Clarion MAGAZINE

Search
Go

Home

COL Archives

Information
Log In
Membership/ Subscriptions
FAQ
Privacy Policy
Contact Us

Downloads
PDFs
Freebies
Open Source

Site Index

Call for Articles

Reader Comments

Subscribe/Renew; Back Issues On Sale

You can save 25% on back issue purchases during our Summer Back Issue Sale. Just follow the link to the order form (login required) and if you're missing any back issues, the order form will calculate the cost and apply a 25% discount. If you don't see any back issue options on the order form, that means you're all caught up. You can also use this form to extend your ClarionMag subscription.

Posted Friday, August 03, 2001

Understanding Recursion - Part 1

What is recursion? Ask that question of any group of developers and chances are pretty good you will receive several different answers. Some of the answers will be correct and others will be...interesting.

Posted Friday, August 03, 2001

Using The Web Browser OCX

Have you ever wanted to display an HTML page from within your application without all the hard work and heartache of interfacing to Internet explorer? How about viewing and editing a document or spreadsheet without loading up Word or Excel? Well now you can! Ever since Internet Explorer 4 Microsoft has been supplying the Web Browser OCX, a wonderful little control which will do all this for you, and what's more it is incredibly easy to use!

Posted Thursday, August 02, 2001

A FileManager For Marked Deleted Records

Dennis Evans recently created a template and two classes to manage records marked as deleted (rather than physically deleted), as described in a

xAppWallpaperManager Version 1.2

xQuickFilter v2.0.2 Released

FQL Clarion Scripting Language Near Release

Clarion Third Party Profile Exchange Update

SealSoft Releases xTipHotKey Class 2.0

recent ClarionMag article. This page contains usage notes on the class, and a link to the source.

Posted Wednesday, August 01, 2001

Weekly PDF for July 22-28, 2001

All Clarion Magazine articles for July 22-28, 2001 in PDF format.

Posted Monday, July 30, 2001

I Didn't Need That Much Detail

Some clients like big reports. Very big reports. Reports that make mortal print drivers quiver and the Clarion IDE barf up cookies. Andrew Guidroz brings out his Binford 6500 IDE Grappler and shows how to fit big reports into that small space in the IDE.

Posted Thursday, July 26, 2001

When Clarion COM Will *Not* Do

Jim Kane reviews some COM fundamentals, and then shows how you can take matters into your own hands and extend Clarion's COM abilities for those times where Clarion's native COM will not do what you want.

Posted Tuesday, July 24, 2001

Weekly PDF for July 15-21, 2001

All Clarion Magazine articles for the week of July 15-21, 2001, in PDF format.

Posted Monday, July 23, 2001

Recovering Deleted Records

Fans of the old Clarion (DAT) file format know that unless you use the RECLAIM attribute, deleted records still exist in the data file. That isn't the case for most other drivers. Here's how to make records recoverable, using two new methods in Clarion 5.5.

Posted Thursday, July 19, 2001

Avoid My SQL Mistakes!

For newcomers to SQL, the pitfalls are many. Mauricio Nicastro has been there and done that. In this article he describes the setbacks he encountered converting his applications to SQL, and the solutions to those problems.

Posted Wednesday, July 18, 2001

Using Procedure Category to Split Apps into DLLs

In Clarion 4 a new field named Category was added to the Procedure Properties window. A new tab with Procedures sorted by Category was also added to the Application Tree window. As Carl Barnes explains, these new features can be a great help when it comes time to split an application up into DLLs.

Posted Monday, July 16, 2001

Weekly PDF for July 1-7, 2001

All Clarion Magazine articles for July 1-7, 2001 in PDF format.

Posted Friday, July 06, 2001

"Sometimes" Lookups

Clarion's ability to validate data with lookups is great, but what happens when you want to only do the lookup sometimes? Steve Parker tells all.

Posted Friday, July 06, 2001

Implementing Read-Only Checkboxes

Many Clarion developers have discovered that the READONLY attribute is not available for checkbox controls. Although you can use the DISABLE attribute, it is often desirable to use the READONLY attribute for the sake of consistency in the user interface. Jeff Slarve shows how it's done.

Posted Thursday, July 05, 2001

Using MATCH In Filters and Regular Expressions

Filters for reports and browses seem to get increasingly complicated over time. In this article Carl Barnes show a trick you can do with MATCH() that will let you create more powerful filters, and which do not require any more code than a simple INSTRING() but can search for multiple substrings.

Posted Tuesday, July 03, 2001

A Column by Any Other Name Is Not A Data Element

This Whitemarsh paper describes an approach to achieve enterprise-wide data standardization through the specification, implementation, and maintenance of data elements within the context of a metadata-repository, CASE-like environment.

Posted Monday, July 02, 2001

Clarion MAGAZINE

Search
Go

Home

COL Archives

Information
Log In
Membership/ Subscriptions
FAQ
Privacy Policy
Contact Us

Downloads
PDFs
Freebies
Open Source

Site Index

Call for Articles

Reader Comments

Understanding Recursion - Part 1

by Dennis Evans

Published 2001-08-03

What is recursion? Ask that question of any group of developers and chances are pretty good you will receive several different answers. Some of the answers will be correct and others will be...interesting.

Why all the different answers and confusion about the subject? I really don't know, but I suspect it has to do with the fact that the definitions found in many programming books are confusing and vary from one to the next. In addition, there are different types of recursion; head, tail and mutual are the common types. Introductory level textbooks seldom discuss the different types, and again the definitions in the advanced level books often vary. Recursion is not a complex subject and it does not require a great deal of study, but you probably won't learn it gradually. Instead you're more likely to struggle with the idea of recursion, then all at once understanding will happen and you will wonder what all the fuss was about.

One of the best definitions of recursion I have ever read is this: "Recursion is nothing more than a different way of looking at repetition."[1] Repetitive problems are commonplace in programming. Attempting to solve a repetitive problem within the constraints of a computer program can be difficult since there are only two ways to solve such a problem: one is recursion, and the other is iteration.

Recursion and iteration

Both recursion and iteration simply perform an action or process on an object a finite number of times. The difference is how the repetition is controlled and what is processed. Procedures using iteration are controlled with a looping statement and will perform the same task on the same object. Recursive procedures are controlled by a conditional statement and perform the same task on a smaller version of the object. Of the two, iteration is used more often because it is generally more efficient, and the solution is often intuitive. What I'll attempt to accomplish in this article is to explain the difference between iteration and recursion,

specifically tail recursion. In addition, I will examine what happens when a program uses recursion, and I'll answer the most important question of when to use recursion instead of iteration.

Tail recursion

I'll be using tail recursion because it is a commonly used type, not because it is simpler or faster or betterthan other types of recursion. Tail recursion does however have one advantage over some of the other types. : any procedure written using tail recursion may also be written using iteration. I will begin with a brief explanation of the stack, local scope and procedure calls. Next, will be an example of iteration from Clarion that is often confused with recursion. Then I will solve a simple programming problem using both recursion and iteration, compare the code from the two solutions, and discuss the details of recursion and what takes place inside the computer during recursion. There is nothing complex in the remainder of this article and you do not need to be a computer science type to understand the text. I will assume you are comfortable with procedures and that you are familiar with parameters and local variables.

The stack

The following description of a stack, scope and procedure calls will be a general account of what these objects are and how they are used. For a detailed discussion see the article 'Understanding Stack and Heap Memory in 32 bit Clarion Applications,' by John Gorter.

A stack is a location in memory used to provide temporary storage. The stack has a fixed stating point and will grow upwards towards the heap using available memory. Review Mr. Gorter's article if you are not sure what the heap is or how it is used. When temporary storage is required the stack will increase in size by the number of bytes required to store the object. A long variable would increase the stack by four bytes; a STRING(10) would cause an increase of ten bytes and so on. Placing a variable on the stack is traditionally called pushing, and removing a variable is call popping.

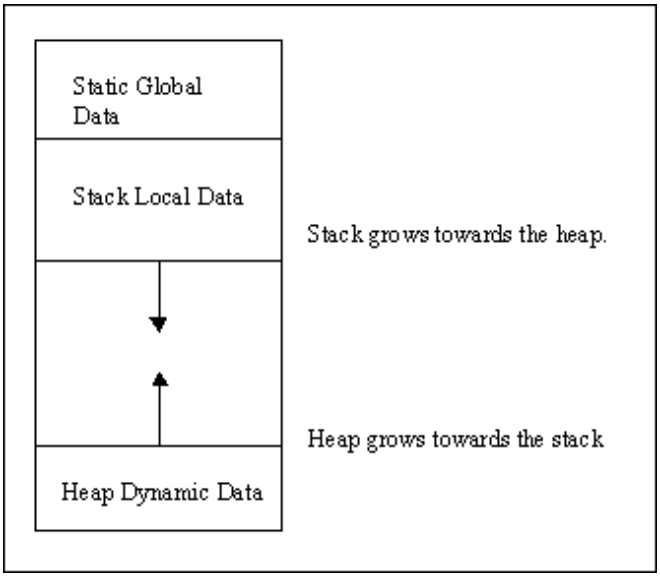

Figure 1. Example memory layout

Objects are pushed and popped on the stack in Last In First Out (LIFO) sequence. In other words, if a program pushes variables A, B and C onto a stack, they will be popped off the stack in the order C, B and A. The location of the variables is tracked internally by the software with a stack pointer. Every procedure call made uses the stack to store some specific information, including the parameters, if any, and data local to the procedure.

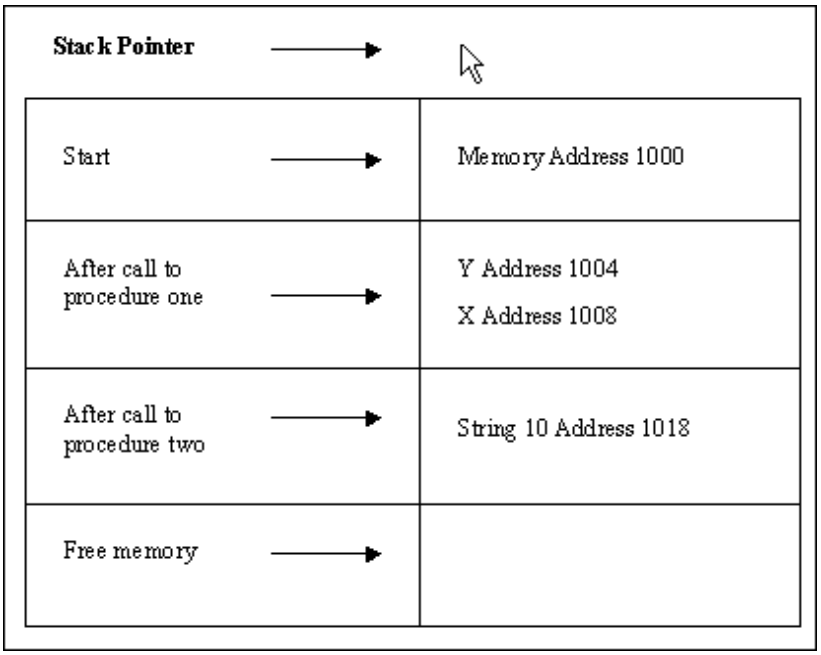

Figure 2. Local scope

Assume for the moment that a program somewhere is running and the stack pointer is currently refers to a memory location with an address of 1000. The program calls a procedure with two parameters, both LONGs. The prototype would be something like:

```
ProcOne procedure(long X, long Y)
```

Inside ProcOne is another procedure call to ProcTwo. ProcTwo has one string parameter of 10 bytes in length. The stack pointer starts at memory address 1000. When the call to procedure one is made the program increases the size of the stack by eight bytes, four bytes for each of the LONG parameters. The call to ProcTwo increases the size of the stack by ten bytes. The result is the scope of the variables X and Y is local to ProcOne. ProcTwo can not access the variables X and Y because it does not know where they are located in memory, or that they even exist. There are other ways that compilers enforce the scope of variables and data, but for the purpose of recursion this is the method of interest.

In addition, the stack is used to store some other information during a procedure call. Exactly what is stored and the sequence the items are stored in is not critical at the programming level. You only need to know that the information is stored and the action does have a significant impact on performance when using recursive procedures.

Procedure calls

Fortunately the compiler, the operating system, and the hardware handle the details of making a procedure call. However in order to understand recursion, you need to understand some of the actions that take place during a procedure call. Again, this is going to be a thumbnail view. Assume a program has a procedure defined with a prototype of MyProc(long X, string S). When a program encounters a call to MyProc execution stops and a very specific set of actions are performed. A piece of memory called a Stack Frame, sometimes referred to as an Activation Record, is created.[2] The term Stack Frame refers to the block of memory used by a single procedure, and allocated on the stack. Part of the stack frame is the caller's responsibility and part belongs to the callee. The caller evaluates the parameters and pushes them onto the stack.

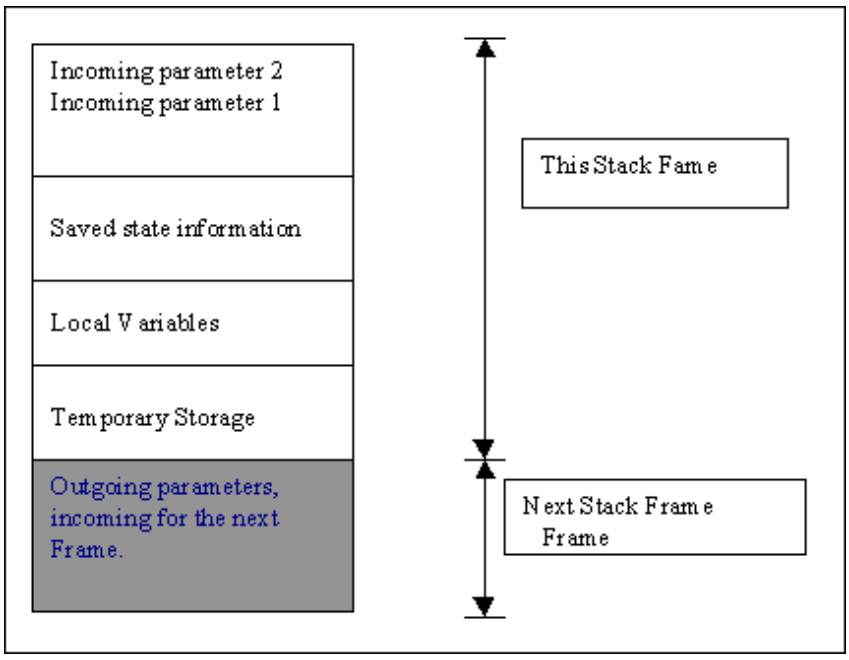

Figure 3. A stack frame

After the caller pushes the parameters onto the stack, information needed to resume execution is placed on the stack. This information includes the current state of the registers and the next program instruction to execute. The callee will then reserve space for local variables created inside the procedure. These are the local named variables you add to the procedure.

Next, the callee will reserve space on the stack for any temporary values created during the execution of the procedure. Temporary values are the result of expressions like `(a + b) * c`. An expression written in that manner will be evaluated in sequence, something like: `Temp_1 = a + b`, `Temp_2 = Temp_1 * c` where `Temp_1` and `Temp_2` are compiler generated variables[2]. Once all those actions are completed the callee's code will execute. If the callee calls another procedure, the entire process is repeated, but this time the callee becomes the caller (the gray shaded area in Figure 3). All of the stack space used for each procedure call is temporary, when the callee completes the memory is returned to the program.

Calling convention

The above example uses the C calling convention or style of passing parameters. Parameters are stored on the stack in the order they are defined, from left to right. The Pascal style is the exact opposite; parameters are stored from right to left. Most compilers, using the C and Pascal style, push the parameters onto the stack. The callee will then remove the parameters from the stack by placing them into a register, or the callee will use the parameter as an address to locate the parameter in memory. Clarion uses a slightly different approach, what is referred to as the JPI calling convention. JPI style places the parameters directly into the registers, bypassing the additional step of pushing the parameters onto the stack. Each of these three styles has some advantages and disadvantages. None of this has anything to do with recursion; I've included it simply as some general information.

Please remember that all of the above is a generic example and the exact sequence will vary slightly from one platform to another and from compiler to compiler. The exact sequence is not all that critical; just know the sequence happens for each and every procedure call, and is very expensive in CPU time and clock cycles. Granted, the CPU deals with nanoseconds, but the calling sequence for a procedure still requires considerable effort on the part of the processor.

You should now have a good grasp of how procedure calls use the stack; this information is essential to understanding how recursion really works. Next week I'll apply this theory to recursive calls in Clarion.

References

1. Nell Dale and Chip Weems, 'Pascal', Second Edition. (One of the better introductory programming texts available.)
2. Ravi Sethi, 'Programming Languages, Concepts and Constructs', Second Edition

---

Dennis E. Evans is retired from the U.S. Army. During his time in the military he spent twelve years in the Armored field and eight years in information management. He currently works as an independent contractor and resides in Marion, Illinois with his wife Beverly and their two children Christopher and Jessica. His hobbies include historical simulations, reading and studying different programming languages.

Reader Comments

Add a comment

As Dave Harms once told me the really difficult part about...
1. By convention followed by Intel processors, the (call)...
Mr. Solovjev, You are absolutely correct, systems...

Clarion MAGAZINE

Search
Go

Home

COL Archives

Information
Log In
Membership/ Subscriptions
FAQ
Privacy Policy
Contact Us

Downloads
PDFs
Freebies
Open Source

Site Index

Call for Articles

Reader Comments

Using The Web Browser OCX

by Matt Grossmith

Published 2001-08-02

Have you ever wanted to display an HTML page from within your application without all the hard work and heartache of interfacing to Internet explorer? How about viewing and editing a document or spreadsheet without loading up Word or Excel? Well now you can! Ever since Internet Explorer 4 Microsoft has been supplying the Web Browser OCX, a wonderful little control which will do all this for you, and what's more it is incredibly easy to use!

The Microsoft web browser OCX allows the developer to display HTML pages, Word documents and Excel spreadsheets on a window. It has a simple set of methods for navigating around and supports all the normal web browsing and document editing stuff.

To use this control in your application, create a new window and place an OLE (a.k.a. OCX) control template on it. Using the right-click properties tab of the OCX control, enter a use variable for the OCX - I used "?Nav". Set the 32-bit check box to true, and the control type to OCX. From the object type drop down box select "Microsoft Web Browser". This will actual display as "Shell.Explorer.2" (See Figure 1)

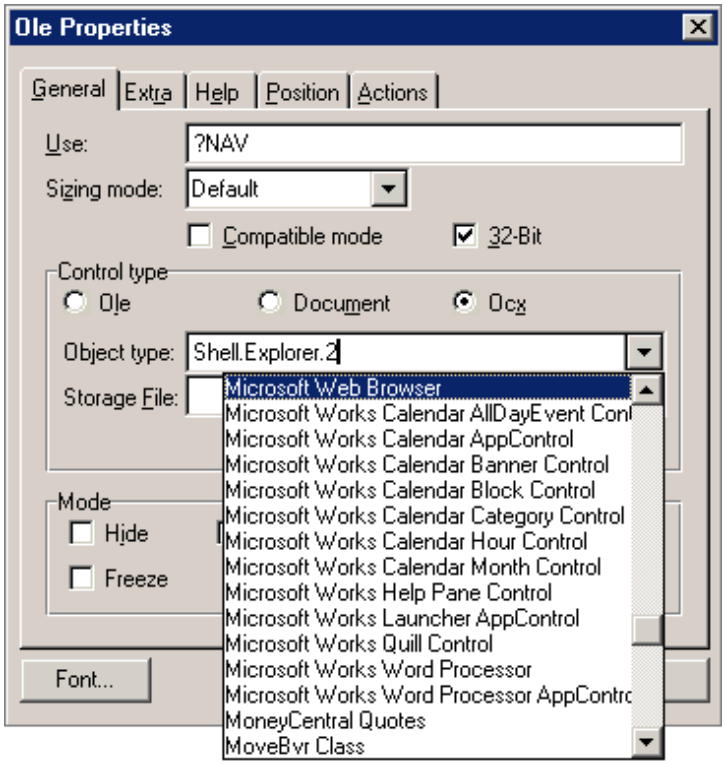

Figure 1. Setting the OCX properties

Next slap on an entry field for a URL. The entry field is a 500 character STRING where the user can enter the document or web address they want to view. I called mine LCL:URL. Place the following code in the Control event handling, after generated code – Accepted embed point:

```
Post(Event:Accepted,?GoButton)
```

In this line of code ?GoButton is the field equate for the "Go" button, which you add next. Make sure the button's field equate matches the equate in the POST statement.

The Go button will instruct the OCX to navigate to the resource described by the LCL:URL variable. Place the following code in the Control event handling, after generated code – Accepted embed point to tell the OCX to load the page specified by the URL. In this code ?Nav is the name given to the OCX and LCL:URL is the variable containing the desired resource.

```
?Nav{'Navigate(URL="' & Clip(LCL:URL) & '",Flags=14)'}
```

Add a file dialog button to allow the user to browse for a local HTML or other file. For the file dialog button place the following code in the Control event handling, after generated code – Accepted embed point.

```
IF FileDialog('Pick a file...', |
LCL:URL,|
'HTML Files|*.html|HTM Files|*.htm|Word documents|*.doc
|Excel spreadsheets|*.xls|All Files|*.*'|
```

```
,10000b)
  display()
end
```

Other buttons you could add are the "Back", "Forwards", "Home", "Stop" and "Refresh" buttons most browsers have. Just place the following code in the "Control event handling, after generated code – Accepted" embed for each button.

Action	Code
Stop	?Nav{'Stop'}
Back	?Nav{'GoBack'}
Forward	?Nav{'GoForward'}
Refresh	?Nav{'Refresh'}

Compile and run the program. Use the file dialog button to find a local HTML file, or type a URL into the entry field, then press the GO button. The page should be displayed. Now try it with a Word document or Excel spreadsheet. I think you will be pleased!

Reader Comments

Add a comment

Well done for documenting this Matt, however the following...
Here are some links with a little information on ...
Source code of the article example. I wrote this example...

Clarion MAGAZINE

Search
Go

Home

COL Archives

Information
Log In
Membership/ Subscriptions
FAQ
Privacy Policy
Contact Us

Downloads
PDFs
Freebies
Open Source

Site Index

Call for Articles

Reader Comments

A FileManager For Marked Deleted Records

by Dennis Evans

Published 2001-08-01

A recent Clarion Magazine article (Recovering Deleted Records) explained how to use new methods in Clarion 5.5 to "delete" records in a table by setting a deleted flag (allowing for recovery of deleted data). I've created a template and two classes to implement this feature. The following are some usage notes on the classes (the source is available below). The zip file contains a readme with usage notes.

The FileManagerDelete Class

`FileManagerDelete` is a class derived from the `FileManager`, and may be used to mark records in a table as deleted. The marked records can then be archived, recovered or permanently deleted. The class contains one data member, `DeleteField` and three methods, `SetDeletedField`, `DeleteRecord` and `Deleted`. Some obvious improvements to the class would be `UnDelete` and `PermanentDelete` methods.

Property	Description
`DeleteField`	The `DeleteField` class member is assigned a reference to the field used to mark deleted records. During program execution the `FileMangerDelete` class uses the `DeleteField` property to compare the current value of the field and assign values to the field.
Method	**Description**

SetDeletedField	This method contains one line of code and will be called once in each browse, process or report procedure that uses the class. The method expects one parameter, Fld, which is a reference to the file field used to mark the records as deleted. The method assigns the reference to the DeleteField property: Self.DeleteField &= Fld After the method is called the DeleteField property points at the same location in memory that the file field does; changes to one are reflected in the other.
DeleteRecord	This method is a virtual method derived from the FileManager class. It accepts one parameter called Query which has a default value of '1' (see the ABC documentation for the specifics on the Query parameter). The DeleteRecord method is called from the RelationManager Delete method, and assigns TRUE to the field used to mark records as deleted and then writes the record to the disk. By default the method returns Level:Benign to the calling method and any errors are handled by the RelationManager class.
Deleted	This is also a virtual method overloaded from the FileManager. The derived method simply compares the value of the DeletedField and returns Level:Fatal for a record that has been marked as deleted, or Level:Benign for a record that has not been marked. The current version uses a BYTE field and the CHOOSE statement: RETURN CHOOSE (Self.DeleteField,| Level:Fatal, Level:Benign) The CHOOSE statement evaluates the DeleteField and returns Level:Fatal if the field is set to true, or level:Benign if the field is set to false.

The BrowseFilterDeleted Class

This class is derived from the ABC BrowseClass and contains one data member and two methods. The data member is used to build the filter criteria so records marked as deleted are not displayed in a browse. The class also uses one equated value, `DeletedPriority equate('9 - zDeleted')`, to set the filter priority.

Property	**Description**
`DeleteFieldStr`	This property is a string variable that will contain the field name used to mark records.
Method	**Description**
`SetFilterField`	This method is used to set the `DeleteFieldStr` property. It will be called once during the procedure, something like: `BrwCustomer.SetFieldStr(| 'Cus:MarkedDeleted').`
`SetDeletedFilter`	This method is used to turn the filtering of marked records on and off. The method accepts one parameter, a `BYTE` with a default value of true. When the method is called with a parameter of true (or without a parameter) the filter criteria is added to the current sort order of the browse. Records marked deleted are filtered from the result set. When called with the parameter set to false, the filter is cleared and the marked records will appear in the browse. SetDeletedFilter uses the equate value `DeletedPriority`, and it is worth noting what exactly is happening. The `ViewManager` class will use filter expressions of different priorities. The filters are added to the view{prop:filter}in descending order. The view engine uses what are called 'short circuit' Boolean evaluations, which means it will stop evaluation expressions when any false condition is found. The filter `priority 9 - zDeleted` will be ordered in the `ViewManager's` filter list before any of the standard filters or QBE expressions from the ABC classes, and after any range limits. Assume for the moment that the browse

	is filtering deleted records and has a QBE filter active. The deleted record filter will be evaluated first, then the QBE criteria. If the view engine determines the filter expression for the marked (deleted) records is true, the QBE filter will not be evaluated and the record is discarded. Arranging filters into logical priorities can have a significant impact on performance.

Performance issues

Everyone is aware that filtering records can be slow, as each record must be read and compared then retained or discarded. Using the `BrowseFilterDeleted` class to filter marked records should not have any noticeable performance issues. The browse will actually be slower, however, since an additional byte must be read from the disk and an additional comparison is performed. Each of these actions takes a certain amount of time.

If used correctly the filtering marked records should not be noticeable to the user. If the table has 2 to 4 percent of the records marked as deleted and the deleted records are distributed throughout the table performance lose will be minimized. On the other hand, if the number of marked records is in the 15 to 20 percent range, performance may degrade noticeably. Actually if the table contains that large of a percentage of marked records you have a lot more serious problems than a slow browse. Records marked as deleted should be removed from the table in a reasonable time frame. They could be permanently deleted, archived or whatever. Just avoid large percentages of deleted records.

Download the source

Dennis E. Evans is retired from the U.S. Army. During his time in the military he spent twelve years in the Armored field and eight years in information management. He currently works as an independent contractor and resides in Marion, Illinois with his wife Beverly and their two children Christopher and Jessica. His hobbies include historical simulations, reading and studying different programming languages.

Reader Comments

Add a comment

Clarion MAGAZINE

Search
Go

Home

COL Archives

Information
Log In
Membership/ Subscriptions
FAQ
Privacy Policy
Contact Us

Downloads
PDFs
Freebies
Open Source

Site Index

Call for Articles

Reader Comments

I Didn't Need That Much Detail

by Andrew Guidroz II

Published 2001-07-26

> Our life is frittered away by detail ... Simplify, simplify.
> ..
>
> Henry David Thoreau

Some of my clients like big reports. Very big reports. Reports that make mortal print drivers quiver and the Clarion IDE barf up cookies.

One client has a many-DLL phone service program with a database of 150 tables. But there is only a single report procedure for all of that data, which comes from various files supported by a host of EXEs. One report! How can that be?

This client likes to mix and match various detail lines and files and file relationships. Sometimes users will generate a report in invoice/detail style. Or perhaps the user only wants the total line from the invoices. Or maybe the user wants the total line of every invoice, plus the credit rating and total profit made on those clients who have a certain zip code, and who buy more than twice a month and less than 10 times a month.

The report that defines all of these details together can get very large. Add one detail too many and the Report Formatter in the Clarion IDE will give the infamous Heap Overflow error. Where do you go? What do you do?

In this article I'll cover various techniques that I have used to make big reports "fit" inside the Clarion IDE better.

Smaller variable names

I employ what Carl Barnes' calls the "Cajun File Naming Convention". I use big, verbose variable names, like `Loc:InvoiceTotalBeforeTaxes`. I have to restrain myself when writing big reports. The smaller the variable name, the better the report formatter likes it because it limits the overall size in bytes of the report structure. In the worst case, you may have to define

some local variables with terribly short names.

Shorten field equates

Long field names can mean long field equate names for variables populated more than once on a report. The IDE adds the third parameter to the USE attribute to prevent duplicate FEQs for multiple instances of the same variable. If you aren't going to refer to that control in your code, then you can shorten the FEQ to something small and nearly meaningless. For instance:

```
STRING(@s25),AT(2813,198),|
   USE(Add:City,,?Add:City:2),#ORIG(Add:City)
```

can become:

```
STRING(@s25),AT(2813,198),|
   USE(Add:City,,?X1),#ORIG(Add:City)
```

Remove unnecessary field equates

This one can save a lot of room. In a report, I drop a string control onto a detail. The Report Formatter generates the following in the report details:

```
STRING('String 30'),AT(1583,146),|
   USE(?String30),TRN,#ORIG(?String30)
```

If this string is never going to be referred to in my code and never modified, I don't need a USE property at all. So it becomes:

```
STRING('String 30'),AT(1583,146),|
    TRN,#ORIG(?String30)
```

This also applies to graphic controls like LINE and BOX.

Unnamed is unnecessary

Many times, when you are setting properties on a control, the Clarion IDE tries to be helpful and reads a USE variable that you previously removed called `?Unnamed` or some derivation of `?Unnamed:ControlNumber`. These can be removed from your report source also.

Unnecessary template anchors

The templates use the `#ORIG` attribute to "anchor" template code to an existing control. If your controls aren't referenced by a template (and *very* few report controls are), you can safely delete this attribute. Keeping in mind the above tips, you can reduce this code:

```
STRING('String 30'),AT(1583,146),|
```

```
   USE(?String30),TRN,#ORIG(?String30)
```

to this code:

```
STRING('String 30'),AT(1583,146),TRN
```

Unnecessary offsets

Most of the numeric controls that I populate on reports only require right justification. Decimal justification with large offsets are overkill and too verbose. This code:

```
STRING(@n-10.2),AT(6531,219),USE(Cus:Balance)|
  ,DECIMAL(12), #ORIG(Cus:Balance)
```

becomes:

```
STRING(@n-10.2),AT(6531,219),USE(Cus:Balance),RIGHT
```

Reduce font settings

If you are using a single set of font attributes for every control of a report, set them on the report and not every control. If a detail has a different set of font attributes from the entire report, set those attributes on a detail basis. If a control will have different font attributes depending on its USE variables value (like BOLD for balances less than zero), then let the default be no font settings and add the font settings at run time. This can be accomplished by using the property syntax in one of two ways. You can use SETTARGET to the report and then set the control font settings. My preferred method is to use the full control target syntax `Report$?MyControl{Prop:XXX} = SomeValue`.

All of these things can help, and can give you room to double the size of a report that the IDE can handle. But what happens when that isn't enough either? How do you put ten pounds of hog fat into a five pound bucket?

Pseudo Hand-Coded Reports

As your report grows even more, all of your detail lines won't fit in the Clarion IDE Report Formatter at one time – you'll get a GPF. But are you really working on every detail at once? Probably not. So here is some light at the end of the tunnel.

In my case, I tend to hand code the actual calls to produce the details printing. By that I mean, I first go to the Report Properties button on the procedure properties screen, and then the Filters tab. There, I type 1 = 0 as the filter expression for every detail within my report. Since 1=0 always evaluates to false, the generated code will never print a detail, so I'll need to hand code each PRINT statement, usually within the TakeRecord method of the report. My reports are originally "wizarded" reports within ABC so I am not talking about hard core hand coding here.

First, go into the source of the Report. You can reach it by pressing the ellipses to the right of the Report button on the procedure properties window. Select the entire report structure and copy it to the clipboard. Go to the embed tree and find

```
Local Data
   Generated Declarations
      Window Structure
```

and add a source embed with a priority of 5050. You now have a procedure with your report declared twice. How do you get rid of the original one?

Beneath the previous added a new source embed. In it, tab once (you don't want the compiler to incorrectly interpret this code as a label) and type:

```
OMIT('TheReport')
```

Add another source embed here with a priority of 6300, tab once, and type:

```
! TheReport
```

The first embed is before the generated report, the second is after, as shown in Figure 1. Now the compiler will ignore the original report altogether, and you have only one report (the one you copied) defined. In order to modify that report, return to the embed point that contains the report structure and type Ctrl-F. This will call up the Report Formatter without as much overhead as is used when you call the formatter from the Report button on the procedure properties window. This Report Formatter is weaker than the standard AppGen one in that it doesn't know how to provide the fields toolbox. So, how can I still build my details with the toolboxes?

I go back to the original Report Formatter and report. I delete all but the details I want to work on. I can then safely design my print lines with access to all variables from the File Tree and not worry about memory space. Then, I can go to the report source and copy and paste my new detail to my Report embed.

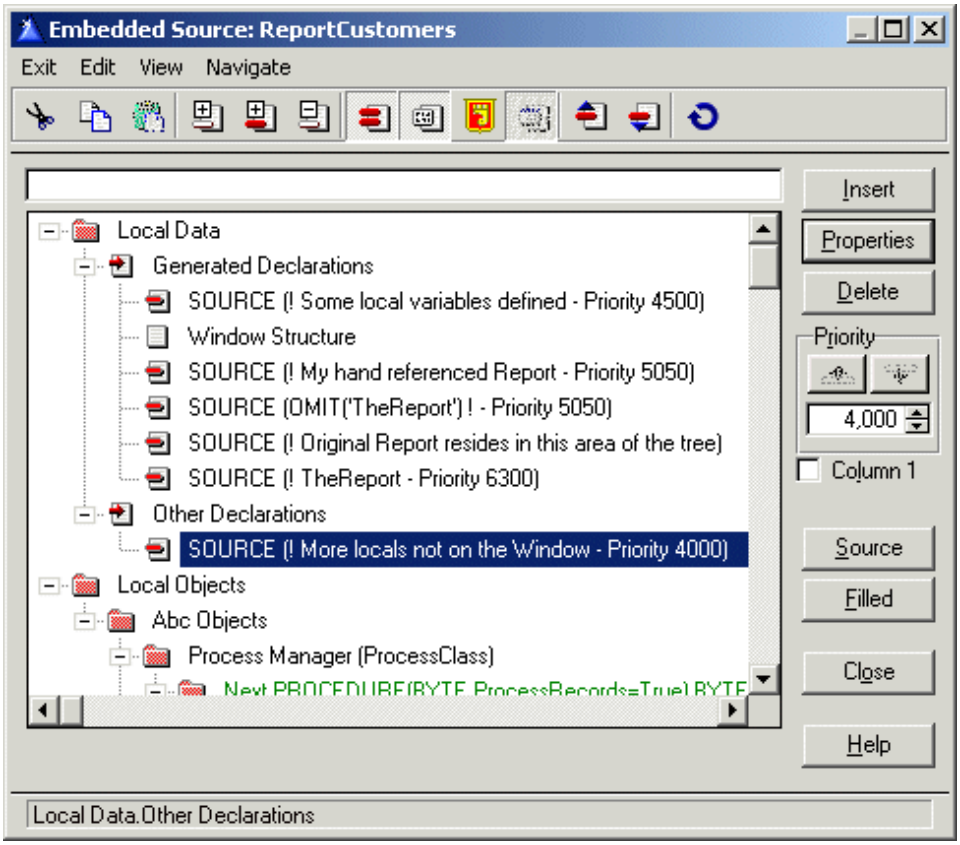

Figure 1. Omitting the generated report.

The only constraint now becomes the size of a source embed point, which is around 64K. If this is not enough, just divide the report into many different embed points.

By trimming the fat from your report structure, economizing on shared attributes, and using a "working copy" of the report, you can fit some really large reports into the IDE. Each of these steps buys you more room and still gives you the flexibility of using the Report Formatter.

Andrew Guidroz II, when he isn't handfeeding the tufted titmouse, writes software for all facets of the insurance industry. His famous Cajun cookouts have become a central feature of Clarion conferences throughout the U.S. Andrew's Cajun website is www.coonass.com.

Reader Comments

Add a comment

Clarion MAGAZINE

Search
Go

Home

COL Archives

Information
Log In
Membership/ Subscriptions
FAQ
Privacy Policy
Contact Us

Downloads
PDFs
Freebies
Open Source

Site Index

Call for Articles

Reader Comments

Recovering Deleted Records

by Dave Harms

Published 2001-07-19

If you've ever used the Clarion (DAT) file driver, you may remember that DAT files don't reclaim used space unless you put the RECLAIM attribute on the data file definition. This means that deleted records still physically exist and can be recovered. The TopSpeed (TPS) driver, on the other hand, automatically reclaims used space, so once a TPS record has been deleted, for all practical purposes it's beyond recovery. You may find the same is true of your SQL database, although most SQL servers provide a data logging capability that lets you track and recover database changes.

There is, however, a fairly easy way to make deleted record recoverable. As Dennis Evans recently pointed out in the newsgroups, Clarion 5.5 added a new `DeleteRecord` method to the `FileManager`, and you can override this method to implement your own record deletion scheme.

To make your data recoverable, first modify the table you want to protect by adding a field to mark a record as deleted. In the example application (below) I modified PEOPLE.TPS (from the MailList example application) by adding a `STRING(1)` field called `Deleted`, in which I can store a value of 'Y' to indicate a deleted record. You could also use a `BYTE` field and store a 1 or a 0; perhaps it would give better performance, perhaps not. I'd be interested to hear opinions on the subject. In any case I don't expect the difference would be great, and 'Y' and 'N' are certainly easily understood by anyone browsing the data.

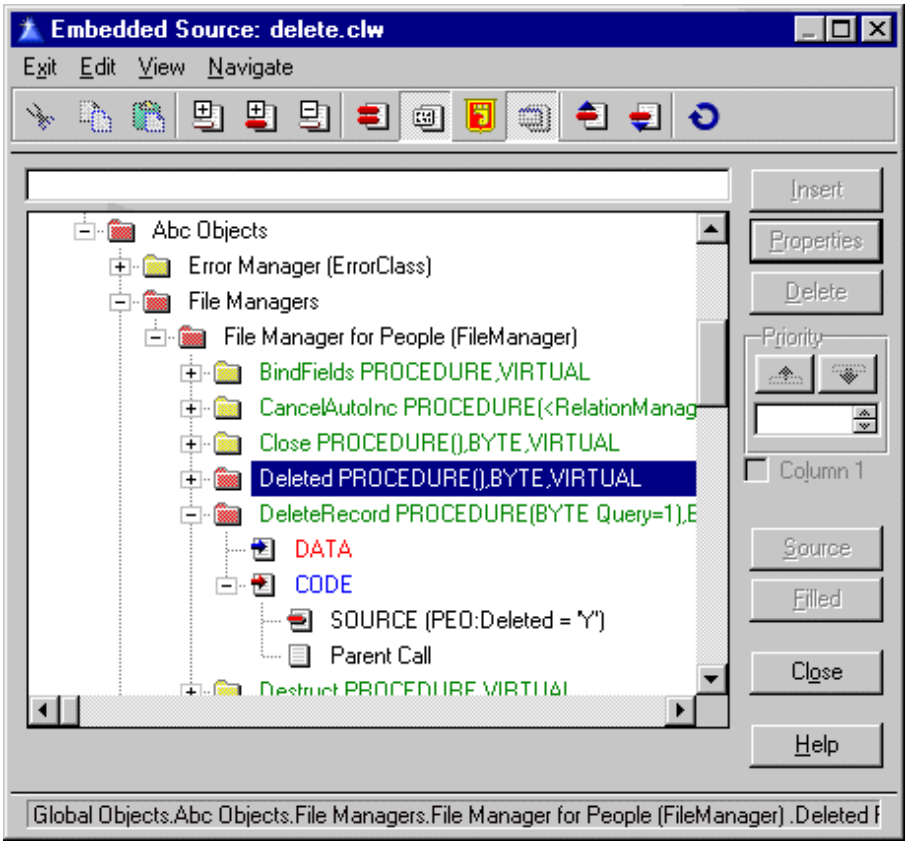

Figure 1. The DeleteRecord embed point

The next step is to derive a new `DeleteRecord` method. Go the Global Embeds, and as shown in Figure 1, under Abc Objects look for the File Managers heading, and choose the `FileManager` object for the table you want to protect. Under the `DeleteRecord` method, add the following source code *before* the parent call, using a priority of 4999 or less:

```
PEO:Deleted = 'Y'
PUT(People)
RETURN LEVEL:Fatal
```

The `DeleteRecord` method is the one ABC uses to delete the record in your table, and by placing this code in the embed point you create a new virtual method. This method replaces (but also calls) the stock ABC method which has the following code:

```
FileManager.DeleteRecord PROCEDURE(BYTE Query)
CODE
DELETE(SELF.File)
RETURN(Level:Benign)
```

As you can see, it's essential that you place your code before the parent method call embed and short circuit that call with a `RETURN` statement, or else the record *will* be physically deleted.

Now that you've marked your record as deleted, you need to filter it from any browses, processes, or reports. In this example, I used the code:

```
PEO:Deleted <> 'Y'
```

in the browse's Record Filter prompt, as shown in Figure 2.

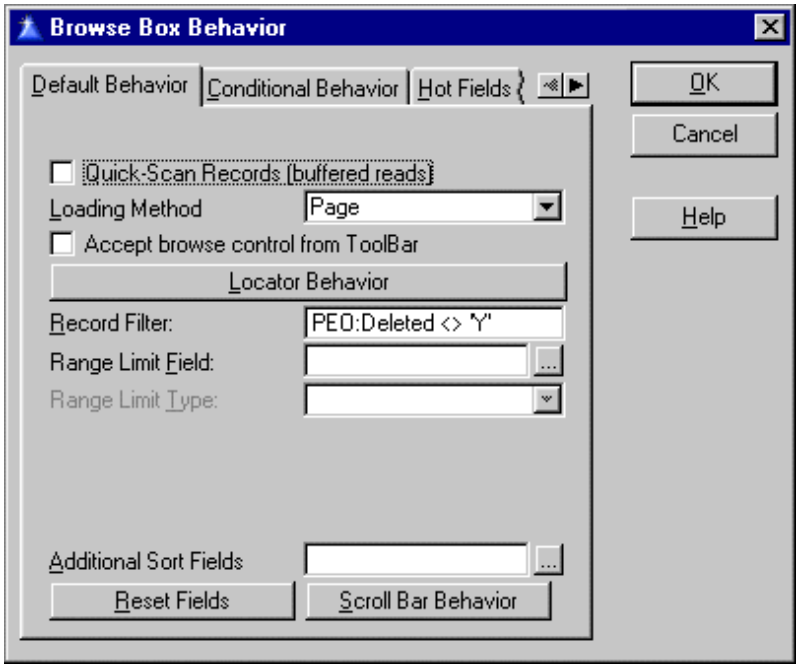

Figure 2. Filtering the deleted records

You're not done yet

There is one other bit of code you probably should add. Along with the `DeleteRecord` method, Clarion 5.5 added a `Deleted` method, which the Relation Manager uses to determine if a related record should be considered active. The default `Deleted` method always returns `LEVEL:Benign`, so as long as the record physically exists, RI code will execute against that related record. This could cause problems if, say, you have a restrict constraint on deletes. Imagine you've implemented an invoice header/detail set of tables, and you're using your own `DeleteRecord` to mark invoice details as deleted. You can delete all of the details, but you won't be able to delete the header because the RI code will still see the deleted details records. Clear as mud?

To get the RI code to behave properly, you'll need to override the FileManager's `Deleted` method as well as the `DeleteRecord` method. Here's the code I used in the example application:

```
IF PEO:Deleted = 'Y' THEN RETURN LEVEL:Fatal.
```

You can return anything other than `LEVEL:Benign`, which has a value of zero - the test in the RelationManager code is `IF SELF.Me.Deleted() THEN CYCLE`.

Figure 3 shows both embed points for the People table's `FileManager` instance.

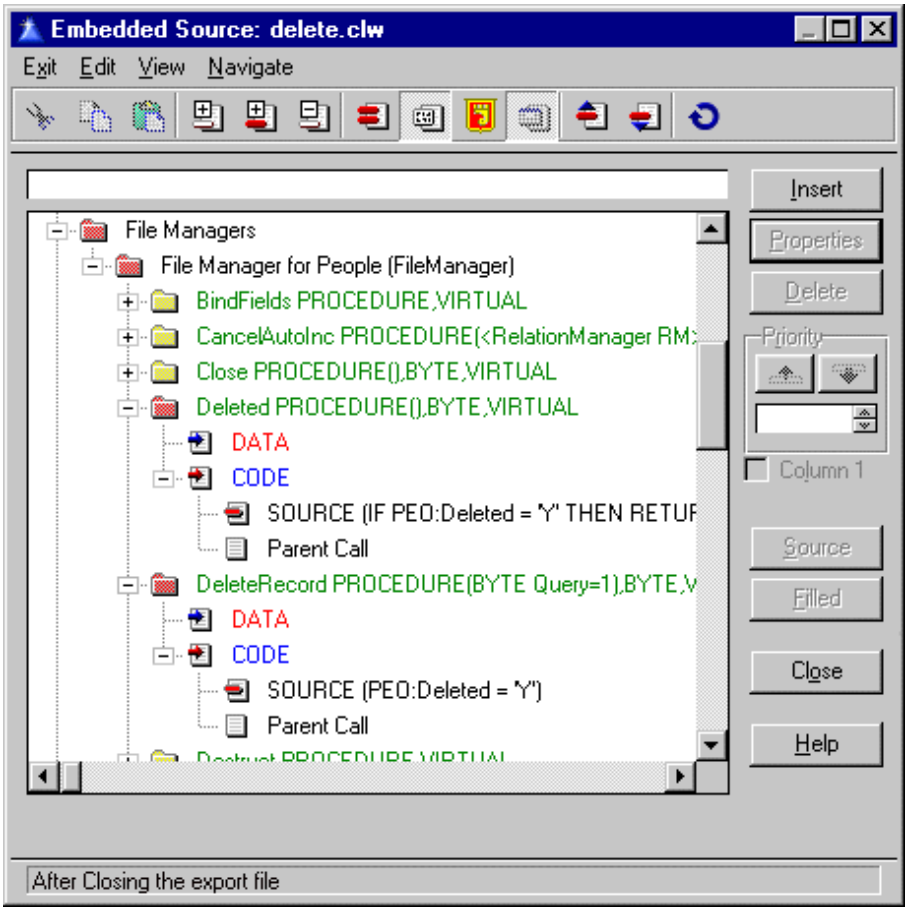

Figure 3. The DeleteRecord and Deleted embed points

If you're creating a multi-DLL application, you don't need to duplicate this code for each DLL. Normally you declare all files as external, except in one files and shared globals DLL. As soon as you mark a table as external, the FileManager embed points become unavailable for that table, so the only place you can (and should) put the code is in the files and shared globals DLL. .

If you're using this approach for all of your tables, you'll probably want to consider writing some templates to streamline the embed creation, particularly for browse, process, and report filtering. Or you might want to consider getting the FileManager to do the filtering on a global basis. I'll leave that as the proverbial exercise for the reader. If you come up with a solution, post it at the end of this article.

Download the source (C55 only)

David Harms is an independent software developer and the editor and publisher of Clarion Magazine. He is also co-author with with Ross Santos of Developing Clarion for Windows Applications, published by SAMS (1995). His most recent book is JSP, Servlets, and MySQL, published by HungryMinds Inc. (2001).

Reader Comments

Add a comment

Dave, I enjoyed this. Give up more of the...
You're most welcome - and thanks to Dennis Evans for...
Template and source available - Dennis Evans has provided a...

Clarion MAGAZINE

Search
Go

Home

COL Archives

Information
Log In
Membership/ Subscriptions
FAQ
Privacy Policy
Contact Us

Downloads
PDFs
Freebies
Open Source

Site Index

Call for Articles

Reader Comments

Avoid My SQL Mistakes!

by Mauricio Nicastro

Published 2001-07-18

The aim of this article is to show all the problems that I had to go through when I began to work with SQL. I'm hopeful that this could be helpful to those who, like me, are still not experts at working with this kind of database. Many of the things that I describe here may seem obvious, but the fact is that after being in a fix over and over you realize that they are not *so* obvious. The difficulties I encountered were the following:

Slow browses

All SQL tables must have at least one unique index. Whenever it is possible, I define autonumber fields that will be part of the principal key. At one time, I had a table with this field in two different keys: Code and Description.

- KeyCode = Code + AutonumberField
- KeyDesc = Description + AutonumberField

I assured my client that the execution of the program would be faster with SQL.! The browse had two tabs and when I changed from a tab to the other, I had to wait 20 seconds in order for the browse could show me something. It wasn't a big table (approx. 20000 records), and as you can imagine, my client wanted to make a programmer martyr of me; I didn't want to be Saint Programmer, not at so young an age!

I revised everything, worked with SQL Query Analyzer, looked for some problematic embedded code, but I couldn't find any anomalies. Just by chance, I found a manual in my bookshelves which contained the solution: all Clarion's indexes must be defined with the Case Sensitive field checked. If you don't do this, the operation is slow because the view engine will turn all the fields into uppercase while it is collecting the data.

Sending commands to the server

I had a requirement to do some mass updates to the data. The

selective replacement of prices in a products table, for example, can be done in two different ways. One way is to make a loop using an index that matches as much as possible the selective criteria (this is the usual way when working with TPS tables). The other is to let the SQL server do the work. For example, in a general price list, I might need to increase by 10% the prices of those products bought from provider X. Here's how I would write this code in SQL:

```
Products{ Prop:SQL } = |
  'update products set price = price * 1.10 ' |
  & 'where ProviderCode = ‘ & Pro:Code
```

Now, is that right? It seems so; however, there is a small mistake: If `Pro:Code` is a CSTRING field, like any string it must be in single quotes. Thus the query, in this case, must be:

```
Products{ Prop:SQL } = |
  'update products set price = price * 1.10 ' |
  & 'where ProviderCode = ' & ''' Pro:Code & '''
```

Remember that it is necessary to write three single quotes, one to open, the real quote, and the last to close.

Date Fields

The Clarion dictionary editor is not able to recognize a SQL date-time field. Instead, you will see a STRING (8) field, followed by a group declared OVER the STRING(8). This group has TIME and DATE fields. Let's say you have to order the browse by date with the use of a locator.

The point here is this: how is this index created? What are the key components? As a way of trying to figure all this out, I used the trial and error method, and then came to learn that it is necessary to use the `Field_DATE` in the index key, because the name of this field will be used in the `ORDER BY` statement sent to the database server. Date ordering won't work if you use the string or the group field.

Filtered locators

Filtered locators are very useful, because while you are typing the browse is filtering. Besides, if the "Find Anywhere" field is checked it is possible to find the string the user types if it matches anywhere in the corresponding table field, not just when the start of the locator value matches the start of the table field. In my case, I don't know if it was my mistake, a Clarion problem or what, but I couldn't get Find Anywhere working. In my browse I had to filter a hot field and when I typed in the locator, the same letters appeared in the field and never worked. I tried with a variable as locator, then with the field, and it was impossible. As a result, I decided that the engine would do the work for me. I created a variable where I typed the string I was looking for, and in the

accepted embedded point, I wrote:

```
IF Loc:Description <> ''
    BRW1::View:Browse{ Prop:SqlFilter } =|
   'Description   LIKE ''%' |
  & clip( Loc:Descripcion ) & '%'''
ELSE
  BRW1::View:Browse{ Prop:SqlFilter } = ''
END !IF
BRW1.ResetQueue( Reset:Queue )
BRW1.ResetFromFile()
```

This code works, and very well, but of course only with SQL databases, as these normally support the LIKE matching function. You can add another variable and make it work as the "Find Anywhere" check.

Refresh the window

Sometimes you will encounter the typical invoice browse: header and detail. You want to add a new invoice, so you open the form, accept the entries and when you go back to the header browse ... the invoice you have just added is not there!! You can do this:

```
BRW1.ResetQueue( Reset:Queue )
BRW1.ResetFromFile()
```

In this way, you refresh the queue and force the program to retrieve the data from the table.

Process

When you work with SQL you have to tell the engine which fields you need, by listing those fields in the browse's Hot Fields tab. With TPS files the entire record is always available; with SQL only those fields the browse/procedure/report knows about, because they're listed in the file schematic or in the hot fields, will be used in the corresponding SQL statement.

Parent-child relationships

Parent-child relationships can also be a problem. Take the invoice case as an example once more. Sometimes you use an autonumber field that is in charge of maintaining this relationship. In this case the question is: which program will autonumber this field? Your Clarion application, or the database server? I have read that I should leave this work to the engine. The problem is that the engine does this job in the same moment it is adding the record; consequently, all the children do not have a number which enables them to keep the relationship. If you want this working properly, you may add some code to retrieve the number of the field, then put this value to each child record and, finally, save everything. But if you are a Clarion Magazine reader (see Stephen Mull's article on converting to SQL) , you may note that it is better to use a

CSTRING(18) field to keep this relationship. In the Field Priming on inset button, you can do the following:

```
RecordID = today() & clock()
```

That does work, though many of you might argue that there is a risk of getting duplicates keys when working with a large number of clients simultaneously. My counter-argument to this point is that I have worked with more than 150 terminals at the same time and that has never happened. I cannot say that it is not possible, however, it is not likely to happen.

These are all the interesting setbacks I had to sort out so far. If you're just getting started with SQL, I hope you find this article useful.

Mauricio Nicastro is a newcomer to the Clarion world. He has been working with databases for over ten years, and began writing programs in Clipper (which is also the name of his dog). In his spare time Mauricio usually plays soccer.

Reader Comments

Add a comment

The statement "All SQL tables must have at least one unique...

Clarion MAGAZINE

Search
Go

Home

COL Archives

Information
Log In
Membership/ Subscriptions
FAQ
Privacy Policy
Contact Us

Downloads
PDFs
Freebies
Open Source

Site Index

Call for Articles

Reader Comments

Using Procedure Category to Split Apps into DLLs

by Carl Barnes

Published 2001-07-16

Most Applications start as a single App file. As time goes by more and more procedures get added and the App file and the resulting EXE can become very large. This can cause problems for the Clarion IDE, compile times get very long, and any change to the application means re-deploying a large EXE. The only cure to this problem is to split the App into multiple smaller App files and compile each of those as DLLs (or LIBs). There are already several articles in Clarion Magazine that talk about the process (listed below); in this article I'll show you how I use the Procedure Category to organize that breakup process.

In Clarion 4 a new field named Category was added to the Procedure Properties window. A new tab with Procedures sorted by Category was also added to the Application Tree window. These features give you a new logical way to group and view procedures in an application.

The reason for adding Category was that under ABC the Browse and Form templates are implemented as the Window template with extension templates. This reduced the effectiveness of viewing the Application by Template since all Browses, Forms and Windows showed under the Window template. Now when a Form procedure is added the template is "Window" but the Category is "Form."

The new Category field can be very useful when moving (or copying) procedures between application files. When breaking a single large EXE into multiple DLLs you will move a lot of procedures between App files. (The example here is small with just 12 procedures and three App files, the last time I did this for hire there were 150 procedures and nine App files.) This is usually messy and somewhat stressful to keep organized. (To reduce stress hum the song "Breaking up is hard to do", although you should only sing if you have a *private* office.) A good way to keep things organized and have a visual plan is to put the name of the future DLL into the Category of each procedure. An example of this is shown in Figure 1.

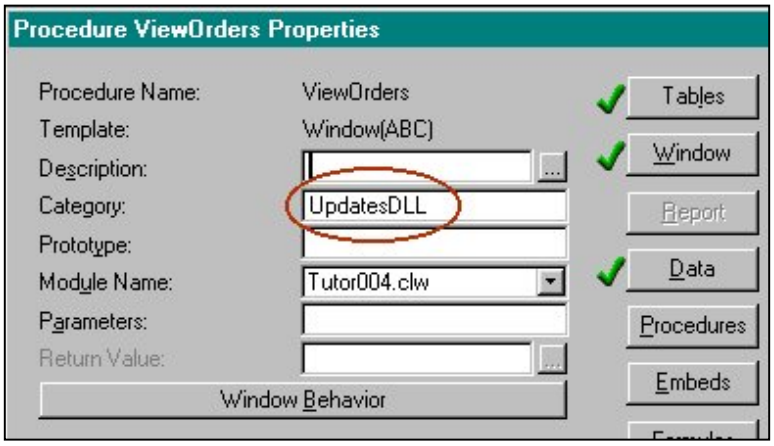

Figure 1. Setting the procedure category.

The future DLL name should be entered for every procedure that will be moved. Any procedures not assigned a DLL category name will default to their template name. (Since I did not want to lose the existing category names identifying Form and Browse windows I placed my DLL name in front of the existing name, e.g. "UpdateDLL Form".) Figure 2 shows the category view of the Tutor example application where the future DLL name has been entered for all procedures. I use this view to show my procedures sorted and grouped by the future DLL name.

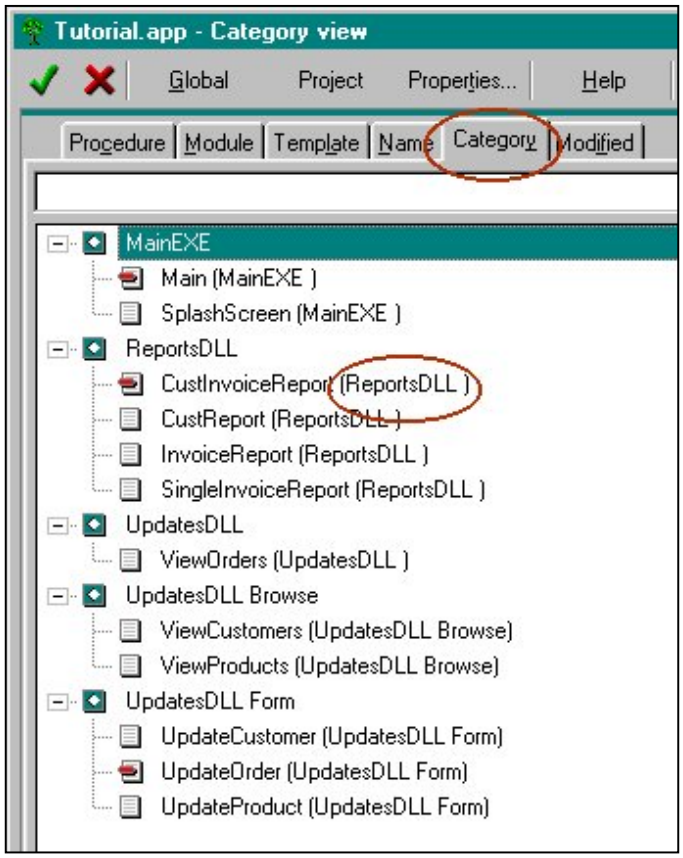

Figure 2. Showing procedures by category.

Since the Category name shows in all views of the Application tree this makes it easy to see and understand exactly how your App will be split up, and how that will affect the calling relationships. Think

of it as a WYSIWYG DLL split tool. In Figure 3 the circled procedure was to be put in the Reports DLL, but after looking at the call tree I decided it might be better to put it in the Updates DLL since that is the only place it is used. This will save having the procedure exported which saves time when running the EXE. On the other hand Gordon Smith suggests putting all reports in their own DLL because reports change frequently and it will be more convenient to patch the Users install.

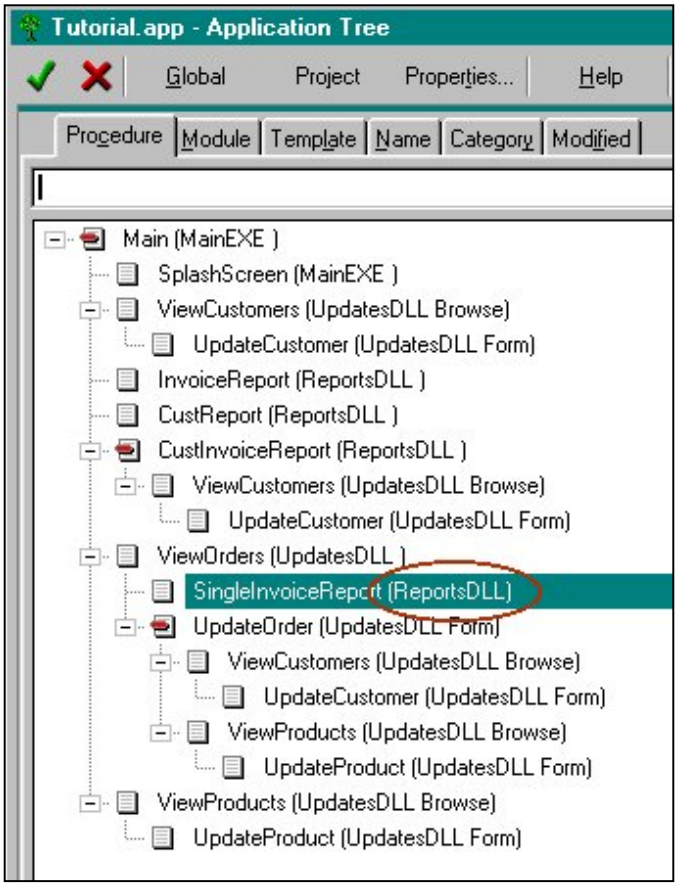

Figure 3. Using the category to check the calling relationship.

Now that I have a clear plan of the future DLL for each procedure I can use Category during the import. Figure 4 is a screen shot of the Import From Application view by Category. This makes it very simple to find the right procedures to import as every procedure that is destined for this DLL is sorted together.

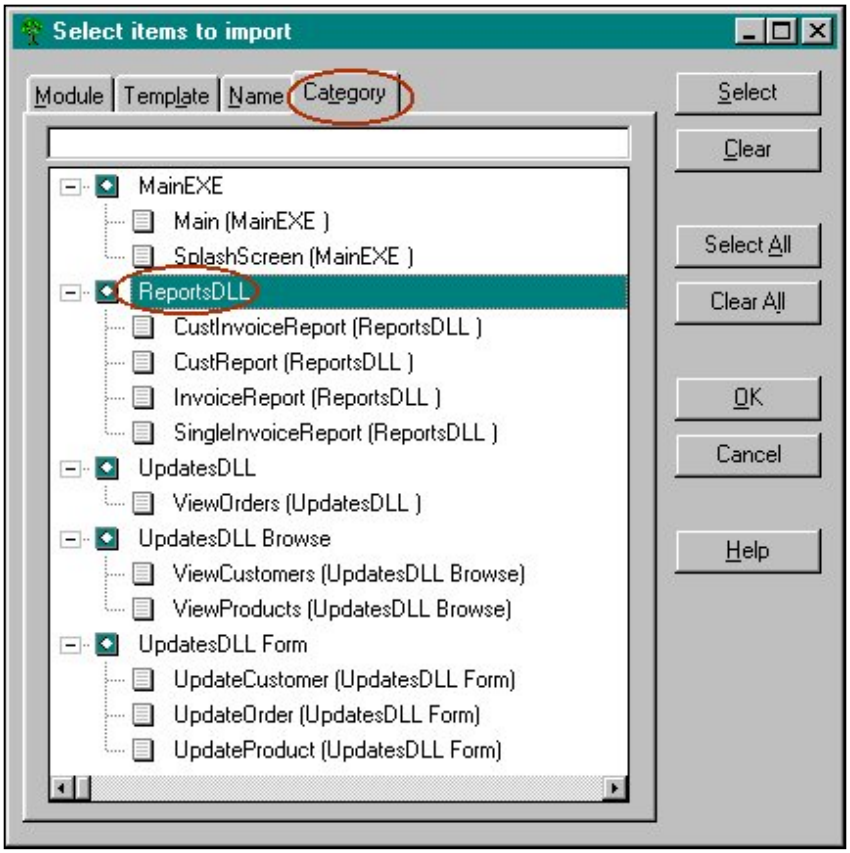

Figure 4. Importing by category.

If you intended to move the procedures out of the current App file, then you can use the Application view by Category when you delete procedures as well, since everything you want to delete will be grouped together.

Getting exports right

You can also use the category to keep track of which procedures will need to be exported. Looking at the original call tree any procedure on level 1 of the tree will need to be exported (unless it's in the EXE). Also any procedure below level 1 in a branch where the calling procedure is in a different DLL will need to be exported, as seen with `SingleInvoiceReport` and `ViewCustomers`. This would also be a good time to check that procedures that are part of the main EXE never appear below level 1 of the tree. Nothing can be exported from an EXE (more on that point below).

To mark a procedure as requiring Export simply add the word "Export" after the name of the future DLL. Once the procedure is imported into the DLL App the "Export" in the Category will serve as a reminder to check the exported box on the Procedure Properties. Figure 5 shows a revised calling tree with the exports marked.

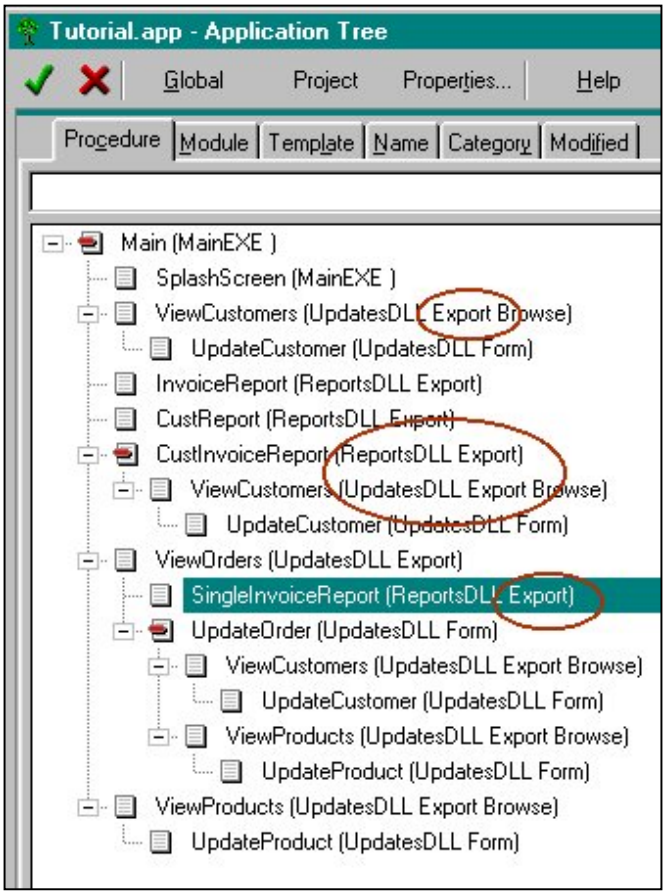

Figure 5. Marking procedures for export.

The WYSIWYG display of this information highlights a potential problem with the design where the Reports and Updates DLLs call each other. `CustInvoiceReports` calls `ViewCustomers`, and `ViewOrders` calls `SingleInvoiceReport`. Most articles on splitting into DLLs suggest avoiding this cross-calling chicken-and-egg situation. On the other hand I have seen it work fine every time. The first compile might not work. It just takes a second make of the project to get the link to work.

If you truly want to avoid this situation of cross calling I would suggest you sketch out a calling tree of the DLL structure and assign each DLL a level number. (An example of this is in the MultiProj tutorial discussed below.) The EXE would be Level 0, the Updates DLL would be level 1. Since Updates call Reports, the Reports DLL would be level 2. Put these level numbers in the Category field too. Then when reviewing your call tree you can verify that along the branches of the tree the level numbers never decrease. (One annoyance with this is if you change the DLL structure and the level numbers change you will have to edit the level in every procedure's Category.) Figure 6 is a screen shot with level numbers, where you can see the problem of a level 2 DLL calling a Level 1 DLL.

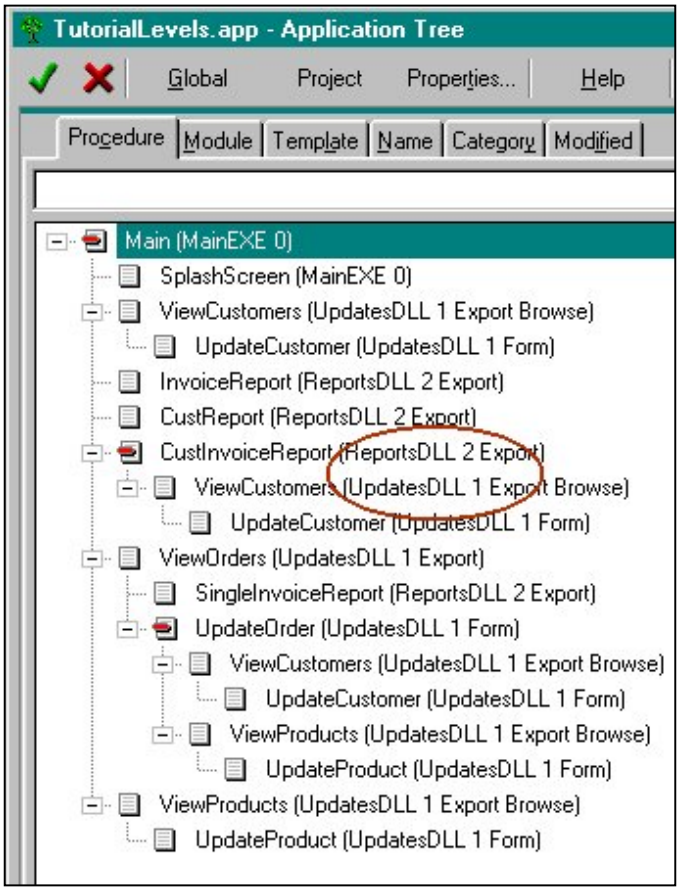

Figure 6. Adding DLL calling levels to the category.

The downside

There is a downside and that is that the Category name replaces the Template name in the Application views (except the Template view). In the screen shots above you no longer see (Frame), (Browse), (Window) or (Report) next to the procedure name. You now see (ReportDLL) which is the Category name. To fix this you have to delete the Category name. Generally I do not care about the template name so I only clean this up and remove the DLL names as needed. Since I have to open each exported procedure to check the exported box, I typically will fix the Category name on those during that process.

Exporting procedures from an EXE

As noted above you cannot export procedures from an EXE. But that does not mean that a DLL cannot call a procedure that resides inside your EXE, it just means the compiler and linker will not help you do it. This type of call is typically referred to as a "callback" and can be implemented easily using the methods described in Larry Sand's excellent article "Loading DLLs at Runtime" http://www.clarionmag.com/cmag/v3/v3n5runtimedlls1.html. You would prototype the procedure exactly as Larry shows using a variable with the `NAME()` attribute to specify a function pointer variable. To get the value of the function pointer you would not use `LoadLibrary()` and `GetProcAddress()` as show in the article. The EXE would have to call the DLL once and pass it the `ADDRESS()` of

the desired procedure, or the EXE could store the address in a global variable that the DLL imported. After this address was assigned to the function pointer variable, you could call the EXE procedure from the DLL. I cannot think of many good reasons to go to all of this work, the main one would be a procedure you wanted to keep secure. Exported procedures from a DLL are public and could be called by anyone without much trouble. A "callback" procedure in an EXE would be much harder to hack. A second possibility might be job security.

Using modules for organization

A second tip for being more organized with your Apps is to group your procedures logically into modules rather than let them be assigned in the order they were added to the App. I like to keep all procedures in a calling tree branch in the same module, for example, a browse and its form. Also group procedures that relate to each other, such as all of the Customer procedures should be in the same module. Then when you're doing modifications you spend less time hunting around for the procedure.

Other tips

When I break an App into DLLs I start by making an APP called Empty.DLL. This is a DLL that will be the starting point for all of my other DLLs. It contains all of the global variables, embed points and application template extensions that will be needed in every DLL. It is also handy to keep around for any time you want to add a new DLL to the project.

The Category method discussed above can be useful any time you want to move procedures from one App to another. Rather than write down a list of procedure names simply type something into the Category. For example, I never really delete procedures, I move them to an OldStuff.App that I keep (but never compile) for obsolete code. To make the move easy I change the procedures' category to "Obsolete", which makes them easy to find them when importing into OldStuff.App and when returning to the original App to delete them.

Summary

As I've show above the Procedure Category can be used to keep you organized when you are moving or copying procedures between App files. Here is a summary of the steps I use when splitting an App into DLLs.

1. Enter the future DLL name into the category of each procedure
2. Retain any existing category names by putting the DLL name first
3. Check Procedures by Category view to find procedures not assigned a DLL
4. Review the App Tree to see if the DLL structure makes sense

5. Use the App Tree to mark exported procedures
6. Create the new DLL App
7. Select Import From Application from the File menu
8. Select the Category Tab
9. Select all the procedures for the DLL and import them
10. Open each procedure and remove the DLL name and handle exports

For More Help on Splitting into DLLs

You will find some excellent information in the past Clarion Magazine articles "Four DLLs And An Executable" by Gordon Smith at http://www.clarionmag.com/cmag/v1/v1n5fourdllsandanexe.html and "Using Dynamic Link Libraries" by Russ Eggen at http://www.clarionmag.com/col/98-05-dlls.html

A nice discussion with plenty of diagrams and lots of helpful instructions can be found in the "Writing Multi-DLL Applications" tutorial that is part of the documentation for CapeSoft's Multi-Proj and can be found online at http://www.capesoft.com/docs/multiproj/mptutor.htm

In your Clarion Examples directory you will find the DLLTutor directory which contains the Tutor example Tutorial.App split out into multiple DLLs. I was not able to find any documentation associated with this example. My above screen shots are based on this App and so is the MultiProj Tutorial. This is a nice-sized example to learn from and is done the correct way.

As well, the Clarion Help "How Do I..." section has a subsection named "Compiling and Linking" with topics that provide information on splitting Apps and using DLLs.

Carl Barnes is an independent consultant working in the Chicago area. He has been using Clarion since 1990, is a member of Team TopSpeed and a TopSpeed Certified Support Professional. He is the author of the Clarion utilities CW Assistant and Clarion Source Search.

Reader Comments

Add a comment

"Sometimes" Lookups

by Steven Parker

Published 2001-07-06

I once characterized lookups as "a subject that just never seems to go away" (see *Lookups: You Don't Always Want to Validate*, Clarion Online, 2, 9, April 1999). It doesn't and it hasn't.

The Clarion templates provide three ways of doing lookups/data validation for fields requiring entries (see *Lookups in C4*, Clarion Online, 2, 1, August 1998). But what if you want to allow the end user to leave a field blank but validate against a list of "approved" choices when the field actually contains data?

In this case, either the field must be empty or its value must validate against another file. In other words, sometimes I want to do a lookup, sometimes I don't.

What if you want to allow users to enter whatever they like, with an optional lookup?. In this case the user can leave the field empty, make an entry or call a lookup.

Where might such a bizarre configuration of an entry field be appropriate? I first ran into the need for this in a checkbook application. Many payees are one-time (or once in a great while) payees. If a payee is infrequent, I see no need to store the name in a lookup file. However, there are many payees to whom checks are written or with whom charges of goods and services are made regularly. It is convenient to look up these payees (i.e., not have to type their names in).

Again, sometimes I want to do a lookup; sometimes I don't (for you Peter Paul Mounds and Almond Joy fans, "sometimes you feel like a nut, sometimes you don't").

Empty or In File

To implement an empty-or-in-file type of "sometimes" lookup, it is important to understand a few basics about the way the templates generate lookup code. Most lookups are created using the entry field's Actions tab:

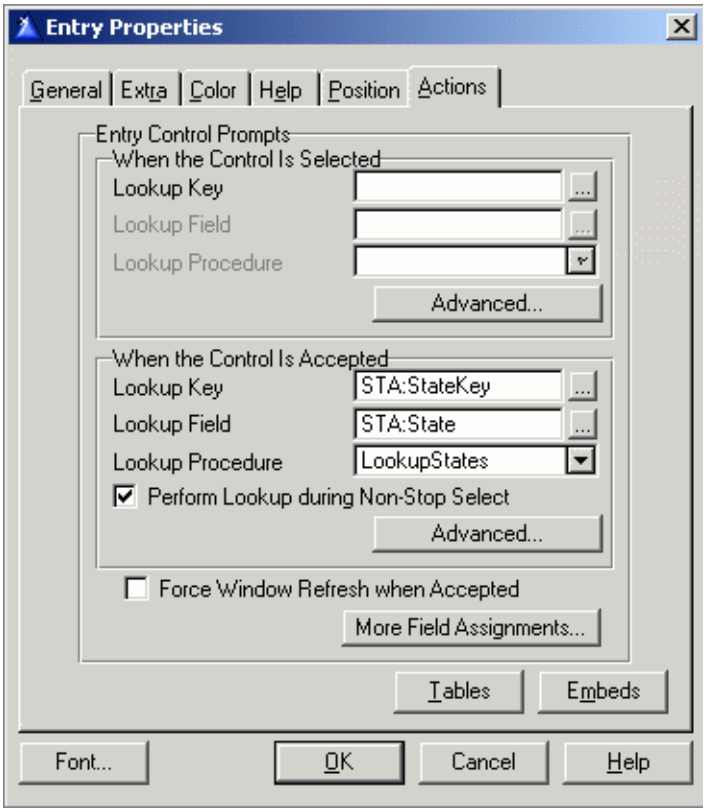

Figure 1. Entry field Actions tab

For most data validation, lookups provided by these prompt are not only appropriate but the easiest and best way to create lookups. They are the easiest because no further code is required.

What is important is that there are two places where a lookup may be called: "When the Control Is Selected" and "When the Control Is Accepted."

If a lookup is called from `Event:Selected`, the lookup is mandatory, whether the field is required or not. That is, if a lookup is nominated on `Event:Selected`, the lookup procedure will always be called. Period. The reason for this is that the lookup is called immediately on tabbing onto the control (that's what "when selected" means, after all). It is possible for the end user to cancel the lookup. This will leave the field empty, as desired. But it is not the way I would want an "empty or in file" scenario to operate.

Moving the lookup call to "When the Control Is Accepted" improves program behavior considerably. With the lookup here, the user may tab through the field and ... will the lookup be called?

If a field may be empty, it is obvious that the field is not required. So, if the "Required" attribute is not checked for the field (see Figure 2 – it doesn't matter what the dictionary says as long as *this* checkbox is off), required field checking is not enforced. In this case, the lookup will be called only if the user has makes an entry in the field.

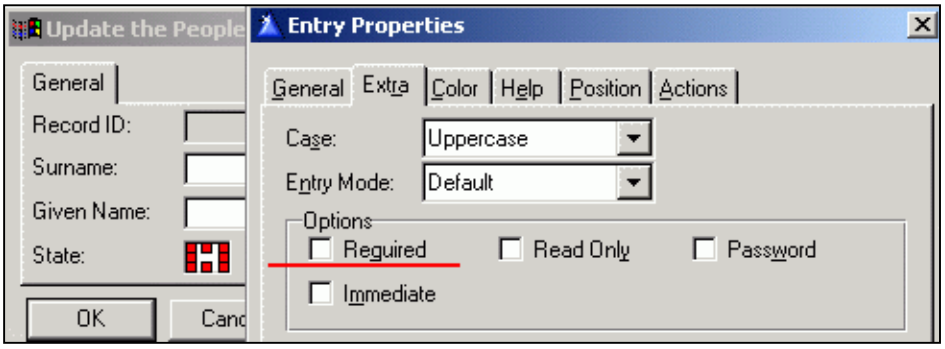

Figure 2. Making a field not required

Examining the generated code (see Figure 3) explains why this works and works so simply. The key is the line:

```
IF PEO:State OR ?PEO:State{Prop:Req}
```

```
OF ?PEO:State
  ! Start of "Control Event Handling"
  ! [Priority 6000]

  IF PEO:State OR ?PEO:State{Prop:Req}
    STA:State = PEO:State
    IF Access:States.TryFetch(STA:StateKey)
      IF SELF.Run(1,SelectRecord) = RequestCompleted
        PEO:State = STA:State
      ELSE
        SELECT(?PEO:State)
        CYCLE
      END
    END
  END
  ThisWindow.Reset()
```

Figure 3: Default code in TakeAccepted for a lookup

The lookup will be called if the field has any contents (`IF PEO:State`) or if the field is required, whether it has contents or not (`?PEO:State{Prop:Req}`). This is precisely what is desired. Therefore, if the form field is *not* required and is empty, no lookup will be called.

Voilà, empty or in file validation!

A More Flexible Alternative

A very valuable but often overlooked template is the CallProcedureAsLookup code template, shown in Figure 4. Notice the prompts for this template. Except for the expected "Lookup Procedure," the procedure to call, each prompt represents an embed. Each represents functionality not available in lookups created from the Actions tab.

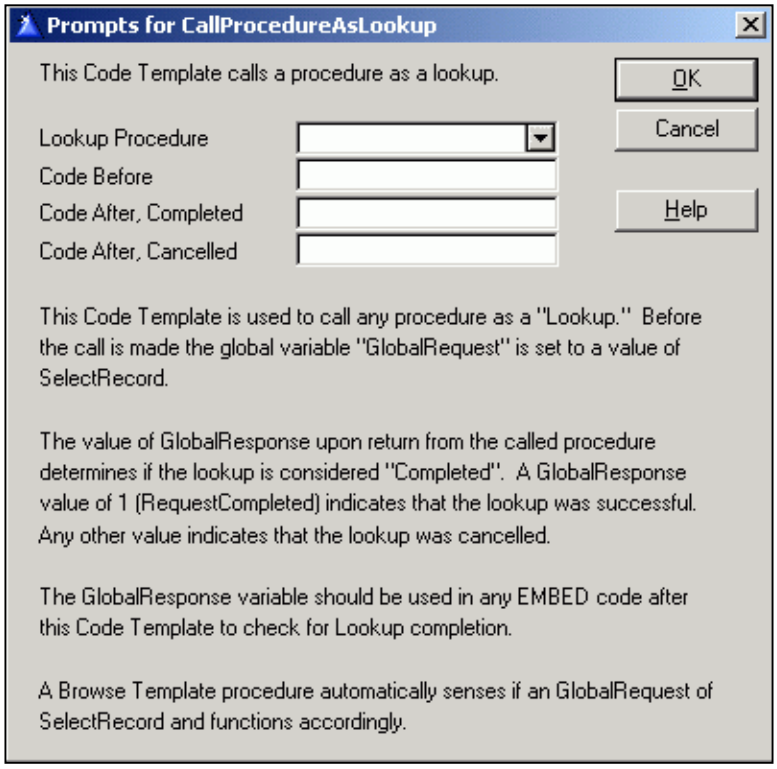

Figure 4: Prompts for CallProcedureAsLookup

The Code Before field is for code that will execute immediately before calling the lookup. Suppose, for example, I don't want to call lookups on Tuesday. The code:

```
If Today() % 7 = 2 then Return Level:Notify.
```

will short circuit the call on Tuesdays.

```
If ~PEO:State then Return Level:Notify.
```

is less ridiculous looking and will prevent the lookup when the State field has no contents (and this is what is wanted here).

The Code After, Completed field allows an assignment to the target field (made automatically in the standard lookup template but, with this template, it must be made manually), and/or assignments to additional fields (similar to the "More Field Assignments" button). But it also allows execution of any code at all. For example:

```
If EV:Type = 'Open' then ?EV:Schedule{Prop:Disable} = True.
```

> **NOTE:** Because the prompts are just a single line, you must adapt your code writing style: no carriage returns are allowed. If this is a burden, `Include()` a file containing your code.

The Code After, Cancelled field lets you embed any code you want to execute before `GlobalResponse` is set to `RequestCancelled`. For example:

```
Select(?)
```

puts the user back on the field when the lookup is cancelled.Because there are no restrictions on code entered here, a function or procedure call is entirely acceptable. Using `Include()` files, quite complex code is possible.

Examining the generated code (Figure 5) shows that both of the "Code After" embeds are generated before the `TakeAccepted` method returns. The standard lookup prompts do not have sufficient granularity to allow code in these places. This is why the CallProcedureAsLookup template can be a very powerful addition to your coding arsenal.

```
!Code Before ------------------------  ! Source before Lookup
GlobalRequest = SelectRecord           ! Set Action for Lookup
LookupStates                           ! Call the Lookup Procedure
IF GlobalResponse = RequestCompleted   ! IF Lookup completed
  !Code After, Completed -------------  ! Source on Completion
ELSE                                   ! ELSE (IF Lookup NOT...)
  !Code After, Cancelled -------------  ! Source on Cancellation
END                                    ! END (IF Lookup completed)
GlobalResponse = RequestCancelled      ! Clear Result
```

Figure 5: Default CallProcedureAsLookup Code

On-Demand Lookups

Now to the case in which the user is to be allowed to enter whatever they want or call a lookup. I once characterized these as "non-validating validation." The desired behavior is that a field be able to contain a value that is not in the validation file. Payees in a checkbook or even in Accounts Payable are a perfect examples.

It is easy to tell whether a control is empty or not. It is easy to tell whether the form is in Non-Stop Mode (i.e., the form has been completed and `0{Prop:AcceptAll} = True`). It is easy to let the user enter whatever they want to enter, that is what entry controls are *for*. What is not so easy is knowing when to call the lookup.

By definition, a lookup requires that whatever value is in the field also be in the validation file. And that is contrary what is needed here. Here, I do not want field contents validated under all circumstances. In fact, I don't want field validation under *any* circumstances. Any validation would force a lookup when a one-time payee was entered and this is not the desired behavior.

The only way to solve this conundrum is to realize that in this case there will be a lookup procedure but there will be no lookup. Since the field must accept anything the user enters, it must be the user that indicates that a lookup is wanted. Think of this more as a quick complete (without the keystroke matching) than a lookup. So, the solution is that user must tell the program that they want to select from a list. This is done by pressing a hotkey (you create the hotkey by right click the control, pressing "Alert," configuring the desired key and regenerating the embed tree to get the AlertKey and PreAlertKey embeds). Add the CallProcedureAsLookup template and name the lookup procedure, as in Figure 6:

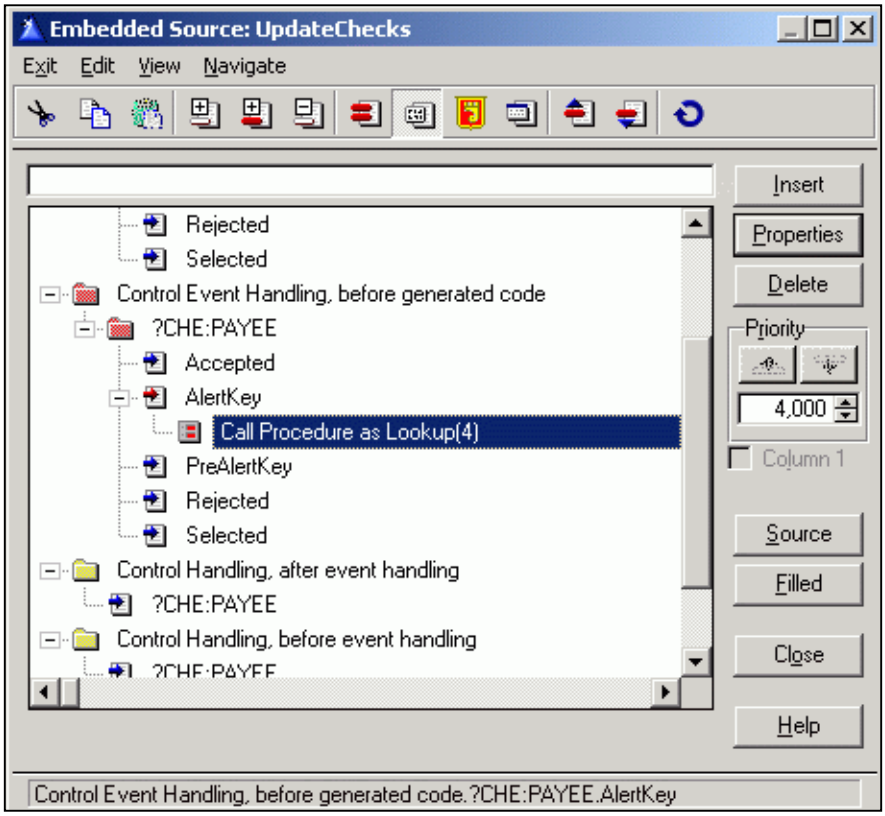

Figure 6: Hotkey lookup

In the `Code After, Completed` prompt, make the assignment and display the selected value (the `Display` statement is not, strictly, required; it *is* required if you want to see the selected value):

```
PEO:State = STA:State;Display
```

In a Hotkey lookup, an entry field is just a standard entry field. That is, it accepts input, any input, from the user. If the user presses the Hotkey (hence the designation "hotkey lookup"), then and only then is the lookup called. On completion of the lookup, the selected value, if any, is assigned to the target field. The developer, using the embeds in the CallProcedureAsLookup template, determines exactly what happens on each possible event, which is exactly what is needed.

In fact, if the user begins the entry and the lookup browse has a locator, the closest match will automatically be selected. For example, if I type "so" and press the hotkey, "SoftVelocity" will be the highlighted record in my payee lookup. All I need to do is press the Enter key and the field is complete:

```
CHK:Payee = PAY:Payee;Display,Select(?+1)
```

In the code shown, the next field is automatically selected after the field assignment is done.

Summary

Lookups are normally used on fields that must have an entry and

must have an "approved" entry, or so says the conventional wisdom. With an understanding of the way lookup code is generated, you can implement an "empty or in file" scenario in one mouse click. With the CallProcedureAsLookup Template, you can also easily add "quick complete" to an application.

What does the conventional wisdom know, anyway?

Download the source code

Steve Parker started his professional life as a Philosopher but now tries to imitate a Clarion developer. A former SCCA competitor, he has been known to adjust other competitors' right side mirrors - while on the track (but only while accelerating). Steve has been writing on Clarion since 1993.

Reader Comments

Add a comment

Clarion MAGAZINE

Search
Go

Home

COL Archives

Information
Log In
Membership/ Subscriptions
FAQ
Privacy Policy
Contact Us

Downloads
PDFs
Freebies
Open Source

Site Index

Call for Articles

Reader Comments

Implementing Read-Only Checkboxes

by Jeff Slarve

Published 2001-07-05

Many Clarion developers have discovered that the READONLY attribute is not available for checkbox controls. Although you can use the DISABLE attribute, it is often desirable to use the READONLY attribute for the sake of consistency in the user interface. For example, a DISABLED control cannot retain focus, and the tooltip does not work for disabled controls (in Clarion). Plus, sometimes a disabled checkbox simply doesn't look too good.

Since I haven't yet seen a truly elegant way to set a checkbox to READONLY, and although I am sure that one exists out there somewhere, I constructed a ReadOnlyCheck class out of rubber bands and duct tape. This class keeps track of the value that you (the developer) say that a checkbox USE variable should be, and reassigns the value if the checkbox is clicked or accepted in some way.

Although there is occasionally a slight flicker when the checkbox is checked, emphasizing the futility that caused me to write this class in the first place, it seems to work okay, and I haven't been able to trick it into allowing the user to change the value yet.

How it works

The ReadOnlyCheck class stores the field equate (FEQ) of each READONLY checkbox in a queue along with the value that each checkbox is supposed to retain:

```
FEQQ            Queue,Type
FEQ               Long
Value             ANY
                End
```

The "Value" field is an ANY because a checkbox can have different data types represented in the TRUEVALUE and FALSEVALUE properties. I guess it would have been pretty safe to use a STRING here, but I opted for an ANY.

You set a checkbox to READONLYwith the AddItem() method, which adds the checkbox's FEQ and value to the queue. Conversely, if you wish to remove the read-only behavior from the checkbox, then you use the RemoveItem() method.

You need to place the ReadOnlyCheck.TakeEvent() method inside the ACCEPT loop (this is a good candidate for an extension template) where it traps Event:Accepted for any of the controls that exist in the queue. When a protected checkbox gets an Event:Accepted, the class slaps the intended value back into the control.

If you need to programmatically change the value of a control that is set to READONLY with this class, then you should use the SetValue() method instead of changing the value of the USE variable itself. If you don't do this the class won't know your intentions, and will change the value back to the way it originally was.

Other than that, the class is relatively low maintenance and seems to serve its intended need.

Download the source

Jeff Slarve is an independent software developer and the creator of the critically-acclaimed In Back automated file safeguard utility. Jeff has been a Clarion developer since 1991, and is a member of the group formerly known as Team TopSpeed.

Reader Comments

Add a comment

Jeff, I've not done this recently, but I was able to set...

Clarion MAGAZINE

Search
Go

Home

COL Archives

Information
Log In
Membership/ Subscriptions
FAQ
Privacy Policy
Contact Us

Downloads
PDFs
Freebies
Open Source

Site Index

Call for Articles

Reader Comments

Using MATCH In Filters and Regular Expressions

by Carl Barnes

Published 2001-07-03

Filters for reports and browses seem to get increasingly complicated over time. In this article I will show a trick you can do with MATCH() that will let you create more powerful filters, and which do not require any more code than a simple INSTRING() but can search for multiple substrings. Steven "Mr. Filter" Parker should love this.

I often find I want to do multiple string searches at once like this one, which uses a number of INSTRINGs and OR conditions:

```
( INSTRING(stateList, State1, 1) |
OR INSTRING(stateList, State2, 1) |
OR INSTRING(stateList, State3, 1) |
OR INSTRING(stateList, State4, 1) )
```

You can replace the INSTRINGs and ORs with a single MATCH statement, which I'll explain in more detail in a moment:

```
MATCH(statelist, State1 &'|'& State2 |
&'|'& State3 &'|'& State4, Match:Regular)
```

Using MATCH in a report filter

It started out as a simple task: I needed to look for a single value in a comma delimited list of values. INSTRING was the obvious choice. For example, in a Dealer file is a field with a list of States in which the Dealer does business:

```
Dlr:BizStates = 'IN,OH,KY,TN,PA'
```

A report requested the State Code to print, stored it in Loc:State2Print and had a Filter of INSTRING(Loc:State2Print,Dlr:BizStates,3).

But as you can image it rapidly became necessary to run the report for multiple states. A simple filter with a single INSTRING() would no longer work. It would have been pretty easy to write

code into the ValidateRecord method and LOOP though multiple INSTRING()s, but I really wanted to use a Filter and not one with twenty INSTRINGs strung together with OR.

My solution was to use the MATCH() statement (new in C5) and a regular expression. In a regular expression, or *regex*, the pipe is the alternate OR operator. So I just need to list the States to be printed in a string with a pipe character between each one as shown:

```
Loc:States2Print = 'NJ|NY|PA|DE'
```

Then on the Report I changed the filter from

```
INSTRING(Loc:State2Print,Dlr:BizStates,3)
```

to

```
MATCH(Dlr:BizStates,Loc:States2Print,|
  Match:Regular+Match:NoCase)
```

and in a single statement I had the equivalent of a many-to-many INSTRING() type search. Notice that the MATCH(string,substring(regex)) statement uses the reverse syntax of INSTRING(substring,string); the string to be searched is the first parameter in MATCH(), the substring or regular expression is second. This potential "gotcha" is noted clearly in the 5.5 help for MATCH() but was not explained well in the 5.0 help. Another difference is MATCH() only returns true or false and not the position of the found substring. The position of a regular expression is available in the new 5.5 STRPOS function, which I'll discuss a little later on.

A regular expression primer

Regular expressions are a very powerful way to search strings, offer many options, and, at least to me, are a lot of fun. Regular expressions came from Unix utilities like sed, awk and grep. The various implementations are very "flavored" – there isn't just one way to do regex. There are also extensions that offer many more options; Perl and JavaScript have many of these.

The implementation in Clarion is fairly basic and a little different from the Unix "standard." Clarion uses curly braces {} for grouping where Unix uses parenthesis () for grouping. Some flavors do use curly braces {min,max} for variable repeat matching (which Clarion does not support). The Clarion Help and LRM incorrectly say that MATCH uses parenthesis for grouping; trust me, it's curly braces.

On a basic level, you can compare regex to DOS wildcards ? and *. Most everyone knows how to use these with the DOS DIR command, for example:

DIR AB*.TP?"

In DOS, a question mark makes a single wild character. In regex the period is used for this purpose so "TP?" would be "TP.".

The DOS wildcard "*" means "any character" repeated an unlimited number of times or not at all. In regex, the * doesn't mean "any character" all by itself; instead, it indicates that the character preceding the asterisk may be repeated zero or more times. So DOS * would be done in regex with period-asterisk ".*" meaning "any character repeated zero or more times." Regex kicks things up a notch by allowing for three different repeaters as described in the below table:

.	Matches a single character and is required unless ? or * follows
?	The previous character may appear 0 or 1 time, this makes the previous character optional
*	The previous character may appear 0 or more times, this makes the previous character optional and it may appear an unlimited number of times
+	The previous character must appear 1 or more times, this makes the previous character required and it may appear an unlimited number of times

If the period specifies any character, how do you match a period? The problem is that the period, asterisk, pipe and all the special characters are used as "meta characters" that perform a function.

To match a regex meta character you must precede it with a backslash. So the DOS "DIR AB*.TP?" would be written in regex as "DIR AB.*\.TP.":

- AB of course means "match AB"
- .* is any character repeated zero or more times
- \. matches a period
- TP is an exact match, like AB above
- . matches any character

MATCH has 10 meta command characters that allow you to do much more than DOS wild cards. They can be combined and nested to perform complex pattern matching. A few of my favorites described in the table below.

|	Alternation aka OR: Allows specifying alternates, e.g. "4|4th|Four|Fourth" would match any of those 'four' strings.

{}	Grouping: It is very useful to combine a group of characters with a repeat count or alternation, e.g. "{4|Four}{th}?" would also match 4, 4th, four, fourth.
[]	Character Set: Lists specific characters to match, e.g. "[0123456789]" would match only digits. A dash "-" in a character set specifies a range, e.g. "[0-9A-Fa-f]" would match a hexadecimal digit "0" through "F". The reason both upper and lower case ranges are given is character sets are always case specific, the Match:NoCase switch does not effect them. Use UPPER() on both strings to work around this. A caret "^" used at the start of character sets specifies the characters are not to be matched e.g. "[^0-9]" would match any character that is not a digit. A character must be present unless you put an * or ? repeat count.
^	Beginning of Line: When placed at the beginning of the line indicates that the match must begin at character one. E.g. to match a CLW file name that begins with "AB" the regex "^AB.*\.CLW" would require the "AB" to be in position one.
$	End of Line: When placed at the end of the line indicates that the match must occur at the end of the line. E.g. "[0-9] [0-9] [0-9]\.CLW$" would match generated Clarion would files that end with a three digit number and the CLW extension.

If you want to learn more about regex the first place to start is the Clarion help on the MATCH() statement. O'Reilly has a book titled "Mastering Regular Expressions" (ISBN: 1565922573) that is very good; it covers all the basics and deals extensively with the various flavors and extensions. If you search the web for "Regular Expressions" you will get many hits that are just as good as the book; some of these are listed at the end of this article. Remember that Clarion will not support many of the regex extensions.

Using a MATCH explorer

If you're going to use `MATCH()` in your code I would strongly suggest building yourself an Explorer program to let you test your expressions to verify that they are working as expected. It's very easy to miss one important character. This will also save you a lot of time compiling and testing. I've included a program I call Match Explorer with this article. It allows testing all forms of `MATCH`: Simple, Wild, Regular and Soundex. It also will format the Clarion `MATCH()` code for you to copy to the clipboard and paste into your App. Figure 1 shows the `MATCH` Explorer..

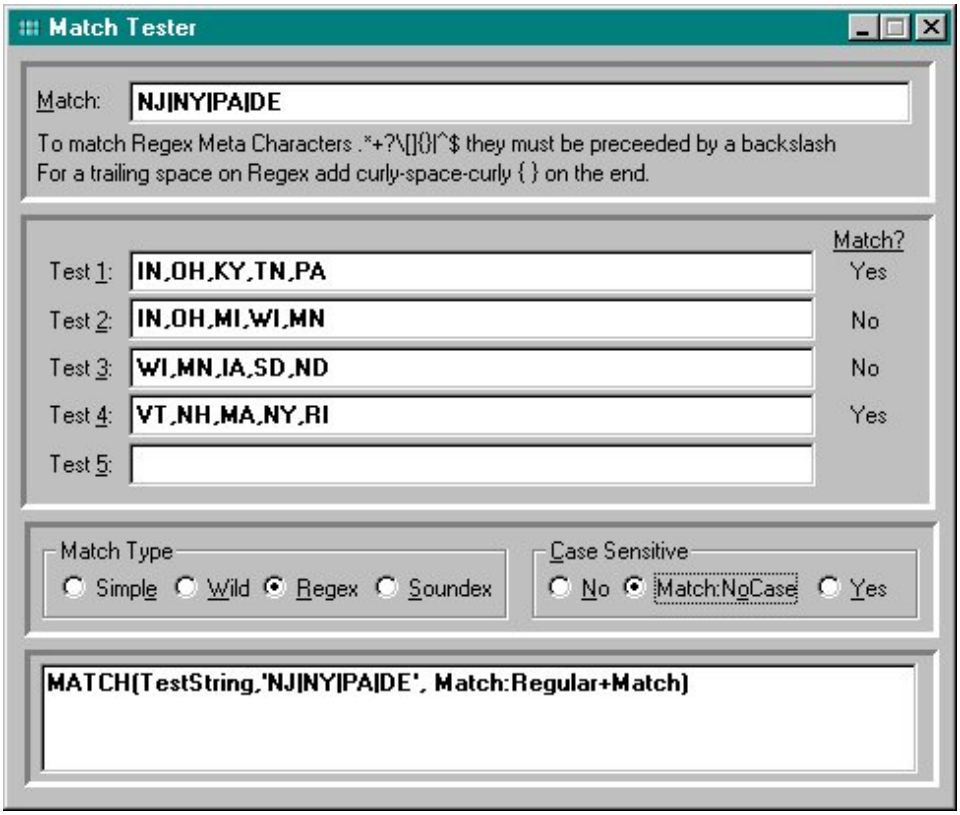

Figure 1. Using the MATCH Explorer to test four different expressions

For a more help with MATCH regular expressions you can download Clarion Source Search from my website. It has a Regular Expression Assistant which can be used free even in an unregistered copy. The Regular Expression Assistant has a parser that explains your regex in words, a tester like Match Explorer, the Clarion regex syntax and a few more bells. Figure 2 shows the expression parser, and Figure 3 shows the syntax help.

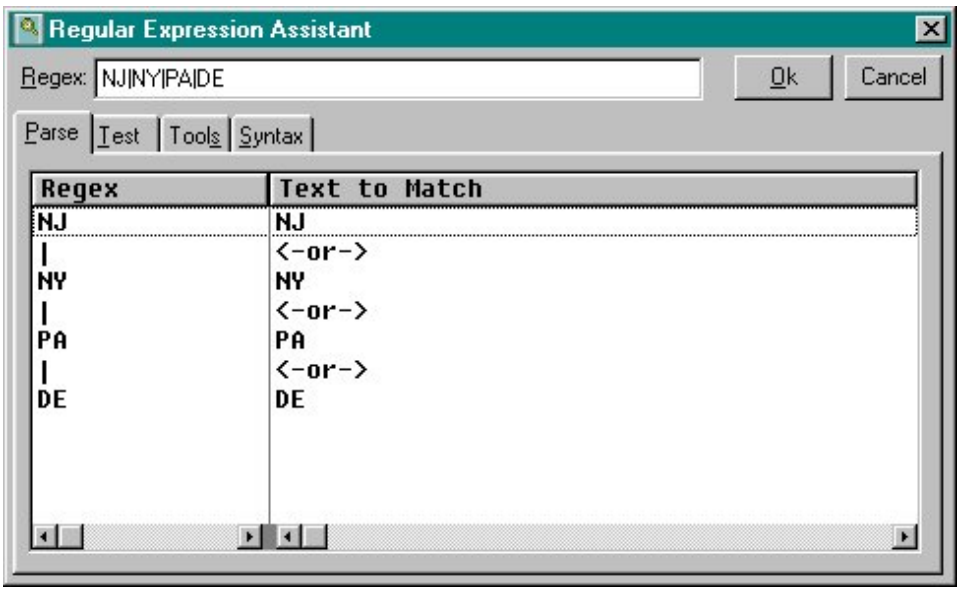

Figure 2. The Regular Expression Assistant parsing a regex

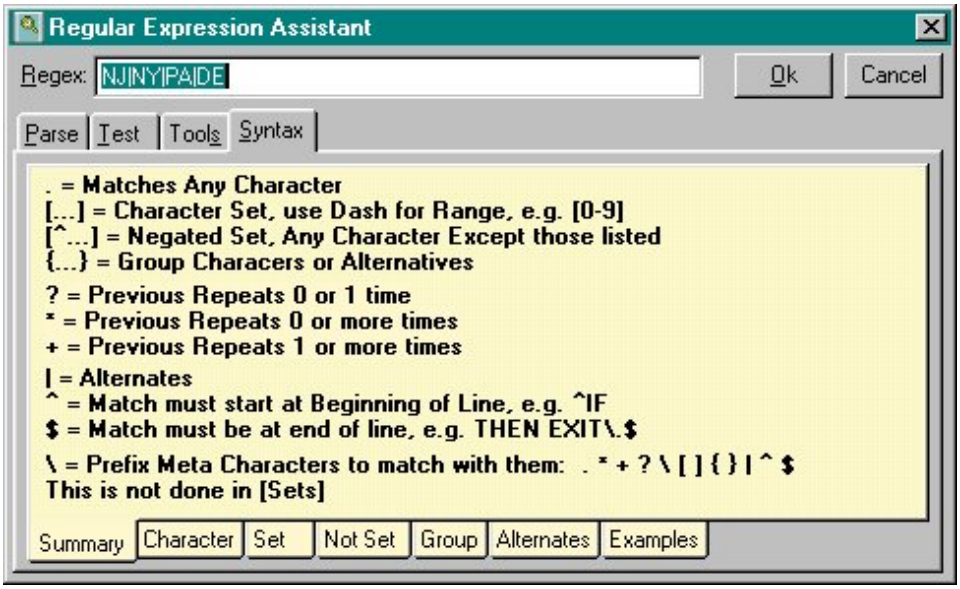

Figure 3. Syntax help in the Regular Expression Assistant

STRPOS()

Clarion 5.5a added a new STRPOS() function that is like MATCH() but returns the position in the string like INSTRING(), rather than just true or false. This function always does a regular expression match and does not allow for simple, DOS wildcard or Soundex matching. This will be very handy for searching large strings and especially raw HTML or RTF code.The syntax is similar to MATCH() and the opposite of INSTRING():

```
Position=STRPOS(string, substring(regex), NoCase)
```

Specify True or "1" for the last parameter for a case insensitive search. If this omitted the comparison will be case sensitive. You may also UPPER() the strings. The prototype from BUILTINS.CLW is shown below:

```
STRPOS(STRING s,STRING p,BYTE
nocase=FALSE),LONG,NAME('Cla$REGULAR')
```

As with MATCH(), I would suggest you use an Explorer program to test your expressions and be certain they are working as you expected. I've also included a STRPOS explorer application in the download at the end of this article. Figure 4 shows the STRPOS Explorer in action.

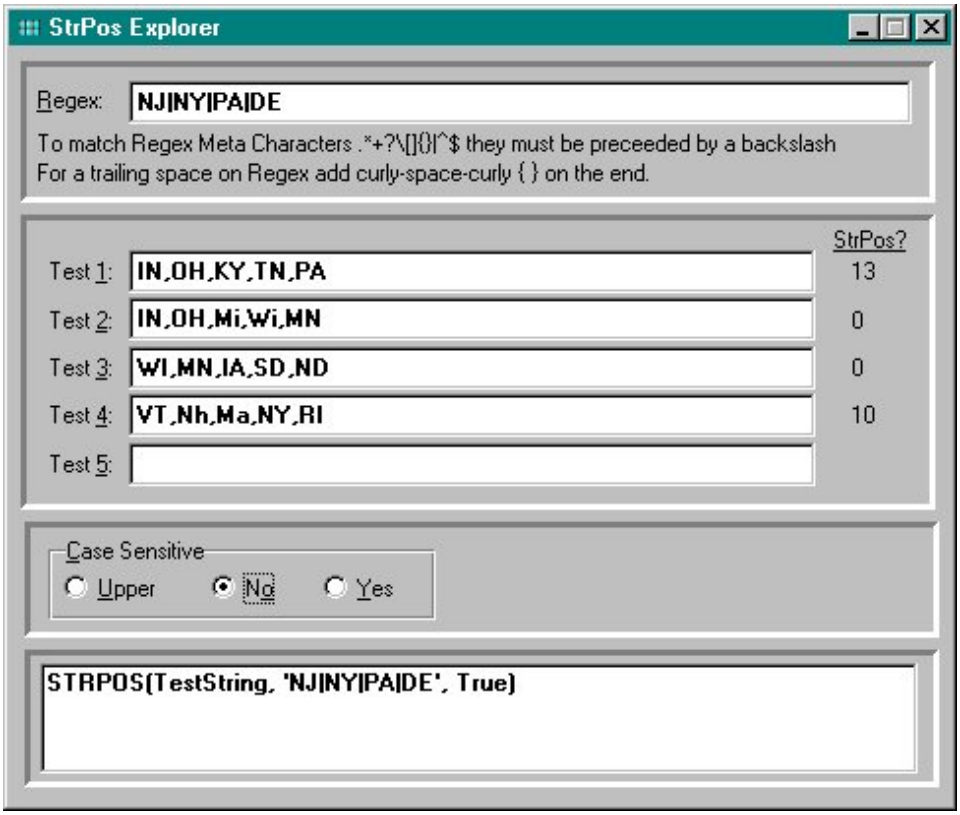

Figure 4. The STRPOS Explorer

Summary

Many times developers use multiple INSTRINGs and logical operators for filtering that result in some long and complex filters. By using the MATCH example given in this article you may be able make your filters smaller and smarter when searching for multiple substrings. Any time you find yourself "OR-ing" together several INSTRINGs think about trying a regular expressions. If you learn a bit more about regular expressions you can do some pretty fancy pattern matching. If you have further questions about regular expressions please email them to me or post them at the end of this article, and I will use them in a follow-up article.

Download the source

Web Resources for Regular Expressions:

- Pattern Matching and Regular Expressions: http://www.webreference.com/js/column5/
- Java Regular Expression Resources http://www.meurrens.org/ip-Links/java/regex/index.html
- GNU Regex Manual http://www.meurrens.org/ip-Links/java/regex/gnu.c.library/regex_toc.html
- Learning to Use Regular Expressions by Example http://www.phpbuilder.com/columns/dario19990616.php3

Carl Barnes is an independent consultant working in the Chicago area. He has been using Clarion since 1990, is a member of Team TopSpeed and a TopSpeed Certified Support Professional. He is the author of the Clarion utilities CW Assistant and Clarion Source Search.

Reader Comments

Add a comment

Carl, First, thanks for the article. Very...
For a report with very specific requirements I normally...

Clarion MAGAZINE

Clarion News

GREGPlus Sale Ends July 31

Gitano Software has put GREGPlus on special until July 31, 2001. All new orders will receive $50 off per product, and all upgrade orders will receive $25 off per product. To obtain your discount you *must*order via the link above. If you buy it from any other source Gitano Software will not be able to deduct the amount. The discount will be deducted from your total before your card is charged.

Posted Monday, July 30, 2001

Service Interruptions At SoftVelocity Web Site

SoftVelocity's ISP is doing some (non-elective) maintenance on its data servers, which means that ASP pages will be intermittently affected. This impacts User Profiles and the Bugs Reporting system.

Posted Thursday, July 26, 2001

PDF-XChange Now Complete

The final set of drivers (NT) are now complete and available to owners of PDF-XChange. The class and template have now been updated to include: simpler Init() and Kill() methods including printer selection; CPCS support now - Larry Teames will make PDF Mailer PDF-XChange Compatible shortly; and updated help. All existing drivers have also upgraded and C5 support has been improved and is now complete. Some simple VB examples are included for those that use VB. There are just a few days left to purchase at the Promo price if you missed the Beta pricing. PDF-

XChange is available at $499 including the API, and the direct output drivers for all Windows operating systems are included, at around a third the price of comparable products.
Posted Thursday, July 26, 2001

SealSoft's xFunction 1.1 (Free)

SealSoft has released version 1.1 of the free xFunction toolkit. New in this version: xGetMemorySize(); xSetEngMonth(); xGetDrivers(). Installs for Clarion 5 and 5.5 are available, as is a demo.
Posted Thursday, July 26, 2001

Clarion Handy Tools O6B-3 Available

Handy Tools build O6B-3 is now available at the subscriber download site. This is an update build to O6B and furthers the enhancements added in O6B-1 and 2. One of the improvements made in this minor build is greater compatibility with other 3rd party products.
Posted Tuesday, July 24, 2001

---

xAppWall Manager v1.0

SealSoft has released xAppWall Manager, a library with extension template for easy management of frame background images. Multiple images can be in fixed order, or random.
Posted Tuesday, July 24, 2001

Win9x/ME Versions Of PDF-XChange Released

Tracker Software Products has added Win9x/Me versions of the PDF-XChange driver, in both end user and Developer SDK editions. The NT version will be available during the week of 30th July; that version will complete the set for all current Windows 32 bit operating systems. The royalty-free developer SDK/API is available for $499 until July 31st - the full retail price of $699 will apply as of August 1st, 2001. The Clarion Class provided free will be extended and released once the NT drivers are complete.
Posted Tuesday, July 24, 2001

CPCS Support Delay

CPCS support will be unavailable from 7/21/01 - 7/29/01. Larry says it's time for a little R&R.

Posted Saturday, July 21, 2001

Clarion to EXCEL Converter Released

Sterling Data's IMPEX now supports export to Excel XLS files, using ExcelBond, an IMPEX add-on written in C by Alexander Ageev. Cost is $79.

Posted Thursday, July 19, 2001

PDF-XChange Beta Pricing Ends Friday, July 20

Tracker Software is about to release the Win9x/Me drivers for PDF-XChange, a royalty-free PDF creation SDK including drivers and an API. PDF-XChange also comes complete with a Clarion-specific template and class set. PDF-XChange pricing is currently at $399, but after Friday, July 20, 2001, the price goes up to $499 until gold release on August 1, when the full retail price of $699.00 goes into effect. Developers will receive the Win9x/Me/2000 drivers (when ready) with the NT drivers to follow (inclusive in price) as soon as they are complete towards the end of the month.

Posted Thursday, July 19, 2001

SysPack Special

Until August 19, 2001, solid.software is offering the SysPack bundle for $199. SysPack contains: SysAni, an animation player for Clarion; SysTrack, a trackbar (aka slider) control; SysHotKey, lets your users specify key combinations; SysList, a list view control with the large icon, small icon, list and detail views; and SysProgress, a progress control. All of these are wrapper classes for the common controls of the WIN32 API, and are written entirely in Clarion with accompanying templates. Documentation and example applications are also included. E-mail support and updates are free.

Posted Thursday, July 19, 2001

Paragon Office Summer Schedule

Paragon Design & Development offices will be closed from 5:00

PM MST (GMT-0700) Tuesday, July 17, until 8:00 AM MST, Tuesday, July 24, 2001. No technical support will be provided during this period. Product sales via the paragondandd.com Web server will be handled normally during this period. See separate notice today of a brief scheduled server outage, unrelated to the office closure.

Posted Wednesday, July 18, 2001

Paragon Server Availability

The Paragon Design & Development server will be unavailable for approximately 4 to 6 hours on Thursday, July 19, due to the planned move of the datacenter housing the server. The down time will begin at approximately 9AM EDT (GMT-0400) on July 19.

Posted Wednesday, July 18, 2001

New Application Prototyper

Riebens Systems has released a buy-in beta of the Application Prototyper, which allows designers/developers to manage the application development project and create project documentation based on the Software Engineering Laboratory (SEL) standards from NASA. Application Prototyper currently allows developers to prototype the software product screens according to the customer requirements. Integration with the business rules application as well as business process prototyping is expected soon. A free distributable remote screen previewer will be available shortly. The price during the beta program is $99, with final pricing expected to be between $199 and $299. In final release Application Prototyper will allow a designer/developer to design business processes associated with a software development product, design a screen map (much like a website map), prototype the application screens and capture the data elements associated with an application screen.

Posted Tuesday, July 17, 2001

SealSoft Releases xPictureBrowse v1.0

SealSoft's xPictureBrowse is a class with control template that makes it easy to preview and select a graphic file. You can use xPictureBrowse to assign pictures to wallpaper or image controls. Demo available.

Posted Monday, July 16, 2001

Clarion MAGAZINE

Search
Go

Home

COL Archives

Information
Log In
Membership/ Subscriptions
FAQ
Privacy Policy
Contact Us

Downloads
PDFs
Freebies
Open Source

Site Index

Call for Articles

Reader Comments

When Clarion COM Will *Not* Do

by Jim Kane

Published 2001-07-24

I thought it might be fun to review some COM fundamentals and then show how you can take matters into your own hands and extend Clarion's COM abilities for those times where Clarion's native COM will not do what you want, such as when you need to use safe arrays. The reason I say it *might be* fun is because I like COM this week – in other words all my COM projects have been going well, so I like it! Don't ask next week, my luck never lasts that long.

COM seems difficult for beginners because it has a unique vocabulary. To quickly review and translate COM to Clarionese, here are some very loose but pragmatic definitions:

COM	Clarion
`Object`	Class
`CoClass`	Class
`Interface`	A group of procedures. Each `CoClass` contains one or more Interfaces. Every interface is a member of a `CoClass`.
`GUID`	Globally unique identifier – a 128 byte number that is said to be globally unique.
`CLSID`	A `GUID` that uniquely identifies a `CoClass`.
`IID`	An interface `ID` or `GUID` that uniquely identifies a `COM` interface.
`ProgID`	A human-friendly string that can substitute for a `CLSID`. There are API functions to translate a `CLSID` to `ProgID` or `ProgID` to CLSID. Both are stored in the registry once a COM object is installed on a computer
`Computer`	A thing that you have a love hate relationship with.

Method	A procedure/function contained in an interface.
Put	A procedure that stores a value inside a COM object.
Get	A procedure that retrieves a value from a COM object.
Iunknown	An interface that starts with 3 particular methods: QueryInterface, AddRef, and Release. Every single COM interface contains the Iunknown methods as its first three methods plus other additional custom methods of the interface designer's choosing
Idispatch	An interface that includes all the Iunknown methods and adds four more methods, the most significant of which is Invoke. Idispatch may or may not contain additional methods.
DispInterface	An interface one that includes all the methods of Idispatch and nothing more.
Dual interface	An interface that includes all the Idispatch methods plus other custom methods. A caller can either call the custom methods directly or call Idispatch.Invoke which in-turn calls the custom method
Vtable	Thank you for asking – this is really an assembler (my eyes glow with delight) concept, but alas it's nothing magical, just a list of addresses for every method in an interface.
Early binding	This is what happens when you call an Iunknown interface or dual interface, which is done using the Clarion Interface keyword. Early binding does not use the Idispatch methods.
Late binding	This means you intend to call Idispatch.Invoke() on an object, which is what the clarion OLE control does. Late binding is slower than early binding.

SafeArray	A safe array is a normal array that has various API functions available so if different threads try to access the array at one time, only one can access the array at a time. It's an array with thread safety, and the API is pretty straightforward. Safe arrays can be implemented in Clarion; in Visual Basic all arrays are safe arrays.
Variant	A variant is a data type much like a Clarion ANY variable in that it can have a lot of different data types. It is prototyped as a group of 16 bytes. The first two bytes of the group contains an equate for the type of data the variant represents. There are 20+ possible types like byte, ushort, short, long, ulong, decimal (not like Clarion's decimal type), SafeArrays of any other type, etc. Another long field contains the value of the data. The rest of the variant group is usually blank.

I would suggest you print out that list and memorize it. Then tell your kids you have a test on it tomorrow and have them quiz you on it. It will delight the wee folks if you get one wrong, but try not to – you parents have a reputation to uphold.

Calling a COM object is really no different than calling any API function. To call a procedure in a DLL you need a LIB or need to load the library with a call to the loadlibrary() API function. To call a COM interface you call CoInitialize() to tell COM you're coming, and then you call CoCreateInstance(), specifying the CLSID and IID for the interface you want. CoCreateInstance returns the address of the interface.

In Clarion COM, when you create an OLE control CoInitialize() is called for you. In the create() statement for the OLE control on the window, or in Prop:Create, you supply the ProgID. Clarion converts the ProgID to a CLSID and calls CoCreateInstance() Either way the COM object is created and the address of the interface of interest is obtained.

If you called CoIntialize() and CoCreateInstance yourself, you now have the address of the interface containing the method you want to call. If you instead used the Clarion OLE control, that control stores the address of the interface internally. You can get the address using PROP:Object. Take for example this simple code that creates an ADO command object:

```
Program
Map
End
Window     WINDOW('ado test'),AT(,,361,184),|
               FONT('MS Sans Serif',8,,FONT:regular),|
```

```
                SYSTEM,GRAY,DOUBLE
              OLE,AT(37,13,70,90),USE(?CmdObj)
              END
            END
Code
     open(window)
     ?cmdobj{prop:Create}='ADODB.Connection'
     ?cmdobj{'OPEN("DATA SOURCE=SERVERNAME;'|
       & 'PROVIDER=SQLOLEDB.1;Initial Catalog=pubs' |
       & ';UID=SA;PWD=",,,)'}
     !cObj is a CSTRING(20)
     cObj=?cmdobj{prop:object}
     if cobj[1]<>'`'
        message('could not connect')
        close(window)
        return
     end
     Message('Address of the ADO Connection Interface: ' |
        & cObj[2 : len(Clip(cObj))])
```

If you wish to run this code, you'll need to edit the connection string contained in the `open` method to reflect your server name, userid, and password. Clarion COM stores addresses of objects in CStrings where the first character is an apostrophe. The `message` statement passes over the first character which is an apostrophe and displays the address of the command object's one and only interface.

Once you have the address of an interface, you can prototype the interface and call any of its methods you want. What you know for sure about this interface is it is an `Idispatch` interface (if it wasn't it couldn't be called from VBScript in web pages). So the following prototype is a subset of the connection interface.

```
IUNKNOWNTYPE        INTERFACE,COM,type
QueryInterface      PROCEDURE (long  iid_Requested, |
                      *LONG lpInterface),HRESULT
AddRef              PROCEDURE (),Long,PROC
Release             PROCEDURE (),Long,PROC
                    END
IDISPATCHTYPE       INTERFACE(IUNKNOWNTYPE),COM,Type
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
```

By calling `Idispatch.Invoke()` with different `dispidMember` constants you can call *any* non-dispatch method in the command object such as `Open` or `OpenSchema`. `Invoke` is a central entry point to all the methods the command object has to offer. Here's the relevant data and code:

Data:

```
Idispatch &Idispatchtype
```

Code

```
Idispatch&=(cObj[2 : len(clip(cobj))])
```

With that done you can now call any method in the interface using `Idispatch.Invoke()`. That puts you in the driver seat Now it's just COM vs. you the programmer. Unfortunately `Idispatch.Invoke` has a lot of parameters, but many of them are simple constants. The prototype is this:

```
 Invoke    Procedure(long dispidmember, |
                 long riid, |
                 unsigned lcid, |
                 ushort flags, |
                 long pdispparams, |
                 long pvarresult, |
                 long pexceptinfo, |
                 *unsigned puArgErr),
                 hresult
```

Fortunately most of the parameters have a constant value and can be largely ignored, while others you just look up. The only difficult parameter is `pDispparams`, which is a group that describes an array of the parameters you are passing in. I'll show how to handle it shortly.

What prompted this article was a request on the newsgroups to call the `OpenSchema` method of the ADO `Command` object. Unfortunately one of the required parameters is a SafeArray, and Clarion doesn't natively support safe arrays. OLEView shows the definition of the OpenSchema method as follows:

```
[id(0x00000013)]
HRESULT OpenSchema(
                [in] SchemaEnum Schema,
                [in, optional] VARIANT Restrictions,
                [in, optional] VARIANT SchemaID,
                [out, retval] _Recordset** pprset);
```

Note the first line containing an ID of 13H. This is the `DispID`. Every

method in a Idispatch interface has a long constant that is essentially an equate for the method name. In this case the OpenSchema method has a DispID of 13. That takes care of the first parameter of Invoke – it is 13H.

The next parameter, called riid, is a dopey API constant. For what ever reason, the address of a constant defined in my StdCom class called IID:null is always passed.

The third parameter is LCID or language ID. I usually pass either 409H, since I only speak English (the Texan variant with a New York accent), or 0 for language neutral. Either way it works. The flags parameter just signals if this is a method or property. Since it is a method just pass Disp_Method or 1. For a property Get use Disp_Get or 2, and for a property Put use Disp_Put or 4. Pexeptinfo and puargErr are involved in error reporting. Since I don't really need these I pass 0 for both.

Although you may think you are on a roll in understanding all these Invoke parameters, the two toughest ones await: pDispParams and pVarResult. Keeping definitions simple, pDispParams is a group that describes an array of input parameters and pVarResult is the one output parameter or return value. In this case the output parameter is a pointer to an ADO RecordSet object. You can identify the output from the OLEView definition above where [out, retval] appears just before the data type RecordSet. What will be returned is the address of a variant group that has a variant type of Idispatch, and the value will be a long or the address of an Idispatch interface. You know this because elsewhere in the type library in OleView the RecordSet type is defined as an Idispatch interface. It also makes sense.

So after the call to Idispatch.Invoke the variant group pointed to by the VarResult parameter will contain a pointer to the Idispatch interface of a record set object. You'll put it back into a form Clarion can use and be able to read the record set using normal Clarion OLE control code.

That leaves the pDispParam parameter of Idispatch.Invoke. It is prototyped like this:

```
dispparms           group
lparrofargs           long
lpdispofargs          long
NumArgs               long
numNamedArgs          long
                    end
```

This is a group that describes the parameters you are passing to the OLE method to be called. The second and fourth fields in the group are for passing named parameters. These are rarely of use so I will not cover them. Set those two items to zero. NumArgs is just the number of input parameters in the method being called. In this case there are three input parameters and one 'retval' so NumArgs is set

to 3. DispParams is the address of an array of variant groups where the first group in the array is the last input parameter. pDispParam can be prototype in Clarion like this:

```
!variant structure
VariantType       GROUP,TYPE
VT                  USHORT
wReserved1          USHORT
wReserved2          USHORT
wReserved3          USHORT
Value               LONG
extrapad            ULONG
                  END

!group of variants - used to pass 3 parameters
varparm           group
var1                like(varianttype)  !schema id not used
var2                like(Varianttype)  !safearray
var3                like(varianttype)  !adschematables=20
                  end
```

Setting up the variants is pretty straight forward. Before using the variant group, call the API function VariantInit() to clear the structure. Then set the VT element to specify the type of data the variant will represent. There is an API constant for every possible type of data. To have the variant represent a long, set the VT field to VT_I4 (which has a value of 3). After the VT field is filled, put the value of the parameter in the Value field. For the three parameters for the OpenSchema method set up the final code this way:

```
variantINIT(address(varparm.var1))
variantINIT(address(varparm.var2))
variantINIT(address(varparm.var3))

!create variants for the parameters -
! first on the line is put at the end of the array
! in other words the array of variants is
! packed backwards

!first parameter is an ADO constant = 20
varparm.var3.vt=vt_I4
varparm.var3.value=20

!second parameter is safe array
varparm.var2.vt=vt_array+vt_variant
varparm.var2.value=SArrayCl.psa

!3rd parameter is schema id and
!will always be empty
varparm.var1.vt=vt_empty
varparm.var1.value=0
```

The second parameter is an array of variants packaged into a safe array. The array is one dimensional and the first dimension has four items in it – three empty variants and a string. To set up the safe array first I load a Clarion array with the maximum number of array

elements in each dimension:

Data:

```
ArrIndex long,dim(1)  !1 dimension safearray
```

Code

```
ArrIndex[1]=4
```

This tells my safe array class that the safe array will have one dimension and the one dimension will have four elements. The other items passed to the safe array class init method are the data type and size of each element in the safe array. In this case I'm using vt_variant and the size of the variantType structure:

```
if SArrayCl.Init(vt_Variant,size(variantType),|
   arrIndex[]) then
  message(SArrayCl.ErrorStr,'init')
  safearrayerror=true
end
```

For each element in the array (four of them) call the PutVariant method with the value of the element to add it to the safe array. Use the same array, arrIndex[] to tell the class what the index of the safe array element you are filling

```
!idx 1 is vt_empty
arrindex[1]=1
if ~safearrayerror and |
  SArrayCl.Putvariant(arrIndex, vt_empty,0) then
    message(SArrayCl.ErrorStr,'put')
    safearrayerror=true
end
!idx 2 is vt_empty
arrindex[1]+=1
if ~safearrayerror and |
  SArrayCl.Putvariant(arrIndex, vt_empty,0) then
    message(SArrayCl.ErrorStr,'put')
    safearrayerror=true
end
!idx 3 is vt_empty
arrindex[1]+=1
if  ~safearrayerror and |
  SArrayCl.Putvariant(arrIndex, vt_empty,0) then
    message(SArrayCl.ErrorStr,'put')
    safearrayerror=true
end
```

For the last item, which is a BString, I set cwstr='VIEW' and then call my string class (strcl) to convert the Clarion string to a BString in one step.

```
!idx 4 is a bstring pointer
arrindex[1]+=1
```

```
!convert to bstring
strcl.cwtoBstrAlloc(cwstr, lpbstr)
!Copy the bstring into the safe array array
if  ~safearrayerror and |
  SArrayCl.Putvariant(arrIndex, vt_bstr,lpbstr) then
    message(SArrayCl.ErrorStr,'put')
    safearrayerror=true
end
```

Now that all the parameters are packed into variants you are ready to call the `invoke` method then free the safe array:

```
!013H is the dispId for open schema from oleview
!iid:null is dopey constant
!409H is English language
!1=dispatch_method since this is a method
!dispparams which you just made
!var is where the result goes
!0,0 is used for error info -
!I'm just too lazy to use it.
hr#=Idisp.invoke(013H,address(iid:null),|
  409H,1,Address(dispparms),address(var),0,0)
!free the safe array and the bstring it contains
if saraycl.kill() then message('safe array kill error').
```

Note in the original type library description cut and pasted from OleView the second parameter, `Restrictions`, which is a safe array, is marked `[In, optional]`. The important part is the `[In]`. This means the method caller allocates the memory for the parameter, and may (must) dispose of the memory any time after the method call returns. If the COM object wants to it can make a copy of the data, but it can not rely on the data continuing to exist after the method returns. You take care of cleaning up the safe array memory by calling the `saraycl kill` method.

At this point if `hr#>=0` there is no error, and the result `or [out,retval] RecordSet` object you want is returned in the `var` group passed to `Invoke`. You can expect the `type` or `VT` field of the variant `var` to be `vt_Dispatch` or an `Idispatch` pointer to a `RecordSet` object. The code to test the result and construct a `CString` containing the `Idispatch` pointer that the Clarion OLE control can understand is shown below, where cRS is a `CString(20)`. Clarion expects an `Idispatch` pointer to be in a `CString` with a ` as the first character. For the record `[Out]` parameters have memory allocated for them by the COM object, and the caller of the method needs to dispose of the memory allocated, if any, for the `[Out]` parameter. In this case the out parameter is just a long and did not require any memory allocation so you have nothing to clean up. Had the `[Out]` parameter been a `BString`, you would have to free the memory allocated by the COM object for the `BString`.

```
!test the hresult. 0 or greater
! is a good result
!also test the output variable is
```

```
! an IDispatch pointer and not blank
if hr#<0 or var.vt<>vt_dispatch or ~var.value then
  message(hr#,'OpenSchema failed')
else
  !the var variant contains a pointer
  ! to the record set
  !now you make the pointer into the
  ! format Clarion understands
  cRS='``'&var.value
end
```

Now you can use normal Clarion code to read the record set. For example to determine the number of fields in each record in the record set:

```
 Loop  !For each record
   if ?cmdobj{crs & '.EOF'}<>0 then break.
   cFields = ?cmdobj{crs & '.FIELDS'}
   count=?cmdobj{cFields & '.Count'}
   Message('This record contains ' & count & ' fields.')
End
```

The complete sample code available for download shows how to read all the schema information returned. To read the value of the schema information, call `Invoke()`again to read the value property of the field object. This provides another example similar to the one above of calling invoke. By calling `Invoke` directly you can better handle some of the returned values which are `NULL`. Clarion OLE doesn't handle `VT_NULL` very well but the code that calls `invoke()` to get the value does.

So there you have it. Once you understand how `Idispatch` works, you can call its `invoke` method and have complete control over the input and output parameters. With that kind of control you an easily supplement the bits of pieces of COM Clarion doesn't handle too well, then go back to letting Clarion do most of the work and continue on. It's nice to know you can have full control when you want to. I guess I'm just a control freak; at least my teen age children think so!

Download the source

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