Reborn Free CLARION online

published by CoveComm Inc.

Clarion MAGAZINE

Search

Go

Home

COL Archives

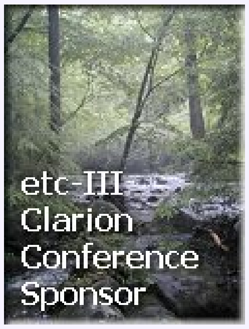
Information
Log In
Membership/
Subscriptions
FAQ
Privacy Policy
Contact Us

Downloads
PDFs
Freebies
Open Source

Site Index

Call for Articles

Reader Comments



Creating #AT Statements The Easy Way

Writing templates can be tricky, especially when it comes to creating #AT statements that correspond to ABC virtual method embeds. Here's a template that makes that task easy.

Posted Friday, June 29, 2001

Handling Multiple Update Forms

In a college placement office application, Dr. Parker finds he needs to call different forms based on whether the client school is permitted to enter both off-campus and on-campus jobs or on-campus jobs only. And, just to keep it interesting, the different forms are to be used only when inserting new records.

Posted Friday, June 29, 2001

Windows-Style List Box Sorting Revisited

Since Steffen Rasmussen's article on Windows-style list box sorting was published, some Clarion Magazine readers have responded with solutions to improve the code. In this article Steffen applies some of these solutions, and offers a small list box sorting template.

Posted Wednesday, June 27, 2001

Maintain Velocity Newsletter

The latest SoftVelocity newsletter is out - topics include: Clarion 5.5 E release upcoming, with over 80 fixes and enhancements; the ClarioNet release; ASP template beta; the upcoming XML beta; SQL templates; the ADO data layer; employment opportunities, and more.

Posted Monday, June 25, 2001

Extending ABC's Edit In Place - Part 2

Russ Eggen has heard it all when it comes to Edit In Place (EIP). Well, here's a dirty

ZipFlash 2.2 Released

SealSoft xQuickFilter Template v1.0

solid.software Closed For Holidays

ABC Free Templates Have A New Home

xSmart Macro Version 2.2

Free Zip Code Template

WinSet Lets Users Change Windows Properties

CapeSoft Tip of the Month: Object Writer

CapeSoft Office Messenger Updated

NetTalk v1 Beta 14 Shipping

Secwin Version 3.1 Beta 1 Released

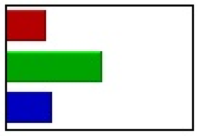
File Explorer Version 1.8b Released

xWord Library Version 1.3.1

PD Translation

SURVEY

How important is it to you to be able to create (i.e. write) COM objects with Clarion?



🟥 Essential
🟩 Important
🟦 Unimportant
Votes: 71

little secret: ABC's EIP features are *fully functional*. Part 2 of 2.
Posted Friday, June 22, 2001

Free ABC/Legacy Embed Utility
Here's a free utility that shows you which ABC embeds correspond to which Legacy embeds. English and French versions available. By Eric Griset and Patrick Corcuff.
Posted Thursday, June 21, 2001

Handcoding Tree Lists Part 2
In Part 2 of this article, James Cooke shows how to handcode around a deficiency in the ABC relational tree control.
Posted Wednesday, June 20, 2001

Tip: How To Start A Browse With The Last-Used QBE Query
Randy Rogers shows how easy it is to have your browse start up with the most-recently used QBE query in effect. (free article)
Posted Tuesday, June 19, 2001

Extending ABC's Edit In Place - Part 1
Russ Eggen has heard it all when it comes to Edit In Place (EIP). Well, here's a dirty little secret: ABC's EIP features are *fully functional*. Part 1 of 2.
Posted Friday, June 15, 2001

ClarioNet Released!
SoftVelocity Debuts New Specialized Thin Client for Clarion Applications
SoftVelocity has released ClarioNet, a specialized thin client for Clarion business applications.
Posted Wednesday, June 13, 2001

Handcoding Tree Lists Part 1
James Cooke considersthe recent shift of the software industry toward tree-rich user interfaces, and decides this might be a good time to examine some of the benefits and key concepts of the tree control, as well as how to make good use of them in Clarion applications.
Posted Wednesday, June 13, 2001

Interview: James Orr On The Public PIM
James M. Orr is the founder and Director

Dictionary

Simshape Templates Special Ends June 21

of Marketing of the OpenDB Alliance, an organization which is promoting the "Public PIM" database design as a proposed industry standard for employing many-to-many relationships and recursive relationships.

Posted Tuesday, June 12, 2001

Using Dynamic Indexes With TPS Files

Dynamic indexes are often overlooked as a way to efficiently access data from TopSpeed (TPS) files, especially if you are dealing with files that hold large numbers of records and a custom sort order and filtered subset is required. By using a dynamic index, you can eliminate the need to create additional file keys.

Posted Friday, June 08, 2001

Understanding Stack And Heap Memory In 32 Bit Clarion Applications

In Clarion, as well as in C and C++ (and unlike Java), you need to be aware of possible memory leaks and thus be aware of the side effects of declaring variables. John Gorter explains how 32 bit Clarion applications use stack and heap memory.

Posted Tuesday, June 05, 2001

Weekly PDF For May 27 - June 3, 2001

All Clarion Magazine articles for May 27 - June 3, 2001.

Posted Monday, June 04, 2001

Reborn Free CLARION online published by CoveComm Inc.

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

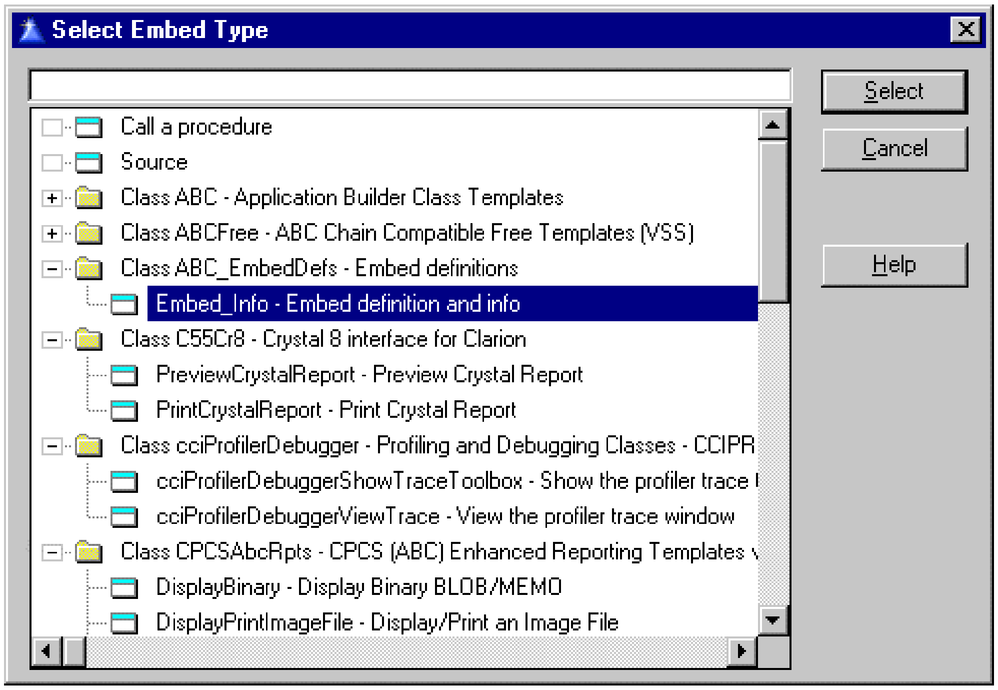
Creating #AT Statements The Easy Way

by Andrew Guidroz II

Published 2001-06-29

Writing templates can be tricky, especially when it comes to creating #AT statements that correspond to ABC virtual method embeds. Here's a template that makes that task easy (available for download at the end of the article). I based this code on something Lee White wrote for Clarion4 (used with permission - thanks, Lee!).

The template is in a TPW, so you'll need to add the statement #INCLUDE('EMBEDS.TPW') to an appropriate TPL file. Once you've done this, open any application, go to embed view, locate the embed you want to generate an #AT statement for, and choose the Embed_Info code template, as shown in Figure 1.



Call a procedure
Source
Class ABC - Application Builder Class Templates
Class ABCFree - ABC Chain Compatible Free Templates (VSS)
Class ABC_EmbedDefs - Embed definitions
Embed_Info - Embed definition and info
Class C55Cr8 - Crystal 8 interface for Clarion
PreviewCrystalReport - Preview Crystal Report
PrintCrystalReport - Print Crystal Report
Class cciProfilerDebugger - Profiling and Debugging Classes - CCIPR
cciProfilerDebuggerShowTraceToolbox - Show the profiler trace
cciProfilerDebuggerViewTrace - View the profiler trace window
Class CPCSAbcRpts - CPCS (ABC) Enhanced Reporting Templates
DisplayBinary - Display Binary BLOB/MEMO
DisplayPrintImageFile - Display/Print an Image File

Figure 1. Selecting the Embed_Info code template.

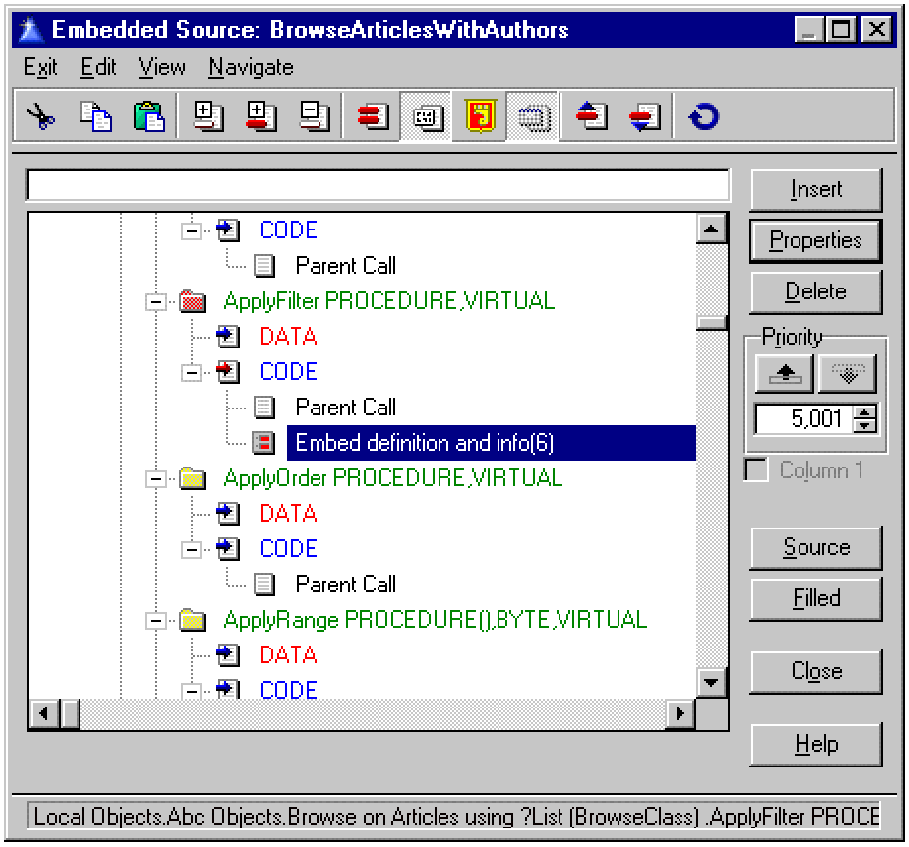

Embedded Source: BrowseArticlesWithAuthors
Exit
Edit
View
Navigate
CODE
Parent Call
ApplyFilter PROCEDURE,VIRTUAL
DATA
CODE
Parent Call
Embed definition and info(6)
ApplyOrder PROCEDURE,VIRTUAL
DATA
CODE
Parent Call
ApplyRange PROCEDURE(),BYTE,VIRTUAL
DATA
CODE
Insert
Properties
Delete
Priority
5,001
Column 1
Source
Filled
Close
Help
Local Objects.Abc Objects.Browse on Articles using ?List (BrowseClass) .ApplyFilter PROCE

Figure 2. The Embed_Info code template in the embed point.

Once you've inserted the template, go to Source View (the easiest way is to click the Source button on the Embeds window). In the example shown in Figures 1 and 2, the template generates the following code (line break added):

```
!>> #AT(%BrowserMethodCodeSection,'1','ApplyFilter',↵
          '()'),PRIORITY( ???? )
!>> !! What follows is debugging info
!>> EmbedID........... = %BrowserMethodCodeSection
!>> EmbedParameters... = 1, ApplyFilter, ()
!>> Description....... = Browser Method Code Section
!>> Process........... = ProcessClass
!>> Object............ = BRW1
!>> TwoParams ........ = 2
!>> ThreeParams....... = 15
```

Just copy and paste the generated #AT statement for use in your own templates. The template source looks like this (line breaks added):

```
#!
#!TEMPLATE(ABC_EmbedDefs,'Embed definitions')↵
    ,FAMILY('ABC')
#!-----------------------------------------
#CODE(Embed_Info,'Embed definition and info')
#!-----------------------------------------
#DECLARE(%TwoParams,LONG)
```

```
#DECLARE(%ThirdParams,LONG)
#DECLARE(%ParamString,STRING)
#IF(%EmbedParameters)
#CLEAR(%TwoParams)
#SET(%TwoParams,INSTRING(',',%EmbedParameters,1,1))
#IF(%TwoParams)
#IF(SUB(%EmbedParameters,(%TwoParams+2),1) <> '(')
#SET(%ThirdParams,INSTRING(',',↵
    %EmbedParameters,1,%TwoParams+2))
#SET(%ParamString,'''' & SUB(%EmbedParameters,1,↵
    (%TwoParams-1)) & ''',''' & SUB(%EmbedParameters,↵
    (%TwoParams+2),((%ThirdParams) - (%TwoParams+2))) ↵
    & ''',''' & SUB(%EmbedParameters,(%ThirdParams+2)↵
    ,LEN(%EmbedParameters)) & '''')
#ELSE
#SET(%ParamString,'''' & SUB(%EmbedParameters,1,↵
    (%TwoParams-1)) & ''',''' & SUB(%EmbedParameters,↵
    (%TwoParams+2),LEN(%EmbedParameters)) & '''')
#END
!>> #AT(%EmbedID,%ParamString),PRIORITY( ???? )
#ELSE
!>> #AT(%EmbedID,'%EmbedParameters'),PRIORITY( ???? )
#ENDIF
#ELSE
!>> #AT(%EmbedID),PRIORITY( ???? )
#ENDIF
!>> !! What follows is debugging info
!>> EmbedID........... = %EmbedID
!>> EmbedParameters... = %EmbedParameters
!>> Description....... = %EmbedDescription
!>> Process........... = %ProcessType
!>> Object............ = %ThisObjectName
!>> TwoParams ........ = %TwoParams
!>> ThreeParams....... = %ThirdParams
```

You can also download the template here.

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

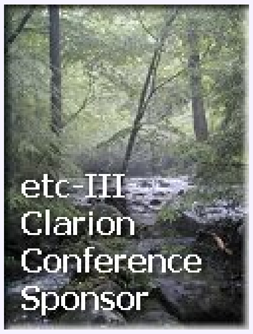
Add a comment

A bad #AT() just never appears in your generated code. That...
C55BinTWriter.EXE is supposed to also provide this feature...
To Carl: I tend to write most templates as some...

Reborn Free CLARION online

published by CoveComm Inc.

Clarion MAGAZINE



Handling Multiple Update Forms

by Steven Parker

Published 2001-06-29

In a college placement office application, I need to call different forms based on whether the client school is permitted to enter both off-campus and on-campus jobs or on-campus jobs only. And, just to keep it interesting, the different forms are to be used only when inserting new records. So, I need three forms. But how to go about this?

It is easy enough to restrict insert or change or delete based on a condition. For instance, after initializing the browse, in the browse's `Init` method, Priority 9300 or later, insert the following code, which disables selected controls based on security levels:

```
Case SecurityLevel
Of 42
  BRWx.InsertControl = 0
  ?Insert{Prop:Disable} = True
Of 87
  BRWx.ChangeControl = 0
  ?Change{Prop:Disable} = 0
Of 3
  SELF.Destruct
End
```

Now suppose I need (or simply want) to call different update forms based on one or more conditions. I envision code that looks something like this:

```
Case ThisWindow.Request
Of InsertRecord ! condition 1
  If CFG:Permit ! condition 2
    OffCampusForm
  Else
    OnCampusForm
  End
Else
  StandardJobForm
End
```

In a checkbook application I wrote years ago, I decided that I wanted three different forms: one for entering a credit (deposit), one for a debit (check) and one for changing an existing record. Not only did I want to prime different values for debits and for credits, I wanted them to look like the forms provided by my bank. In this case, multiple forms were not required, I simply wanted them.

Multiple update forms are a little less straightforward than conditionally restricting access to a form. There are no obvious template prompts or ABC methods to affect this.

How Forms Are Called

First, some basics. Clarion uses the "Request-Response" model. It is important to understand how this is used to accomplish record updates. A global variable called `GlobalRequest` stores the requested action, based on the button or key the user presses. The form reads the value of this variable and "knows" what it is expected to do.

Another global variable, `GlobalResponse`, is set by the form. If the user completes the form, `GlobalResponse` is set to `RequestCompleted`; if the user cancels, `GlobalResponse` is `RequestCancelled`. Thus, the browse "knows" what happened in the form.

`GlobalResponse` can be set in one of three distinct places. The Ok button, obviously, is one of them. The Cancel button is not (well, it *is* obvious, it just doesn't happen to be one of the places where `GlobalResponse` is set). Because the default value of `GlobalResponse` is `RequestCancelled` (early in the `INIT` method, this assignment is made) the Cancel button simply does not re-set `GlobalResponse`.

The third is after a user cancels and is asked whether to save the edits and answer "Yes" (in other words, if "Offer to save changes" is the action selected for "On Aborted Add/Change" in the form's "Messages and Titles"). This is in `TakeCompleted` and, consequently, code embedded in the Ok button will be by-passed if the user clicks "Yes." For this reason developers like Dennis Evans and Jim Katz who really know their ABCs recommend that `TakeCompleted` be used instead of the Ok, Accepted embed.

Both of the request-response variables are threaded, so update forms should not be called with the `Start` statement. If a form is `Start`ed, the value in `GlobalRequest` will be lost (the value in the form's thread does not know the value in the browse's thread).

Similarly, a form expects the record buffer to be empty for an Insert (or empty except for primed variables). If the form is called for an update (Change) or Delete, the buffer must contain the record. Normally, the browse fills the buffer before calling the form (updates the buffer with the selected record). But if the update procedure is Started and files are threaded, the buffer on the form's thread will always be empty, including fields that should have been primed (buffer contents are not shared by the threads).

This is not to say that update forms cannot be Started, but it does mean that an update form on another thread requires additional care and code to ensure that the requested action and the target record are provided to the form. (see Mike Hanson's article Multi-Threaded Browses and Forms).

Also, both GlobalRequest and GlobalResponse are stored in local variables almost immediately after they are primed. In a form, for example,

```
SELF.Request = GlobalRequest
```

is set at the very beginning of the INIT method, right after that method sets up the Error Manager. While GlobalRequest is cleared a few lines later, SELF.Request (a.k.a. ThisWindow.Request) is the variable actually used in the form. So, once this value has been stored locally, anything that may happen to GlobalRequest is irrelevant.

GlobalResponse is read into Response immediate on return from the form. Again, because the local variable is used, GlobalResponse ceases to be relevant immediately. While this may not entirely satisfy the worry over mashing globals and doesn't address any principles about the use of globals, it at least minimizes the opportunity for problems.

The ABC Way

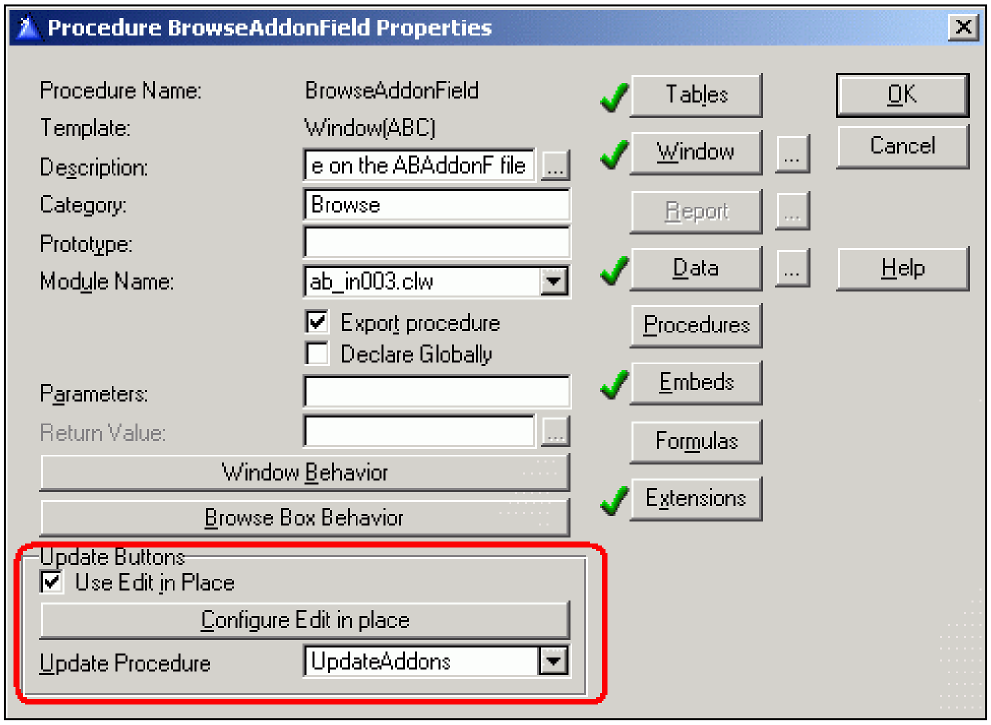
The ABC Libraries call an update procedure with the Run method and two parameters. The first parameter is a number (an ordinal, actually), the second is an Equate for the request (1 for Insert, 2 for Change, 3 for Delete and 4 for Select). The generated code for a real browse-form looks like this:

```
ThisWindow.Run PROCEDURE(USHORT Number,BYTE Request)
ReturnValue          BYTE,AUTO
  CODE
  ReturnValue = PARENT.Run(Number,Request)
  IF SELF.Request = ViewRecord
    ReturnValue = RequestCancelled
  ELSE
    GlobalRequest = Request
    UpdateABCUSICD ! update procedure
    ReturnValue = GlobalResponse
  END
  RETURN ReturnValue
```

While not critical to the current question, it is useful to know that the Parent call is where record priming (including auto incrementing keys) takes place. Priming in the browse allows support for edit-in-place, explaining why this happens in the browse, not the form where one would expect it. The import of this is that a form called without a browse will not prime properly on its own; you have to call the priming methods explicitly (see my COL article [Calling Form Procedures]).

The code above shows how `GlobalRequest` is set, the form called and `GlobalResponse` read. Note, however, that the first parameter, `Number`, does not appear to be used. And, in fact, for the typical browse-form combination, it isn't.

The `Number` parameter is the value of `BRWx.AskProcedure` and in the typical browse-form there is only one procedure called. (The code generated looks remarkably like Clarion for DOS code, doesn't it?) If the browse uses only edit in place, there is no procedure call and the update code shown above isn't generated. In fact, when edit in place is configured and there is no named update procedure, this `Run` method isn't generated at all. However, if edit in place is checked *and* an update form is named, as shown in Figure 1. the `Run` method is generated.


Procedure BrowseAddonField Properties

Procedure Name:	BrowseAddonField
Template:	Window(ABC)
Description:	e on the ABAddonF file
Category:	Browse
Prototype:	
Module Name:	ab_in003.clw
[✓] Export procedure	
[ ] Declare Globally	
Parameters:	
Return Value:	

Update Buttons
[✓] Use Edit in Place
Update Procedure: UpdateAddons

Figure 1. EIP and standard update form

In this case, it is possible to programmatically control whether edit in place or a form is used by setting:

```
BRWx.AskProcedure = 0> !use EIP
```

or

```
BRWx.AskProcedure = 1!use form
```

at appropriate places (see Jim DeFabia's [Dr. De Phobia article] which discusses Edit-in-Pace and forms). The Number parameter, oddly, is more likely to be used in a form, not a browse:

```
ThisWindow.Run PROCEDURE(USHORT Number,BYTE Request)
ReturnValue          BYTE,AUTO
  CODE
  ReturnValue = PARENT.Run(Number,Request)

  IF SELF.Request = ViewRecord
    ReturnValue = RequestCancelled
  ELSE
    GlobalRequest = Request
    EXECUTE Number
      BrowseDepartments
      BrowseUserSorts
      BrowsePriceRules
      BrowseMixMatch
      BrowseSpecialTax
      BrowseBottleDeposit
      BrowseAddonField
      UpdateBOMItem
      UpdateSaleItem
      UpdateChildPLU
    END
    ReturnValue = GlobalResponse
  END
  RETURN ReturnValue
```

Each of the procedures named in this list are lookups.

Each lookup procedure named on the Actions tab of a control, and this procedure has quite a few, is inserted into an Execute structure, assigned a value (in the order populated) and, at the appropriate time, the value of AskProcedure is updated and the correct lookup called. This allows a single method to call all needed procedures for all needed purposes. It may not contribute to readability, but it certainly reduces the amount of code generated.

The ABC Way: Part Deux

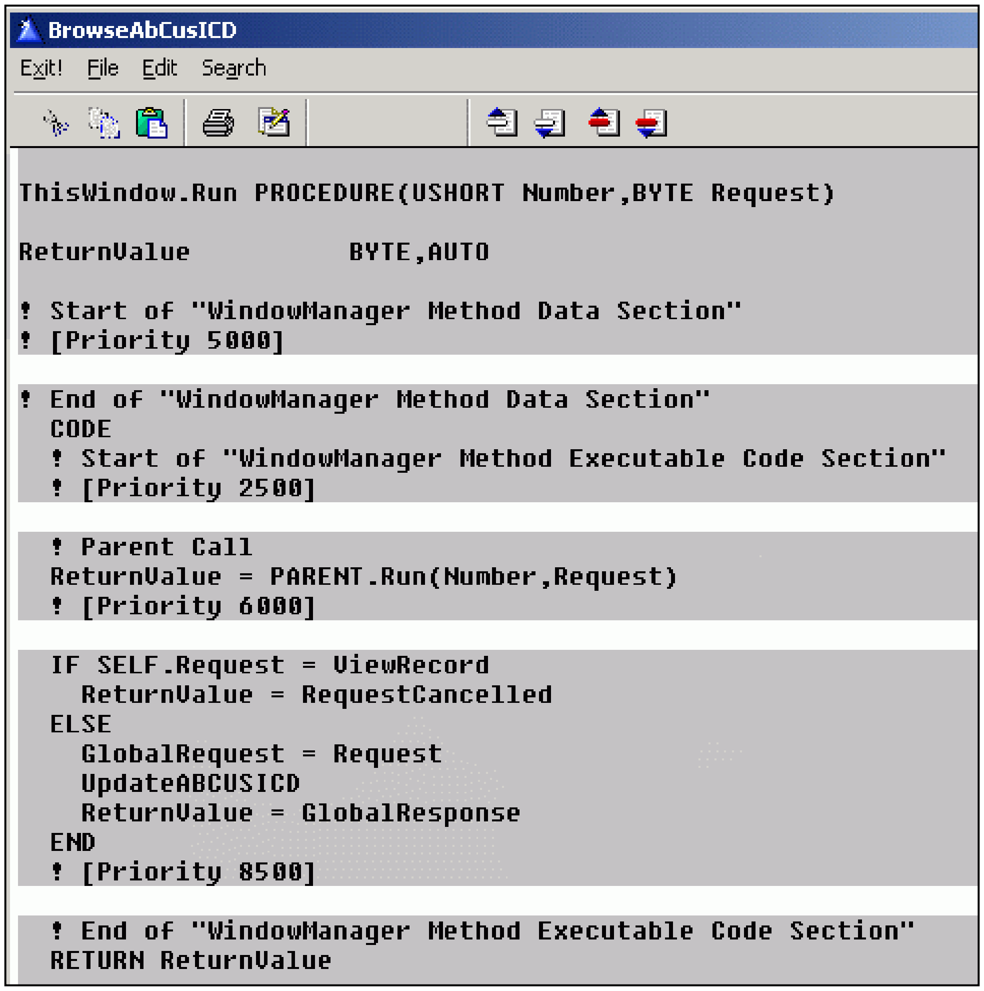
If ThisWindow.Run calls a procedure or creates an Execute structure after setting GlobalRequest, multiple update procedures should not be a great challenge. It would seem easy enough to emulate what the templates generate.

First, anywhere before the update call, the AskProcedure value must be set. For example,

```
Of InsertRecord
  If CFG:Permit
    BRWx.AskProcedure = 14
  Else
    BRWx.AskProcedure = 17
  End
```

```
Else
  BRWx.AskProcedure = 2
End
```

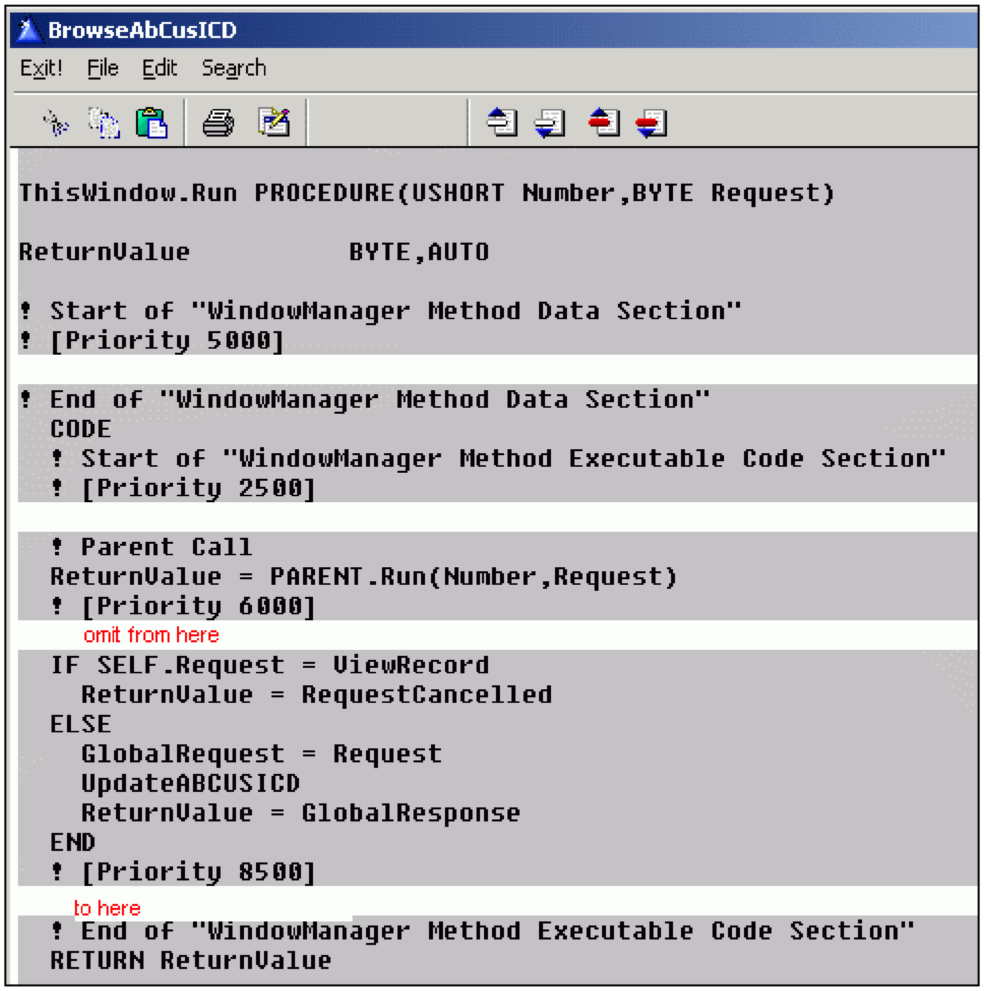
Now is it just a matter of constructing or modifying the Execute structure. Unfortunately, there are no embeds available in ThisWindow.Run to get code where it is needed in or around the Execute structure (which isn't even generated in the standard case):



```
ThisWindow.Run PROCEDURE(USHORT Number,BYTE Request)

ReturnValue          BYTE,AUTO

! Start of "WindowManager Method Data Section"
! [Priority 5000]

! End of "WindowManager Method Data Section"
  CODE
  ! Start of "WindowManager Method Executable Code Section"
  ! [Priority 2500]

  ! Parent Call
  ReturnValue = PARENT.Run(Number,Request)
  ! [Priority 6000]

  IF SELF.Request = ViewRecord
    ReturnValue = RequestCancelled
  ELSE
    GlobalRequest = Request
    UpdateABCUSICD
    ReturnValue = GlobalResponse
  END
  ! [Priority 8500]

  ! End of "WindowManager Method Executable Code Section"
  RETURN ReturnValue
```

Figure 2. Embeds in Run procedure

Oh me, oh my, oh dear! Whatever is a poor developer to do? In the end, the easiest thing to do seems to be to remove the standard code using the OMIT directive, as shown in Figure 3. At least, this is the most often recommended solution.



```
ThisWindow.Run PROCEDURE(USHORT Number,BYTE Request)

ReturnValue          BYTE,AUTO

! Start of "WindowManager Method Data Section"
! [Priority 5000]

! End of "WindowManager Method Data Section"
  CODE
  ! Start of "WindowManager Method Executable Code Section"
  ! [Priority 2500]

  ! Parent Call
  ReturnValue = PARENT.Run(Number,Request)
  ! [Priority 6000]
    omit from here
  IF SELF.Request = ViewRecord
    ReturnValue = RequestCancelled
  ELSE
    GlobalRequest = Request
    UpdateABCUSICD
    ReturnValue = GlobalResponse
  END
  ! [Priority 8500]
   to here
  ! End of "WindowManager Method Executable Code Section"
  RETURN ReturnValue
```

Figure 3. Omitting the standard code

Omitting the Run code shown in Figure 3 results in the following code being generated:

```
  omit('xxx') ! embed before
  IF SELF.Request = ViewRecord
    ReturnValue = RequestCancelled
  ELSE
    GlobalRequest = Request
    UpdateABCUSICD
    ReturnValue = GlobalResponse
  END
    xxx ! embed after
```

While the omitted code must be duplicated, it can be completely customized and do exactly what is needed:

```
IF SELF.Request = ViewRecord
  ReturnValue = RequestCancelled
ELSE
  GlobalRequest = Request
  Execute Number
    Proc_1
    Proc_2
    Proc_3
  End
  ReturnValue = GlobalResponse
END
```

When an update procedure must be called, simply ensure `BRW`*x*`.AskProcedure` has been set. A simple assignment does the job.

But Wait, There's More!

The same effect can be achieved using a more legacy-oriented style, that is, without having to set any properties (the `OMIT` is still required but setting the `AskProcedure` isn't):

```
IF SELF.Request = ViewRecord
  ReturnValue = RequestCancelled
ELSE
  GlobalRequest = Request
  Case Request
  Of InsertRecord
    If CFG:Permit
      OffCampusForm
    Else
      OnCampusForm
    End
  Else
    StandardForm
  End
  ReturnValue = GlobalResponse
End
```

or

```
IF SELF.Request = ViewRecord
  ReturnValue = RequestCancelled
ELSE
  GlobalRequest = Request
  Case SecurityLevel
  Of 42
    Proc_1
  Of 87
    Proc_2
  Else
    Proc_3
  End
  ReturnValue = GlobalResponse
End
```

Parameters

The technique of `OMIT`ting the standard code and substituting customized code allows an often requested thing: calling update procedures with parameters.

```
IF SELF.Request = ViewRecord
  ReturnValue = RequestCancelled
ELSE
  GlobalRequest = Request
  MyUpdateProc(LOC:myParameter)
  ReturnValue = GlobalResponse
End
```

And, in my opinion, passing parameters to the update procedure is the *only* reason (legitimate or otherwise) for using the OMIT technique. For simply calling one of several update procedures, it is too much work and it is too confusing; too many i's to dot and t's to cross.

The Right Way, the Wrong Way ...

There is an expression that goes "There are three ways to do things: the right way, the wrong way and the Navy way." (I am aware that there are many variations on this, all claiming to be the original. My research shows this version is the oldest and, therefore, *the* original.)

The Easy Way

Omitting a block of template code and then re-typing in order to customize it seems awfully kludgey, to say the least. Overriding the Run method sounds ... unappetizing. (Maybe a pair of embeds in a future version of the templates?)

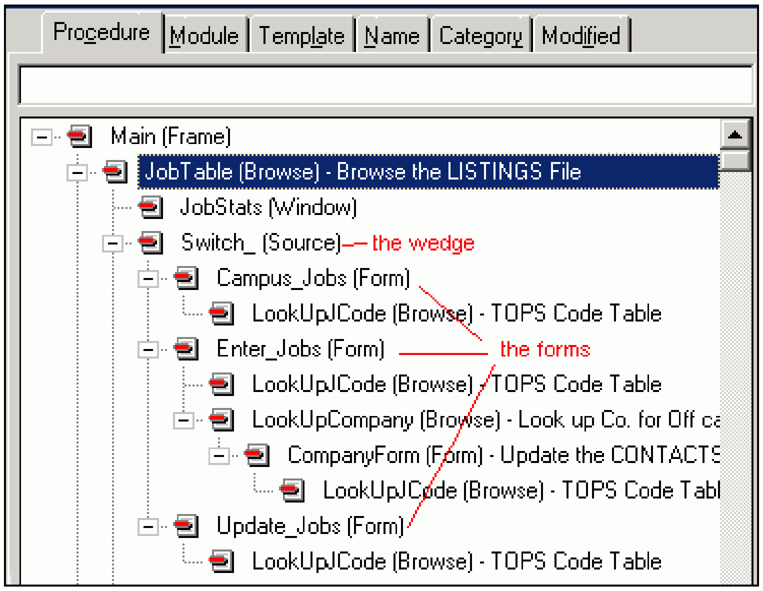
There is a legacy way of calling multiple update forms. And, by "legacy," here, I mean Clarion Professional Developer 2.0 (1988): wedge procedures, so named because a procedure is wedged between two other procedures and originally created to wedge in a lookup where no provision for a lookup had been made.

It is important to realize that an "update" procedure does not have to be a form. It can be *any* type of procedure. For example, the "update" procedure for a browse of purchase orders is usually another browse, a browse of the items in the highlighted purchase order. Similarly, the update procedure for a browse of invoices is usually a browse of line items. In other words, a browse can be an "update" procedure. So, why not use a Source procedure for an "update" procedure?-Why not indeed.

If the update procedure is created with the Source template, I can test the value of GlobalRequest or any other variable or condition, I can do so at any level of complexity I can handle and I can then call the desired procedure:

```
Case GlobalRequest
Of InsertRecord
  If CFG:AllowAdd <> ‘Y’
    CampusJobs
  Else
```

```
    EnterJobs
  End
Else
  UpdateJobs
End
```



```
Main (Frame)
  JobTable (Browse) - Browse the LISTINGS File
    JobStats (Window)
    Switch_ (Source)
      Campus_Jobs (Form)
        LookUpJCode (Browse) - TOPS Code Table
      Enter_Jobs (Form)
        LookUpJCode (Browse) - TOPS Code Table
        LookUpCompany (Browse) - Look up Co. for Off ca
          CompanyForm (Form) - Update the CONTACTS
            LookUpJCode (Browse) - TOPS Code Tabl
      Update_Jobs (Form)
        LookUpJCode (Browse) - TOPS Code Table
```

Figure 4. Procedure tree with a wedge

The procedure tree shown in Figure 4 is from a production app and is the entire code for the "update" procedure. Any of the code samples above, forced into the Run method can be made into a stand-alone Source procedure and, therefore, serve as an update procedure.

All that I *must* do is ensure that the value of GlobalRequest is preserved. And as long as a Source procedure is called on the same thread and does not call another procedure along the way, that value *will* be preserved.

If a lookup or validation is needed after calling the form but before that form is opened, GlobalRequest will be re-set in the lookup procedure. So, before calling the other procedure, save GlobalRequest. Restore it afterward:

```
!save request
SaveRequest = GlobalRequest
If GlobalRequest = InsertRecord
  !prime switch
  EVE:EventType = GetEventType()
  If GlobalResponse = RequestCancelled
    Return
  End
End
!restore request
GlobalRequest = SaveRequest
!use switch
Case EVE:EventType
```

```
Of ‘Interview’
  InterviewForm
Of ‘Workshop’ orof ‘Presentation’
  WorkShopForm
Of ‘Career Fair’
  CF_Form
End
```

This, too, is from a production app.

Summary

Pass a few parameters to an update procedure? `Omit` and code. A kludge but it works. Otherwise, wedges. In fact, wedges are used all the time. "Are you sure?" or log-in screens are wedges. They are usually used at the beginning of a procedure. But why not on the back end?

Wedge procedures, a tried and true technique allow multiple update procedures and I commend them to your attention. In the last example, if you trace the code, you’ll see I used two wedges (a cosmic wedgie?).

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

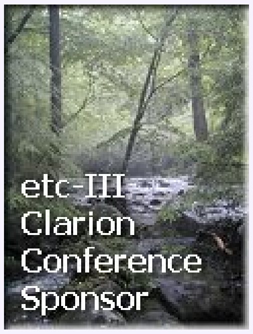
Add a comment

The BrowseUpdateButtons template should allow having either...
Carl, excellent idea.

Reborn Free CLARION online

published by CoveComm Inc.

Clarion MAGAZINE



Windows-Style List Box Sorting Revisited

by Steffen Rasmussen

Published 2001-06-27

Since my previous article on Windows-style list box sorting was published, some Clarion Magazine readers have responded with solutions to improve the list box sorting code. I have used some of these solutions and modified them a bit so they fit into the existing code structure. To finish off I have created a small List Box sorting template.

Depressing the header in the List Box

In the previous article I mentioned a couple of flaws in the code. One flaw that I did not mention was the absence of an explanation on how to refresh the list box after it had been depressed. Although I mentioned that you needed to know when the mouse is lifted from the region, I never showed where to implement the code or what code to implement. My apologies to the reader. What I forgot to mention was that in the embedded points MouseOut and MouseUp for the region, the source should contain the code `ThisWindow.Reset`.

One flaw that I did mention was when the column sorts in ascending and descending order and uses a key and a non key to accomplish this process, the user will not be able to see the header button being depressed, when a field name is used to sort the list box.

To solve this problem Nick Grasso came with an alternative solution. In stead of implementing code in the region to update the List Box when the mouse is moved out of the header (`MouseOut`) or lifted from the header (`MouseUp`), Nick suggested using the API function `Sleep()` to accomplish the same task.

To use an API function you have to include it into the Clarion language so it can be interpreted correctly at compile time. In the Global Properties select Embeds, then select the embedded point Inside the Global Map. Choose source and type the following:

```
MODULE('WIN32.LIB')
```

```
  Sleep(ulong),pascal !Windows API Sleep()
END
```

How does this then work? Lets look at the MODULE structure as defined in the LRM:

```
MODULE(sourcefile)
  Prototype
END
```

The MODULE structure contains function prototypes. The source file could be the name of the file containing the definitions for the different procedures. But in this case the MODULE is being used to create a new procedure definition in Clarion by calling the external library for the Windows API call. The source file has to have a unique name, which in this case is 'WIN32.LIB', but it could be anything you like. You don't have to specify the exact LIB file in this case because WIN32.LIB is always in memory.

In the prototype section you define the Windows API Sleep() function as

```
Sleep(ulong),pascal
```

where ulong is the data type and pascal is the calling convention. The calling convention Pascal is compatible with the Windows API and it specifies how the ulong parameter is passed. When implementing this code remember that the MODULE cannot begin in column 1 in the source file.

Now that the API Sleep() function has been defined it is just a matter of implementing it into the existing code:

```
F KEYCODE () = MouseLeft
  IF ?Browse:6{PROPLIST:MouseDownRow} = 0
    LOC:SortKey = ?Browse:6{PROPLIST:MouseDownField}
    ?Browse:6{PROP:Edit,LOC:SortKey} = ?Region1
    ?Region1{PROP:YPos} = 0
    ! New Sleep():
    DISPLAY(?RegionHeader)
    ! Call the SLEEP API function
    Sleep(200)
    IF LOC:SortKey = LOC:PreSortKey
      LOC:SortKey=LOC:SortKey-(LOC:SortKey*2)
      ?Browse:6{PROPLIST:Header,|
         ABS(LOC:SortKey)} = '« ' & LOC:PreHeader
    ELSE
      ?Browse:6{PROPLIST:Header,|
        ABS(LOC:PreSortKey)} = LOC:PreHeader
      LOC:PreHeader = ?Browse:6{PROPLIST:Header,|
         ABS(LOC:SortKey)}
      ?Browse:6{PROPLIST:Header,ABS(LOC:SortKey)} |
         = '» ' & LOC:PreHeader
    END
    LOC:PreSortKey = LOC:SortKey
  ELSE
    ?Browse:6{PROP:Edit,LOC:SortKey} = 0
    BRW6.TakeNewSelection()
  END
  ! New Reset:
  ThisWindow.Reset
END
```

In this case I have chosen to let the depressed header button sleep for 200 milliseconds before it continues executing of the rest of the code. In the period where the system is sleeping the ?Region1 is displayed as a depressed button.

Originally, instead of using `Sleep()` I had the code monitoring the header in the list box for the mouse activities right click down, lift mouse up and move mouse out of header, to control when the depressed button is visible or not. Now when using the `Sleep()` function it is only necessary to monitor for the mouse right click down. So when the program has finished, executing the code for depressing the header and sorting the list box it is just a matter of resetting the window and making the changes take place right after the 200 milliseconds has passed. The disadvantage is that the user loses control over the amount of time passed before changes take effect because the code is not waiting for the user to lift the mouse up or out of the header. But then again, is this necessary? If the user uses this sorting function as it is supposed to be used, with a quick right mouse click and release on the header, he or she will never notice it.

Resizing the columns

In Clarion you can resize each column by selecting the columns right border and thereby adjusting the width of the column. A minor flaw that I did not mention in the previous article is that it is not possible to select the columns right border in the header area, because this will just cause the column to sort instead of resize. Charles Patnoi came with a simple solution to this problem.

In the existing code there is an `IF` structure which checks for a left mouse button press in the header area. So in order to make it possible to resize the columns in the header area, without sorting the column, the program just has to see if the left mouse button click is outside the field's right border resize zone; if so, execute the following:

```
IF KEYCODE () = MouseLeft
  IF ?Browse:6{PROPLIST:MouseDownRow} = 0
  ! New code to determine mouse position:
    IF ?Browse:6{PropList:MouseDownZone} |
      = ListZone:Right
      CYCLE
    END
    LOC:SortKey = |
      ?Browse:6{PROPLIST:MouseDownField}
    ?Browse:6{PROP:Edit,LOC:SortKey} = ?Region1
    ?Region1{PROP:YPos} = 0
    DISPLAY(?RegionHeader)
    Sleep(200)
    IF LOC:SortKey = LOC:PreSortKey
      LOC:SortKey=LOC:SortKey-(LOC:SortKey*2)
      ?Browse:6{PROPLIST:Header,ABS(LOC:SortKey)} |
        = '« ' & LOC:PreHeader
    ELSE
      ?Browse:6{PROPLIST:Header,|
         ABS(LOC:PreSortKey)} = LOC:PreHeader
      LOC:PreHeader = ?Browse:6{PROPLIST:Header,|
         ABS(LOC:SortKey)}
      ?Browse:6{PROPLIST:Header,ABS(LOC:SortKey)} |
         = '» ' & LOC:PreHeader
    END
    LOC:PreSortKey = LOC:SortKey
  ELSE
    ?Browse:6{PROP:Edit,LOC:SortKey} = 0
    BRW6.TakeNewSelection()
  END
  ThisWindow.Reset
END
```

Deselecting columns for sorting

Since Clarion Magazines readers have solved most of the flaws in the previous article, I might as well get rid of the problem, namely the one where clicking on a column header, which does not contain any conditional browse behavior, will show the button selection as well as a non-existing sort direction.

Each column is represented by a number from one to the maximum number of columns in the list box. This number is used to determine which column to sort. So the next step is to determine which columns not to sort, and when one of these is selected, skip the execution of the code. Ideally determining which column not to sort should be based on the non-existence of any predefined Conditional Browse Behavior. Unfortunately I haven't been able to automatically determine this, so if any of you readers have a solution for this task let me know.

For the time being I am going to use a CASE structure that contains the number for each column not to sort and if the number isn't among them execute the code:

```
IF KEYCODE () = MouseLeft
  IF ?Browse:6{PROPLIST:MouseDownRow} = 0
    IF ?Browse:6{PropList:MouseDownZone} |
       = ListZone:Right
      CYCLE
    END
    !New CASE structure:
    CASE ?Browse:6{PROPLIST:MouseDownField}
    OF 3 ! Do not sort this column
    OF 5 ! Do not sort this column
    ELSE
      LOC:SortKey = |
         ?Browse:6{PROPLIST:MouseDownField}
      ?Browse:6{PROP:Edit,LOC:SortKey} = ?Region1
      ?Region1{PROP:YPos} = 0
      DISPLAY(?RegionHeader)
      Sleep(200)
      IF LOC:SortKey = LOC:PreSortKey
        LOC:SortKey=LOC:SortKey-(LOC:SortKey*2)
        ?Browse:6{PROPLIST:Header,ABS(LOC:SortKey)}|
           ='« ' & LOC:PreHeader
      ELSE
        ?Browse:6{PROPLIST:Header,|
           ABS(LOC:PreSortKey)} = LOC:PreHeader
        LOC:PreHeader = ?Browse:6{PROPLIST:Header,|
           ABS(LOC:SortKey)}
        ?Browse:6{PROPLIST:Header,ABS(LOC:SortKey)}|
           ='» ' & LOC:PreHeader
      END
      LOC:PreSortKey = LOC:SortKey
    END
  ELSE
    ?Browse:6{PROP:Edit,LOC:SortKey} = 0
    BRW6.TakeNewSelection()
  END
  ThisWindow.Reset
END
```

In the above code columns three and five are the columns not to sort.

Now that all the code for this list box sorting has been created the next step is to create a template for Clarion to automate the future use of the code. I have included a small template, which can be studied by the interested reader. I won’t go into the details of how I wrote this template since there are already some excellent articles in Clarion Magazine that cover the basic principles of template creation.

Download the source

Steffen S. Rasmussen has graduated in Computer Science from Copenhagen Business College. Since then he has worked as a programmer, system technician and network administrator, and is currently IT manager. Clarion is a quite a new language to Steffen since his only been working with it since January 2000. But what better way to learn it than by trying to teach others! Steffen has also set up a web site to collect as many examples of different user interfaces as possible to inspire Clarion developers.

Reader Comments

Add a comment

Nice little template! One suggestion is to add an option...

Reborn Free
CLARION online
published by CoveComm Inc.
Clarion MAGAZINE

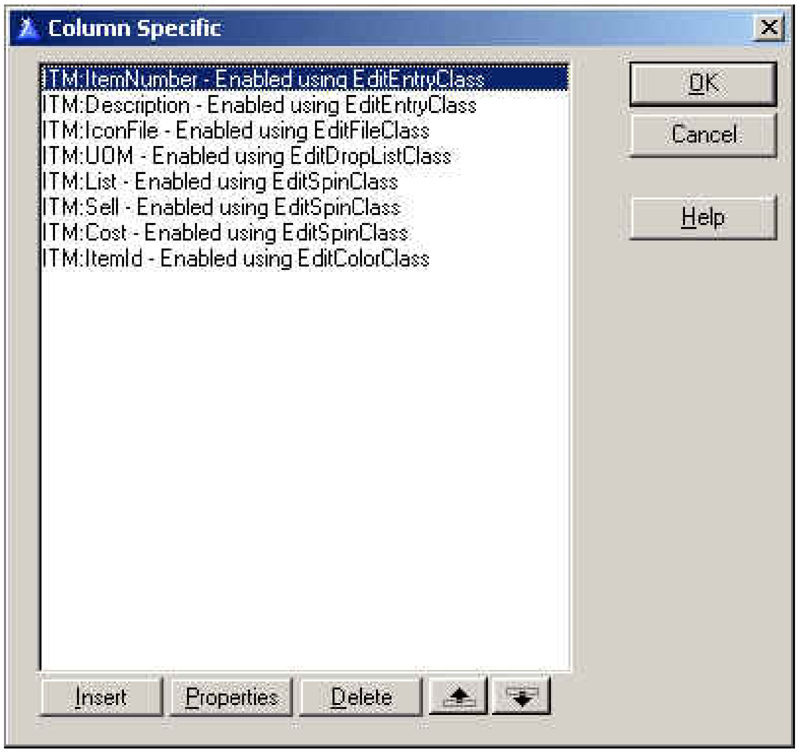
Extending ABC's Edit In Place - Part 2

by Russell Eggen

Published 2001-06-22

Last week I discussed the theory of Edit In Place (EIP), so now it’s time to apply the theory. I’ve made a sample Clarion 5.5 application, available at the end of this article. It requires nothing more than the ABC templates.

The application is a simple order entry application. If you downloaded the application already, look at the `BrowseItems` procedure. Open the Extensions dialog and highlight Update a record from Browse Box on Item. Press Configure Edit In Place, and you’ll see the dialog in Figure 8.



```
ITM:ItemNumber - Enabled using EditEntryClass
ITM:Description - Enabled using EditEntryClass
ITM:IconFile - Enabled using EditFileClass
ITM:UOM - Enabled using EditDropListClass
ITM:List - Enabled using EditSpinClass
ITM:Sell - Enabled using EditSpinClass
ITM:Cost - Enabled using EditSpinClass
ITM:ItemId - Enabled using EditColorClass
```

Figure 5. List box columns using EIP.

If you highlight `ITM:List` and press Properties, you see that this column uses an `EditSpinClass` object.. The templates name these objects a bit strangely. I don’t care for the names, but I do agree with the reasoning behind it, as this scheme reduces the chances of duplicate object names. As a general rule, I change the object name to something that describes what it does and who it is. This makes it easy when and if you need to refer to it in embedded source. This case, I named it EIP:ITM:List for **E**dit **i**n **P**lace on the **List** price field in the **It**e**m** file.

It makes sense to make this a spin control as the user can click on the increment or decrement buttons to adjust the value. I want control to stop on a dime (pun intended), not a dollar amount (which is the default). As part of the Init method, I need only one line of code as shown in the Embeditor:

```
EditInPlace::IND:Cost.Init PROCEDURE(↵
    UNSIGNED FieldNumber,UNSIGNED ListBox,*? UseVar)

! Start of "Edit-In-Place Manager Method ↵
    Executable Data Section"
! [Priority 5000]

! End of "Edit-In-Place Manager Method ↵
    Executable Data Section"
  CODE
  ! Start of "Edit-In-Place Manager Executable ↵
    Code Section"
  ! [Priority 2500]

  ! Parent Call
  PARENT.Init(FieldNumber,ListBox,UseVar)
  ! [Priority 5001]
  SELF.FEQ{PROP:Step} = .1 ! added code

  ! [Priority 5001]
  IF LOC:NoEdit
    EditInPlace::IND:Cost.SetReadOnly(1)
  END
  ! End of "Edit-In-Place Manager Executable ↵
    Code Section"
```

Remember, whenever you are coding in a method, use the keyword SELF when calling other methods in this method's object, or referring to variables that are a part of this object (but not local to the method).

I have three price type fields in the list box, so I add the above code to each one. If you look at the Help for a Spin control, there are more attributes than simply "step". You can even change the appearance of the spin arrows. I've done this on two of the columns, each with different properties. This gives you an idea of how creative you can get with simple embedded code. Run the example application to see this, and inspect the embed points for these column controls to see how this is done.

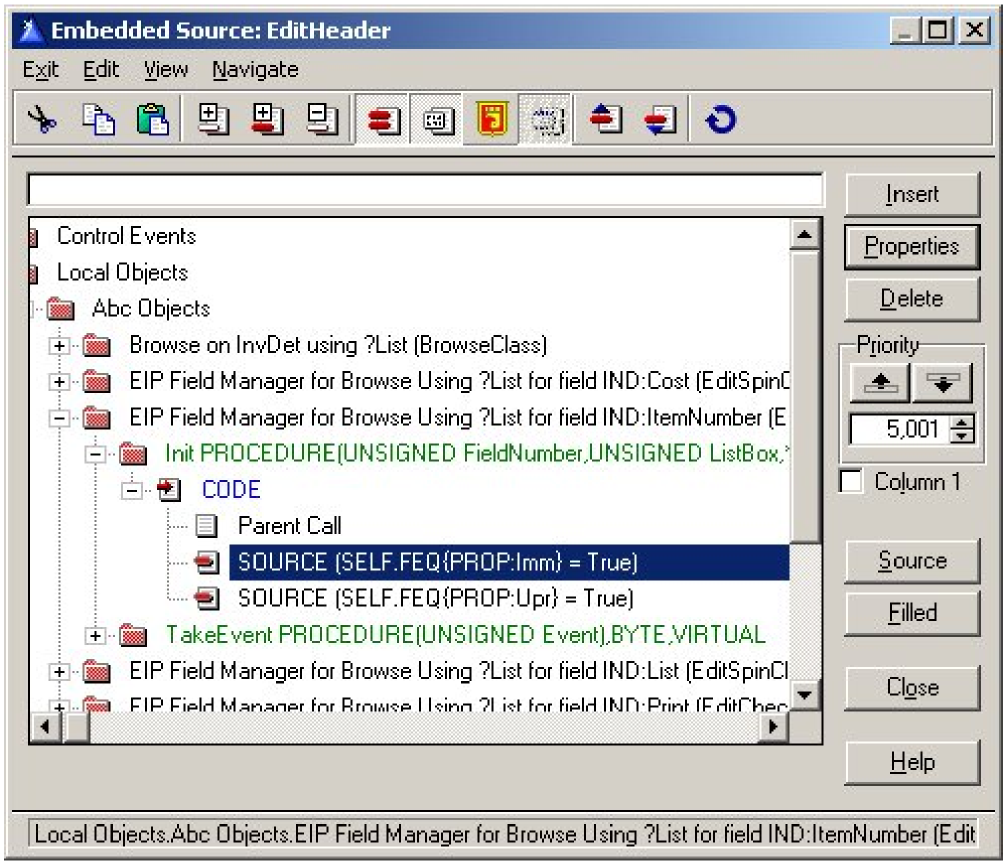
Auto-complete (or automatic lookups)

In some other applications like Outlook Express or Quicken, a nifty feature is "auto-complete." This is where you start typing the name of the person you wish to send email or write a check to, and the rest of the entry is completed for you. You can do this with Clarion's EIP features too, once you set it up.

Obviously, I am talking about entry controls, as they are simple to use and set up, but you can apply these principles to drop combos.

In the example application, the procedure where the auto-complete is located is `EditHeader`. This procedure is a Form for updating the `InvHdr` (Invoice Header) table. It contains a child list box for editing the `InvDet` (Invoice Detail) table. There is one interesting thing to point out with this list control. A local variable called `LOC:ExtendedAmount` is disabled when EIP is active. This is controlled with the same dialog covered previously. The point is that you can disable edits on columns where applicable.

The auto-complete feature is within the embeds for the `IND:ItemNumber` column. This column is a `STRING` variable. The example data files contain letters and numbers (although any data type could be used). To make this work, I wrote code to do a lookup as the user enters a character. Remember, EIP controls are entry controls, created at runtime, with no attributes active as a default. Thus two attributes are needed. These are `UPR` which ensures everything is in upper case and `IMM` which generates an `EVENT:NewSelection` for each keystroke. The code looks like this as a source embed:



```
Control Events
Local Objects
  Abc Objects
    Browse on InvDet using ?List (BrowseClass)
    EIP Field Manager for Browse Using ?List for field IND:Cost (EditSpinC
    EIP Field Manager for Browse Using ?List for field IND:ItemNumber (E
      Init PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,
        CODE
          Parent Call
          SOURCE (SELF.FEQ{PROP:Imm} = True)
          SOURCE (SELF.FEQ{PROP:Upr} = True)
      TakeEvent PROCEDURE(UNSIGNED Event),BYTE,VIRTUAL
    EIP Field Manager for Browse Using ?List for field IND:List (EditSpinCl
    EIP Field Manager for Browse Using ?List for field IND:Print (EditChec
```

Local Objects.Abc Objects.EIP Field Manager for Browse Using ?List for field IND:ItemNumber (Edit

Figure 6. The two property statements to ensure successful functionality.

> **Note:** Normally you would not use two embeds for two lines of code; I did it here only to expose the code in the embed tree.

The next thing is to trap the events. There is only one event embed point for any EIP control. This is named TakeEvent. As the name implies, TakeEvent traps (takes) any event. This is a no-brainer as to which embed is needed for event trapping; you have only one!

Before I dive into this code, let's review what the code should do. As a user types characters into this control, the code needs to see if there are any matching records based on what was typed. Sounds a bit like an incremental locator, doesn't it? It does help to think how an incremental locator works, finding the closest match as the user enters data. You do need to find the closest match. To make this work nicely, the other columns are filled with data per the code for this control. All the user needs to do from that point onward is decide if he found the correct data or not. All without leaving the control.

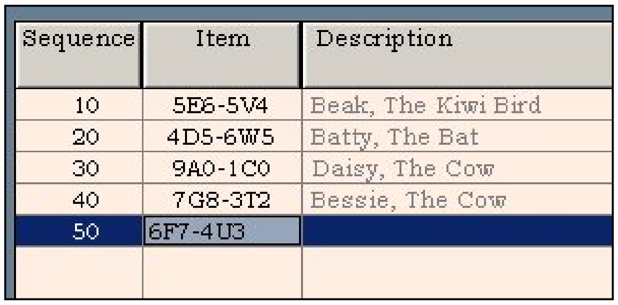
I am talking about simple lookups. All Clarion programmers can do simple lookups. So let me now show you the code where this is done.

Listing 1. The code to lookup a related record.

```
CASE EVENT()
OF EVENT:NewSelection
  UPDATE(SELF.FEQ)
  OffSet = SELF.FEQ{PROP:SelStart} - 1
  IF OffSet
    !Try to fill in remaining item number
    ITM:ItemNumber = |
      SUB(DetailList.Q.IND:ItemNumber,1,OffSet)
    SET(ITM:ItemNumberK,ITM:ItemNumberK)
    IF Access:Item.NEXT() |
       OR UPPER(SUB(ITM:ItemNumber,1,OffSet)) <> |
       UPPER(SUB(DetailList.Q.IND:ItemNumber,1,OffSet))
      DetailList.Q.IND:ItemNumber = |
        SUB(DetailList.Q.IND:ItemNumber,1,OffSet)
      DetailList.Q.IND:Quantity   = 0
      DetailList.Q.IND:Cost       = 0
      DetailList.Q.IND:List       = 0
      DetailList.Q.IND:Sell       = 0
      DetailList.Q.IND:Print      = ''
    ELSE
      DetailList.Q.IND:ItemNumber = ITM:ItemNumber
      DetailList.Q.IND:Quantity   = 1
      DetailList.Q.IND:Cost       = ITM:Cost
      DetailList.Q.IND:List       = ITM:List
      DetailList.Q.IND:Sell       = ITM:Sell
      DetailList.Q.IND:Print      = '0'
    END
    SELF.FEQ{PROP:SelStart} = OffSet + 1
  ELSE
      DetailList.Q.IND:ItemNumber = ''
      DetailList.Q.IND:Quantity   = 0
      DetailList.Q.IND:Cost       = 0
      DetailList.Q.IND:List       = 0
      DetailList.Q.IND:Sell       = 0
      DetailList.Q.IND:Print      = ''
  END
  DISPLAY()
END
```

In essence, as the user types each character, the code tries to find a matching record. If it does not, leave everything blank. When you find the first matching record, fill it in. The user may continue typing characters if it is not the correct record.

The key is trapping the `EVENT:NewSelection` event. This is available only if the `IMM` attribute is active for the entry control. `OffSet` is a local data variable defined in the data embed for this method.



Sequence	Item	Description
10	5E6-5V4	Beak, The Kiwi Bird
20	4D5-6W5	Batty, The Bat
30	9A0-1C0	Daisy, The Cow
40	7G8-3T2	Bessie, The Cow
50	6F7-4U3	

Figure 7. The application showing the auto-completion.

The above screen shot shows what happens when you are in insert mode and after typing the number 6. There are many other features available with EIP. They may not be obvious in the above screen shot, but let me point out these features starting with the most salient feature. Notice the use of images in the "Print?" column. In the List Box Formatter, this column has the Icon attribute added. This attribute is required.

The icon use is conditional. Two icons are used, one to indicate that it will print on an invoice, and the other indicating it won't. See below for how this is done via template dialogs.

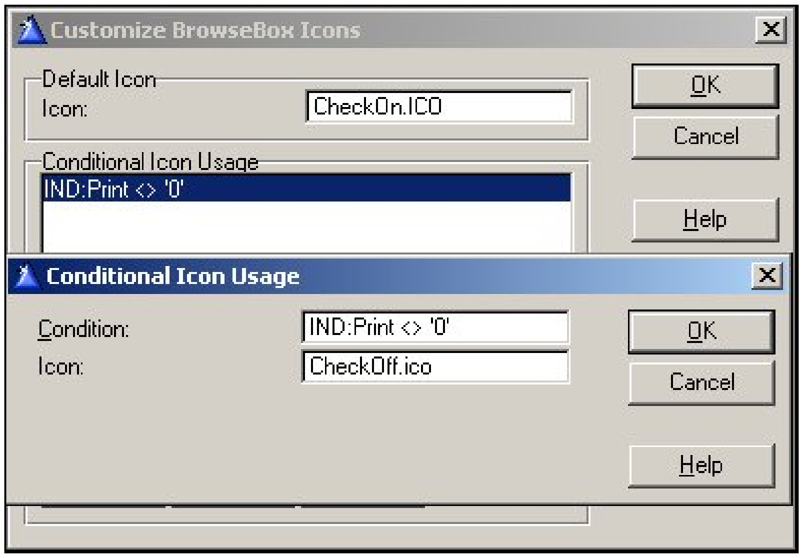
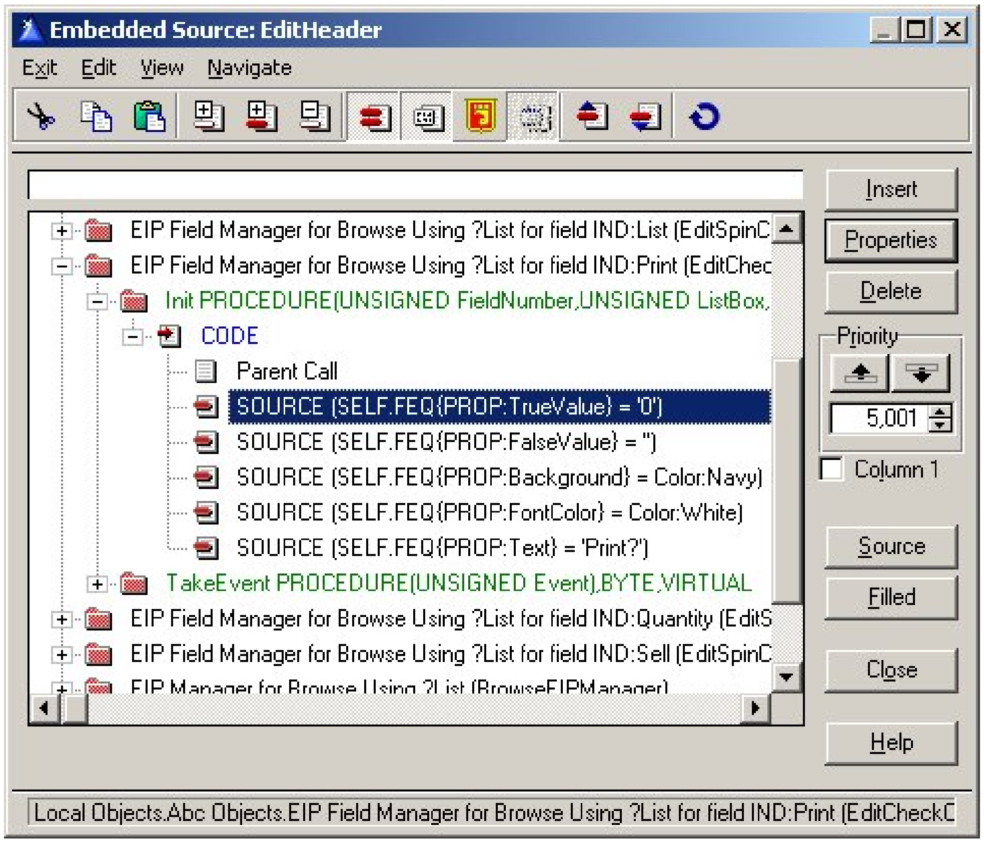


Figure 8. Conditional icon dialog.

This column is a check control, using the `EditCheck` class. Figure 9 illustrates how this works.

Figure 9. Embed tree showing the code for the `EditCheckClass`.



```
EIP Field Manager for Browse Using ?List for field IND:List (EditSpinC
EIP Field Manager for Browse Using ?List for field IND:Print (EditChec
  Init PROCEDURE(UNSIGNED FieldNumber,UNSIGNED ListBox,
    CODE
      Parent Call
      SOURCE (SELF.FEQ{PROP:TrueValue} = '0')
      SOURCE (SELF.FEQ{PROP:FalseValue} = '')
      SOURCE (SELF.FEQ{PROP:Background} = Color:Navy)
      SOURCE (SELF.FEQ{PROP:FontColor} = Color:White)
      SOURCE (SELF.FEQ{PROP:Text} = 'Print?')
  TakeEvent PROCEDURE(UNSIGNED Event),BYTE,VIRTUAL
EIP Field Manager for Browse Using ?List for field IND:Quantity (EditS
EIP Field Manager for Browse Using ?List for field IND:Sell (EditSpinC
EIP Manager for Browse Using ?List (BrowseEIPManager)
```

```
Local Objects.Abc Objects.EIP Field Manager for Browse Using ?List for field IND:Print (EditCheckC
```

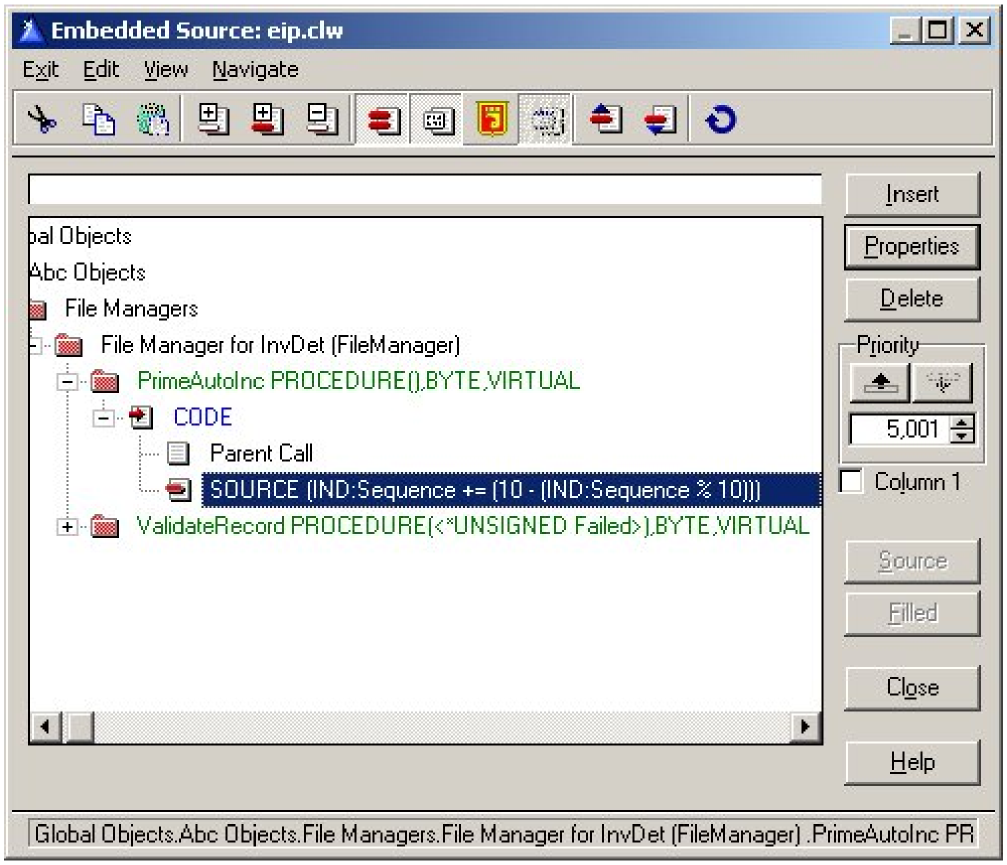
Each property statement is shown in the embed tree. Looking at the status message, you can see the location of the embed in addition to the object it belongs to. The two property expressions, `PROP:TrueValue` and `PROP:FalseValue` set the attributes of the true and false states of the control. For display reasons, numeric strings work best here, but you do not have to use them. It depends on whether or not you wish to show the actual values in the column. In this case, the picture string is `@N1B`.

The next two embed lines show how to set the colors when the EIP control is active. Of course, the `PROP:Text` attribute places a text string in the control. In this instance, the text asks the user if the record should be printed. The Description and Extended Total columns are disabled while EIP is active. Simply uncheck the "Allow Edit in Place" box to do disallow editing of this column at runtime. You find this option located in the Column Specific dialog discussed earlier.

The Sequence column increments by values of 10 when you add a new line. This feature allows editing of this number if the user wishes to override the value. For example, to insert a new line between line 20 and line 30, change the next default value of 40 to 25. This is a nice feature allowing a user to place or order where the line items display.

So how do you change the auto incrementing values to add 10 to the last number? While this does not really have anything to do with EIP, it is a nice feature. You see this effect while running a procedure, but there is nothing in any particular procedure that can control such behavior.

Remember, files are considered global data. Thus anything affecting the behavior of files is in the global embeds. Look at the screen shot below:



```
bal Objects
Abc Objects
  File Managers
    File Manager for InvDet (FileManager)
      PrimeAutoInc PROCEDURE(),BYTE,VIRTUAL
        CODE
          Parent Call
          SOURCE (IND:Sequence += (10 - (IND:Sequence % 10)))
      ValidateRecord PROCEDURE(<*UNSIGNED Failed>),BYTE,VIRTUAL
```

Global Objects.Abc Objects.File Managers.File Manager for InvDet (FileManager) .PrimeAutoInc PR

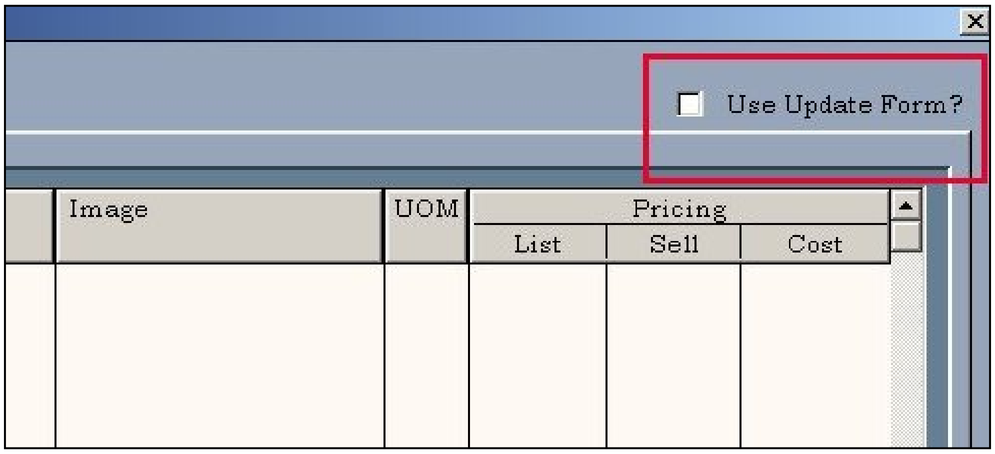
Figure 10. Changing the behavior of an auto incrementing field

As Figure 10 shows, it only takes one line of code needed to make that work. Also notice the location of the embed by looking at the status line. It reflects the tree structure. I also have an override in the ValidateRecord method. I'll leave that to you to explore.

EIP or update form?

This question of how to use EIP or an update form comes up often enough it is worth discussing. When should you use EIP vs. the Form procedure to edit the record? Can you do both? The answer to the latter question is "Yes." Often some customers like a certain feature in your program, but others do not like it and want to do a task a different way. Some may not like EIP at a certain point in your application and wish to use the Form procedure, or vice versa.

I've always supported giving customers more than one way of doing things, and the example application provides such a feature. Look at the `BrowseItems` procedure. Open the window formatter (or run the application). There is a check box that gives the user the choice of how the wish to edit records.



The checkbox control is a local variable named `LOC:UseForm`. In the `ThisWindow.Update` method (embed point), is the following code:

```
IF LOC:UseForm
  !If user wants to use edit form
  !tell ABC to make it so.
  ItemList.AskProcedure = True
ELSE
  !else use EIP for all edits
  ItemList.AskProcedure = False
END
```

Figure 11. Application showing the EIP or Form option to the user.

This code is instructing ABC how to edit records. The `AskProcedure` property identifies the procedure to update a browse item. The `AskProcedure` property is setup for you in the procedure's `Init` method by the templates. A value of zero (0) indicates the `BrowseClass` object's own AskRecord method used to do updates. Any other value uses a separate procedure registered with the `WindowManager` object.

Note: The ABC default browse object name is `BRW1`. I changed this object name to an easier to understand name *before* I wrote the coded. If I had done this afterwards, I would get compiler errors, as the templates do not interpret hand written source.

Summary

I started Part 1 of this article with the basics of how EIP works at runtime, and why. I then explained why you would want to set attributes for EIP controls. I showed you where to find the areas you wish to change to meet your design specifications. I also showed you nothing complex or difficult.

ABC provides some wonderful functionality for EIP and it does not take much work on your part to add these features. Some may argue that embedding hand code is not very RAD and SoftVelocity should have better templates. I counter that argument by stating EIP as it is done now is quite RAD; you have flexibility configuring EIP to meet your needs. A better template is always a good idea, but that should not detract from what is quite a workable system.

Run the example application that accompanies this article and study the code. If you can improve on it, please add your comments to this article and I will be more than pleased to post a follow-up article. You will, of course, be given proper credit. In a later article, I'll discuss adding some template support to the features covered here.

Download the C5.5 source code

Download the C5b source code

Russ Eggen has been using Clarion since 1986. Until about 1996, he was using it for business applications, mostly accounting programs. Afterwards he joined Topspeed as a consultant, and later as an instructor. He was a founding member of SoftVelocity when that company formed from Topspeed in May 2000. He left SoftVelocity in January 2001 and now works for a private NY firm. Russ enjoys flying, scuba, and applied philosophy, and with great effort you might coax him into political discussions.

Reader Comments

Add a comment

A Clarion 5b version of the example source is now available...
Thank you Dave. Anyone having any problems with it, drop...

Reborn Free CLARION online published by CoveComm Inc.

Clarion MAGAZINE

Clarion News

ZipFlash 2.2 Released

Sterling Data has released ZipFlash 2.2, a low cost way to add or look up ZIP Codes. ZipFlash does ZIP and city/state lookups. Options include: latitude/longitude; area code; time zone; and a map showing the location of the city and a button to calculate the distance between any two cities/ZIP codes. Demo available.
Posted Friday, June 29, 2001

SealSoft xQuickFilter Template v1.0

xQuickFilter is a control template which adds filtering capabilities to list boxes. Filter by INSTRING or by currently selected data. Hot keys are configurable in the template. Demo available.
Posted Wednesday, June 27, 2001

---

solid.software Closed For Holidays

The solid.software office will be closed from June 28 until July 10. Products can still be ordered at ClarionShop; support questions will be answered as soon as the office opens.
Posted Wednesday, June 27, 2001

ABC Free Templates Have A New Home

The ABCFree Templates and Tools have moved to www.authord.com - the most recent update to the set was June 22.
Posted Tuesday, June 26, 2001

xSmart Macro Version 2.2

SealSoft's xSmartMacro is a programmer's macro utility which makes coding embeds easier. Macros are stored in a tree structure, and can contain variables which can be edited visually (similar to the template #PROMPT statement). Macros can also be exported/imported.
Posted Monday, June 25, 2001

Free Zip Code Template

Shane Vincent has made a free zip code template available for download. This is just a code template right now, but may be fleshed out more if there is a demand.
Posted Thursday, June 21, 2001

WinSet Lets Users Change Windows Properties

New from SealSoft is xWinSet, a product that lets end users change Windows standard screen properties at runtime, including font size, foreground and background colors, wallpaper, etc. Styles are supported. Demo available.
Posted Thursday, June 21, 2001

CapeSoft Tip of the Month: Object Writer

Not in the mood to fork out some money this month? No matter. If you're writing your own objects, or extending the ABC objects, then check out the FREE CapeSoft Object Writer template.
Posted Thursday, June 21, 2001

CapeSoft Office Messenger Updated

One product built on NetTalk functionality is the CapeSoft Office Messenger. This release improves on the core product, smoothing a few rough edges, and has improved stability. At $6 per seat (less for educational, not-for-profit, and volume sites) it certainly doesn't break the budget.
Posted Thursday, June 21, 2001

NetTalk v1 Beta 14 Shipping

Beta 14 of NetTalk now includes full support for UDP as well as TCP. This means you can now send, and receive UDP packets. Beta 14 also includes some internal tweaks which adds to the stability of the product. NetTalk is rapidly approaching a version 1.0 Gold release, expected by the end of the month. Shortly after that the special price of $199 will end and the normal price of $299 will apply.
Posted Thursday, June 21, 2001

Secwin Version 3.1 Beta 1 Released

It's been over a year since the last release of Secwin. As you may know version 3.0 was never officially out of beta although it was certainly live in a large number of systems. The main problem keeping it from going gold is the SQL support files. This version still has not addressed the SQL issues completely, but there are too many sufficient new features to hold back a release. The primary goal of this release is to provide full compatability with the Web Builder templates, and the recently released ClarioNet product. It's also now compatible with Makeover. Another major feature of this release is the ability for you to re-create any of the built-in Secwin screens. The administration functions have also been extended to allow full programmatic control of the security files. Secwin costs $99.
Posted Thursday, June 21, 2001

File Explorer Version 1.8b Released

Version 1.8a of FileExplorer is now shipping. This release adds Flash to the list of file formats supported. This means you can now include Macromedia Flash files on your application windows. Great for Login screens, as well as About screens etc. There's also a whole trainload of new properties and methods giving you more control than ever before. FileExplorer costs $99.
Posted Thursday, June 21, 2001

---

xWord Library Version 1.3.1

New in the latest release of SealSoft Company's xWord library are methods for importing plain text from MS Word, and inserting text in the clipboard. A new demo is also available.
Posted Thursday, June 21, 2001

PD Translation Dictionary

The PD Translation Dictionary is now available to contributors on the download page of the ProDomus web site. This dictionary contains environment files and a dictionary consisting of translations from English to other languages. You can help out in expanding this dictionary by becoming a contributor. Anyone sending either translations files or environment files by September 15, 2001 will be provided access to this file. Current languages (some partial) include Danish, French, German, Norwegian, Spanish, Czech, Turkish, and Polish. All could use additional environment and translation files. Translation files which we can use include clarion .env and .trn files as well as files associated CWIntl,PD Translator, or PD Translator Plus). The dictionary package also includes a small source code application for extracting translations from the dictionary or adding translations to the dictionary.

Posted Thursday, June 21, 2001

Simshape Templates Special Ends June 21

Eric Churton has announced the release of a new product for C5 and C5.5 (ABC only). The Simshape Templates let you create buttons in different shapes, use various images for mouseover events, and use images for checkboxes, all by making images and regions behave like buttons. SimShape Templates are now available from www.clarionshop.com at an opening special of $39 US up to June 21 2001, thereafter $49 US. Demo available.

Posted Tuesday, June 19, 2001

Reborn Free CLARION online

published by CoveComm Inc.

Clarion MAGAZINE

Search

Go

Home

COL Archives

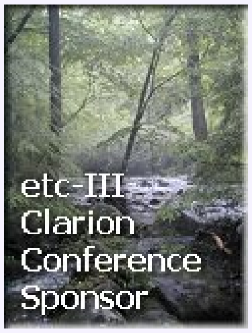
Information
Log In
Membership/
Subscriptions
FAQ
Privacy Policy
Contact Us

Downloads
PDFs
Freebies
Open Source

Site Index

Call for Articles

Reader Comments



Handcoding Tree Lists Part 2

by James Cooke

Published 2001-06-20

Last week I explained the fundamentals of Clarion tree lists. Simply put, a Tree Control is a standard listbox with the "Tree" checkbox set to true, and one way to create a tree list is to create the data yourself. It’s not that difficult to do.

Clarion also supplies the ABC Relational Tree Template, which generates code to populate the simple Windows Tree Control with relational database tables. However, the relational tree does not give you everything! This week I will show you how to do what the ABC relational tree cannot do.

Database Loaded Trees

ABC’s Relational Tree template is fantastic – but it only loads a single one-to-many relationship at one time. For example, each customer has many orders. What if you want to represent another relationship on the same tree? For example, each customer has many orders, but each customer has many backorders too. This means that the Customer node needs to have two child nodes. One will indicate orders, with all the customer’s orders listed in child nodes below, and the other will indicate backorders, with each backorder similarly listed.

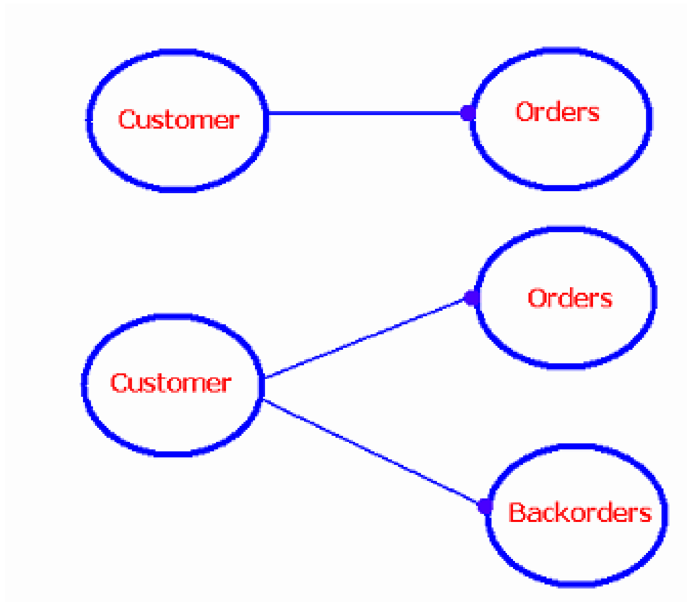


Figure 8. ABC's relational tree does not directly support dual relationships

To handle this kind of dual relationship you will need to code the tree manually. Try this out: define these files in a dictionary and link up the relationships:

```
Customer    FILE,DRIVER('TOPSPEED'),↵
               PRE(Customer),CREATE,BINDABLE
KeyCustomer   KEY(Customer:Customer),↵
                NOCASE,OPT,PRIMARY
KeyName       KEY(Customer:Name),DUP,NOCASE,OPT
Record        RECORD,PRE()
Customer        LONG
Name            STRING(20)
              END
            END

Order       FILE,DRIVER('TOPSPEED'),PRE(Order),↵
               CREATE,BINDABLE
KeyOrder      KEY(Order:Order),NOCASE,OPT,PRIMARY
KeyCustomer   KEY(Order:Customer),DUP,NOCASE,OPT
Record        RECORD,PRE()
Order           LONG
Customer        LONG
Item            STRING(20)
              END
            END
BackOrder   FILE,DRIVER('TOPSPEED'),↵
               PRE(BackOrder),CREATE,BINDABLE
KeyBackOrde   KEY(BackOrder:BackOrder),NOCASE,OPT,PRIMARY
KeyCustomer   KEY(BackOrder:Customer),DUP,NOCASE,OPT
Record        RECORD,PRE()
BackOrder       LONG
Customer        LONG
Item            STRING(20)
              END
```

```
           END
```

Populate the database files with some data, and then embed the following code in a button labeled "Load database data"

```
Free(MyQueue)   !Empty Tree
Post(Event:Accepted,?Button:AssignIcons)
MyQueue:Display = 'Customer Orders & Backorders'
MyQueue:Level = 0
MyQueue:Icon = 1
Add(MyQueue)

Set(Customer)
Loop
  next(Customer)
  if error() then break.
  MyQueue:Display = Customer:Name
  MyQueue:Level = 1
  MyQueue:Icon = 2
  Add(MyQueue)
  MyQueue:Display = 'Orders'
  MyQueue:Level = 2
  MyQueue:Icon = 3
  Add(MyQueue)
  Order:Customer = Customer:Customer
  Set(Order:KeyCustomer,Order:KeyCustomer)
  Loop
    Next(Order)
      If error() then break.
      If Order:Customer <> Customer:Customer |
             then break.
      MyQueue:Display = Order:Item
      MyQueue:Level = 3
      MyQueue:Icon = 4
      Add(MyQueue)
    End
    MyQueue:Display = 'Back Orders'
    MyQueue:Level = 2
    MyQueue:Icon = 3
    Add(MyQueue)
    BackOrder:Customer = Customer:Customer
    Set(BackOrder:KeyCustomer,BackOrder:KeyCustomer)
    Loop
      Next(BackOrder)
      If error() then break.
      If BackOrder:Customer <> Customer:Customer|
          then break.
        MyQueue:Display = BackOrder:Item
        MyQueue:Level = 3
        MyQueue:Icon = 4
        Add(MyQueue)
    End
End
```

This section of code basically loops through the database , and for each customer it creates a set of nodes for the order and a set of backorders. Compile and run the app, click the button "Load Database Data" and your tree should look something like Figure 9:

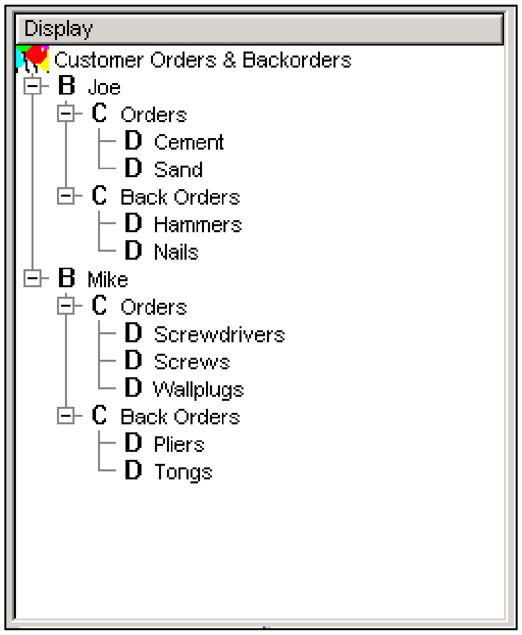


Figure 9. This tree displays multiple database relations

How about creating update forms for these nodes? This is not as simple as in the relational tree, mainly because unlike the relational tree a different update form may be called for "D" level items. For example, if the parent is Back Orders then call `FrmBackOrder`; if the parent is Orders then call `FrmOrder`. If the node is a "B" level item, then it is simple – just call `frmCustomer`, and if the node is a "C" level item then don’t do anything. The simplest way to do this might be to use custom Insert/Change/Delete buttons and embed the logic in each button’s *accepted* embed point. Here’s how:

Declare a `LONG` local variable called `Counter`

Declare a `BYTE` local variable called `CurrentLevel`

Now place three buttons underneath the tree and label them Insert, Change and Delete.

Embed the following code in the Change button:

```
Get(MyQueue,Choice(?List))
Case MyQueue:Level
  Of 1
    Customer:Customer = MyQueue:PK
    If Access:Customer.Fetch(Customer:KeyCustomer)
      Stop(error() & errorfile())
    Else
      GlobalRequest = ChangeRecord
      FrmCustomer
      If GlobalResponse = RequestCompleted
```

```
        Post(Event:Accepted,?Button:LoadDB)
      End
    End
    Of 3
      CurrentLevel = ABS(MyQueue:Level)
      Loop Counter = Choice(?List) to 1 by -1
        Get(MyQueue,Counter)
        !Found the parent!
        If ABS(MyQueue:Level) < CurrentLevel then break.
      End
      Case Clip(MyQueue:Display)
      Of 'Orders'
        Get(MyQueue,Choice(?List))
        Order:Order = MyQueue:PK
        If Access:Order.Fetch(Order:KeyOrder)
          Stop(error() & errorfile())
        Else
          GlobalRequest = ChangeRecord
          FrmOrder
          If GlobalResponse = RequestCompleted
             Post(Event:Accepted,?Button:LoadDB)
          End
        End
      Of 'Back Orders'
        Get(MyQueue,Choice(?List))
        BackOrder:BackOrder = MyQueue:PK
        If Access:BackOrder.Fetch(BackOrder:KeyBackOrder)
          Stop(error() & errorfile())
        Else
          GlobalRequest = ChangeRecord
          FrmBackOrder
          If GlobalResponse = RequestCompleted
            Post(Event:Accepted,?Button:LoadDB)
          End
        End
      End
End
```

To allow for deletions, take the same code and embed it in the delete button then do a search & replace for "Change" and set it to "Delete". This is obviously a waste of code which is okay for the purpose of this article, but in a production environment you might place the code in a routine and put in a condition for Add/Change/Delete.

When adding a record, there are two things you need to consider. Firstly, to add a row your cursor needs to be over the *parent* record. Secondly, primary and foreign keys need to be primed correctly. Embed the following code into the Insert button: (The APP that accompanies this article also includes source comments, which have been removed here to improve readability)

```
Get(MyQueue,Choice(?List))
Case MyQueue:Level
  Of 0
```

```
    GlobalRequest = InsertRecord
       Clear(Customer:Record)
       Access:Customer.TryPrimeAutoInc()
       FrmCustomer
       If GlobalResponse = RequestCompleted
          Post(Event:Accepted,?Button:LoadDB)
       End
    Of 2
       Case MyQueue:Display
       Of 'Orders'
          GlobalRequest = InsertRecord
          Clear(Order:Record)
          Access:Order.TryPrimeAutoInc()
          Loop Counter = Choice(?List) to 1 by -1
             Get(MyQueue,Counter)
             If MyQueue:Level = 1 then break.
          End
          Order:Customer = MyQueue:PK
          FrmOrder
          If GlobalResponse = RequestCompleted
             Post(Event:Accepted,?Button:LoadDB)
          End
       Of 'Back Orders'
          GlobalRequest = InsertRecord
          Clear(BackOrder:Record)
          Access:BackOrder.TryPrimeAutoInc()
          Loop Counter = Choice(?List) to 1 by -1
             Get(MyQueue,Counter)
             If MyQueue:Level = 1 then break.
          End
          BackOrder:Customer = MyQueue:PK
          FrmBackOrder
          If GlobalResponse = RequestCompleted
             Post(Event:Accepted,?Button:LoadDB)
          End
       End
End
```

By now it should be clear that using a filter would be a breeze – all you need to do is check a condition before adding a node. This means that you can handcode all levels of nodes to be range-limit filtered (using keys) or simply by using sequential access.

People tend towards simplicity – a window full of listboxes and buttons only serves to confuse. Without a tree, the example application used in this article would have required a screen with three listboxes and nine buttons and probably a bunch of listbox headers and tooltips, panels and groups to clarify the relationship between the listboxes. The tree control has enabled me to display clearly the relationships between the data entities, as well as manage updates – all using four controls instead of 12!

Coding this tree might take a bit longer, but making the effort to achieve a solid understanding of this control will result in many benefits. As demonstrated in this example the simple elegance of the Clarion language still allows programmers to deliver higher quality hand-coded material faster than other mainstream tools. Compliance with user expectations and industry standards, providing simpler, intuitive user interfaces and meaningful data representation will contribute to a greater acceptance of your software in the marketplace.

Download the example application

James Cooke has been using Clarion since 2.1 days and has been a die hard for "the cause" ever since. He and his family recently moved from South Africa to Texas and is currently working in the banking industry. He spends most of his free time basking in the sun by the pool with a good book or succumbing to that hard-to-kick addiction that persistently haunts the Western cosmopolitan neighborhoods - the yard sale.

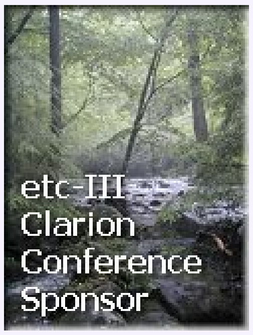
Reader Comments

Add a comment

Reborn Free CLARION online

published by CoveComm Inc.

Clarion MAGAZINE



Tip: How To Start A Browse With The Last-Used QBE Query

by Randy Rogers

Published 2001-06-19

I recently added Query By Example (QBE) to a browse and was pleased with the ease of implementing this functionality. I wanted my application to open the browse using the most recently used query (the default is to open the browse without any QBE filters). After a lot of experimenting, I was able to accomplish what I needed with two strategically placed lines of code.

Here's how to have a browse start with the most recent QBE query. In the WindowManager.Init Method [8505] (after process field templates) embed I added the following code:

```
BRW1.Query.Restore('tsMRU')
```

In the Browse on filename using ?Browse:1 (Browse Class) ApplyFilter Method [4500] (before parent call) I placed this code:

```
SELF.SetFilter(SELF.Query.GetFilter(),'9 - QBE')
```

That’s all there is to it. Calling the `Restore` method with ‘`tsMRU`’ causes the `QueryClass` to fill its `FieldQueue` with the `tsMRU` section of the program’s INI file. This is a queue of the field contents for the most recently used query (hence the acronym MRU). Other saved queries are stored in the INI file too! `GetFilter` returns a properly constructed filter expression based on the contents of the `FieldQueue` which I loaded earlier with `Restore`.

The `SetFilter` method appends the query filter to any existing filters because I specify the ‘`9 – QBE`’ id, which is used by the templates. See the `ViewManager SetFilter` method help for details.

Randy Rogers is a data processing professional with over 35 years of experience in a wide variety of industries including accounting, municipal government, insurance, printing, and pharmacoeconomics. He is the president of Keystone Computer

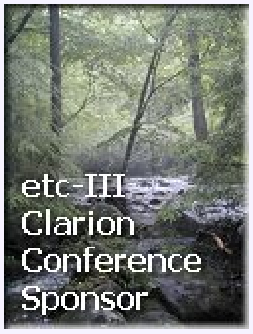
Resources and creator of NetTools, Queue Edit-in-Place and Screen Capture Tools for Clarion application developers. Randy has a degree in Mathematics from Florida State University and has taught programming at the community college level.

Reader Comments

Add a comment

Reborn Free CLARION online published by CoveComm Inc.

Clarion MAGAZINE



Extending ABC's Edit In Place - Part 1

by Russell Eggen

Published 2001-06-15

Over time I've noticed a number of complaints about ABC's Edit in Place (EIP) features, or lack thereof. Let me fill you in on a dirty little secret: ABC's EIP features are *fully functional*.

I'll admit, that statement may be a bit controversial. I assert that the true cause of dissatisfaction with EIP is the lack of full template support. In other words, you are required to add some embed code yourself to make EIP work per your application's design specifications. While some say adding embed code is not very RAD, I say adding a few lines of code does not detract from RAD.

As readers of Clarion Magazine may note, I like to start with the basics (assume no prior knowledge) and then move the reader to higher understanding. I plan to continue that practice with this topic. My goal here is to start at the beginning and work my way up to a template (in a future article) that you may add to your existing ABC applications. A template makes sense as it alleviates the need to write the same code in embeds for the same controls. It won't be the fully functional template I have in mind, but enough to give you the idea.

The basics of EIP

How does EIP work anyway? The first place to look is the ABC classes. Look at the following code.

```
EditClass.Init PROCEDURE( |
    UNSIGNED FieldNo,UNSIGNED ListBox,*? UseVar)
  CODE

  SELF.ListBoxFeq = ListBox
  SELF.CreateControl()
  ASSERT(SELF.Feq)
  SELF.UseVar &= UseVar
  SELF.Feq{PROP:Text} = |
    ListBox{PROPLIST:Picture,FieldNo}
  SELF.Feq{PROP:Use} = UseVar
  SELF.SetAlerts
```

What is that code doing? If you study it briefly, you notice this code is doing more than may first appear!

The SELF object name is whatever the name of the local object is. By local object, I mean a new object is created in this procedure, but ABC has no idea what the name is. SELF as a placeholder for this object's name. This object name defaults to BRW1. The use of SELF as the name of the object means that this method (procedure) is not hard coded and is reusable. The use of this placeholder shows the OOP advantage of "write once, use anywhere" as SELF could be any object name.

Notice the use of the property expressions. Since UseVar is declared as an ANY variable (see below), this means it takes on whatever characteristic (data type, properties) is it assigned. This is done via the reference assignment. Since the entry control is created on the fly (runtime, not development time), it needs a field equate (FEQ). This is done via the two lines starting with SELF.Feq. Notice the use of the property expressions for SELF.Feq. The FEQ also inherits the picture from the column, so any valid picture token is inherited.

```
EditClass   CLASS,TYPE
Feq           UNSIGNED
UseVar        ANY
ListBoxFeq    SIGNED
ReadOnly      BYTE
END
```

There is also a call to CreateControl, which is a method in this class. Here is the code for CreateControl:

```
EditClass.CreateControl     PROCEDURE
  CODE
  ASSERT(False)
```

A bit boring, isn't it? . This is a "stub method", meaning that overriding the method is expected. It is also a VIRTUAL method. This is one of the benefits of VIRTUAL methods; the parent class (ABC) will call the child method *instead* of its own method. In other words, you have changed the behavior of ABC without touching its code. For more on virtual methods, see The ABCs of OOP - Part 3.

There are a few more methods in this class, but their discussion is not important to what you need to know at this time. The EditClass itself is rather small anyway, so you do not need to devote much study effort to it.

Since I am discussing entry controls, which are the default ABC EIP controls, here is the entire class definition for the EditEntryClass (module and link attributes deleted for readability):

```
EditEntryClass    CLASS(EditClass),TYPE
CreateControl       PROCEDURE,VIRTUAL,PROTECTED
      END
```

This definition inherits from the parent EditClass. Inheritance gives you the ability to use everything in a parent as if it was defined in a child class. Thus, you only need to code what is different or missing. Inspect the code for this new method and you see why the ABC method had to be overridden – you specifically need an entry control, which the parent EditClass.CreateControl method does not create:

```
EditEntryClass.CreateControl     PROCEDURE
  CODE

  SELF.Feq = CREATE(0,CREATE:Entry)
```

The zero (0) parameter for the CREATE statement means to use the next available field equate number. CREATE:Entry is an equate for the type of control you wish. Inspect EQUATES.CLW if you wish to see the actual value, but all you really need to know is the equate.

Let’s assume you have a description for an Item file. When you browse the Item file, ITM:Description is one of the columns. When you have EIP activated for this column, the templates write the following code (edited to avoid wrapping):

```
EIP:ITM:Description CLASS(EditEntryClass)
Init      PROCEDURE(UNSIGNED FieldNumber,|
UNSIGNED ListBox,    |
*? UseVar),DERIVED
        END
```

Remember, the above local class is derived from the EditEntryClass, which itself is a child of EditClass. The Init method is defined in EditClass. Inheritance gives any child access to this method as if it was defined in its CLASS structure.

The DERIVED attribute means you want the compiler to flag this method as an error *if* the parent prototype changed. DERIVED methods are implied VIRTUALs (you do not need to add that attribute too). Imagine you have a class with a method declared with the VIRTUAL attribute, and a derived class that overrides that same method. Now you change the parent method’s parameter list. That means that the child method’s prototype and the parent method’s prototype no longer match, so the child is really no longer a child, but a new "top level" method. If you use the DERIVED attribute, the compiler will verify that there is an identical virtual method signature in a parent class. Remember the rule about VIRTUAL attributes: you must define a method with the same name and prototype in the parent class and the child class, with the VIRTUAL attribute on both.

One interesting aspect of the prototype is the use of the `*?` parameter. Remember that the `UseVar` passed to it is an `ANY` data type and is passed by address. You do not know at design time what kind of data is passed to an `ANY` data type; as the name implies, it could be anything.

The code for the generated `INIT` method is as follows:

```
EIP:ITM:Description.Init PROCEDURE|
  (UNSIGNED FieldNumber,UNSIGNED ListBox,|
   *? UseVar)
  CODE
  PARENT.Init(FieldNumber,ListBox,UseVar)
```

This method, when run, calls the code in the parent `EditEntryClass`, discussed earlier. And how is the Init method called in your local procedure? Look in the `BRW1.Init` method, which is executed when the browse window opens. If EIP is used, the ABC templates write the code to call the Init methods for each column that use EIP:

```
SELF.AddEditControl(EIP:ITM:Description,2)
```

The first parameter to `AddEditControl` is the object name; the second parameter is the corresponding list box column for this edit control. For each list column that uses an edit control for EIP, the `AddEditControl` method executes the *same* code, not copies. This is why the `SELF` keyword is so important. As each `AddEditControl` method executes, it calls the same ABC code, but `SELF` contains a different object name.

Now that the salient ABC class code is out of the way, you do have embed points available to further modify how the control is setup. You do this with property expressions. How do you know what properties are associated with what control? Look at the Help for Entry controls. There are a number of attributes listed for entry controls. You control some of these in dialogs while designing your window. For example, one could turn on or off `CAP`s. Look in the text portion of the Help, you see the property statements for each attribute. In this example, `PROP:Cap` is seen and this turns on or off the `CAP` attribute at runtime.

Simply use this one line of code in the embed point:

```
Self.Feq{Prop:Cap} = True
```

`SELF.Feq` is the field equate for the just-created entry control explained previously. How would you add tool tip text for this control? How about changing the font? Justification?

So this code, at runtime, when entering a description, causes each word to begin with an upper case letter.

More ABC EIP classes

ABC ships with a few more classes to assist you in defining precisely how you wish EIP to work. These classes are fully functional, but you are still required to tune them for your application's design.

In order to change the default entry control to something else, you need to know how to get to the Class dialog. A Class dialog exists for every ABC object in your application, overridden or not. Using these dialogs is easy, but it may not always be obvious how to get to them. See Figure 1 below. The class dialog is always the last tab on these dialogs. Since the topic is EIP, an by EIP, I mean the ability to update the highlighted record.

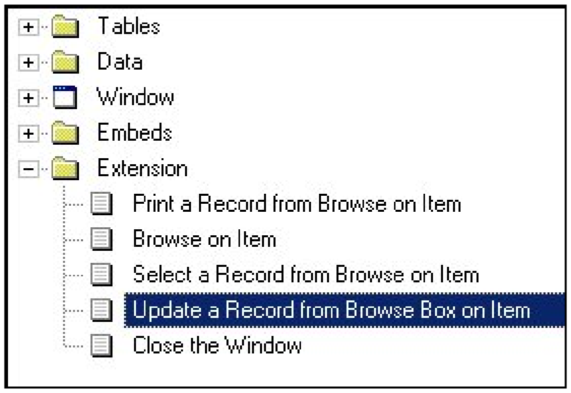
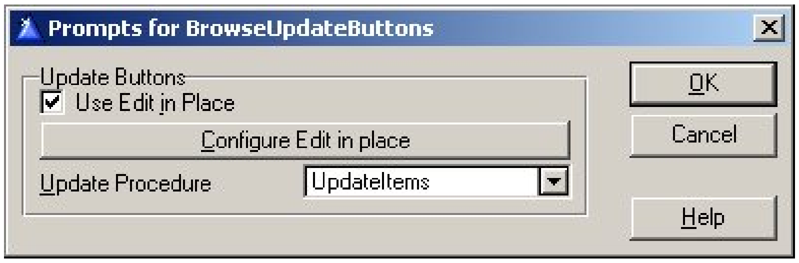

Tables
Data
Window
Embeds
Extension
Print a Record from Browse on Item
Browse on Item
Select a Record from Browse on Item
Update a Record from Browse Box on Item
Close the Window

Figure 1. Locating the update extension.

Selecting the properties for this extension shows this dialog (see Figure 2):



Use Edit in Place: ✓

Update Procedure: UpdateItems

Figure 2. The `BrowseUpdateButtons` dialog.

Pressing the Configure Edit in place button shows a dialog with default edit behaviors. What you are interested in the Column specific button. Pressing it shows an empty list box (when you first use it). This dialog (shown below as Figure 3 in Insert mode) allows you to insert capabilities for each column you wish to change.

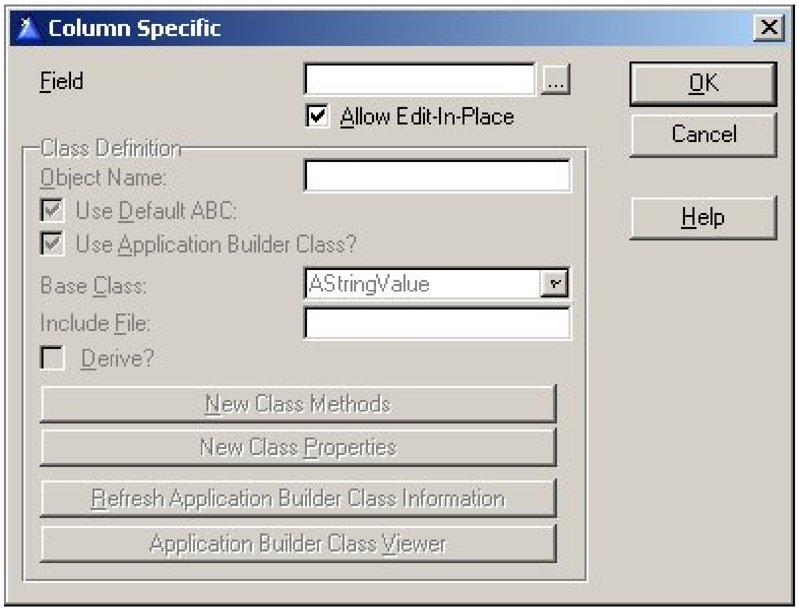


Figure 3. EIP dialog in Insert mode.

The Field entry is required. There is a lookup button next to the entry control to allow you to pick the field you wish to override. For example, suppose you have a field in your Inventory that controls the list price of the item. It would be nice to set the list price without using the keyboard. A spin control is ideal for this. Here is what the dialog would look like if you wish to change to a spin control:

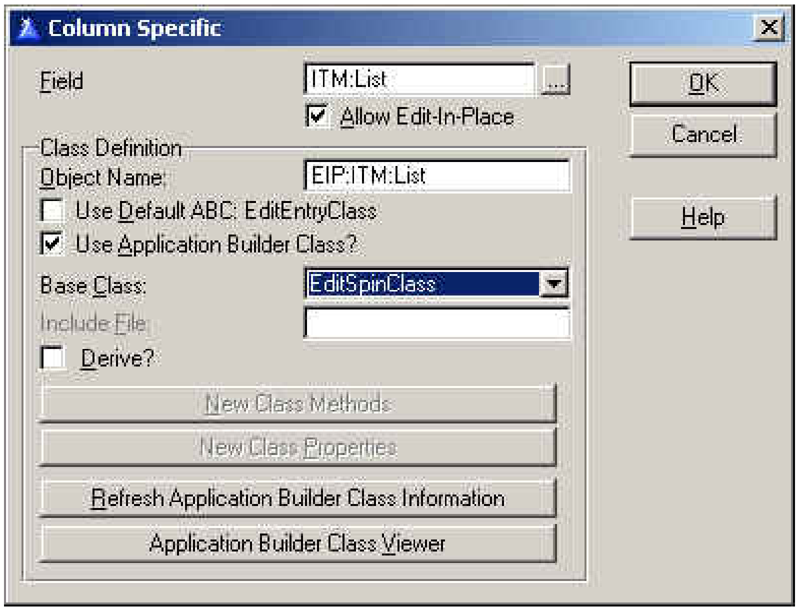

Field: ITM:List
Allow Edit-In-Place: ✓
Object Name: EIP:ITM:List
Use Default ABC: EditEntryClass: ☐
Use Application Builder Class?: ✓
Base Class: EditSpinClass
Derive?: ☐

Figure 4. Using the EditSpinClass instead of the default EditEntryClass.

To activate the Base Class drop down, you first must uncheck the Use Default ABC: EditEntryClass. You don't want an entry control. Notice that ABC has a class called `EditSpinClass`. There are quite a few other classes based on EIP (which are revealed when you drop down the base class list control). If you look in the ABC source for the `EditSpinClass`, you see this class definition:

```
EditSpinClass  CLASS(EditClass),TYPE
CreateControl    PROCEDURE,VIRTUAL,PROTECTED
   END
```

There isn't much to this class; it just creates a control and that's it. What does the *CreateControl* method look like? Inspect the code as shown below:

```
EditSpinClass.CreateControl PROCEDURE
  CODE

  SELF.Feq = CREATE(0, CREATE:Spin)
  SELF.Feq{PROP:Step}=1
  SELF.Feq{PROP:RangeLow}=80000001h
  SELF.Feq{PROP:RangeHigh}=7FFFFFFFh
```

I've already covered the `CREATE` statement. If you look up `CREATE:Spin` in the help, you will find the property statements used in this code. It is expected that you override these ranges in your application. Inspect the other `CreateControl` methods; you will see they are simple, and in some cases set some properties for you.

> **TIP:** The purpose of ABC objects is to be overridden. By themselves, ABC objects do absolutely nothing of value. They exist to be derived based on your design. Once this happens, you have working code.

So much for the theory. Next week, I'll show you how to apply the theory in a sample application.

Russ Eggen has been using Clarion since 1986. Until about 1996, he was using it for business applications, mostly accounting programs. Afterwards he joined Topspeed as a consultant, and later as an instructor. He was a founding member of SoftVelocity when that company formed from Topspeed in May 2000. He left SoftVelocity in January 2001 and now works for a private NY firm. Russ enjoys flying, scuba, and applied philosophy, and with great effort you might coax him into political discussions.

Reader Comments

Add a comment

Good article Russ. After reading it I was finally able to...
Dave, Glad it helped! Stay tuned for the next issue...
Thanks for explaining the underlying code for EIP. Up till...
Jan Jacob, That was the whole point of writing the...

Reborn Free CLARION online

published by CoveComm Inc.

Clarion MAGAZINE

Search
Go

Home

COL Archives

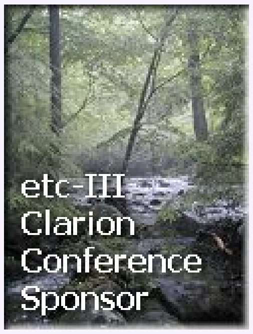
Information
Log In
Membership/
Subscriptions
FAQ
Privacy Policy
Contact Us

Downloads
PDFs
Freebies
Open Source

Site Index

Call for Articles

Reader Comments



ClarioNet Released! SoftVelocity Debuts New Specialized Thin Client for Clarion Applications

Published 2001-06-13

Media Relations: John Iacovelli, 954-785-4555 john.iacovelli@softvelocity.com
Sales Information: David Merenstein, 800-367-5444 david.merenstein@softvelocity.com

Internet/Intranet Clients Indistinguishable From Desktop Apps in Looks and Performance

Pompano Beach, Florida. June 13 – SoftVelocity™, Inc., manufacturer of the Clarion™ development environment, today announced the release of ClarioNET™. "ClarioNET is a new class of specialized thin client for line of business applications," said Bob Zaunere, CEO of SoftVelocity. "Imagine building a Windows application with all the features your users expect, and then being able to run it over the Internet without any license fees or royalties, and without having to change any of your code. ClarioNET enables one code base for deployment on your internal network or the Internet. A Clarion application combined with the ClarioNET client renders an exact duplicate of the application at any remote location... globally or locally."

Specializing in business database applications, Clarion is in widespread use as a development environment for line of business applications. The ClarioNET solution divides a new or existing Clarion application into an n-tier solution comprised of presentation, application, and database layers. An end user downloads a launcher, as small as 500KB, which can be used for multiple sessions and/or applications. The launcher connects to a server using the HTTP protocol in a series of compressed, encrypted transactions. The server starts an instance of the application, and downloads instructions to the client to create a window and controls. To the end user, it looks and feels exactly like any other Windows application, except for a small "LED" indicator on the caption bar that indicates when the client is sending or receiving information to or from the server. The client supports all functions of a typical Windows application, including printing reports to the client's local printer.

Typically, when a new window is displayed, the client and server exchange usually between one and nine kilobytes of data.

Thereafter, only instructions to update data are exchanged, usually less than one kilobyte per exchange. Unlike other remote client products, no constant communications stream needs be maintained with the server, helping to keep bandwidth requirements down, and server activity to a minimum. In tests on a Windows 2000® server with twenty clients opening views into database tables, server CPU activity remained below 15%, and memory usage below 25% on a 512MB system. Only the HTTP protocol is used, insuring compatibility with most firewalls. The ClarioNET solution requires either Microsoft Internet Information Server, or the Clarion Application Broker (a specialized web/application server) on the server side.

SoftVelocity Professional Services is deploying a ClarioNET application for CSRS, (http://www.csrs.ca) a Canadian registry service company based in Richmond, BC. CSRS offers registration and search services throughout Canada. The PPSA Management System delivers centralized database access and reporting for clients wishing to register their loan securities with provincial Personal Property Registries across Canada. Clients include all major chartered banks as well as the big three automobile financing companies.

ClarioNET was created by ClarioNET Solutions, Inc. of Corona Del Mar California. The ClarioNET client is written in the Clarion language, and is fully customizable. ClarioNET development requires either Clarion Professional Edition or Clarion Enterprise Edition, and the ClarioNET library. White papers containing additional information are available at http://www.softvelocity.com/clarionet.

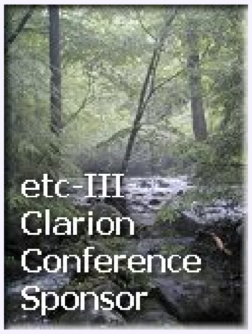
About SoftVelocity

SoftVelocity, Inc. is the provider of the Clarion line of rapid application development tools. Application developers have used Clarion for over two decades. Developers value its blazing speed and ease-of-use. SoftVelocity offers Clarion sales, education, and technical assistance to the worldwide community of Clarion users. SoftVelocity acquired the Clarion product line from TopSpeed Corporation on May 1, 2000. SoftVelocity is a privately held company based in Pompano Beach, Florida with distributors throughout the world. Access the www.softvelocity.com site for further information.

Reader Comments

Add a comment

Awesome product. My only hope is that SV creates client...



Handcoding Tree Lists Part 1

by James Cooke

Published 2001-06-13

One of the primary reasons for the increasing popularity of tree controls in user interfaces is that the user can be presented with a wide variety of related information, but without being overwhelmed with too much information at once. The user is able to achieve a bird's eye perspective of the information and optionally "drill down" into the data for more detail. Clarion is one of the few tools on the market that has built in functionality to represent relational data using the Tree Control – and even so, most Clarion programmers tend to avoid trees.

Considering the recent shift of the software industry toward tree-rich user interfaces, this might be a good time to examine some of the benefits and key concepts of the tree control, as well as how to make good use of them in Clarion applications.

What is a tree control?

It is important to differentiate between the ABC Relational Tree Template and the Tree Control. The ABC Relational Tree Template is an ABC template that generates efficient code to populate the simple Windows Tree Control with relational database tables. Simply put, a Tree Control is a standard listbox with the "Tree" checkbox set to true. Checking the Tree option will change the FORMAT attribute of the tree by adding 'T'. To illustrate, in a standard list box, the FORMAT string might look like this:

```
FORMAT('80L(2)|M~Customer Name~@s20@')
```

With the "Tree" attribute the FORMAT statement gets a 'T':

```
FORMAT('80L(2)|MT~Customer Name~@s20@')
```

The 'T' tells Clarion that it must display the data of the control's queue as a tree and not as an ordinary listbox. It's all very well for the control to know it needs to display its data in tree format, but how does the tree know at which level it needs to display the data? That will mean some additional data added to the queue.

The Tree Control's Queue

With a standard listbox there are no "system required" fields. You just slap in the columns as and when you need them. The tree control however requires several columns placed at the beginning of the queue and in a particular order. These columns are as follows:

```
Text     STRING(255)
Icon     SHORT
Level    LONG
```

These columns are used to tell the tree a bit more about itself:

- `Text` is the text to display for that node on the tree
- `Icon` specifies the node's icon
- `Level` specifies the amount of indentation from the left border of the listbox – The root of the tree is always Zero, and every subsequent shift to the right denotes an increase in level. Looking at Figure 1, the black text is level 0; the green text level 1; the red text level 2 and the blue text is level 3.

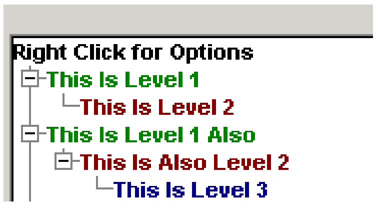


Figure 1. Denoting levels on a tree

Once these required columns are in place (at the start if the queue) you can place your own specific columns after them. Consider a simple example:

Create an application, define a window and declare the following queue in the data formatter:

```
MyQueue      QUEUE,PRE(MyQueue)
Display        STRING(200)
Level          LONG
Icon           SHORT
             END
```

Place a listbox control on the window, and when prompted with Select control template to use, select "Populate control without control template." :

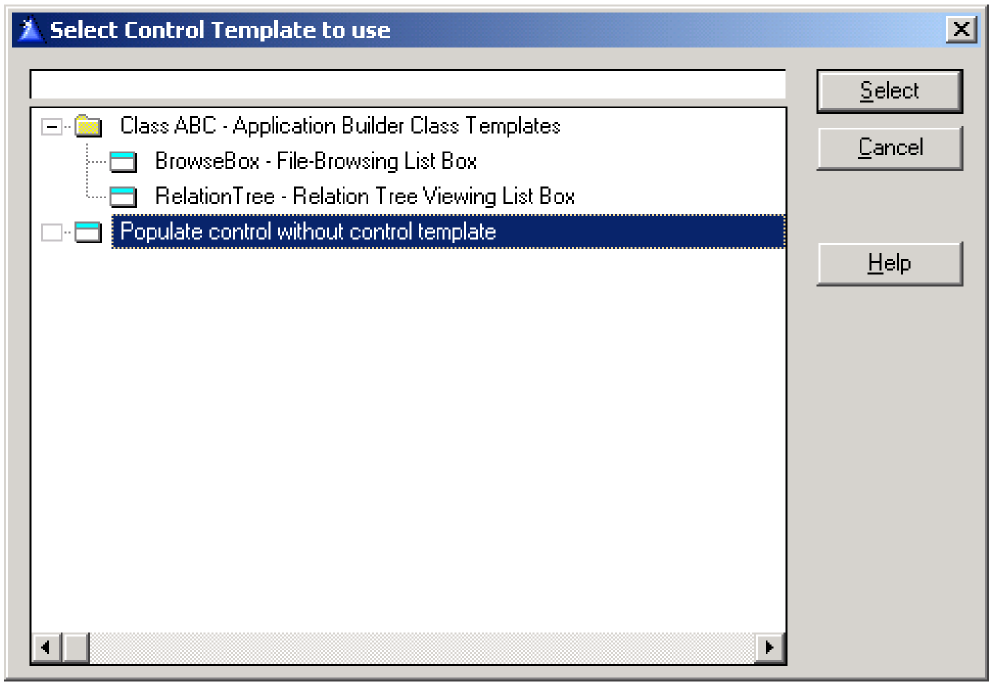

Select Control Template to use
Class ABC - Application Builder Class Templates
BrowseBox - File-Browsing List Box
RelationTree - Relation Tree Viewing List Box
Populate control without control template
Select
Cancel
Help

Figure 2. Populating a basic listbox control

Now select the column `MyQueue:Display` to populate in the listbox:

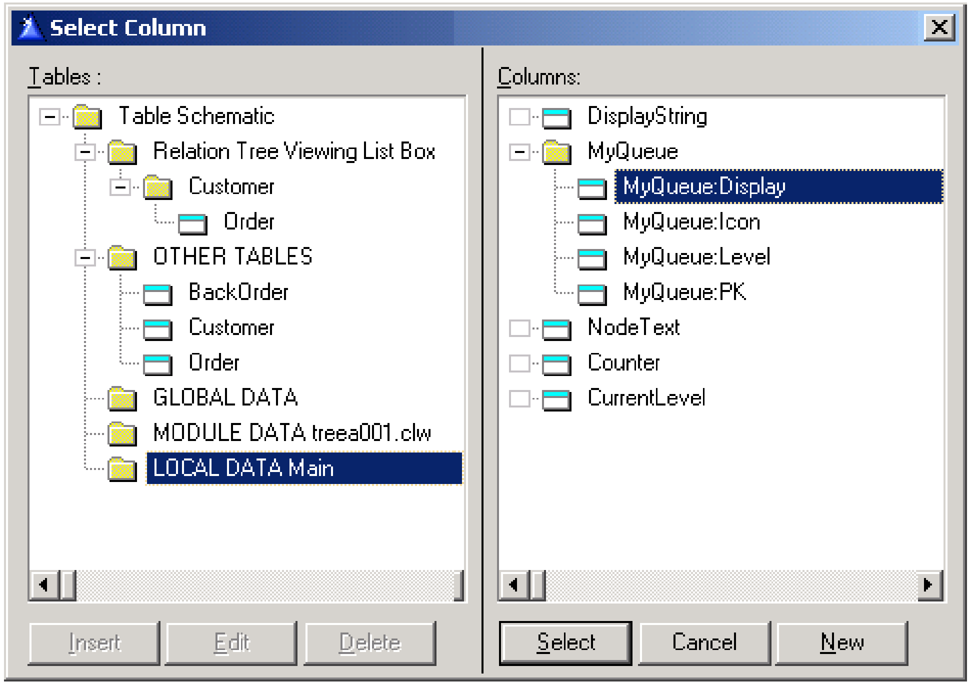
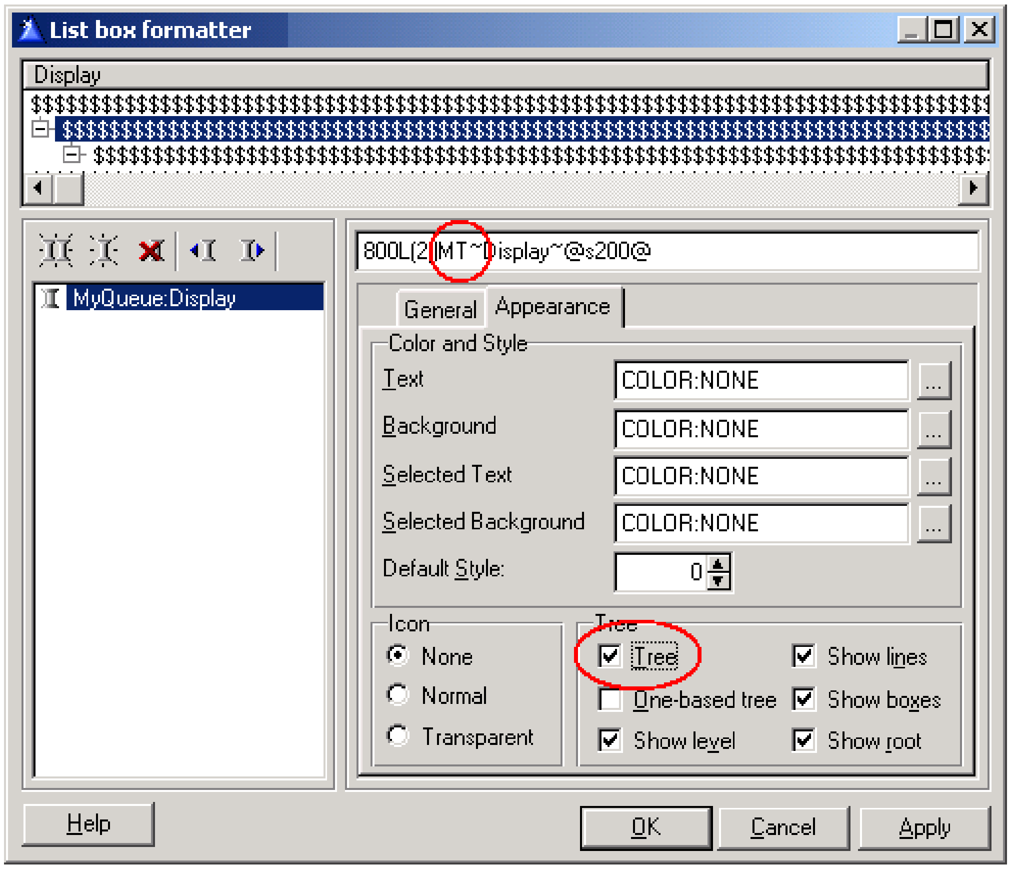


Figure 3. Select the column to display in the listbox

Now click the Appearance Tab for that column set the Tree checkbox value to checked:



List box formatter

Display

800L(2)|MT~Display~@s200@

MyQueue:Display

General | Appearance

Color and Style

Text	COLOR:NONE
Background	COLOR:NONE
Selected Text	COLOR:NONE
Selected Background	COLOR:NONE
Default Style:	0

Icon: [✓ None] Normal Transparent

Tree: [✓ Tree] [✓ Show lines] One-based tree [✓ Show boxes] [✓ Show level] [✓ Show root]

Figure 4. setting the listbox to display as a tree

Click OK (Save listbox formatter changes), right click on the listbox and click Properties on the popup menu. In the General Tab, set the From value to MyQueue.

The formatted window data should now look like this:

```
Window WINDOW('Caption'),AT(,,400,184),|
    FONT('Arial',8,,),GRAY
  LIST,AT(41,9,210,139),USE(?List1),|
    FORMAT('800L(2)|MT~Display~@s200@'),|
    FROM(MyQueue),#ORIG(?List1),  |
    #FIELDS(MyQueue:Display)
END
```

You need code to add items to this queue: place a button on the window with "Load Data" as its caption and embed this code to execute when the button is accepted:

```
MyQueue:Display = 'zero'
MyQueue:Level = 0
Add(MyQueue)
MyQueue:Display = 'one'
MyQueue:Level = 1
Add(MyQueue)
MyQueue:Display = 'two'
MyQueue:Level = 2
Add(MyQueue)
MyQueue:Display = 'three'
```

```
MyQueue:Level = 3
Add(MyQueue)
```

Run the program and click on the button, and you should have something that looks like Figure 5:

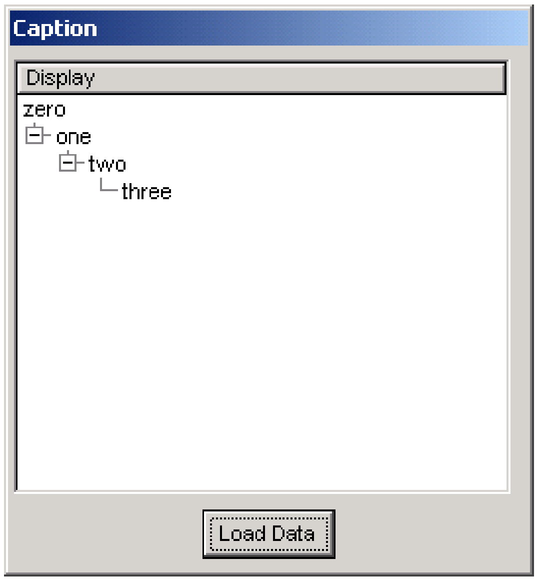
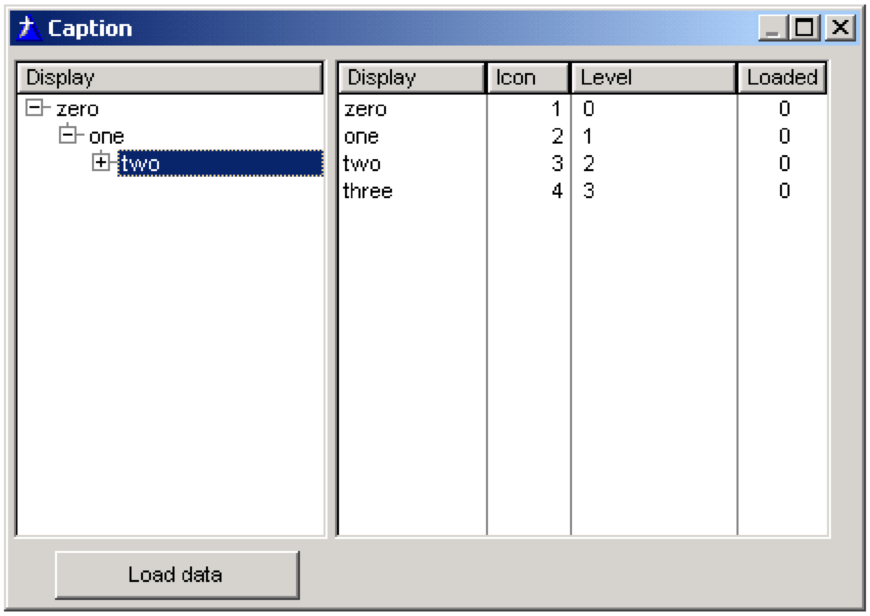


Figure 5. After adding a root and three nodes

This is the essence of the tree control – you have created a queue, bound it to a tree control and populated it with data. The buck does not stop here, however. There are issues to be resolved like expanding and contracting nodes, being able to add, change and delete nodes, displaying node icons, database driven node creation and filters. Lets look at these one by one:

To get a better picture of what is happening with the queue while navigating the tree, populate another listbox next to the tree with all the same attributes except the "Tree" attribute, but this time populate all the columns (setting the Display width to 50) Your window should now look like Figure 6.



Display
zero
one
two

Display	Icon	Level	Loaded
zero	1	0	0
one	2	1	0
two	3	2	0
three	4	3	0

Figure 6. Two listboxes on one window, one with tree attribute, one without, but using the same queue.

You may notice that in the tree, there only appear to be three lines in the Tree whereas in the listbox on the right there are still four items in the queue. Because both controls are showing the same queue, this shows that the only difference between a tree listbox and a standard listbox is the manner in which the queue data is presented. Hidden nodes only *seem* to appear from nowhere when the parent node is expanded.

Changing node properties

Retrieving a node with a mouse click and changing its properties is a fundamental aspect of tree management. Since this is a queue, the first thing you need to do is make sure the queue pointer is in sync with the currently highlighted listbox row. Add a new button, give it the caption Expand/Contract, and embed the following code:

```
Get(MyQueue,choice(?List))
If MyQueue:Level > 0
   MyQueue:Level = ABS(MyQueue:Level) * -1
else
   MyQueue:Level = ABS(MyQueue:Level)
End
Put(MyQueue)
```

This code will retrieve the queue record relative to the currently highlighted listbox pointer, and if the level of that node is greater than zero (the node is expanded) it will be made negative; if the level is less than zero it will be made positive. This effectively toggles the expanded/contracted state of the node.

Changing the text of a node is just as simple: Create a local variable called `NodeText` and place it under the tree as an entry box. Next to it place a button with the caption Change. In this button, embed the following code:

```
Get(MyQueue,choice(?List))
MyQueue:Display = NodeText
Put(MyQueue)
```

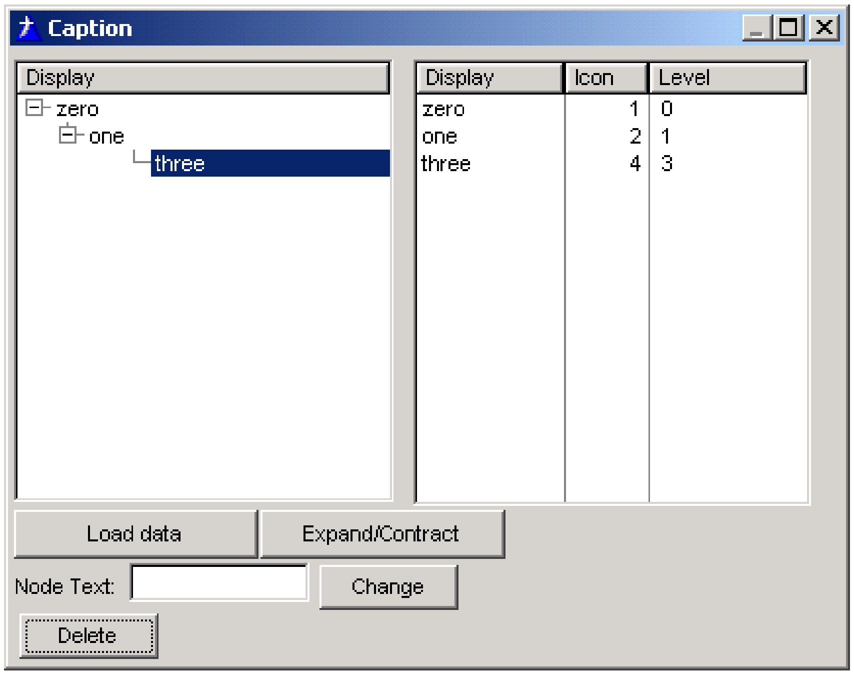
Once again, this retrieves the queue record according to its position relative to the listbox pointer, the value is changed and then saved to the queue. That was easy. Now, how about deleting items?

Deleting Tree Nodes

To delete a node place a button under the tree control with the caption "Delete" and embed the following code:

```
Get(MyQueue,choice(?List))
Delete(MyQueue)
```

Now, run the program, click on the Load Data button, highlight the last node and click Delete. The node disappears. That was easy, right? Not quite. Run the program again then click on the node that says "two". Delete it. Your listbox should now look like Figure 7:



Display	Icon	Level
zero	1	0
one	2	1
three	4	3

Figure 7. The tree list after deleting a node in the middle of a queue

Do you see the problem? Node number two was correctly deleted, but its child (three) was left dangling. That is certainly incorrect – the child node should have been removed automatically. Clearly there is no such thing as an automatic cascading delete with a tree. To do this, it will be necessary to remove each child node explicitly. Do this by looping till the end of the current branch and then looping backwards by the same number of nodes, deleting each one as you go. The last node in a branch is indicated by the fact that the one following it will have a level less or equal to the level of the node you are trying to delete.

Using Icons

Using icons in a listbox is not as straightforward as it may appear, primarily because a row (or tree node) is given an icon *number* rather than the icon name. Clarion automatically allocates the icon number based on the order in which the icon was associated with the listbox control using property syntax.

Huh? All right – here's an example to clarify. Embed the following in a button labeled "Assign icons to listbox".

```
?List{Prop:IconList,1} = 'a.ico'
?List{Prop:IconList,2} = 'b.ico'
?List{Prop:IconList,3} = 'c.ico'
?List{Prop:IconList,4} = 'd.ico'
```

This code associates each one of those icons to the listbox and each icon may be referenced using its respective number.

Now go and change the code that actually adds the rows to the queue (the button labeled Load Data) to the following:

```
MyQueue:Display  = 'zero'
MyQueue:Level    = 0
MyQueue:Icon     = 1     ! new code
Add(MyQueue)
MyQueue:Display  = 'one'
MyQueue:Level    = 1
MyQueue:Icon     = 2     ! new code
Add(MyQueue)
MyQueue:Display  = 'two'
MyQueue:Level    = 2
MyQueue:Icon     = 3     ! new code
Add(MyQueue)
MyQueue:Display  = 'three'
MyQueue:Level    = 3
MyQueue:Icon     = 4     ! new code
Add(MyQueue)
```

All you have done at this point is state that node 1 will use icon number 1, which is "a.ico". Node 2 will use icon number 2, which is "b.ico" etc. Compile and run the application and you will see that each node displays its own icon.

That takes care of the basics of tree lists. Next week I'll look at the ABC Relational Tree template, and show how to handle a non-standard type of file self-relationship.

James Cooke has been using Clarion since 2.1 days and has been a die hard for "the cause" ever since. He and his family recently moved from South Africa to

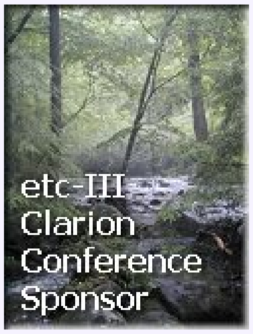
Texas and is currently working in the banking industry. He spends most of his free time basking in the sun by the pool with a good book or succumbing to that hard-to-kick addiction that persistently haunts the Western cosmopolitan neighborhoods - the yard sale.

Reader Comments

Add a comment

Reborn Free CLARION online published by CoveComm Inc.

Clarion MAGAZINE



Using Dynamic Indexes With TPS Files

by Bill Florek

Published 2001-06-08

Dynamic indexes are often overlooked as a way to efficiently access data from TopSpeed (TPS) files, especially if you are dealing with files that hold large numbers of records and a custom sort order and filtered subset is required. By using a dynamic index, you can eliminate the need to create additional file keys.

You can retrieve data from a TopSpeed file by either sequential or random access. However, to access records in some specified sort order, you are limited in the options that are available to you. You can use keys, views, or dynamic indexes.

Keys may be used to read data in a predefined sequence. This methodology is very fast, regardless of the number of records in the file. However, if you need to filter the data on fields other than the key elements, you must read all records from the file to determine which records do not belong to the filtered data subset.

Views may be used to create a user-defined sort sequence. Also, if a filter is applied to a view, only the records that match the filter criteria are returned. The problem with this methodology is speed. If a data file contains a few hundred to a few thousand records, this is a viable option. However, when dealing with tens or hundreds of thousands of records, views give the illusion that the application is "locked-up".

It should be noted that *generated* browse, report and process procedures use views to access data. Views, when coupled with a key from the primary file, produce acceptable results, as long as record filtering on fields other than the key elements is not used.

Keys and views are not the only options available to you. Dynamic Indexes, or "static keys", incorporate the features of keys and views into a single structure.

Dynamic index basics

Before you can use a dynamic index with a TPS file, you must declare the index as part of the file structure. This is accomplished by creating a file key and selecting Runtime Index as its type.

Before using a dynamic index, you need to *build* the index. . The following defines the Clarion language `BUILD` statement as it pertains to creating a dynamic index:

> BUILD (*index* , *components* , *[filter]*) where:
>
> > *index* is the label of the dynamic index.
> >
> > *components* is a comma-delimited list of fields to sort on
> >
> > *filter* is an optional expression to filter the records

See the Clarion language reference for a complete description of the BUILD statement.

Dynamic indexes create a temporary file that is exclusive to the user who built it (when the file is closed, the temporary file is deleted). This allows multiple users to create indexes specific to their needs without affecting anyone else. However, because an index is a *static* structure, updates to the file are not reflected in the index after it is built.

After the index is built, you may use it to access records in a sequential or random access manner. The `RECORDS()` function will return the actual number of records in the index, which is very useful when creating a process procedure that uses a progress bar.

> Note: when using Clarion versions after 5501, the `RECORDS()` function returns the total number of records in the index *plus* the records in the file.

If an application is using the legacy templates, dynamic indexes may be used as the key on generated procedures such as browses and reports. However, if an index is used on a browse, the locator must be set to NONE. If a locator is required, you must handle it manually (that is, hand-code it).

In the ABC templates, generated procedures will not use a dynamic index. If you specify an index as the *key*, no sort order will be used. The reason ABC template procedures do not directly support runtime indexes lies in the `FileManager` class. When a file is "registered" with the `FileManager`, the file's keys and associated key fields are saved using the `FileManager.AddKey` method. This method retrieves a key's component fields using the key property `PROP:Components`. Since a dynamic index has no fields defined until the index is built, the `FileManager` has no components to register. The key definition stored by the `FileManager` is used when setting sort orders, range limiting files or processing locators. Therefore, since no component fields are initially defined for a dynamic index, the `FileManager` does not know how to what indexed fields it is dealing with. This would be similar to creating a generated browse procedure where the primary file for the browse does not use a key but has "additional sort fields" defined. The view engine has to handle the record sorting internally.

Why use a dynamic index?

If the generated procedures using the ABC templates do not allow the use of developer-defined runtime indexes, what possible use could there be for them?

The volume of data stored in today's business applications continues to grow, and files with hundreds of thousands to millions of records are becoming very common. When an application is first designed, it is almost impossible and definitely impractical to incorporate every conceivable sort order that may ever be required by the application into a file definition. However, by adding a dynamic index to the file definition, you essentially eliminate this problem. Remember that this discussion on dynamic indexes applies to TPS files and not SQL databases, although indexes are vital there as well.

As I stated earlier, views suffer from a speed problem in situations that require record filtering. The following example illustrates this fact:

Test file	two fields defined as string(10) with 262,000 records and no keys
Test Criteria	sort on field1 and filter on field2 by using SUB(field2,1,1) = 'M', resulting in 468 selected records
Results: Dynamic Index	1.08 seconds
Results: View using Order and Filter	9.09 seconds
Results: File using Sequential Access	4:06 minutes

This simple test shows that a dynamic index is much faster than a view, and processing a file sequentially should not even be considered unless the order is unimportant and few records will be filtered out. Although this is a very simple test, the same type of result holds true when very complex file structures are used.

Therefore, if you are presented with a situation that requires sorting and filtering a file's records so that they can be processed in some manner, and the file has a large number of records, a dynamic index may very well be the perfect solution.

Typical use

Dynamic indexes can be substituted for keys or views in almost any situation. One of the deciding factors in whether or not an index should be used is the number of records in the file, although performance will generally be better using a runtime index. As I mentioned earlier, the ABC template generated procedures do not directly support dynamic indexes, so the developer (a.k.a. programmer) will need to do something that is becoming more foreign every day: *write code*.

To illustrate a simple, yet powerful use of runtime indexes, look at the following pseudo-code, which uses a BUILD statement to determine exactly which records a report will process, and in what order:

```
Access:file.open
Access:file.usefile
Open(ProgressWindow)
Display
Open(Report)
Build(DynNdx,sortorder,filter)
ProgressBar{prop:rangehigh} = |
  records(DynNdx) – records(File)
Set(DynNdx)
Loop
  If Access:file.next() then break.
  ProgressBar{prop:progress} = |
    ProgressBar{prop:progress} + 1
  Print(ReportDetailBand)
End
Close(Report)
Access:file.close
```

In this example, a report may be printed in *any* sort order and filtered on *any* fields. Simply set the sortorder and filter parameters of the BUILD statement to whatever is required to generate the report. Also, as a side benefit, the progress bar is truly accurate. (Note: when calculating the records contained in the dynamic index, remember to subtract the file record count when using Clarion versions after 5501, as shown in the listing) Although you could use a view with order and filter properties, the report would take much longer to generate and the end-user would not be informed as to the *true* progress of the report.

By simply replacing the "report specific" code (such as the print statement) with some other type of processing code, you can accomplish any record-specific task.

As is evident in the example, no range or filter checking exists in the main processing loop. Since all filtering, which is synonymous with range checking, is done in the BUILD statement, none of this code needs to written. On this premise, multi-file filtering becomes a simple task with very little additional coding required. For example, the following code will process `file1` in some key order and only include records on the report if a related record exists in the filtered subset of records in `file2`:

```
Access:file1.open
Access:file1.usefile
Access:file2.open
Access:file2.usefile
Open(ProgressWindow)
Display
Open(Report)
Build(File2DynNdx,sortorder,filter)
ProgressBar{prop:rangehigh} |
  = records(DynNdx) – records(File)
Set(File1Key)
Loop
  If Access:file1.next() then break.
  ProgressBar{prop:progress} = |
    ProgressBar{prop:progress} + 1
  File2.DynNdxSortField = File1.RelatedField
  Set(File2DynNdx, File2DynNdx)
  If ~Access:file2.next() AND |
    File2.DynNdxSortField = File1.RelatedField
    Print(ReportDetailBand)
  End
End
Close(Report)
Access:file1.close
Access:file2.close
```

The statement

```
If ~Access:file2.next() AND |
  File2.DynNdxSortField = File1.RelatedField
```

takes into account that there may be multiple `file2` records that match the `file1` related field. The purpose of this type of coding technique is not to process (or in this case, print) `file2` records, but to include `file1` records in the result set if any related record exists in the filtered subset of `file2`.

By replacing this statement with

```
If ~Access:file2.fetch(File2DynNdx)
```

and removing the `SET(File2DynNdx, File2DynNdx)` statement, a unique relationship between `file1` and `file2` is accomplished. The following example illustrates this technique:

- `file1` is an invoice header file that contains a customer code that relates to a customer file
- `file2` is a customer file that contains various customer information
- a dynamic index is built on the customer file in customer

code order and only includes records that have a specific zip code

- When processing through `file1`, records (invoices) may be included or omitted from processing based on whether or not the `file1.fetch(DynNdx)` is successful.

The methodology presented in this example may be easily adapted to ABC generated procedures. To do this, place the file with the dynamic index into the procedure's file schematic under Other Files. Do not place it under the primary file as a related file (it would become part of the generated view and the purpose of the dynamic index would be defeated). In the embeds for the procedure, place the build index statement(s) after the files have been opened in `INIT`. The record checking code that uses the index could be placed into a variety of places, such as the `ValidateRecord` or `TakeRecord` embeds..

Summary

Dynamic indexes, also known as runtime indexes, can be a useful tool when dealing with TPS files that hold a large number of records. When you need a custom sort order and/or filtered subset of records for a processing task, a dynamic index will generally produce much faster results than a view. However, to be able to realize these benefits, you must first overcome the fear of hand coding a procedure.

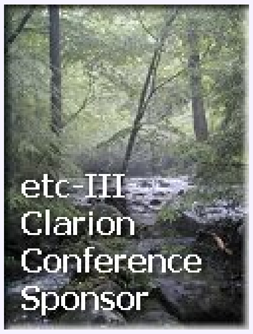
Bill Florek is an Electrical Engineer who has been using Clarion since the first release of the 2.0 DOS version. For the last 16 years, he has developed (and redeveloped) an A/R system for the health care industry that includes extensive use of EDI files in various flavors, mainly for insurance claim transmittals. Many of Bill's electrical circuit designs have been patented, and he has also designed various PC add-on cards for specific engineering purposes, with control software written in Clarion. In his spare time Bill plays snare drum with a competition-level Scottish Pipe and Drum band, and used his own all-Clarion software to create sheet music and MIDI playback files for Celtic-style drumming.

Reader Comments

Add a comment

Outstanding Article!!! extremely timely , worth the cost of...
Very good article: I've used Dynamic Indexes to some...
To answer Jim's question: Dynamic Indexes are specific...

Reborn Free
CLARION online
published by CoveComm Inc.
Clarion MAGAZINE



Understanding Stack And Heap Memory In 32 Bit Clarion Applications

by John Gorter

Published 2001-06-05

In Clarion, as well as in C and C++ (and unlike Java), you need to be aware of possible memory leaks and thus be aware of the side effects of declaring variables. There are three pools of memory where any of your declared variables can reside:

The Stack	This is where the local variables reside for the duration of the procedure call (the scope).
The Heap	Variables allocated reside here until they're explicitly freed or de-allocated. MS Windows32 mentions two types of heap in the article Managing Heap Memory in Win32, by Randy Kath, which can be found in the MSDN. I won't elaborate on this as the differences are of no relevance to this article.
Static Storage	This is where the global and static variables reside while the program is executing.

The Stack

Every procedure or function has its own stack. The stack is a piece of memory used to store information related to procedure or method calls. When a procedure is called, the system places the return address of the next machine instruction to execute on the stack so the CPU can go back were it came from and continue, after the procedure finishes executing. The stack is a single contiguous, strictly linear block of memory. Besides the bytes it contains, a stack has a pointer that indicates which memory address is the top of the stack.

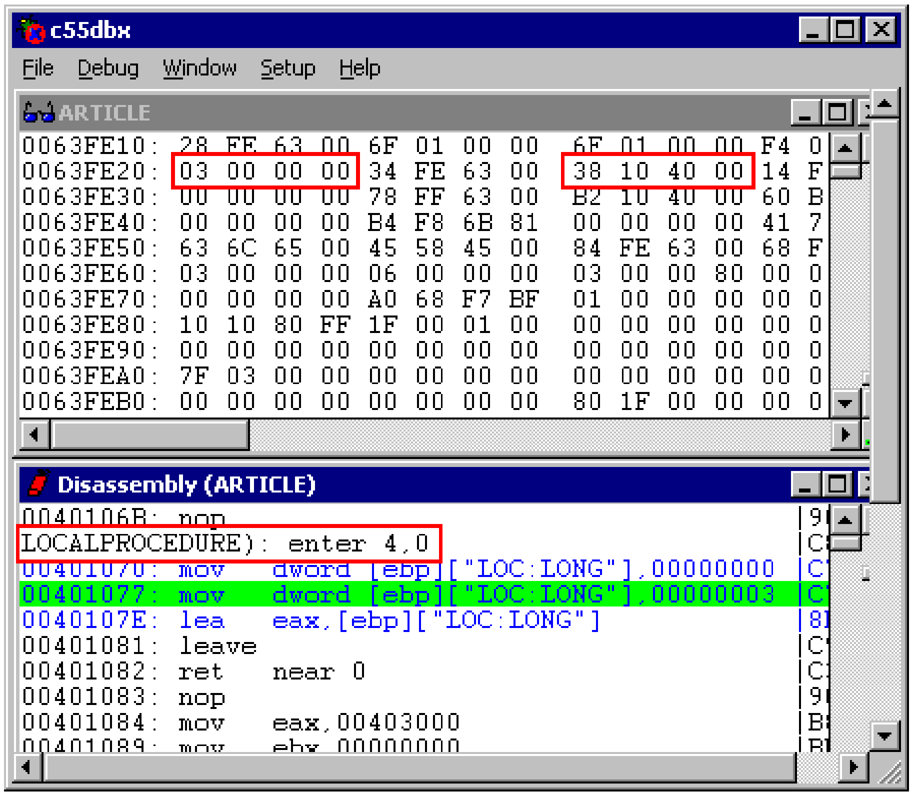
Whenever a procedure is called, a *stack frame* is pushed onto the stack, which includes room for all of the procedure's parameters, local variables and the return address (and possibly other information). The size of the stack frame is partly calculated by the number and type of declared local variables. When local variables are larger than a certain size, they are automatically allocated on the heap, but as I want to show the different allocation methods I won't use extraordinary large variables in the example code.

Whenever a procedure returns, it de-allocates its own stack frame by re-adjusting the stack pointer to the position where it was before the procedure was called. Procedures called later can and will overwrite the local variables of the terminated procedure. This re-adjusting of the stack pointer is also known as *stack-unwinding* or simply *unwinding the stack*.

Here's a small example program which calls a procedure and assigns a `LONG` to a `&LONG`:

```
  program

  map
    LocalProcedure (), long, pascal
  end

glo:long &long

  code
  glo:long &= (localprocedure())
  stop(glo:long)
  stop(glo:long)

LocalProcedure procedure()
loc:long long
  code
  loc:long = 3
  return address(loc:long)
```

This (obvious) example shows a problem one might encounter. When `LocalProcedure` is called, room for variable `loc:long` is allocated on the stack. Lets see what happens. As Figure 1 shows, `LocalProcedure` is initiated, the return address is stored on the stack (`00401038`), and room is allocated for local variables (enter 4,0).

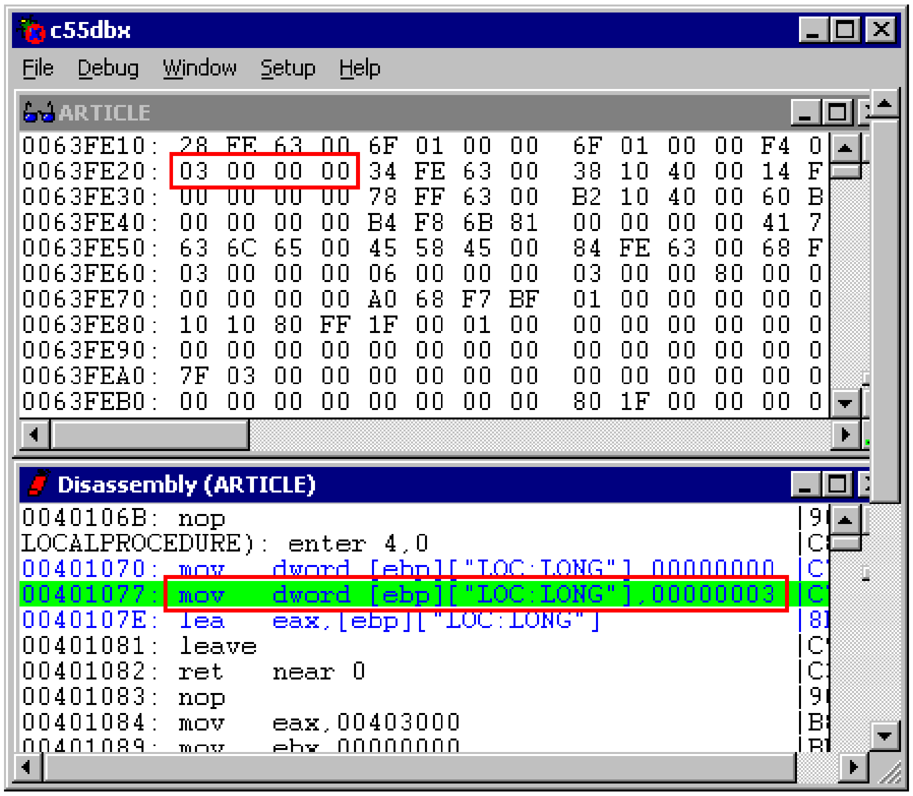


```
0063FE10: 28 FE 63 00 6F 01 00 00  6F 01 00 00 F4 0
0063FE20: 03 00 00 00 34 FE 63 00  38 10 40 00 14 F
0063FE30: 00 00 00 00 78 FF 63 00  B2 10 40 00 60 B
0063FE40: 00 00 00 00 B4 F8 6B 81  00 00 00 00 41 7
0063FE50: 63 6C 65 00 45 58 45 00  84 FE 63 00 68 F
0063FE60: 03 00 00 00 06 00 00 00  03 00 00 80 00 0
0063FE70: 00 00 00 00 A0 68 F7 BF  01 00 00 00 00 0
0063FE80: 10 10 80 FF 1F 00 01 00  00 00 00 00 00 0
0063FE90: 00 00 00 00 00 00 00 00  00 00 00 00 00 0
0063FEA0: 7F 03 00 00 00 00 00 00  00 00 00 00 00 0
0063FEB0: 00 00 00 00 00 00 00 00  80 1F 00 00 00 0
```

```
0040106B: nop                                 |9
LOCALPROCEDURE): enter 4,0                    |C
00401070: mov  dword [ebp]["LOC:LONG"],00000000 |C
00401077: mov  dword [ebp]["LOC:LONG"],00000003 |C
0040107E: lea  eax,[ebp]["LOC:LONG"]          |8
00401081: leave                               |C
00401082: ret  near 0                         |C
00401083: nop                                 |9
00401084: mov  eax,00403000                   |B
00401089: mov  ebx,00000000                   |B
```

Figure 1. Disassembly of the call to localprocedure (part1).

Next, the value 3 is stored in the local variable `loc:long` at address `0063FE20h`.

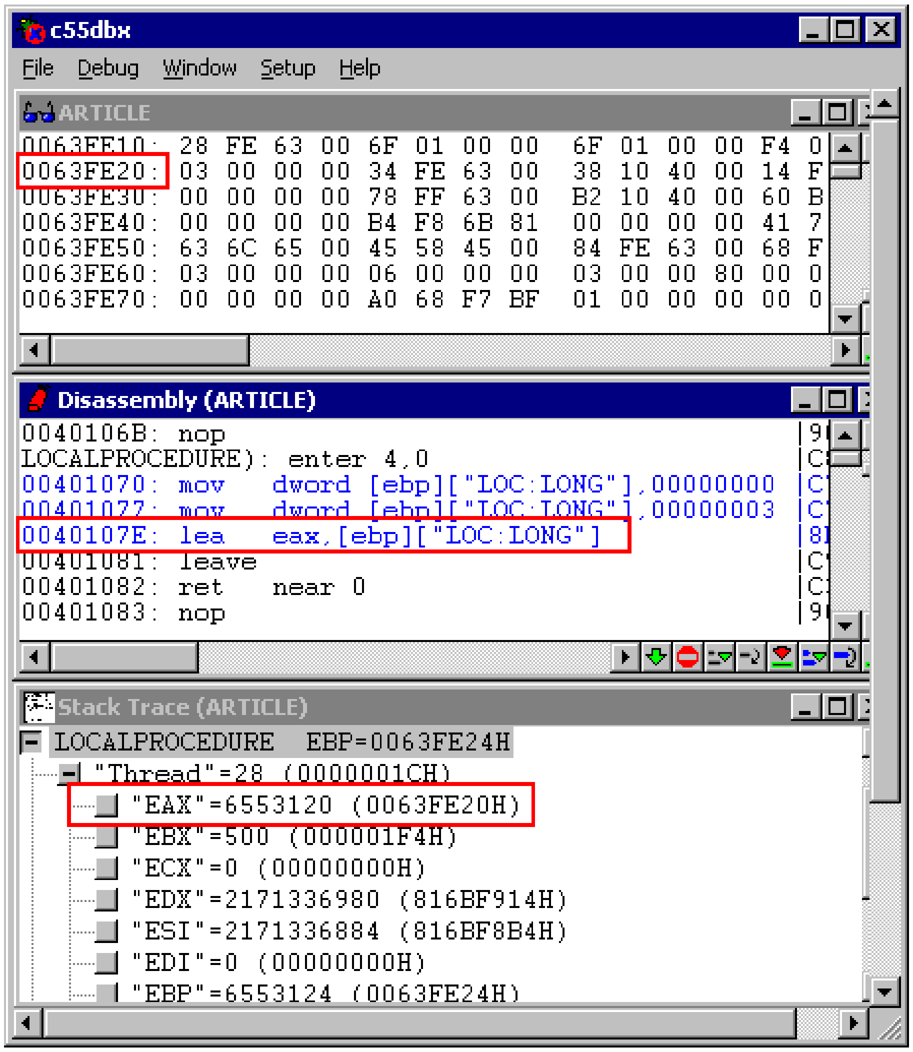


```
0063FE10: 28 FE 63 00 6F 01 00 00  6F 01 00 00 F4 0
0063FE20: 03 00 00 00 34 FE 63 00  38 10 40 00 14 F
0063FE30: 00 00 00 00 78 FF 63 00  B2 10 40 00 60 B
0063FE40: 00 00 00 00 B4 F8 6B 81  00 00 00 00 41 7
0063FE50: 63 6C 65 00 45 58 45 00  84 FE 63 00 68 F
0063FE60: 03 00 00 00 06 00 00 00  03 00 00 80 00 0
0063FE70: 00 00 00 00 A0 68 F7 BF  01 00 00 00 00 0
0063FE80: 10 10 80 FF 1F 00 01 00  00 00 00 00 00 0
0063FE90: 00 00 00 00 00 00 00 00  00 00 00 00 00 0
0063FEA0: 7F 03 00 00 00 00 00 00  00 00 00 00 00 0
0063FEB0: 00 00 00 00 00 00 00 00  80 1F 00 00 00 0
```

```
0040106B: nop                                 |9
LOCALPROCEDURE): enter 4,0                    |C
00401070: mov  dword [ebp]["LOC:LONG"],00000000 |C
00401077: mov  dword [ebp]["LOC:LONG"],00000003 |C
0040107E: lea  eax,[ebp]["LOC:LONG"]          |8
00401081: leave                               |C
00401082: ret  near 0                         |C
00401083: nop                                 |9
00401084: mov  eax,00403000                   |B
00401089: mov  ebx,00000000                   |B
```

Figure 2. Instruction storing value in local variable.

The return value, the address of `loc:long` at address `0063FE20`, is stored in register EAX, the register that holds the return value.



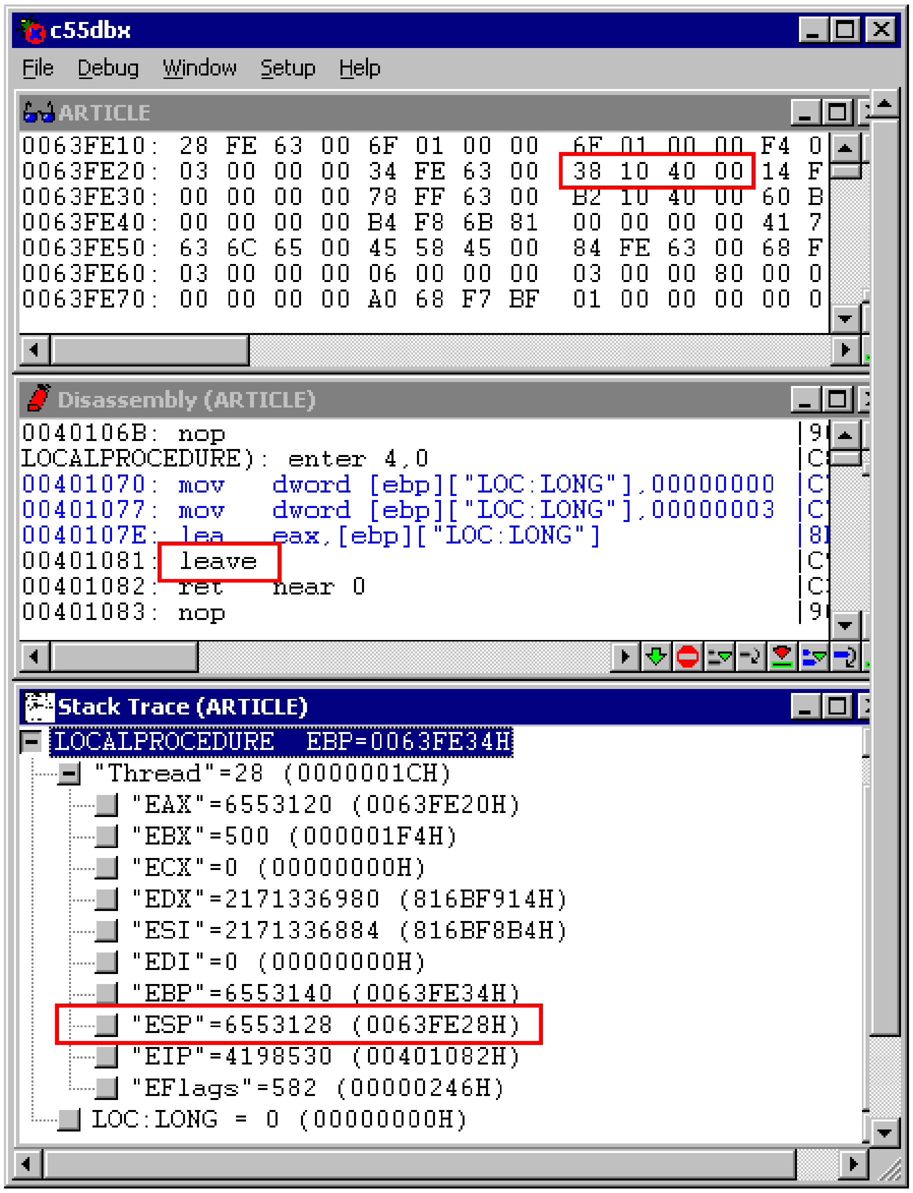
```
0063FE10: 28 FE 63 00 6F 01 00 00  6F 01 00 00 F4 0
0063FE20: 03 00 00 00 34 FE 63 00  38 10 40 00 14 F
0063FE30: 00 00 00 00 78 FF 63 00  B2 10 40 00 60 B
0063FE40: 00 00 00 00 B4 F8 6B 81  00 00 00 00 41 7
0063FE50: 63 6C 65 00 45 58 45 00  84 FE 63 00 68 F
0063FE60: 03 00 00 00 06 00 00 00  03 00 00 80 00 0
0063FE70: 00 00 00 00 A0 68 F7 BF  01 00 00 00 00 0
```

```
0040106B: nop                                          |9
LOCALPROCEDURE): enter 4,0                             |C
00401070: mov    dword [ebp]["LOC:LONG"],00000000      |C
00401077: mov    dword [ebp]["LOC:LONG"],00000003      |C
0040107E: lea    eax,[ebp]["LOC:LONG"]                 |8
00401081: leave                                        |C
00401082: ret    near 0                                |C
00401083: nop                                          |9
```

```
LOCALPROCEDURE   EBP=0063FE24H
  "Thread"=28 (0000001CH)
    "EAX"=6553120 (0063FE20H)
    "EBX"=500 (000001F4H)
    "ECX"=0 (00000000H)
    "EDX"=2171336980 (816BF914H)
    "ESI"=2171336884 (816BF8B4H)
    "EDI"=0 (00000000H)
    "EBP"=6553124 (0063FE24H)
```

Figure 3. Arranging the return value.

The leave instruction restores the ESP (SP = Stack Pointer) to point to the return address 00401038.



```
0063FE10: 28 FE 63 00 6F 01 00 00  6F 01 00 00 F4 0
0063FE20: 03 00 00 00 34 FE 63 00  38 10 40 00 14 F
0063FE30: 00 00 00 00 78 FF 63 00  B2 10 40 00 60 B
0063FE40: 00 00 00 00 B4 F8 6B 81  00 00 00 00 41 7
0063FE50: 63 6C 65 00 45 58 45 00  84 FE 63 00 68 F
0063FE60: 03 00 00 00 06 00 00 00  03 00 00 80 00 0
0063FE70: 00 00 00 00 A0 68 F7 BF  01 00 00 00 00 0
```

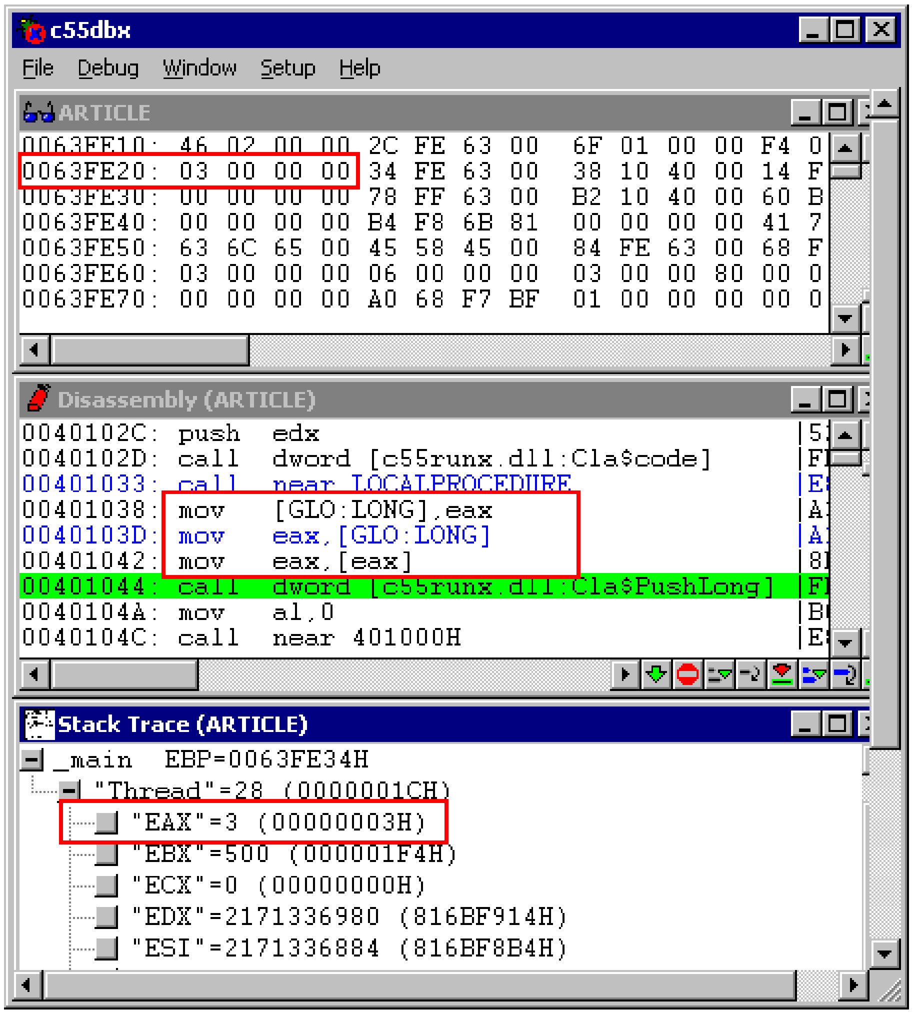
```
0040106B: nop                                     |9
LOCALPROCEDURE): enter 4,0                        |C
00401070: mov   dword [ebp]["LOC:LONG"],00000000  |C
00401077: mov   dword [ebp]["LOC:LONG"],00000003  |C
0040107E: lea   eax,[ebp]["LOC:LONG"]             |8
00401081: leave                                   |C
00401082: ret   near 0                            |C
00401083: nop                                     |9
```

```
LOCALPROCEDURE   EBP=0063FE34H
  "Thread"=28 (0000001CH)
    "EAX"=6553120 (0063FE20H)
    "EBX"=500 (000001F4H)
    "ECX"=0 (00000000H)
    "EDX"=2171336980 (816BF914H)
    "ESI"=2171336884 (816BF8B4H)
    "EDI"=0 (00000000H)
    "EBP"=6553140 (0063FE34H)
    "ESP"=6553128 (0063FE28H)
    "EIP"=4198530 (00401082H)
    "EFlags"=582 (00000246H)
  LOC:LONG = 0 (00000000H)
```

Figure 4. Stackpointer re-adjustment.

The procedure finalizes by unwinding the stack and returning the address of `loc:long`, which in turn is assigned to `glo:long`. The first `STOP()` receives a parameter with the value equal to the value pointed to by `glo:long`, which at the moment of calling is still 3.

As Figure 5 shows, just before the call to `STOP()`, the value `glo:long` points to is stored in the EAX register. Note that it is not overwritten yet and still contains the value 3!



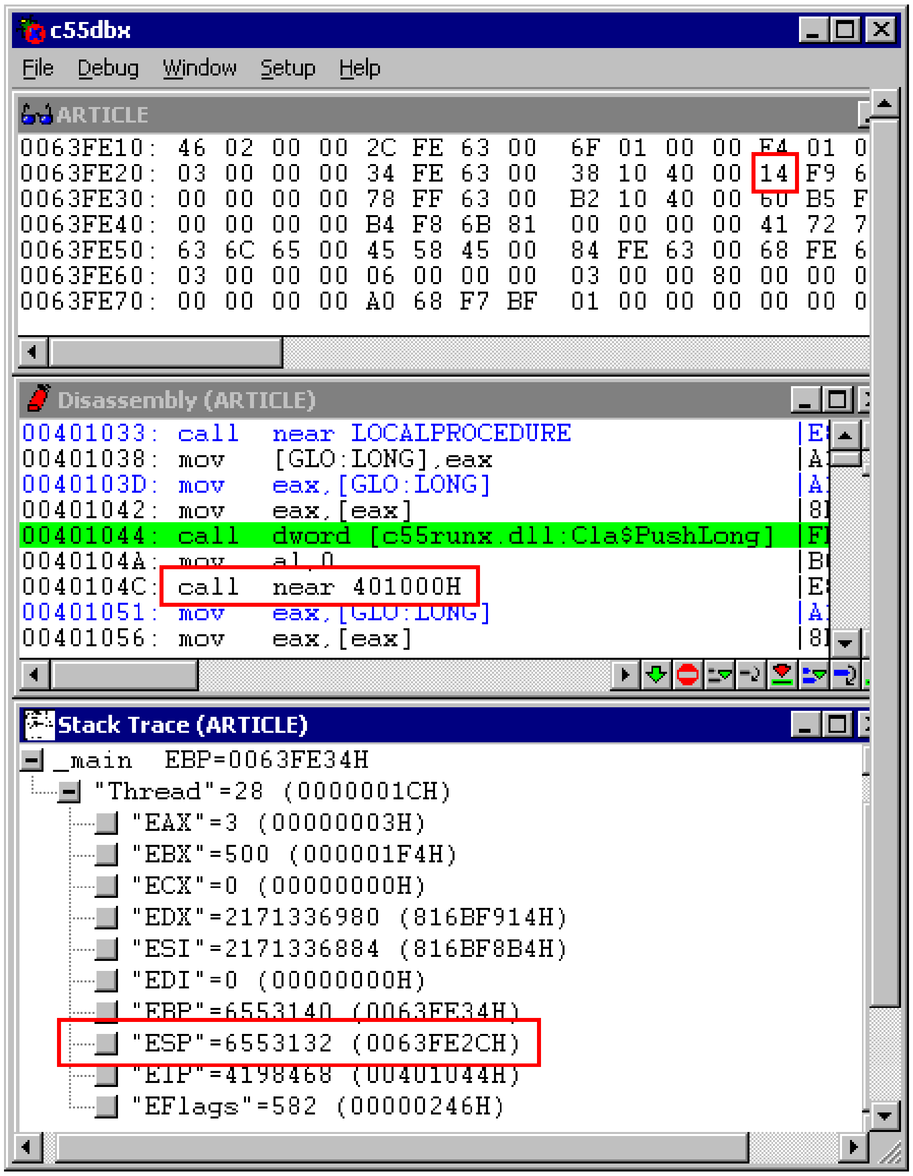
```
0063FE10:  46 02 00 00 2C FE 63 00  6F 01 00 00 F4 0
0063FE20:  03 00 00 00 34 FE 63 00  38 10 40 00 14 F
0063FE30:  00 00 00 00 78 FF 63 00  B2 10 40 00 60 B
0063FE40:  00 00 00 00 B4 F8 6B 81  00 00 00 00 41 7
0063FE50:  63 6C 65 00 45 58 45 00  84 FE 63 00 68 F
0063FE60:  03 00 00 00 06 00 00 00  03 00 00 80 00 0
0063FE70:  00 00 00 00 A0 68 F7 BF  01 00 00 00 00 0
```

```
0040102C:  push   edx                               |5
0040102D:  call   dword [c55runx.dll:Cla$code]      |F
00401033:  call   near LOCALPROCEDURE               |E
00401038:  mov    [GLO:LONG],eax                    |A
0040103D:  mov    eax,[GLO:LONG]                    |A
00401042:  mov    eax,[eax]                         |8
00401044:  call   dword [c55runx.dll:Cla$PushLong]  |F
0040104A:  mov    al,0                              |B
0040104C:  call   near 401000H                      |E
```

```
_main  EBP=0063FE34H
  "Thread"=28  (0000001CH)
    "EAX"=3  (00000003H)
    "EBX"=500  (000001F4H)
    "ECX"=0  (00000000H)
    "EDX"=2171336980  (816BF914H)
    "ESI"=2171336884  (816BF8B4H)
```

Figure 5. Passing the parameters to STOP().

The stack pointer in ESP points to the memory address which will be used by the next procedure at address 00401000h. This is illustrated in the following screenshot.



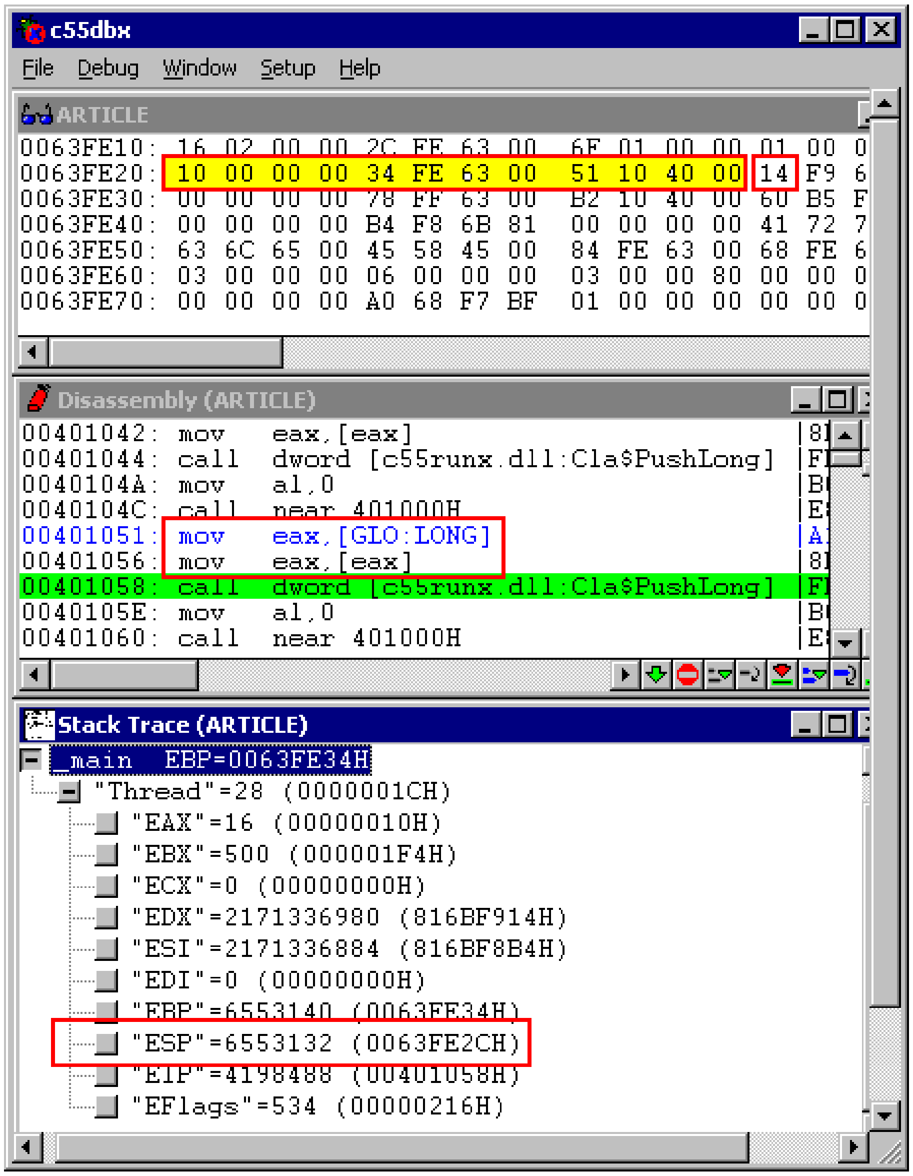
```
0063FE10: 46 02 00 00 2C FE 63 00  6F 01 00 00 F4 01 0
0063FE20: 03 00 00 00 34 FE 63 00  38 10 40 00 14 F9 6
0063FE30: 00 00 00 00 78 FF 63 00  B2 10 40 00 60 B5 F
0063FE40: 00 00 00 00 B4 F8 6B 81  00 00 00 00 41 72 7
0063FE50: 63 6C 65 00 45 58 45 00  84 FE 63 00 68 FE 6
0063FE60: 03 00 00 00 06 00 00 00  03 00 00 80 00 00 0
0063FE70: 00 00 00 00 A0 68 F7 BF  01 00 00 00 00 00 0
```

```
00401033: call  near LOCALPROCEDURE
00401038: mov   [GLO:LONG],eax
0040103D: mov   eax,[GLO:LONG]
00401042: mov   eax,[eax]
00401044: call  dword [c55runx.dll:Cla$PushLong]
0040104A: mov   al,0
0040104C: call  near 401000H
00401051: mov   eax,[GLO:LONG]
00401056: mov   eax,[eax]
```

```
_main  EBP=0063FE34H
  "Thread"=28 (0000001CH)
    "EAX"=3 (00000003H)
    "EBX"=500 (000001F4H)
    "ECX"=0 (00000000H)
    "EDX"=2171336980 (816BF914H)
    "ESI"=2171336884 (816BF8B4H)
    "EDI"=0 (00000000H)
    "EBP"=6553140 (0063FE34H)
    "ESP"=6553132 (0063FE2CH)
    "EIP"=4198468 (00401044H)
    "EFlags"=582 (00000246H)
```

Figure 6. Disassembly of the call to Stop() .

The internal procedure STOP allocates its own local variables etc, executes and returns. Then the next STOP() is called, but now the value pointed to by glo:long is a leftover from the preceding STOP(), in this case 16. After the call to the STOP() procedure, the stack is restored to the value it was before the call, address 0063FE2Ch. Note the leftovers from the procedure call in yellow displayed in the following figure.



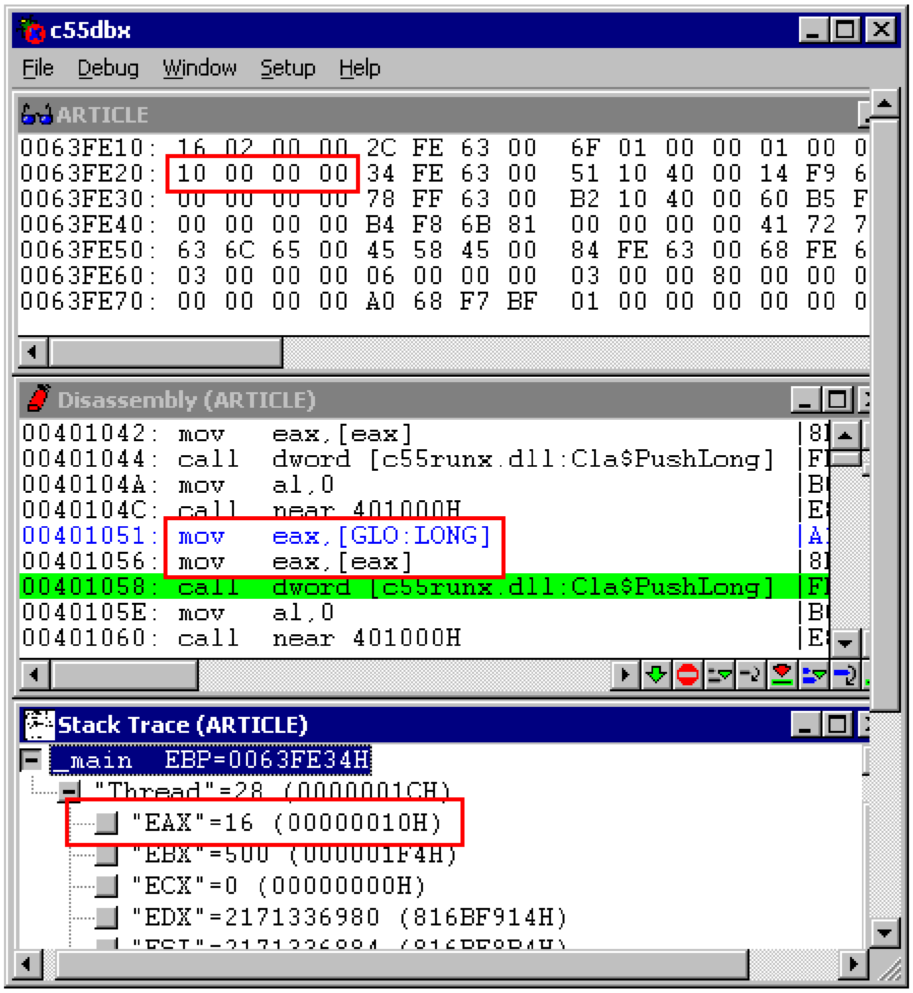
```
0063FE10: 16 02 00 00 2C FE 63 00  6F 01 00 00 01 00 0
0063FE20: 10 00 00 00 34 FE 63 00  51 10 40 00 14 F9 6
0063FE30: 00 00 00 00 78 FF 63 00  B2 10 40 00 60 B5 F
0063FE40: 00 00 00 00 B4 F8 6B 81  00 00 00 00 41 72 7
0063FE50: 63 6C 65 00 45 58 45 00  84 FE 63 00 68 FE 6
0063FE60: 03 00 00 00 06 00 00 00  03 00 00 80 00 00 0
0063FE70: 00 00 00 00 A0 68 F7 BF  01 00 00 00 00 00 0
```

```
00401042: mov   eax,[eax]                          |8
00401044: call  dword [c55runx.dll:Cla$PushLong]   |F
0040104A: mov   al,0                               |B
0040104C: call  near 401000H                       |E
00401051: mov   eax,[GLO:LONG]                     |A
00401056: mov   eax,[eax]                          |8
00401058: call  dword [c55runx.dll:Cla$PushLong]   |F
0040105E: mov   al,0                               |B
00401060: call  near 401000H                       |E
```

```
_main  EBP=0063FE34H
  "Thread"=28 (0000001CH)
    "EAX"=16 (00000010H)
    "EBX"=500 (000001F4H)
    "ECX"=0 (00000000H)
    "EDX"=2171336980 (816BF914H)
    "ESI"=2171336884 (816BF8B4H)
    "EDI"=0 (00000000H)
    "EBP"=6553140 (0063FE34H)
    "ESP"=6553132 (0063FE2CH)
    "EIP"=4198488 (00401058H)
    "EFlags"=534 (00000216H)
```

Figure 7. The memory after the call to Stop().

The value glo:long points to, this time 00000010h, is stored in the EAX register as a parameter to CLA$PushLong, resulting in a dialog box showing the value 16.



```
0063FE10: 16 02 00 00 2C FE 63 00  6F 01 00 00 01 00 0
0063FE20: 10 00 00 00 34 FE 63 00  51 10 40 00 14 F9 6
0063FE30: 00 00 00 00 78 FF 63 00  B2 10 40 00 60 B5 F
0063FE40: 00 00 00 00 B4 F8 6B 81  00 00 00 00 41 72 7
0063FE50: 63 6C 65 00 45 58 45 00  84 FE 63 00 68 FE 6
0063FE60: 03 00 00 00 06 00 00 00  03 00 00 80 00 00 0
0063FE70: 00 00 00 00 A0 68 F7 BF  01 00 00 00 00 00 0
```

```
00401042: mov   eax,[eax]
00401044: call  dword [c55runx.dll:Cla$PushLong]
0040104A: mov   al,0
0040104C: call  near 401000H
00401051: mov   eax,[GLO:LONG]
00401056: mov   eax,[eax]
00401058: call  dword [c55runx.dll:Cla$PushLong]
0040105E: mov   al,0
00401060: call  near 401000H
```

```
_main  EBP=0063FE34H
  "Thread"=28 (0000001CH)
    "EAX"=16 (00000010H)
    "EBX"=500 (000001F4H)
    "ECX"=0 (00000000H)
    "EDX"=2171336980 (816BF914H)
    "ESI"=2171336884 (816BF8B4H)
```

Figure 8. Disassembly of the next call to Stop().

The Heap

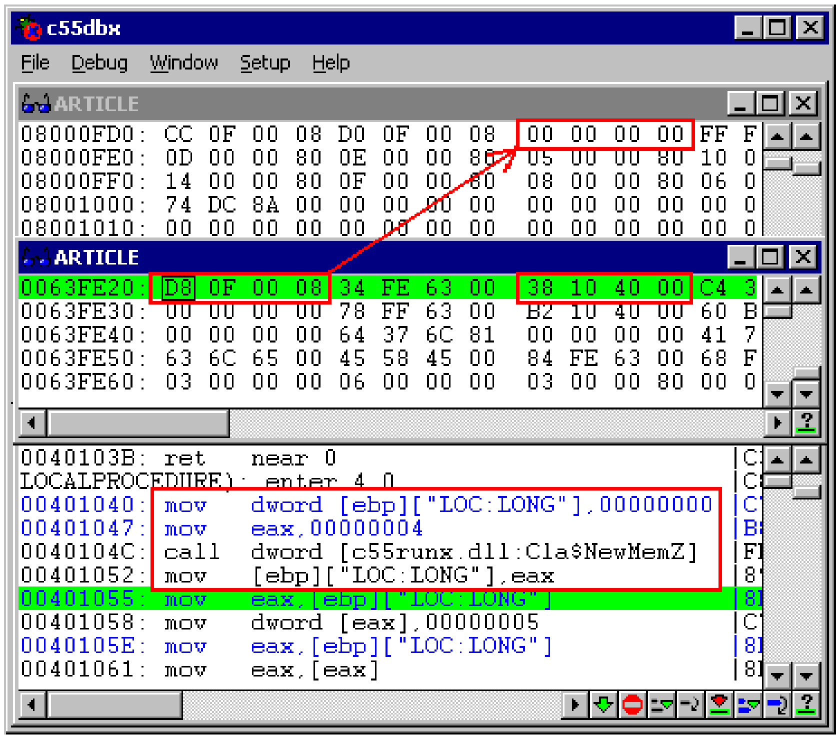
Each process also has its own heap, which is a large pool of memory. A process, in the simplest terms, is an executing program. The heap, unlike the stack, is not allocated in contiguous order. You can allocate heap memory using the NEW() function. The heap manager, which I won't discuss here, allocates the memory and returns a pointer to this memory. The heap is where most memory leaks occur.

Here's another example, this time using the heap.

```
  program

  map
    LocalProcedure (), pascal
  end

  code
  LocalProcedure()

LocalProcedure procedure()
loc:long &long

  code
```

```
loc:long &= new long
loc:long = 5
stop(loc:long)
dispose(loc:long)
```

Figure 8 shows the disassembly of `LocalProcedure`, which starts by storing the return address (`00401038h`), allocating room for the pointer (enter 4,0), and assigning a value to the pointer through a call to `Cla$NewMemZ`.


c55dbx

File Debug Window Setup Help

ARTICLE

```
08000FD0: CC 0F 00 08 D0 0F 00 08  00 00 00 00 FF F
08000FE0: 0D 00 00 80 0E 00 00 80  05 00 00 80 10 0
08000FF0: 14 00 00 80 0F 00 00 80  08 00 00 80 06 0
08001000: 74 DC 8A 00 00 00 00 00  00 00 00 00 00 0
08001010: 00 00 00 00 00 00 00 00  00 00 00 00 00 0
```

ARTICLE

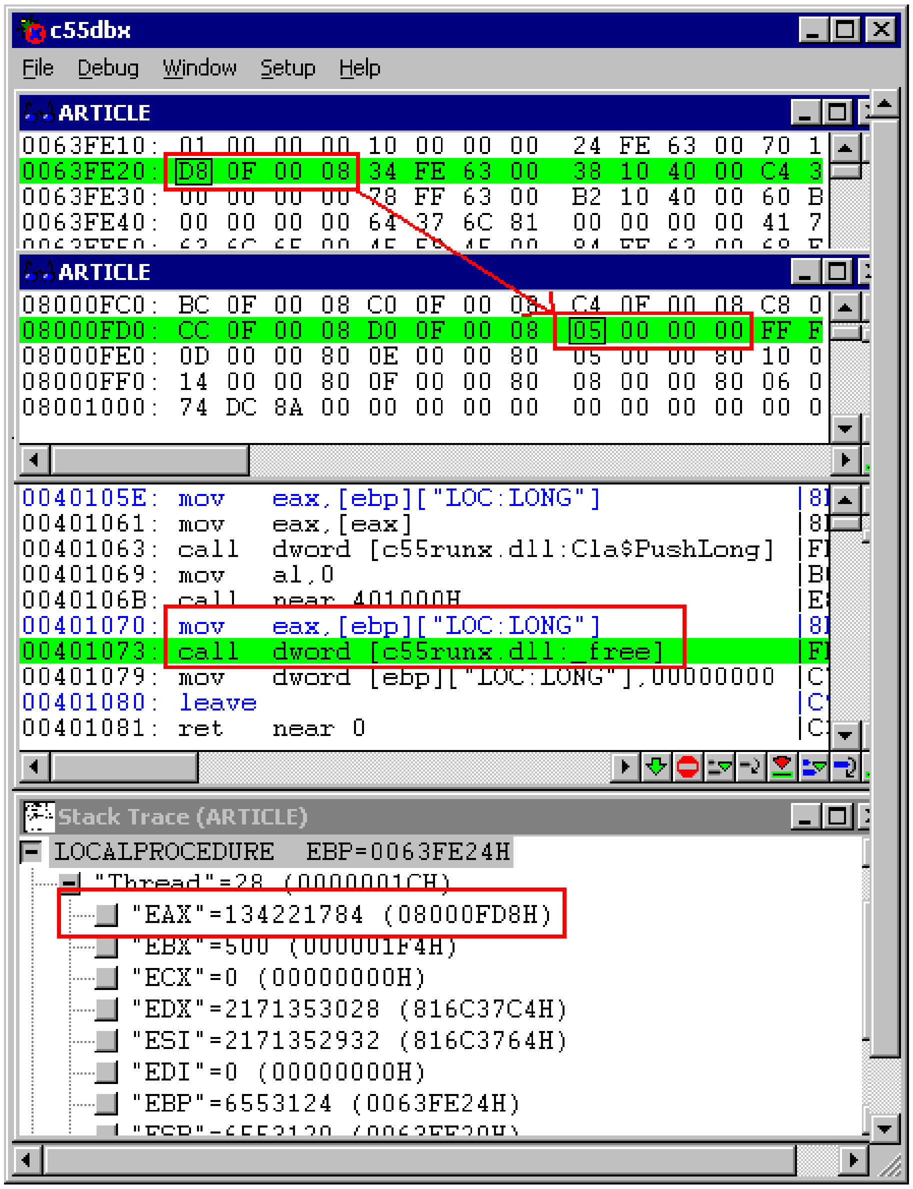
```
0063FE20: D8 0F 00 08 34 FE 63 00  38 10 40 00 C4 3
0063FE30: 00 00 00 00 78 FF 63 00  B2 10 40 00 60 B
0063FE40: 00 00 00 00 64 37 6C 81  00 00 00 00 41 7
0063FE50: 63 6C 65 00 45 58 45 00  84 FE 63 00 68 F
0063FE60: 03 00 00 00 06 00 00 00  03 00 00 80 00 0
```

```
0040103B: ret    near 0                                 |C
LOCALPROCEDURE): enter 4 0                              |C
00401040: mov    dword [ebp]["LOC:LONG"],00000000       |C
00401047: mov    eax,00000004                           |B
0040104C: call   dword [c55runx.dll:Cla$NewMemZ]        |F
00401052: mov    [ebp]["LOC:LONG"],eax                  |8
00401055: mov    eax,[ebp]["LOC:LONG"]                  |8
00401058: mov    dword [eax],00000005                   |C
0040105E: mov    eax,[ebp]["LOC:LONG"]                  |8
00401061: mov    eax,[eax]                              |8
```

Figure 9. Disassembly of heap memory allocation.

If `Cla$NewMemZ` is successful, it returns a pointer to the allocated memory. The pointer is stored in the appropriate variable, which is a local variable, which is allocated on the stack! When the procedure terminates, the unwinding of the stack deletes the pointer to the allocated memory. If there wasn't a `DISPOSE()` action before this unwinding starts, I've just caused a memory leak! The system marks the memory allocated as reserved, but the program cannot reach it because the pointer is not available anymore.

The next figure shows the call to `_free()`, which takes a parameter that points to the memory that can be released, in this case the memory pointed to by `loc:long` (address `08000FD8h`), so the allocated memory is de-allocated before the stack-unwinding starts.



```
0063FE10: 01 00 00 00 10 00 00 00  24 FE 63 00 70 1
0063FE20: D8 0F 00 08 34 FE 63 00  38 10 40 00 C4 3
0063FE30: 00 00 00 00 78 FF 63 00  B2 10 40 00 60 B
0063FE40: 00 00 00 00 64 37 6C 81  00 00 00 00 41 7
```

```
08000FC0: BC 0F 00 08 C0 0F 00 08  C4 0F 00 08 C8 0
08000FD0: CC 0F 00 08 D0 0F 00 08  05 00 00 00 FF F
08000FE0: 0D 00 00 80 0E 00 00 80  05 00 00 80 10 0
08000FF0: 14 00 00 80 0F 00 00 80  08 00 00 80 06 0
08001000: 74 DC 8A 00 00 00 00 00  00 00 00 00 00 0
```

```
0040105E: mov   eax,[ebp]["LOC:LONG"]
00401061: mov   eax,[eax]
00401063: call  dword [c55runx.dll:Cla$PushLong]
00401069: mov   al,0
0040106B: call  near 401000H
00401070: mov   eax,[ebp]["LOC:LONG"]
00401073: call  dword [c55runx.dll:_free]
00401079: mov   dword [ebp]["LOC:LONG"],00000000
00401080: leave
00401081: ret   near 0
```

```
LOCALPROCEDURE   EBP=0063FE24H
  "Thread"=28 (0000001CH)
    "EAX"=134221784 (08000FD8H)
    "EBX"=500 (000001F4H)
    "ECX"=0 (00000000H)
    "EDX"=2171353028 (816C37C4H)
    "ESI"=2171352932 (816C3764H)
    "EDI"=0 (00000000H)
    "EBP"=6553124 (0063FE24H)
```

Figure 10. Freeing the allocated memory.

Static Storage

Global and static variables are stored in the static storage area of the program. This is memory allocated for the entire duration of the program. There are some reasons not to use global memory:

- If two different modules each have a public global variable with the same name, the modules cannot be put together into a single program.
- Re-entrant code usually modifies its own local data. This is not possible when using global variables. Global memory is shared by all procedures in the running process.
- Because global variables can be read and written from everywhere in the code, it is difficult to analyse their effect on a program's behavior. Seemingly unrelated procedures can have unexpected side effects when they depend on global variables.

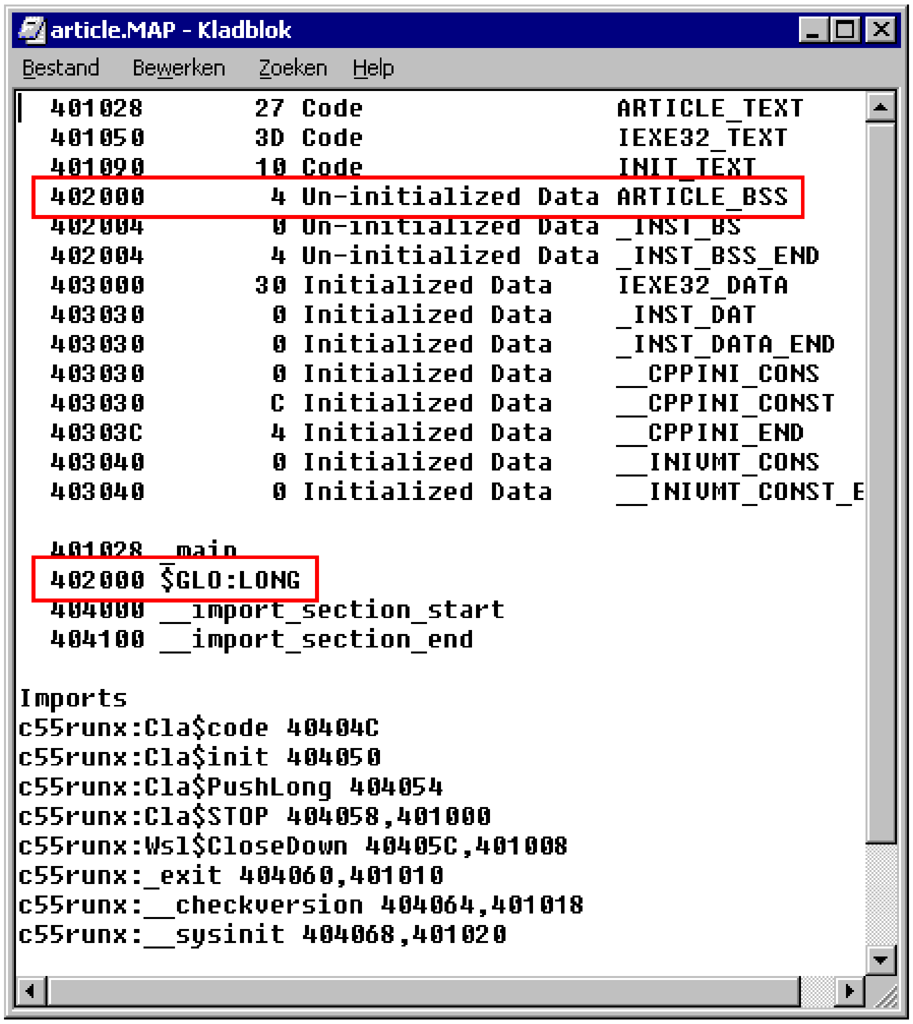
Here’s one final example:

```
  Program

  map
  end

glo:long long

  code
  Glo:long = 7
  stop(glo:long)
```

Where the global variables reside can be read from the generated map file, however, I assume that these globals are shown at Relative Virtual Addresses (RVAs) based on a load-address of 00400000h. Since the EXE is the first module that is loaded when an application starts, reallocations are not necessary because there never will be a conflict at the given address and thus the addresses in the map file for the EXE always display correct values.

This is not always the case when dealing with DLLs. DLLs are loaded at a "preferred address." Based on this address, function pointers receive their values. However sometimes, when a DLL is loaded in an address space of the executing procedure, another DLL is already loaded at the given address. When this occurs, the system redirects the function pointers to the right address with help of the relocation table in the DLL, in a process called relocating. For more information on this topic see Microsoft’s online documentation, specifically the article Peering Inside The PE: A Tour of the Win32 Portable Executable File Format by Matt Pietrek.

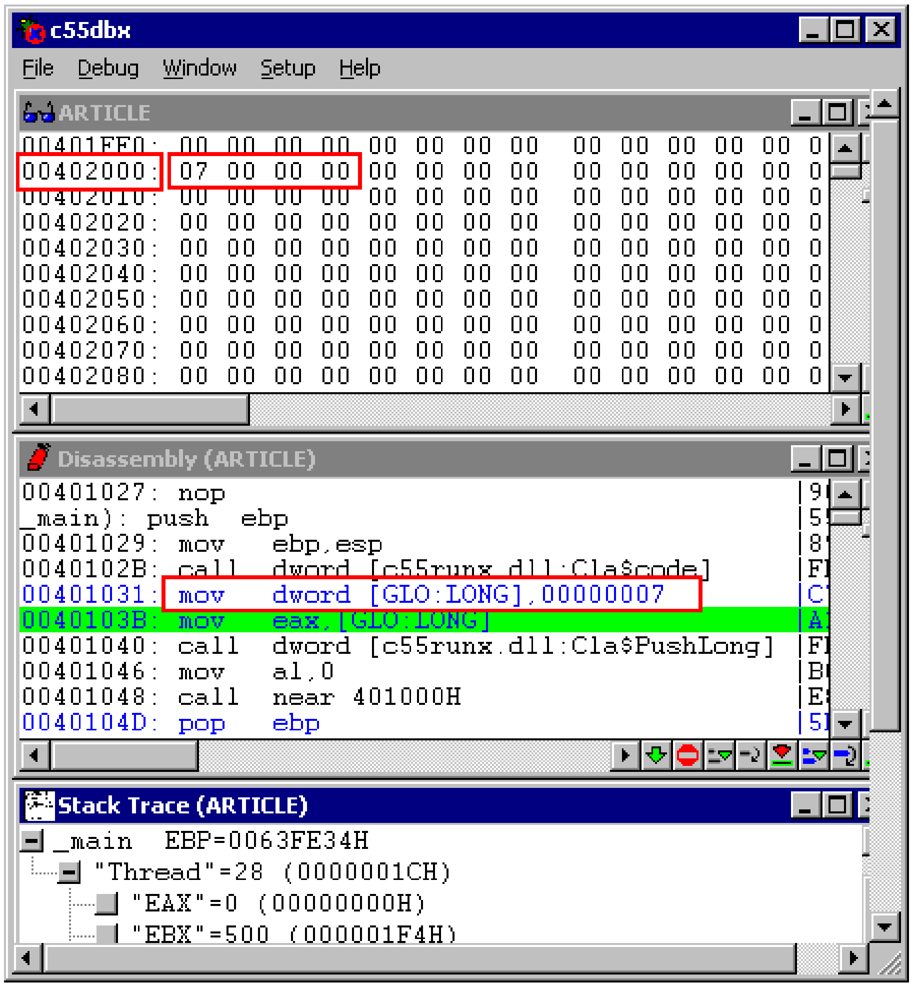
The following figure shows the generated map file, which shows where the uninitialised data is going to be mapped when loaded, in this case address `00402000h`.



```
  401028       27 Code                ARTICLE_TEXT
  401050       3D Code                IEXE32_TEXT
  401090       10 Code                INIT_TEXT
  402000        4 Un-initialized Data ARTICLE_BSS
  402004        0 Un-initialized Data _INST_BS
  402004        4 Un-initialized Data _INST_BSS_END
  403000       30 Initialized Data    IEXE32_DATA
  403030        0 Initialized Data    _INST_DAT
  403030        0 Initialized Data    _INST_DATA_END
  403030        0 Initialized Data    __CPPINI_CONS
  403030        C Initialized Data    __CPPINI_CONST
  40303C        4 Initialized Data    __CPPINI_END
  403040        0 Initialized Data    __INIVMT_CONS
  403040        0 Initialized Data    __INIVMT_CONST_E

  401028 _main
  402000 $GLO:LONG
  404000 __import_section_start
  404100 __import_section_end

Imports
c55runx:Cla$code 40404C
c55runx:Cla$init 404050
c55runx:Cla$PushLong 404054
c55runx:Cla$STOP 404058,401000
c55runx:Wsl$CloseDown 40405C,401008
c55runx:_exit 404060,401010
c55runx:__checkversion 404064,401018
c55runx:__sysinit 404068,401020
```

Figure 11. Map file generated for the current project.

When executing the program, `glo:long`, at address `00402000h`, is assigned a value of 7. This memory is allocated for the entire duration of the program, hence pointers to this variable are safe to use.



```
00401FF0: 00 00 00 00 00 00 00 00  00 00 00 00 00 0
00402000: 07 00 00 00 00 00 00 00  00 00 00 00 00 0
00402010: 00 00 00 00 00 00 00 00  00 00 00 00 00 0
00402020: 00 00 00 00 00 00 00 00  00 00 00 00 00 0
00402030: 00 00 00 00 00 00 00 00  00 00 00 00 00 0
00402040: 00 00 00 00 00 00 00 00  00 00 00 00 00 0
00402050: 00 00 00 00 00 00 00 00  00 00 00 00 00 0
00402060: 00 00 00 00 00 00 00 00  00 00 00 00 00 0
00402070: 00 00 00 00 00 00 00 00  00 00 00 00 00 0
00402080: 00 00 00 00 00 00 00 00  00 00 00 00 00 0
```

```
00401027: nop                                          |9
_main): push  ebp                                      |5
00401029: mov    ebp,esp                               |8
0040102B: call   dword [c55runx.dll:Cla$code]          |FI
00401031: mov    dword [GLO:LONG],00000007             |C
0040103B: mov    eax,[GLO:LONG]                        |A
00401040: call   dword [c55runx.dll:Cla$PushLong]      |FI
00401046: mov    al,0                                  |B
00401048: call   near 401000H                          |E
0040104D: pop    ebp                                   |5I
```

```
_main  EBP=0063FE34H
  "Thread"=28 (0000001CH)
    "EAX"=0 (00000000H)
    "EBX"=500 (000001F4H)
```

Figure 12. Storing values in global variable.

Summary

There are three important memory areas, these are:

- Stack - local variables and parameters
- Heap - memory allocated with NEW(), pointer is placed on stack
- Static storage - global and static variables

The main reason to use stack memory is simplicity; the compiler takes care of allocating and de-allocating memory without giving cause for concern over memory leaks.

There are two reasons I can think of to use heap memory. The first reason concerns the extra overhead of stack page allocation. Because the stack is set aside by the operating system in pages of fixed memory, it is possible that slightly more memory is being allocated than the same allocation on the heap. The second reason involves the control over scoping issues. If an object is allocated on the heap, a pointer to the object is placed on the stack. You can then give the value of the pointer to another pointer outside the scope of the function and use the object after the function terminates.

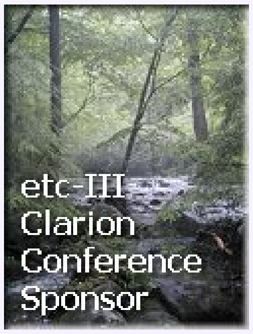
If you need complete control for the scope of the variable, use `NEW()` to allocate memory, which uses heap memory. Remember always to `DISPOSE()` or you can have serious memory leaks.

John Gorter has been programming in Clarion for three years, before which he studied business informatics. He has just passed the MCSD exams and is now busy creating web applications with C55 Internet Connect. John lives in the Netherlands, and when not programming, reads about programming.

Reader Comments

Add a comment

A must have book for any advanced developer library is...



Interview: James Orr On The Public PIM

Published 2001-06-12

James M. Orr is the founder and Director of Marketing of the OpenDB Alliance, an organization which is promoting the "Public PIM" database design as a proposed industry standard for employing many-to-many relationships and recursive relationships.

Where did you get the idea for the OpenDB Alliance, and more specifically the Public PIM, your concept for a common database structure for personal information manager applications?

James: The idea came over a whole lot of years, really. I had gone through just about every database resource, program, 4GL, 3GL, 5GL known to man, and I always seemed to hit a wall. I always wanted many-to-many [relationships between tables], but trying to find some information about many-to-many was like pulling hen's teeth. You go to all the major bookstores, and you find all these books that are three inches thick, and you find a half a page about many-to-many relationships. I've sort of self-taught myself about foreign keys and many-to-many and the whole nine yards. And it's come down to those six tables that are in this [Public PIM] working example that I've derived. It'll do many-to-many, it'll do recursive [relationships], it'll do whatever you want it to do.

If you're successful, what will happen?

Everybody will be using this structure, and communication between programs could be an awful lot simpler. [Developers] will think along the lines of Public PIM and its extensible structure. It'll be the status quo way of doing things, or you'll be able to recognize it as a style, and if it's written in this style then you know that you can do certain things.

So this is kind of like a design pattern for databases?

Exactly. And that's really all it is. It's a way of doing something. But now I'm trying to present it to others, the Clarion world, DB2, Informix, all of that, to just use it as a model, like a Hello World [program]. If there's anything open source about it, it is the structure, the naming conventions that I'm using, I'm hoping to use those to make it intuitive to people to explain many-to-many, to do things in a more aggressive way than they do in those three inch thick books I was talking about. I think it certainly helps a lot for newbies to the world of relational databases, and especially with the web, because people are going to have to use more many-to-many relationships [in web databases].

Your web site says: "This is an attempt to create a defacto structure standard for accessing database data." Are you interested in more than just PIM databases?

This goes back to my history of going to a lot of seminars. There is always somebody asking "who's got a problem?" There'll be someone back in the room of a hundred people, and he wants to talk about blue widgets. There's 99 other people in the room who are trying to convert his problem into their problem and then think about the particular database that they're working with. My thought is that the [Public PIM] [design] that we're talking about here is something that everybody else in the world knows about, at least if they're trying to do databases.

So it's something that everybody can communicate on. It happens to be using the same tables that a majority of the PIMs use. The term Public PIM was really coined by Jesse Berst, and I just borrowed it from him. Berst and one of the other authors [at his site]. So here comes OpenDB, and the working example is just a way to demonstrate [the concept], and how it works, and how it can be very powerful. You can turn it into almost anything you want to, plus it shares data.

The way I think of this database structure, it is as extensible as XML is. If you want to design something, use the basis here and you can create any kind of application you would choose to use. That's a bit of a problem with XML too. Before you sit down and work with it, you think it must be some amazing thing. And really it's just a way of describing data. That's pretty simple. It can be used in sophisticated ways, but the underlying concept is simple.

That's the challenge. You've got to get it down to a common terminology, a common problem. Do you know how many different thousands of PIMs there are? There are a tremendous number of these PIMs that would become a lot more powerful if they used [the Public PIM database design].

Does your design specify the actual table structures?

Dave : I guess a better answer would be that there are rudimentary things that establish the structure and the things that will allow multiple apps to share data so what a person learns from the basic Tables would give them the tools to go farther with the design even though I have some pretty strong feelings about keeping it simple.

Do you define the foreign keys?

Yes, we use a particular naming convention as part of what we consider an intuitive way to recall their use in the schema

You mentioned XML. Have you looked at defining a DTD for the XML data interchange?

We don't have a DTD at this time but the standard has been set now for XML Schema so I am sure we will use it.I am working with a company called Popkin right now, you might be familiar with them. Their Envision XML product should be able to turn this [database design] into a DTD and also be able then to export this to Oracle or MySQL or any of those.

Is everything open source?

I don't give everything away. My working example, I don't give the code away on that to everybody. But if I preloaded a database with a buyer's guide, and sold it to a magazine, then they have not only a buyer's guide but they also have a program to put their own stuff in. I would love to be able to massage Outlook into doing some of the things I'm now doing with [the Public PIM]. I just haven't found the right people to work with yet.

Do you have the example Public PIM in Access files only?

I have it in SQL 7 as well and always looking for people who want to use different databases and LDAP

What's your focus at the moment?

Well, I'm talking to you, and I've had a couple of other folks in the publishing world [show interest]. That buyer's guide? I'd like to promote to publishing firms. The database itself [is well suited to] keeping track of projects, and leads, and things like that. And maybe they'll write about it.

I'm just being very patient, and improving on everything that I can, and chatting with everybody I can. It's made some headway. I'm trying to get going with the SQL 7 version [of the web PIM].

You have an online PIM database available to the general public. Does that raise any security concerns? Could it become a resource for, say, spammers?

I guess I'll have to face that issue when I come to it. You can't be so cautious that you don't get anything done, but I understand what you're saying.

What's your background?

[I went] out of the Navy into working at IBM as a librarian in their Houston office, then at the Carbide Employees Federal Credit Union where I got into programming. I went to Service Bureau

Corporation, and to California with them. Then I switched to sales, initially selling the credit union package. When I know something, I can sell it really well. It's just like [Public PIM] – if I can get the tools in front of me, this will be easy to sell because it is simple.

Resources

Web site	http://www.opendb.org
Mailing Address	OpenDB Alliance 5203 Highway 3 Dickinson, Texas 77539-6833
Email	asaptt@wt.net
ICQ	22993101
Telephone	281/337-0268
GIF of main screen	ftp://208.150.237.25/publicpim/publicpim.gif
GIF of ER diagram.	ftp://208.150.237.25/publicpim/publicpimER.gif Other ER diagram is done with relation from www.msbsoftware.ch/relation.htm
ZIP of new Access97 MDB with recursive structured join tables	ftp://63.111.238.120/publicpim/
Working Example (download)	ftp://63.111.238.120/PublicPIM
Working Example (online)	www.opendb.org/WEBpp/test.asp

Reader Comments

Add a comment

OK, I'll ask the stupid question -- what does PIM stand...
Personal Information Manager, I believe.