyping, but this kind of late binding is slow process with a lot of overhead. Given a choice, never call IDispatch.Invoke. Because Invoke is so slow, Microsoft currently recommends creating dual interfaces. A dual interface includes all the methods of IDispatch, plus all the methods IDispatch supports which can be called directly. If the hypothetical ITestCom interface were a dual interface it would look like this:

```
ITestCom   Interface(IDispatch),COM
DoIt         procedure(),hresult
           End
```

With a dual interface you can call DoIt via IDispatch.Invoke or you can call the DoIt method directly. Why bother? If you create COM objects with a dual interface, scripting languages can call the object via IDispatch and languages that support early binding can have the speed and efficiency of early binding; these languages can call the methods directly without going through IDispatch.Invoke. The original Clarion OLE control only called COM methods via IDispatch (late binding). Now, with the advent of the Interface structure, you can also call the methods of a dual interface directly (early binding).

To call a COM object you need an interface definition, and an object that implements that definition. An interface contains no code – it's just a group of prototypes that tells Clarion how to call a COM object. Working with COM is all about matching COM objects up with Clarion interface definitions, using the following sequence of events:

- get the CLSID and IID from the typelib
- call coIntialize to wake up COM
- call CoCreateInstance to get the pointer to the interface
- feed the pointer obtained from CoCreateIntance to the interface definition
- you're ready to call OLE methods!

Getting back to ISaxReader, the two most important methods you need to call are putContentHandler and ParseURL:

```
putContentHandler procedure(long lpISaxContentHandler)↵
```

```
                        ,hresult
ParseURL          procedure(long lpURL),hresult
```

PutContentHandler is how you tell IsaxReader what the address of the ISaxContentHandler interface it should send results to is. ParseURL is where you specify the name of the file to parse as a wide string. Because ParseURL expects a resource in URL format, precede the name of the file with file://.

Next you need to create the ISaxContentHandler interface and class. The IXMLSaxReader is a generic XML parser; it knows how to read an XML file and extract the tags and tag attributes. But that's all it does; you still have to supply code to do something with that data. In my case I'm writing XML file data to a TPS file. The code that does this is in the ISaxCl.EndElement method in saxxml.clw.

When you call a COM object you only need the interface, but to implement or create a COM object, you need both the interface for others to call (well it would be a shame to create a COM object and not have any one call it!) and the class itself that does the work. The IXMLSaxReader requires you to create a class that implements the ISaxContentHandler interface, which you can find in the MSXML3 typelib. Prototype this interface as follows:

```
ISaxContent    INTERFACE(IUnknowntype),COM,type
documentLocator      Procedure(long lpLocatorObj),hresult
StartDocument        Procedure(),hresult
EndDocument          Procedure(),hresult
StartPrefixMapping   Procedure(long lpWPrefix, ↵
                       long cbPrefix, long lpWURI,↵
                       long cbURI),hresult
EndPrefixMapping     Procedure(long lpWPrefix, ↵
                       long cbPrefix),hresult
StartElement         Procedure(long lpWNamespaceURI,
                       long cbNameSpaceURI,↵
                       long lpWLocalName, ↵
                       long cbLocalName, ↵
                       long lpWQName, long cbQName,↵
                       long lpObjAttributes),hresult
EndElement           Procedure(long lpWNameSpaceURI, ↵
                       long cbNameSpaceURI, ↵
                       long lpWLocalName,↵
                       long cbLocalName, ↵
                       long lpWQName,↵
                       long cbQName),hresult
Characters           Procedure(long lpWChars,↵
                       long cbChars)↵
                       ,hresult
IgnoreableWhitespace Procedure(long lpWchars, ↵
                       long cbChars),hresult
processingInstruction Procedure(long lpWTarget, ↵
                       long cbTarget,long lpWData,↵
                       long cbData),hresult
SkippedEntity        Procedure(long lpWName, ↵
                       long cbName),hresult
```

```
                    End
```

Notice first that ISaxContent a type definition. The "real" interface is the class as you will see shortly. Among the parameter names I use the prefix lp to signify long pointer or an address. My usage of lp tells you I'm very old. Back in the days of 16 bit there was a distinction between long and short pointers. Out of habit I still refer to addresses as lpSomething. I use lpW as my abbreviation for the address of a wide string. Keep in mind that the IXMLSaxReader works in wide (Unicode) strings , so the COM object that handles the parser's output needs to work in wide strings also.

Most of the methods in ISaxContent, such as StartElement, contain the address of a wide string followed by its length. I use cb to stand for count byte or length of the string. For the most part, the code in each of these methods simply takes the address of the wide string with it's length and converts the wide string into a Clarion string. Once converted into a Clarion string, I pass the Clarion string back to my IsaxCl class.

The complete circle is this: IsaxCl calls the IXMLSaxReader interface and tells it to parse a file; IXMLSaxReader sends the output to a class which implements an ISaxContent interface. Once ISaxContent gets the information, it converts it to a Clarion format and passes it back to the IsaxCl. The IsaxCl can then store the information in a file or queue or use the information directly.

Interpreting XML

The ISaxContentCl class interprets the XML elements the parser extracts from the XML file. Here is a basic XML file with one record:

```
<?xml version="1.0" ?>
<ORDER>
  <ITEM>Widgets</ITEM>
  <QTY>5</QTY>

</ORDER>
```

The IXMLSaxReader handles the sequence of calls required to correctly read the XML data. First the parser calls the content handler's StartDocument method, then for each element in the XML file the parser calls the content handler's StartElement, Characters, and EndElement methods.

In StartDocument you open or create a file. StartElement tells you the record type or field name, and EndElement indicates that the parser is done with that element. In between StartElement and EndElement, one or more calls to Characters supply the element value. When you encounter a call to EndElement with a record name of Order, you save the record.

Repeat this process repeats for each element in the XML file until you get an error or the IXMLSaxReader automatically calls

EndDocument. In both the error handler and EndDocument you should close the file you are saving to.

The best way to learn how the XML file is parsed is to put MESSAGE statements into each of the IsaxCl methods corresponding to an IsaxContext method and display the information it receives. Believe me, it’s not very complex, VB programmers have done it.

Now with out further ado, but perhaps a drum role, here is a COM Class implementing the ISaxContent interface:

```
ISaxContent        INTERFACE(IUnknowntype),COM,type
documentLocator      Procedure(long lpLocatorObj),hresult
StartDocument        Procedure(),hresult
EndDocument          Procedure(),hresult
StartPrefixMapping   Procedure(long lpWPrefix, ↵
                       long cbPrefix,long lpWURI, ↵
                       long cbURI),hresult
EndPrefixMapping     Procedure(long lpWPrefix, ↵
                       long cbPrefix),hresult
StartElement         Procedure(long lpWNamespaceURI, ↵
                       long cbNameSpaceURI, ↵
                       long lpWLocalName, ↵
                       long cbLocalName, ↵
                       long lpWQName, ↵
                       long cbQName, ↵
                       long lpObjAttributes)↵
                       ,hresult
EndElement           Procedure(long lpWNameSpaceURI, ↵
                       long cbNameSpaceURI, ↵
                       long lpWLocalName, ↵
                       long cbLocalName, long lpWQName, ↵
                       long cbQName),hresult
Characters           Procedure(long lpWChars, ↵
                       long cbChars),hresult
ignoreableWhitespace Procedure(long lpWchars, ↵
                       long cbChars),hresult
processingInstruction   Procedure(long lpWTarget, ↵
                          long cbTarget, long lpWData, ↵
                          long cbData),hresult
SkippedEntity           Procedure(long lpWName, ↵
                          long cbName),hresult
                   End

ISaxContentClType  Class,Implements(ISaxContent),↵
                     Module('ISaxCl.clw')↵
                      ,DLL(_ABCDllMode_),type
refcount              long(0)
debugmode             byte(0)
ISaxCl                &IsaxClType !reference to parent
StrCl                 &StrClType  !helper class
    End
```

Here is one typical class method:

```
ISaxContentclType.ISaxContent.EndElement ↵
                Procedure(long lpWNameSpaceURI,
                long cbNameSpaceURI, ↵
                long lpWLocalName,
                long cbLocalName, long lpWQName, ↵
                long cbQName)

Res          byte(0)
NameSpaceURI &string
LocalName    &string
QName        &String

  Code
  !convert each of the wide
  !strings to Clarion strings
  NameSpaceuri&=SELF.StrCl.WideStrToCwAlloc↵
    (lpwNamespaceURI,res,cbNameSpaceUri)
  if ~res then localname&=SELF.StrCl.WideStrToCwAlloc↵
    (lpwLocalName,res,cbLocalName).
  if ~res then QName&=SELF.StrCl.WideStrToCWAlloc↵
    (lpwQName,res,cbQName).
  !pass the Clarion strings to
  !the corresponding IsaxCl method
  if ~res then res=SELF.ISaxCl.EndElement↵
    (NameSpaceURI,Localname,QName).
  !dispose of the Clarion strings new()ed by strcl.
  dispose(namespaceuri)
  dispose(localname)
  dispose(qname)
  !message if an error and debug mode is on.
  if res and SELF.DebugMode  |
    then message('endelement failed').
  !convert our nice simple 0 code
  !to a com hresult (0=S_OK)
  return Choose(res=0,0,E_Fail)
```

The corresponding `IsaxCl` method does just about nothing other than clean up:

```
ISaxClType.EndElement              ↵
  Procedure(*String NameSpaceURI, *String LocalName, ↵
  *string QName)!,byte,proc

  Code
  !At this point, SELF.ElementText contains
  !the text of the element. Insert your code here
  !to save it to a queue or derive this method
  !Clear the inelement flag used by the
  !character method
  SELF.Inelement=False
  clear(SELF.ElementText)
  return return:benign
```

Perhaps the odd thing is you never call the `EndElement` method – the Microsoft `IXMLSaxReader`'s class calls this method via the `ISaxContentCl` when it has data from the parsing of the XML file.

For those of you familiar with the term, this is a classic callback function.

Despite all the hype about creating a COM object, all you really need to do is create a normal Clarion class that implements a Clarion COM interface.:

```
ISaxContentClType
Class,Implements(ISaxContent),Module('ISaxCl.clw'),↵
   DLL(_ABCDllMode_),type
```

And:

```
ISaxContentclType.ISaxContent.EndElement Procedure(↵
    long lpWNameSpaceURI, long  cbNameSpaceURI, ↵
    long lpWLocalName, long cbLocalName, ↵
    long lpWQName, long cbQName)
```

That's it! There's nothing up my sleeve; just add an interface with the COM attribute and your class become a COM object callable by standard COM protocols! Now all you need to do is connect the dots, or in this case connect the `IsaxCl` class to the `ISaxContent` XML content handler. That happens in the `IsaxCl.init` method, like this:

```
!create a few classes
!string class for Clarion to wide string conversion
 SELF.StrCl &= NEW StrClType
!create the ISaxContectCl
SELF.ISaxContentCl &= New ISaxContentClType
!intialize the ISaxContentCl
!Pass the IsaxCl or SELF to the ISaxContentCl
!you just created
SELF.ISaxContentCl.ISaxCl&=SELF
!Give ISaxContentCl it's own string class
SELF.ISaxContentCl.StrCl&=New StrClType
```

At this point the `ISaxContent` interface and class, and the `IsaxReader` interface are all set up and ready to call. Notice that you pass the address of the `ISaxContent` interface and not the address of the class because the interface is what `IXMLSaxReader` will use.

```
!if pass the content Interfaces
!to the IXMLSAXReader Interface
hr=SELF.ISaxReader.PutContentHandler↵
  (address(SELF.ISaxContentCL.ISaxContent))
if hr<0 then !standard com error handling

res=return:fatal end
```

To begin the parsing, convert the XML file name to URL form and call `ParseURL` with the address of that string:

```
!this starts the actual parsing of the file
```

```
!pXMLSourceURL='file://order.xml'
wurl&=SELF.StrCl.CWToWideStralloc(pXMLSourceURL)
hr=SELF.ISaxReader.ParseUrl(address(wURL))
dispose(wURL)
if hr<0 then
  res=return:fatal
end
return res
```

In the downloadable source you'll find the test application, made up of SaxXML.Prj and SaxXml.clw. That little test application uses the IsaxCl to parse Order.XML and save the information into outfile.tps, which has this format:

```
OutFile   FILE,DRIVER('TOPSPEED'),CREATE,BINDABLE
Record      RECORD,PRE()
Item          string(20)
Qty           long
            END
          END
```

The test program declares IsaxCl like this:

```
ISaxCl class(ISaxClType)
EndElement Procedure(*String NameSpaceURI, ↵
  *String LocalName, *string QName),byte,proc,virtual
End
```

Whenever the parser calls EndElement, EndElement looks for a field completion, or a record completion. When the EndElement LocalName is Order it's time to add the record:

```
ISaxCl.EndElement   Procedure(*String NameSpaceURI, ↵
    *String LocalName, *string QName)

anyfield  any

  code
  if Upper(localname)='ORDER' then add(Outfile) else
    loop Idx=1 to Outfile{prop:fields}
      if Upper(localname)=|
        Upper(Outfile{prop:Label,Idx})
        anyfield&=what(Outfile.Record,idx)
        anyfield=SELF.ElementText
        anyfield&=NULL
      end
    end
  end
  return(Parent.EndElement|
    (NameSpaceURI,LocalName,QName))
```

Note that I've used prop:label and What to save myself the trouble of hard coding field names.

As you look through the code you will see a few other objects that handle errors and XML attributes. That code follows the same pattern as the code I've already described: you prototype an interface, define a corresponding class that implements the interface, and set things up by passing the address of the interface to IXMLSaxReader so IXMLSaxReader's class knows what to call.

The only other detail in the code worth special mention is the use of AddRef and Release. As I discussed in Part 1 of this series, when the reference count inside an object gets to zero, the COM object self destructs. If IXMLSaxReader passes you an object, the first thing you should do with it is call AddRef to increment that object's internal counter. That ensures that the object won't be disposed until after you call Release.

If an error occurs, IXMLSaxReader passes the address of a locator object's IsaxLocator interface along with the other error information. This happens in the IsaxClType.ErrorHandler method. The ErrorHandler method then calls the ReadLocator method just to read the locator and get the row and column in the XML file where the error occurred. The code in IsaxClType.ReadLocator method shows how to call the interface on the locator object. The first call is to AddRef and the last to Release. In between those calls the locator reads the column and row. You must be very careful to balance calls to AddRef and Release. Here is the code:

```
ISaxClType.ReadLocator     procedure(long lpLocator, ↵
    *long pColumn, *long pLine)

res         byte(Return:Fatal)
ISaxLocator &ISaxLocatorType

  code
  if ~lpLocator then return res.
  ISaxLocator&=(lpLocator)
  !you let the locator know you are using
  !it so it doesn't self destruct
  ISaxLocator.AddRef()
  if ISaxLocator.GetColumnNumber(pColumn)<0 then
    ISaxLocator.Release()
    clear(pColumn)
    clear(pLine)
    return res
  end
  if ISaxLocator.getLineNumber(pLine)<0 then
    ISaxLocator.Release()
    clear(pLine)
    clear(pcolumn)
    return res
  end
  !let the locator know you are done with it and it
  ! can self destruct any time it wants
  ISaxLocator.Release()
  return return:benign
```

If you would like to see the error handling and the Locator in action, find the `jni_server.xml` file in the downloadable zip. If you load this XML file with Internet Explorer you will find out it has a syntax error. If you try to read it with the demo program, it will report the same syntax error with the location. That is the locator object in action reporting the column and row where the error is.

In the above example calls to `AddRef` and `Release` are balanced. If you call `Release` without `AddRef`, the object will probably be destroyed while your program or some other program is trying to use it.. Since your code didn't create the `ISaxReader` object, it shouldn't release it. On the other hand your code did call `CoCreateInstance` (by calling `stdcom.GetInterface`) to create the class that contained the IXMLSaxReader interface so your code should dispose of the object by calling `Release`. The last few lines of code in the `init` method do just that:

```
!if ISaxReader was created, then Release it
if lpISaxReader then
  SELF.ISaxReader.Release
End
```

If you want to check that the reference count is okay, put a message around the Release like this:

```
Message(SELF.IsaxReader.Release(),'Ref Count')
```

The value displayed should be 0 indicating the object was disposed.

As I was building this project, my first implementation of `ISaxContent` just had a method statement in each `ISaxContent` method. I had no wide string conversion, no call to `IsaxCl`, and I was ignoring the parsed data. You may think it corny, but when that `MESSAGE` statement popped up and I knew the Microsoft `IXMLSaxReader` was calling my COM object, I was quite excited and did a little dance of joy. Fortunately no video tape was rolling – at least I hope not!

After that little victory dance the floodgates opened, and I've built little COM objects that can be called by both the usual Clarion OLE Control and by early binding Interface methods. I've also written little COM objects to modify the behavior of Internet Explorer (Browser helper objects) and MS Office but have run into some trouble getting them to unload.

The only thing that could make using COM in Clarion even better is a utility that accepts information about the COM object you want to create and produces the needed typelib and some starter files. Want to guess what I'm working on? Okay I'm perhaps easily entertained, but I think the COM possibilities for Clarion programmers are endless. That's a good feeling.

Download the source

Some additional resources (courtesy of Carl Barnes)

- Dr. GUI articles on COM
- Download OLEView
- Inside COM - a good starter COM book
- Jumpstart for creating a SAX2 application with C++
- A Visual Basic SAX example
- MSDN article on XML
- Write an XML composer in VB

Jim Kane was not born any where near a log cabin. In fact he was born in New York City. After attending college at New York University, he went on to dental school at Harvard University. Troubled by vast numbers of unpaid bills, he accepted a U.S. Air Force Scholarship for dental school, and after graduating served in the US Air Force. He is now retired from the Air Force and writing software for ProDoc Inc., developer of legal document automation systems. In his spare time, he runs a computer consulting service, Productive Software Solutions. He is married to the former Jane Callahan of Cando, North Dakota. Jim and Jane have two children, Thomas and Amy.

Reader Comments

Add a comment

Reborn Free CLARION online published by CoveComm Inc.

Clarion MAGAZINE

Clarion News

RSBackUp 1.2 Released

Robert Stanic has released version 1.1 of RSBackUp, a standalone EXE backup utility for your applications. Features include: compression using addZIP DLL; automatic calculation of required floppies; large capacity/ZIP drive support; and selective restore.

Posted Tuesday, February 27, 2001

MessageEx Updated

MessageEx, a MESSAGE() enhancement tool from solid software, is now in release 1.4. New features include: random message window placement, centering of text, WAV files (32 bit only), and an HTML help file. This update is free for registered users. If you purchased at ClarionShop, please download the update from their site. If you purchased directly from solid software, you can download from there using the old passwords. Demo available.

Posted Tuesday, February 27, 2001

Free Outlook Style Template

Ronald van Raaphorst has created a free template to assist developers in creating Outlook-style applications with a menu on the left, a tree (visible or invisible) in the middle, and a browse or other window on the right. This is an adaptation of the free Locus templates and the WinTree template. This is a work in progress, not production quality code. For C55EE.

Posted Monday, February 26, 2001

HTML Designer Version 1 Release Imminent

HTML Designer Version 1 is still scheduled for release at the end of February 2001. The price of the HTML Designer will then increase from $59.00 to $99.00. HTML Designer allows you to bypass the cwHH class and automatically implements HTML Help on ALL versions of Clarion for Windows for both Legacy and ABC. You can add a template to your application to create a complete bare bones HTML Help system for your application, and you can export your application's help information and build a complete HTML Help system for your application making use of the custom WYSIWYG HTML Editor and the custom HTML Help Project Editor.
Posted Monday, February 26, 2001

VariView FAQ Page

Simon Burrows has created a FAQ page to answer the many questions developers have been asking about the new VariView templates.
Posted Monday, February 26, 2001

Freeware Template Handles Capitalization

New from Sterling Software, CapFlash is a freeware extension template to be used on a Process. The template coverts file data from upper case to proper case with the following options: You can enter into the template a list of words which are always lower case - such as del, la, de etc.; You can define a list of words which are always upper case, such as ABC, USA, AFB etc.; Individual fields (such as State) can be excluded; Names beginning with Mc,Mac or O' are handled correctly. Compatible with CW2002 to C5.5, ABC and Legacy
Posted Monday, February 26, 2001

Solace VariView Demo Available

Solace Software has a demo of the VariView debugging tool, which is simply the Event Manager C55 example with the VariView templates added. The new templates add a toolbox to your application which can be called up by a programmer defined key press. The toolbox shows the current values of all your Global and Local variables together with all fields from your files. As your progress through your program, the toolbox shows the

values of these variables without interfering with the flow of your program. The templates are available for ABC and Legacy apps and for 16 and 32bit.
Posted Monday, February 26, 2001

MenuTree Template Includes TXAs

The Menutree template includes two simple example applications in Clarion 5.5 format. For those who don't have Clarion 5.5, the template also ships with the application TXA files.
Posted Monday, February 26, 2001

Registry Function Library 1.5 Available

Registry Function Library 1.5 is now available for download. This release includes three new functions for enumerating the registry: EnumReg will load the entire contents of a key into a queue structure; EnumValue and EnumKey return the name, type, and content of a value or key by position. Also all remaining 16-bit API calls have been replaced by the 32-bit version.
Posted Tuesday, February 20, 2001

Intuit To Provide QuickBooks Interface

Intuit has announced qbXML, an interface to its popular QuickBooks accounting system. Still under development, qbXML will allow developers direct access to QuickBooks data. An application communicating with QuickBooks will create XML data (in memory or in a file) and pass that data to QuickBooks via a COM object. The XML data type definitions are available for download.
Posted Tuesday, February 13, 2001

Logic Central Adds Remote Reporter

Logic Central has released Remote Reporter, an add on to IFT:HTTP server. Remote Reporter web-enables Clarion reports quickly, and without coding. Your report is converted to a JPEG, PNG, or WMF and sent to the client's browser or a customized HTTP client. Requires Windows 95 or better, Clarion 5 or better, and Internet Framework Tools HTTP Server 2.5 or better.
Posted Monday, February 12, 2001

FrameText Adds Text To Application Frame

New from solid software is FrameText, a small extension that allows you to display text on your application frame's client area, where you normally can't display any information. With FrameText you can specify all text parameters including font name, size and style, character set, color, light and shadow color (for 3D effect), and justification. FrameText works with Clarion 5 and 5.5, legacy and ABC, 16 and 32 bit.
Posted Monday, February 12, 2001

CLARION.COM Sold For $30,000

Sensium Corporation recently put the CLARION.COM domain name up for auction, with a minimum bid of $25,000. A user named cagle22 purchased the domain for $30,000 on February 4.
Posted Friday, February 09, 2001

Clarion ABC Templates In Spanish

A version of the c55 Gold ABC templates, translated into Spanish, is now available. Back up your current templates, just in case.
Posted Friday, February 09, 2001

GREGPlus Update Available

A free update is now available for GREGPlus for all v4.5 registered users.
Posted Tuesday, February 06, 2001

Reborn Free CLARION online

published by CoveComm Inc.

Clarion MAGAZINE

The Clarion Advisor: Locating Records With PROP:SQLFilter

by Dave Harms

Published 2001-02-13

As much as I like the TopSpeed file format, the more I use SQL, the more I wonder why I didn't switch sooner. SQL is an amazingly powerful way to deal with data, and one of my favorite SQL features is the ease with which I can locate data in a table using `PROP:SQLFilter`.

Typically, I add an entry field to a browse window, and on that field's `Accepted` event I place code something like the following (assuming the View is called `BRW4::View:Browse` and the entry field is called `ArticleFilter`):

```
BRW4::View:Browse{prop:sqlFilter} = ArticleFilter
ArticlesBrowse.ResetFromBuffer()
```

I can then type any valid filter statement into `ArticleFilter` and restrict the number of records the browse retrieves from the database. Some of the statements I commonly use on the Clarion Magazine Articles table are as follows:

```
ArticleType='CMAG' and Status='V'
```

```
ExtraInfo <> ''
```

```
ArticleType='NEWS' and
month(PublicationDate) = month(now())
and year(PublicationDate) = year(now())
```

Since I'm executing SQL statements directly on the server, I can use all of the available functions, such as the month() and year() functions. My all-time favorite filter function is `like`:

```
lower(URI) like '%oop%'
```

The % character is a wild card, and substitutes for any string of characters so this filter will retrieve all articles with the phrase 'oop' in the file name.

As handy as PROP:SQLFilter is, it doesn't sit well with ABC. If you enter an incorrect filter string (say you forget a quote character), ABC reports the error, and then quite unpleasantly terminates the browse procedure. At first I tried to detect the bad filter by executing a NEXT() on the view after applying the filter, Clarion never reported the error. Then Larry Teames suggested I use PROP:SQL to test the filter, and that did the trick. My code now looks like this:

```
Articles{prop:sql} = |
 'select ArticleID from Articles where ' |
 & clip(ArticleFilter) & ' limit 1'
next(Articles)
if errorcode()
   beep
   select(?ArticleFilter:2)
else
   BRW4::View:Browse{prop:sqlFilter} = ArticleFilter
   ArticlesBrowse.ResetFromBuffer()
end
```

The PROP:SQL statement attempts to retrieve just one record from the table using the filter I've typed. If there's an error, or there are no records that match the filter, the code beeps and returns the focus to the entry field.

I find PROP:SQLFilter so useful that I almost never use locators anymore. I store commonly-used filters in a separate table, and instead of an entry field I use a file loaded drop combo loaded from that table. That way I can still type free-form filters when I want to.

David Harms is an independent software developer and the co-author with Ross Santos of Developing Clarion for Windows Applications, published by SAMS (1995). He is also the editor and publisher of Clarion Magazine.

Reader Comments

Add a comment

Reborn Free
CLARION online
published by CoveComm Inc.
Clarion MAGAZINE

COM: Getting Easier By The Minute (Part 2 of 3)

by Jim Kane

Published 2001-02-13

Part 2 of 3

In Part 1 of this series I explained how Windows stores COM information in the registry, and I showed how to declare a Clarion interface that corresponds to the IUnknown interface that every COM object contains. In this article I'll show how to build on IUnknown to call COM methods.

Adding the getFeature method

The IUnknown interface is the base interface for ISaxXMLReader. Interfaces are just like classes – you can derive one interface from another. To create an interface you can use to actually call the COM object, derive the interface from IUnknown, and add the additional methods (in this case the methods in ISaxXMLReader). Here is the information from the typelib about the getFeature method.

```
HRESULT _stdcall getFeature(
  [in] unsigned SHORT* pwchName,
  [out, retval] VARIANT_BOOL* pvfValue);
```

Again you see the familiar HRESULT (which is a LONG); the function looks like this in Clarion:

```
Hresult        equate(LONG)

ISaxXMLReader Interface(IUnknown),COM
GetFeature       procedure(pwchName, pvfValue),hresult
               End
```

You've already declared the IUnknown interface so you don't need to redeclare those methods, but since those three methods already exist, getFeature is the fourth method in the interface. You might recall from my earlier articles that you had to count the position of a method in the interface to know how to call it. I love inheriting interfaces and saving all that typing.

Now you need to figure out the types for each of the two parameters. One of the very best things about a typelib is each parameter is marked with [in], [out] or [in,out]. Just like in Clarion, if something is going to come out of a COM method, that data has to be passed by address. In that case the C++ data type has a * after it. All [IN] parameters are passed by value and have no *.

Strings and groups

It looks like I just talked myself into a corner. I said [in] parameters are passed by value, yet the first parameter is an [in] and passed by address. The reason for that is STRINGs and GROUPs are a special cases. These are too big to put on the stack and pass by value, so instead just the address of the STRING or GROUP is passed. That would explain it, except the type says USHORT and not STRING! Well, a bunch of C and VB programmers got together one day and evidently did a lot of drinking. Either that or they're under a Microsoft spell that causes them to do silly things. The result was a decision to refer to wide (a.k.a. UniCode) strings as unsigned shorts since each character in the STRING is 2 bytes or a SHORT. It's bizarre to me that unsigned SHORT * is used to represent the address of a wide string. There really is no way to tell from the prototype alone if the author meant a single USHORT passed by address (*USHORT) or an array of ushorts representing a wide string.

Clarion uses STRING and CSTRING which are both single byte strings, but fortunately converting a Clarion STRING or CSTRING to a wide string is not very difficult. My original COM article covered it (yet another shameless plug). I also have included a new and improved string class with the downloadable code for this article. This string class has functions to convert wide strings to Clarion strings and vice versa. With that code in hand conversion becomes an easy process, so I won't dwell on wide strings. There's too much other stuff to fill your heads with, and you wont get any cool points for converting strings anyway.

Since the unsigned SHORT* is the address of a wide string, for the purpose of prototyping I declare the wide string as LONG (all addresses are can be prototyped as longs). The second parameter is Variant_Bool*. Variant_Bool is not a variant but it is Boolean. It is a SHORT and how VB programmers express true and false. To those folks (remember they evidently drink a lot and don't make good decisions) true is -1 or 0FFFFH, and false is 0. That conversion can mess up Clarion and C programmers since true is 1 to us. Since VARIANT_BOOL is an [Out] (parameter, not belly button) you pass it by address, so in the Clarion prototype you add a * to the SHORT. With that information the interface becomes:

```
Hresult        equate(LONG)
```

```
ISaxXMLReader  Interface(IUnknown),COM
GetFeature        procedure(LONG pwchName, ↵
                    *SHORT pvfValue),hresult
               End
```

Now that's pretty amazing. Two long paragraphs and all you've added to your interface is two words: LONG and *SHORT! Well, at least it's progress.

I know you'd love me to do a blow-by-blow description of every parameter in the entire interface but instead I put the completed prototypes for the ISaxXMLReader interface in the IsaxCl.Inc file in the download. The pattern is the same. When you pass an item such as a STRING, GROUP, or interface as a parameter, you pass it by address because the item is too big to fit on the stack. In each case you'll see a * in the typelib prototype but in Clarion you use a LONG since you're just passing a pointer, or address. If you're passing the address [out] then make it a *LONG, since to get the information out of the method you'll need to give the address's, ah, address.

I think you'll find it worth your time to compare the typelib prototypes with the prototypes in IsaxCl.inc in the downloadable code and work your way through the conversions. These are not the only possible ways to prototype these functions but workable. Other possibilities you might think would work, don't. If you tried to prototype the unsigned SHORT* used for a wide string literally as *USHORT, Clarion would insist you use a variable with a type of USHORT for the actual parameter, and that would not work.

Now I said calling an interface was easy. Here you are well into Part 2 and I've not done it yet. Guess I better, but heck, at least I haven't written any assembler code yet. I think that shows excellent restrain on my part.

Creating the COM object

Before I create the COM object (another delay!) I'm going to give you an overview of how my parser program works. If you look in saxxml.clw you'll see these two lines of code:

```
ISaxCl.Init('file://D:\Clarion5\apps\sax\order1.XML',1)
ISAXcL.KILL()
```

Those two lines of code do all the parsing. ISaxCl is an instance of the ISaxClType class, which you can find in ISaxClType.inc and ISaxClType.clw. This class is a wrapper around the Microsoft SAX parser. The ISaxCl.init method (mainly by means of utility classes) initializes the COM object, obtains the necessary interfaces, and calls interface methods to parse the XML file.

To create the COM object you want (that is, SaxReaderClass) and get the pointer to the interface you need (that is, ISaxXMLReader), you call the Windows API CoCreateInstance procedure. In addition, you need to call CoIntialize just to wake up COM on the thread you are using. CoCreateInstance takes a number of parameters but the only three of consequence are the CLSID of the class you want to turn into an object, the IID of the interface you want, and a pointer which CoCreateInstance will set to the ISaxXMLReader object.

Within my code, to save typing, I named the ISaxXMLReader as IsaxReader. Since you always refer to the COM interfaces by IID, the label in the code in unimportant. The fact that Microsoft refers to this COM interface as ISaxXMLReader and I call the interface IsaxReader makes no difference at all – it's the IID that is important. To make life easier, I wrote a little class that contains a ton of constants needed for COM work, and included in that class is a wrapper function for CoCreateInstance, called GetInterface:

```
GetInterface procedure(LONG lpClsid, ↵
LONG lpiid),LONG
```

You can think of CoCreateIntance, at its simplest, as a way to pass in the CLSID and IID and get back the address of the requested interface. Previously I showed CLSIDs as long strings of numbers in curly braces. As if that wasn't ugly enough, when you use UUIDs of any type, such as CLSIDs and IIDs, in a program, you have to represent them in GROUPs.. For example, here is the CLSID from the typelib for the CoClass containing the ISaxXMLReader interface:

```
!079aa557-4a18-424a-8eee-e39f0a8d41b9
CLSID_ISax  Group
data1         ULONG(079aa557H)
data2         USHORT(4a18H)
data3         USHORT(424aH)
data4         string('<8eH><0EEH><0E3H><9FH>↵
                      <0AH><8DH><41H><0B9H>')
            end
```

Parsing the XML file

The IsaxClType.init method (from isaxl.clw) contains the code that will eventually grow to serve your parsing needs:

```
!make the stdcom class
SELF.StdComCl&=New StdComclType
if SELF.StdComCl&=NULL then return return:fatal.
!call coinit to initialize com on
! this thread, call couninit later
SELF.StdComCl.InitCom()
!use the clsid for ISax to get the
! address of an ISaxReader interface.
lpISaxReader=SELF.StdComCl.GetInterface(|
  address(Clsid_ISax), |
  address(IID_ISaxReader))
```

```
if ~lpISaxReader
  !0 Value indicates an error
  return return:fatal
end
```

First `init` creates `StdCom` helper class, and then calls this class's `InitCom` method. `InitCom` calls `CoIntialize` to initialize the COM object. Then `init` calls the `GetInterface` method with the address of the `CLSID` and `IID` for the class and interface you want. Remember how I said groups do not fit on the stack so they are always passed by address? The above is an example - I'm passing the addresses of the `CLSID` and `IID` groups.

The `GetInterface` call returns a pointer to the list of addresses (or v`Table`, but you don't need to worry about `vTables` any more) for the `ISaxReader` Interface. On the next line I feed the address to the interface. This way the interface knows where to find the information it needs:

```
SELF.ISAXReader &= (lpISaxReader)
```

> **NOTE:** You must use the parentheses around `lpISaxREader`. This forces the compiler to evaluate what is inside the parentheses to an address, which is then assigned to the interface on the left side of the statement.

With that done, the interface is ready for use.

The syntax to call a method of an interface is quite simple. Here is some sample code to call the `AddRef` method, which increases the reference count, and then the release method to restore the reference count.

```
SELF.IsaxReader.AddRef()
SELF.IsaxReader.Release()
```

When you Call the methods of an interface directly using the Clarion Interface structure, you are using a technique called early binding. Early binding is fast because the methods to be called are determined at compile time, but it does require you to know the method prototypes.

In addition to serving as a list of methods and providing the calling methodology, interfaces also help with the version control problem common with ordinary DLLs. Once an interface is defined and receives an IID, the interface definition may not change. If you (the COM class developer) need new functions, you must create a new interface; usually this interface has the same name as the original, with a 2 at the end. For example, `IcreateTypeLib` and `IcreateTypeLib2` are the first and second versions of an interface; `IcreateTypeLib2` has a few additional functions not found in `IcreateTypeLib`. Everything still runs fine because older programs can still request and receive `IcreateTypeLib` via

CoCreateInstance or QueryInterface . Newer programs requiring the IcreateTypeLib2 interface can request it, and if it is not available, take appropriate action (such as shutting down gracefully). The invariant nature of interfaces is a tremendous strength of COM.

Although I doubt reading type libraries to write interface definitions will ever become a national pastime, it is a necessary step before either calling or creating a COM objects. Fortunately, OleView shows the needed information from a type library. In the third article I'll apply what I have learned and build both the interfaces I need to call and the interfaces I need to create. With that done, I'll add a modest amount of Clarion code to the interfaces I will create, and the SAX Parser will begin to function. Along the way I'll compare the IUnknown derived interfaces discussed so far with the Idispatch interfaces commonly used with the Clarion OLE control.

Read Part 3

Download the source

Some additional resources (courtesy of Carl Barnes)

- Dr. GUI articles on COM
- Download OLEView
- Inside COM - a good starter COM book
- Jumpstart for creating a SAX2 application with C++
- A Visual Basic SAX example
- MSDN article on XML
- Write an XML composer in VB

Jim Kane was not born any where near a log cabin. In fact he was born in New York City. After attending college at New York University, he went on to dental school at Harvard University. Troubled by vast numbers of unpaid bills, he accepted a U.S. Air Force Scholarship for dental school, and after graduating served in the US Air Force. He is now retired from the Air Force and writing software for ProDoc Inc., developer of legal document automation systems. In his spare time, he runs a computer consulting service, Productive Software Solutions. He is married to the former Jane Callahan of Cando, North Dakota. Jim and Jane have two children, Thomas and Amy.

Reader Comments

Add a comment

Reborn Free
CLARION online
published by CoveComm Inc.
Clarion MAGAZINE

COM: Getting Easier By The Minute (Part 1 of 3)

by Jim Kane

Published 2001-02-06

Part 1 of 3

Have you ever received an email from an old customer you haven't heard from in a while? Those emails always makes me pause, because it's rare to get good news out of the blue. Well, a few months ago I got an email from an old customer. With the usual amount of trepidation I opened the email, and was quite relieved to find a request for a bid on new work. Even better, the request was for something I knew how to do: work with COM!

From my experience on the Clarion newsgroups I know many Clarion users are not comfortable with Microsoft's Component Object Model, or COM. Once you get past the terminology, however, basic COM is fairly simple and quite similar to calling any API function. Heck, if VB programmers can do it you ought to be able to! You can get perhaps 90% of the functionality of COM just knowing the basics. You don't need to conquer the more difficult topics like threading and aggregation. All you really need is a love of acronyms and redundant terminology, and once through the terminology you'll just sound cooler – and may get a few things done you wouldn't otherwise!

Getting down to business

My customer's request was to parse a XML file (that the company receives on a regular basis from an e-business partner) and update some TPS files. My customer had attached a sample file that was quite simple, plain XML. Since I already had a class that used the Microsoft Document Object Model (DOM) to parse XML, I thought it would be an easy project. I gave a low bid which was quickly accepted.

That was the last of the good news! After I built the application and sent it off, I got a quick response that it didn't work. After assessing the damage, I realized the problem was the real file to be parsed was a huge quarterly summary of all transactions.

DOM is a notorious memory hog. Even on a server with 512MB of memory the application couldn't parse the file. Fortunately, I always have my nose in something, often a computer book, so I knew the alternative was to use the Microsoft (or other) SAX (Simple Api for

Xml) parser, and in particular the ISAXXMLReader interface which is much better on memory usage than DOM.

I do like challenges, but when I downloaded the latest (3.0) release of the SAX parser (from http://msdn.microsoft.com/xml/default.asp) I realized that to use the SAX parser not only did I need to call a COM interface, I also needed to create one.

As I read through the documentation that came with the SAX parser I realized I needed to first call the ISaxXMLReader interface to pass in the name of the file to parse. The SaxXMLReader object then calls a ISAXXMLContext interface that my program must supply. When I discovered I needed to create a COM interface, I said a bad word (or two), sighed, but realized this was my chance to explore two new Clarion features: Interface and Implements. From my earlier cursory inspection of those new features I thought they were everything I needed to call COM interfaces and create my own. It was clearly time to jump in and see if I could save my bacon and get this project done. Well, I'm happy to report that about one week later I emerged victorious. I was able to call a COM interface more easily than ever before, and I even created my own COM interface and object.

Understanding COM

I think the easiest way to understand COM is to look at how a COM object is registered, called, and eventually released or unloaded. A COM object can be packaged in an EXE, DLL, or a few other file types. When in the form of a DLL, your program loads the DLL and the COM object becomes a part of your program or process. As a result, that type of COM object is called an in-process server. A COM object in a separate EXE is an out-of-process server. The SAX COM object is a in-process server so I'll confine my discussion to that type of object.

Each COM DLL contains one or more classes. An object is just an instance of a class, or, said another way, a class that is in use. For most purposes you can use the terms component, object and class interchangeably.

If more than one version of a DLL is available, sometimes a copy other than the intended one is loaded, causing a program to misbehave and programmers to commit suicide. COM seeks to avoid these problems. One of the requirements for creating a COM class (the original class, not the instance used by a program) is to assign that class a unique CLSID. CLSIDs are those long, slightly insane numbers you've probably seen in the registry with this format: {079aa557-4a18-424a-8eee-e39f0a8d41b9}.

To load a COM object, the program looks up the COM object in the registry. This is a very important step. Take a moment and run regedit.exe – you can usually find this program in the Windows system directory, or you can just run it from the Windows menu. Find the HKEY_CLASSES_ROOT key. Open this key and look down until you find the CLSID subkey. Under the CLSID subkey you will see all the COM objects on the computer. Assuming you have downloaded and installed the SAX parser mentioned above, you will find its CLSID under this key:

```
HKEY_Classes_Root
|_{079aa557-4a18-424a-8eee-e39f0a8d41b9} SAX XML Reader
|_Inprocserver32 %systemRoot%\system32\msxml3.dll
|_ProgID Msxml2.SAXXMLReader
```

There is also a similar key based on the more programmer friendly `ProgID` of `MSXML2.SAXXMLReader`. Beware though `ProgID`s are not globally unique; they are just easier to type and add an extra step in getting to the CLSID:

```
Hkey_Classes_Root
|_MsXML2.SAXXMLReader
|_CLSID {079aa557-4a18-424a-8eee-e39f0a8d41b9}
```

If you tell your program to create the COM object from the ProgID (I'll explain how to do this shortly), the program searches the registry using the `ProgID` key shown just above and retrieves the `CLSID`. Once the program has the `CLSID`, which is a unique value identifying this particular COM class, it looks up the `CLSID` key above and gets the path and name of the DLL. Your program can then load the DLL.

As you can imagine, if the registry entries are not there or the DLL is moved after the registry entries are created, this isn't going to work. If a COM object doesn't work or stops working, you can often solve the problem by tracing through these registry entries. Most installation programs create these required entries. If this hasn't happened, or if you don't have an installation program, simply run `RegSvr32`, and pass the name of the DLL on the command line. It is likely you will find regsvr32 on your system all ready but if not, you can download it from Microsoft. If you are in doubt about the accuracy of the registry entries, just run `RegSvr32` again to re-write the correct registry entries. `RegSvr32` calls the code to write the registry located in the DLL itself.

Loading the COM object

If the registry entries are in place, your program can load the COM object. Now it's time to look at how to call the loaded DLL. To link to a conventional DLL you use a corresponding LIB (which gets linked into your program). If you don't have a lib you can create one with `LibMaker`. Then you write some prototypes and call the functions. Guess what? COM is the same. Only in the case of a COM object, the tool of choice is a free Microsoft program called OLEView.

If you plan to use COM, you must have OLEView. Without OLEView you're working in the dark. OLEView exposes all kinds of information about a COM object, and who wouldn't want to expose a sexy object? You can download OLEView and `RegSvr32` by going to the Microsoft download home page http://msdn.microsoft.com/downloads/default.asp and searching for each program.

While LibMaker only shows the name of the functions, OLEView shows a lot (almost an excessive amount) of information. OLEView reads something called a type library or typelib for short. The typelib, like the registration code, is usually inside the COM object. The typelib can also be in a separate file usually with the extension of OLB or TLB. Whenever I get my hands on a COM object, I register it, and then try to open it with OLEView. While usually this works, not all COM objects contain or come with a typelib.

After I open a COM object with OLEView I first look for `CoClass` sections. Remember when I said the only requirement for working with COM is a love of acronyms and redundant terminology? Well, here we go. The `CoClass` section simply shows the `CLSID` for the class and a list of the interfaces in the class. For example, for SAX, I found from the registry entries that the SAX parser's DLL is MSXML3.DLL, located in the Windows `system32` directory. I opened MSXML3.DLL with OLEView and searched for a CoClass that contained the `ISAXXMLReader` interface I knew I wanted to call. This is what I found:

```
[
  UUID(079AA557-4A18-424A-8EEE-E39F0A8D41B9),
  helpstring("SAX XML Reader (version independent) coclass")
]
coclass SAXXMLReader {
    [default] interface IVBSAXXMLReader;
    interface ISAXXMLReader;
    interface IMXReaderControl;
};
```

The `UUID`, or Universally Unique Identifier, is just another term for `GUID`, or Globally Unique Identifier. `CoClass` is another term for CLASS. `UUID` and `GUID` are exactly the same and a `CLSID` is a `GUID` that identifies a CLASS or `CoClass`. Redundant enough for you? Next I searched for `ISAXXMLReader` and found both the interface and finally the `ParseURL` method I knew I needed to eventually call. This is what it looks like:

```
[
  odl,
  UUID(A4F96ED0-F829-476E-81C0-CDC7BD2A0802),
  helpstring("ISAXXMLReader interface"),
  hidden
]
interface ISAXXMLReader : IUnknown {
    HRESULT _stdcall getFeature(
                    [in] unsigned short* pwchName,
                    [out, retval] VARIANT_BOOL* pvfValue);
  ...Many more methods deleted for now...
};
```

What I saw was another UUID or GUID this time for an interface. From previous exploits in COM land, I knew a GUID applied to an interface is an called `IID` or, as you might guess from the context, interface id. There may be a million of these abbreviations but at least they are fairly short and easy to figure out.

Understanding interfaces

So just what is an interface? In many ways, it's the most important thing in COM. In Clarion, to use a class you need the class definition in the INC file. To call a regular DLL you use a prototype in a map. To call a COM Class or object you use an interface. An interface lists all the methods in a specific order with their prototypes. All methods in a COM interface use the pascal calling convention, like most Windows functions. The difference is that the address for each method in an interface is supplied in a table called a vtable. If you are interested in the assembler level mechanics of a vtable, please read my first three articles on COM. While it deeply saddens me to say this, it is no longer necessary to understand vtable mechanics; Clarion can now handle calling a COM interface without the programmer writing the low level details. All you need to do is convert the typelib information above into a Clarion interface definition. Clarion will then take care of the rest.

Defining an interface

By way of example, let's look SAXXMLREADER Class's getFeature method, using the ISAXXMLReader Interface. Keep in mind the Class or object is the code and the interface is just the calling convention or instructions. Here again is the interface definition from OLEView.

```
[
  odl,
  UUID(A4F96ED0-F829-476E-81C0-CDC7BD2A0802),
  helpstring("ISAXXMLReader interface"),
  hidden
]
interface ISAXXMLReader : IUnknown {
    HRESULT _stdcall getFeature(
                    [in] unsigned short* pwchName,
                    [out, retval] VARIANT_BOOL* pvfValue);
  ...Many more methods deleted for now...
};
```

Notice the first line of the interface definition contains the text ISAXXMLReader : IUNKNOWN. Iunknown is not a description of my knowledge of COM; it does mean that the ISaxXMLReader interface is based on or starts with the three methods of the IUNKNOWN interface. IUNKNOWN is the interface every single COM interface is based on. In Clarion you prototype IUNKNOWN like this:

```
Iunknown       Interface,COM
QueryInterface   PROCEDURE (long  iid_Requested, ↵
                   *LONG lpInterface),HRESULT
AddRef           PROCEDURE (),Long,PROC
Release          PROCEDURE (),Long,PROC
               END
```

You call the QueryInterface method to get the address of any other interface the class supports. Rather than asking for the interface by name, you pass an IIDand the method returns a pointer to the requested interface. QueryInterface, or QI for short, returns an HRESULT, which is just a long. If QI returns a value less than zero you have an error. The AddRef method takes no input and returns the number of users the COM object you're querying currently has. This counter is just like a file opening counter in Clarion. Every time some piece of code tries to open a file in a Clarion application, the count on the file is increased by one. For every close request it's decreased. When the count gets to 0, then the file is really closed.

COM objects use the exact same type of reference counting. When you create a COM object (by calling the COM API functions) that object's internal reference count starts life at 1. If another user comes along and calls `AddRef` the internal count goes to 2. When either the original or new user calls Release the count goes to 1. When the last user calls Release, the count goes to 0 and the object self-destructs. If Release is not called enough, the COM object stays in memory. Reference counting is pretty important because if you don't call Release for a COM object when you're done with it, you'll have a memory leak, and if you call Release before you're done, the COM object is destroyed before you want it to be destroyed. If you then try to call the COM object, the chances of GPF are great. Take your shoes and socks off and count it out if you need to; as long as the count is correct, the COM object will do just what you want! In fact, if your boss catches you with your feet up on your desk, you have my permission to quote this article and explain you were just reference counting.

That's all for Part 1 of this series. In Part 2 I'll explain how to add the methods you really want to call to this interface.

Download the source

Some additional resources (courtesy of Carl Barnes)

- Dr. GUI articles on COM
- Download OLEView
- Inside COM - a good starter COM book
- Jumpstart for creating a SAX2 application with C++
- A Visual Basic SAX example
- MSDN article on XML
- Write an XML composer in VB

Jim Kane was not born any where near a log cabin. In fact he was born in New York City. After attending college at New York University, he went on to dental school at Harvard University. Troubled by vast numbers of unpaid bills, he accepted a U.S. Air Force Scholarship for dental school, and after graduating served in the US Air Force. He is now retired from the Air Force and writing software for ProDoc Inc., developer of legal document automation systems. In his spare time, he runs a computer consulting service, Productive Software Solutions. He is married to the former Jane Callahan of Cando, North Dakota. Jim and Jane have two children, Thomas and Amy.

Reader Comments

Add a comment

Reborn Free CLARION online published by CoveComm Inc.

Clarion MAGAZINE

The Five Minute Developer: Understanding Interfaces

by Dave Harms

Published 2001-02-06

If you've dabbled in Clarion object-oriented programming you've probably come across the term "interface." In OOP interfaces have a particular and well-defined meaning, which I'll paraphrase as follows:

An interface is a description of the methods a class must implement.

I know what you're thinking. Big deal, right? As it turns out, interfaces are a very big deal.

Just as a procedural application is made up of procedures, any of which can call each other, object-oriented applications consist of numerous objects, many of which can call methods in other objects. The more classes you design, and the more objects you create (an object being an instance of a class), the more important this interoperability becomes.

Getting classes to talk to each other can be a problem. Consider the ABC `WindowManager` class. This class manages all typical window-related functions for all sorts of procedures, including browses. When you create an ABC browse procedure, the templates create an instance of `WindowManager`, and as many instances of `BrowseClass` as you have browses. Let's say you have a file-loaded drop combo on the window as well, and the browses are using the (ugh) toolbar control.

`WindowManager` has to know about all of these classes. Each time the user does something (clicks the mouse, presses a key), `WindowManager` traps the event, and then passes that event along to the appropriate object.

In the pre-interface ABC implementation, `WindowManager` had a different type of reference for each kind of object it needed to work with. For instance, to register a `BrowseClass` object with the `WindowManager`, the template-generated code called a method with `BrowseClass` as one of its parameter types. This worked fine, but it was highly restrictive.

Let's say you create a funky new kind of grid class and you want to register an object of this type with WindowManager. Since the pre-interface WindowManager only knows about certain object types, your only option would be to derive your class from a type WindowManager already knows how to register. It might suit your needs to derive from one of these other classes, but chances are that will add a lot of unnecessary baggage to your class.

Interfaces to the rescue! In Clarion 5.5 WindowManager uses an interface that's defined like this:

```
WindowComponent INTERFACE
Kill              PROCEDURE
Reset             PROCEDURE(BYTE Force)
ResetRequired     PROCEDURE,BYTE
SetAlerts         PROCEDURE
TakeEvent         PROCEDURE,BYTE
Update            PROCEDURE
UpdateWindow      PROCEDURE
                END
```

The WindowManager.AddItem method takes a WindowComponent object as a parameter:

```
AddItem PROCEDURE(WindowComponent WC)
```

and stores component objects in a queue:

```
ComponentList QUEUE,TYPE
WC              &WindowComponent
              END
```

Note that the ComponentList queue is actually declared in abwindow.clw, not abwindow.inc – an interesting bit of forward referencing (but not particularly important to this discussion).

You can think of WindowComponent references as object references, except that WindowComponent doesn't actually have any code. It's simply a definition of methods that must exist in a class. When the compiler sees you passing an object as if it were a WindowComponent, or assigning an object to a WindowComponent reference, it examines the object to see if it implements the WindowComponent interface:

```
BrowseClass CLASS(ViewManager),IMPLEMENTS(WindowComponent)
... lots of stuff omitted
END
```

The compiler won't let you create a class that implements an interface without that class containing all of the methods defined in the interface. This makes perfect sense, when you think about it. The WindowManager, for instance, needs to know that it can call a TakeEvent method on any object in its component list. If that method didn't exist, you'd get a GPF.

And now that all `WindowManager` expects of components is a defined set of methods, you can adapt any class you wish to function as a `WindowManager` component. Just implement the interface and its methods, and pass the object to `WindowManager.AddItem`.

An interface is just a description of the methods a class has to implement. It's like a window on the class that allows the calling code to see only that part of the class defined in the interface. Of course, you can implement the interface code any way you choose. All the calling code should care about is that your class implements the methods defined in the interface.

For an in-depth treatment of interfaces, see the David Bayliss article in the January 2000 issue.

David Harms is an independent software developer and the co-author with Ross Santos of Developing Clarion for Windows Applications, published by SAMS (1995). He is also the editor and publisher of Clarion Magazine.

Reader Comments

Add a comment