Clarion magazine

Main Page
Log In
Subscribe
Open Source
Links
Mailing Lists
Advertising
Submissions
Contact Us
Site Index
ClarionMag FAQ
Download PDFs
Search ClarionMag

Volume 1, Number 5 - June 1999

Issue Index

Four DLLs And An Executable
Gordon Smith explains step-by-step how to split a large application up into a global DLL, program DLLs and an EXE.
Posted on June 7, 1999

Sliders!
Sliders are useful 32-bit Windows controls, but they're not part of the Clarion toolset. Pierre Tremblay shows how you can create and use sliders using the Windows API and a set of wrapper classes..
Posted on June 7, 1999

Call For Photos
Clarion Magazine wants your Clarion-related photos. If you have pictures from a Clarion gathering (of any size) over the past several years and they're not so terrifying that they'll cause other subscribers to panic, send them in. Selected images will be published in a ClarionMag photo album page.
Posted on June 7, 1999

Problems With PDFs?
If you've been having problems with Adobe Acrobat 4 and Internet Explorer not displaying ClarionMag PDFs properly read the PDFs page for a useful tip.
Posted on June 7, 1999

The Novice's Corner - Many-To-Many Relationships
One of the trickier problems in designing databases is handling many-to-many relationships. This month's Novice's Corner looks at how to define such a relationship in the data dictionary.
Posted on June 14, 1999

David Bayliss On The FileManager Part 2
David Bayliss continues his series on the FileManager with a look at the administrative functions, error handling, and the snapshot mechanism.
Posted on June 14, 1999

Get On Our Mailing List
If you'd like to receive weekly emails of changes to Clarion Magazine, visit the mailing lists page and add your name to the notification list.
Posted on June 14, 1999

Search Clarion Magazine
See that button on the bottom of the menu bar? It'll take you to the Clarion Magazine search page. We use the htdig search engine which provides sophisticated searching of all ClarionMag articles.
Posted on June 14, 1999

New Improved Clarion Challenge!
Well, you can't quite stick a fork in it yet. This was supposed to be the results article for last month's Clarion Challenge, but because of some disparities between implementations, the challenge has been reformulated. The requirements are the same (create a string parser),

but all you need to do now is fill in the code for four or five methods! This makes a great OOP learning experience - try it and see!
Posted on June 21, 1999

How ABC Handles Multiple Sort Orders (Part III)
A need for speed (sorts) takes Steve Parker down a winding path to ABC's handing of multiple sort orders. Part 3 of 3.
Posted on June 21, 1999

Developers' Open Source Public License Version 1.0
It's been a long haul, but the CoveComm Developers' Open Source Public License is back from legal and ready for prime time! If you missed the earlier open source articles, it became clear after reviewing the existing open source agreements that none were well suited to the Clarion development community. We hope this license will become the standard for Clarion open source contributors.
Posted on June 21, 1999

June 1999 News
News, notes, announcements, bulletins, dispatches, communiques and (yes, and) proclamations for the Clarion world.
Posted on June 28, 1999

Clarion Advisor: Debugging Tricks
Clarion Magazine has featured several articles on debugging applications (go to the Search page and search for "debug"). But you can never have too many tricks up your sleeve, and Clarion developers get as wily as anyone when it comes to using alternative debugging techniques.
Posted on June 28, 1999

The ABCs Of OOP - Part 3
In this third article in the ABCs Of OOP series Dave Harms explains virtual methods, one of the most powerful features of object-oriented programming.
Posted on June 28, 1999

Clarion Magazine Best Read With Verdana
Clarion Magazine is formatted using the Verdana font, which is designed specifically for on-line use. Get Verdana and give your eyes a break!
Posted on June 28, 1999

Detecting Crashes With DDE
David Podger shows how to use a tiny DDE server to detect (and potentially recover from) application crashes.
Posted on June 28, 1999

Copyright © 1999 by CoveComm Inc. All Rights Reserved. Reproduction in any form without the express written consent of CoveComm Inc., except as described in the subscription agreement, is prohibited. If you find this page on a site other than www.clarionmag.com, email covecomm@mbnet.mb.ca.

Vol 1, No 5
June 1999

Clarion magazine

Feature Article

Four DLLs And An Executable

by Gordon Smith

Eventually a time will come when you will want to split your APP (and consequently EXE) into smaller more manageable parts. This can be quite a daunting task, not because it is difficult (it is after all just a matter of setting a few Global Properties on the General and File Control tabs), but because it seems to be "black magic" and when it goes wrong it seems to go horribly wrong! In this article I will outline an approach to creating DLLs and take a look at what actually happens.

Before starting, make sure your APP and DCT are copied to an easily accessed backup folder. You will be creating new APPs, and they will need to import their procedures from this original copy.

The Objective

The objective is to convert the main APP into several smaller ones, giving several benefits:

- A quicker development cycle: After the initial design phase, development normally takes place in one or several of the smaller APPs. Since the APP only contains a subset of the overall program (Procedures and Files), generation and compile times are reduced.
- Multi-user development: Different APPs make it easier to distribute the workload; programmers can be easily assigned specific APPs to develop and manage.
- Simplified distribution: Minor changes only necessitate minor upgrades (several changes may only affect one DLL, for example).

Each APP will either create an executable program or a library of procedures and data. There are two main types of library:

- Static: All data and procedures are linked into a library file (.LIB), which can then be included (in total) in other applications.
- Dynamic: All data and procedures are linked into a dynamic link library (.DLL), this is a separate unit that needs to be distributed with the application EXE. NOTE: A library file is also created but it doesn't contain any procedures or data, it just contains information about the DLL (such as procedure entry points etc.).

  NOTE: This article only deals with the Dynamic Link Library (DLL).

The proposed layout will contain the following elements (each element corresponds to an individual APP):

- The global DLL (one only).
- The program DLLs (several).
- The main EXE (one only).

To distinguish between the different types of DLL and EXE, extra information needs to be provided:

BKO Enterprises, Inc.
Great Opportunity!
Salary to $125,000+
Pre-IPO Stock Opts
Sunny Boca Raton

etc 2000
If you're interested take our poll & let us know!

- Whether the application is a DLL or EXE (project settings).
- Specify which items (data/classes/procedures) are to be linked into the application (the default behaviour).
- Specify which items (data/classes/procedures), are to be published (exported) for other applications to use. This information is specified in the module definition or export file which has the application's name and an EXP extension and includes the entry points in the associated library file.
- Specify which items (data/classes/procedures) reside in other applications. This is specified with the EXTERNAL and DLL attributes. The associated library file will also need to be included in the project.

The Global DLL

Scope of data is important in a multi DLL application. Simply creating data on the global properties tab is not the same as creating global data (data which is visible across several DLLs). It is still desirable to have "application" global data which is visible throughout only one APP. Care must be taken to differentiate between these different types.

To simplify matters global data can now (Clarion 5) be declared in data dictionaries, allowing each APP to either declare and instantiate it (in the global data APP), or simply define it and reference it as external.

The same goes for classes. Class methods should only exist once (think of them as procedures) and class properties should only exist once per instance (think of them as data). This means that the class "code" should only be linked into one APP (the global one), while the class instances will depend on usage (instances of the window manager will be local to a procedure, while the single instance of the file manager will be global).This DLL will contain the master copy of all file declarations, global data, ABC class library and the generated database referential integrity code. It will not access any other DLLs (except for the Clarion and Windows runtimes).

To create the global DLL perform the following steps:

Step 1: To enable the templates to generate the global data correctly, this data must be moved from the Data section (in the Global Properties) to a global data file in the dictionary. Subsequently each APP that uses this dictionary will be able to generate the global data with the appropriate attributes.

Step 2: Create a new APP, using the existing dictionary.

Step 3: On the Project Editor|Global Options dialog set the Target Type to DLL. This tells the compiler/linker to create a DLL, and it determines which items to publish (export) by reading the application's EXP file. A further discussion on the export file would warrant an article in itself!

Step 4: On the Global Properties dialog General tab set the Generate Template Globals and ABC's As External option to unchecked. When generated, all template global data and ABC class declarations will NOT have any `EXTERNAL` or `DLL` attributes. This also links the ABC class library into the DLL by setting `_ABCLinkMode_` to `TRUE`.

Step 5: On the Global Properties dialog File Control tab make sure that Generate All File Declarations is checked. This will ensure that all file declarations in the dictionary are generated. Since the global APP won't contain any browses or reports, without this switch it wouldn't generate any file definitions at all.

Set File Attributes/External to None External. This is similar to Step 2 and ensures generated file declarations will NOT have any `EXTERNAL` or `DLL` attributes.

Set File Attributes/Export files/Export all file declarations to checked. This tells the AppGen to add the list of files and global data to the export (EXP) file, thus making them visible to the other APPs.

Step 6: Compile.

The Program DLLs

The program DLLs contain the body of the program: browses, forms and reports. The size of the program will dictate how many of these DLLs you will require. From experience I usually put all reports into one APP and divide the rest into functional groups (all employee related information in one DLL, all sales/invoice information in another etc.). Another approach would be to break them up by programmer. I also tended to limit the number of procedures to about 20 per APP, as this number coincides with my tolerance level for a complete generation/compile.

> **NOTE:** The reason for separating the reports into their own DLL is not one of efficiency, as it would be more efficient to keep the report in its logical DLL, but rather one of practicality. In my experience the report DLL is the one that gets changed the most and keeping all the reports together subsequently makes distribution a lot easier.

To create the program DLLs:

Step 1: For each DLL, create a new APP and import the required procedures from the original APP and follow Step 3 in the Global DLL section.

Step 2: On the Application menu select Insert Module and choose ExternalDLL as the module type. This presents the Module Properties dialog. The global DLL library file must be entered into the in the Name field (filename.LIB NOT filename.dll).

Step 3: On the Global Properties dialog's General tab set the following options:

- Set Generate Template Globals and ABC's as External to checked (True). When generated, all global data declarations and ABC class declarations will have the `EXTERNAL` and `DLL` attributes. This also sets `_ABCLinkMode_` to `FALSE` so the ABC class library won't be linked into the DLL.
- Set External Globals and ABC's Source Module to Dynamic Link Library. This tells the linker that all global data declarations and ABC class library are present in another DLL by setting `_ABCDllMode_` to `TRUE`. This is strictly only required for 32bit DLLs as the compiler needs to add an extra de-reference to distinguish between statically linked and dynamically linked libraries.

Step 4: On the "Global Properties dialog File Control tab" set the following options:

- Set Generate All File Declarations to OFF. There is no need to generate superfluous file definitions.
- Set File Attributes/External to All External. This is similar to Step 3 and ensures generated file declarations will have the appropriate `EXTERNAL` and `DLL` attributes.
- Set File Attributes/External files/All files are declared in another app to checked. This was used in the Legacy Templates to optionally generate an EXTERNAL attribute on the `File:Open` flag.

Step 5: Decide which procedures need to be exported. Typically this is any procedure that is called from the main menu (normally browses and reports, but not forms as they tend to be in the same DLL as their browse counterparts). On the Procedure Properties dialog set Export Procedure to checked. This tells the generator to create the appropriate line in the export file.

During these steps there will be references to procedures that don't exist in the current DLL and appear as a ToDo. They exist in one or more external DLLs and are covered in the next section. These should be left as To Do's until they have been created in their appropriate DLL; this can take several iterations depending on how the procedures have been arranged.

Step 6: Compile.

The Main EXE

The main EXE will typically contain the Application Frame and the Splash and About procedures. Being an EXE it can't export any procedures and will call procedures from the Procedure DLLs

Step 1: Create a new APP and import the required procedures from the original APP (Frame, Splash etc.).

Step 2: Follow Steps 2, 3 and 4 from the Program DLL section.

Step 3: There should be several ToDo procedures relating to each menu item that calls a procedure. Before these procedures can be called, their related DLL/Library file must be first added to the module list (this will resolve the external procedure calls for the linker). For each DLL perform Step 2 from the Program DLL section.

Step 4: For each ToDo procedure select the External procedure template and ensure the correct library is selected in the Module Name field. This adds the procedure declaration to the global map, in the correct module.

Step 5: Compile the main exe.

Summary

If all has gone well that should be it. If it has gone wrong it can appear a lot worse than it usually is. Don't be afraid of the error messages; the majority of them will be one of the following three:

- **Unresolved External XXX in YYY.obj**: This means you have declared some item as being external, but the linker was unable to locate it in any of the included libraries.
- **XXX Is unresolved for export**: Some item has been listed in the EXP file, but the linker was unable to locate it in the APP.
- **XXX is duplicated (dll)**: This happens when XXX has been either exported from more than one included library or has been declared in the current APP and one (or more) included library.

Well that's it, now your program can grow with style! In a future article I will look at how you can include Clarion's runtime libraries inside a library of your own. This creates smaller programs with your own DLL naming conventions and less distribution headaches.

Prior to joining TopSpeed Development Centre, Gordon Smith worked for an Irish company developing software for multi-national pharmaceutical companies. He was also a member of the 3rd party accessories program (Compile Manager 2) and developed the Clarion Class Browser.

Copyright © 1999 by CoveComm Inc. All Rights Reserved. Reproduction in any form without the express written consent of CoveComm Inc., except as described in the subscription agreement, is prohibited. If you find this page on a site other than www.clarionmag.com, email covecomm@mbnet.mb.ca.

Vol 1, No 5
June 1999

Clarion magazine

BKO Enterprises, Inc.
Great Opportunity!
Salary to $125,000+
Pre-IPO Stock Opts
Sunny Boca Raton

etc 2000
If you're interested take our poll & let us know!

Feature Article

A Slider, Did You Say?

by Pierre Tremblay

A while ago I was reading the newsgroup and I saw a message from a Clarion programmer asking if a slider control was available somewhere. I was wondering why Clarion doesn't support this control natively and I decided that I should see if there was any possibility of adding this to my toolbox. After a quick look in my MSDN library I discovered that the control is only 32bit. This wasn't a big deal, so I added it to my toolbox.

A control is simply a child window which belongs to a parent window. For the creation of a slider control (which Microsoft calls a trackbar), you need to make some Windows API calls and subclass the parent window procedure in order to track messages that the control sends to the parent window.

A slider can be horizontal or vertical and depending on this it will notify the parent window by sending a `WM_HSCROLL` or `WM_VSCROLL` message. An application can also send messages to the control to set the range, the slider position, and many more properties.

The slider control can also be restricted to a selected range of values within the usual range of the control (see Figure 1 below).

I decided the ideal interface to manage the control functionality would be a class, and in this article I will set the foundation of that class. I say ideal because it seems to be very natural to ask the control to do things that the application needs. For example, it looks friendlier in the source code to see:

```
cTrack1.SetLimit(1, 50)
```

instead of using the SendMessage API call, where you have to look up the right message to send and find out the way to set the low and high word of the `LPARAM` parameter. Of course, there is a need to make that call somewhere but once it is inside the class, it doesn't need to be looked at again!

What is needed?

There are three very important elements needed for managing what I will called a "foreign" control on a Clarion window.

The first one is a subclassing procedure which will trap messages sent by the control. Basically this procedure will trap the `WM_HSCROLL` and `WM_VSCROLL` received by the window procedure.

Subclassing a window procedure is the process of defining a new procedure within the application which will be called by Windows instead of the Clarion internal message handler. The new procedure should do whatever needs to be done for the `WM_HSCROLL` and `WM_VSCROLL` messages and then call the original Clarion internal procedure for messages that it doesn't process.

Another important element is a way to call a class method as if it were a Windows callback procedure. This is not directly possible because a class method has an implicit first parameter of SELF which is a reference to the object itself, and Windows expects the callback procedure to have a specific set of parameters, none of which is a Clarion object. The way the code will address this is by using a queue as a dispatcher. The layout of the queue is shown in Listing 1.

Listing 1. The dispatcher queue.

```
HandlerQueue            QUEUE
ParentHWND                HWND
ChildHWND                 HWND
QControl                  &cControl
                        END
```

At the creation of the control, the code stores the parent window handle, the control handle itself and a reference to the class object managing the control. The queue is sorted using the parent and the child handles. Once in the window procedure, the subclassed message handler code will simply make a lookup in the queue and call the `cControl.WndProc` method of the referenced object. All of this makes it possible to have more than one trackbar on the same window, or different control classes derived from the same base class.

Listing 2. The subclassed message handler.

```
TBTestWinProc        PROCEDURE  (unsigned MyHWnd, unsigned usMsg↵
        ,UNSIGNED WParam,LONG lParam) ! Declare Procedure

  CODE
  ! Look in the queue if there is a corresponding
  ! object to handle this message
  HandlerQueue.qHWnd = MyHwnd
  HandlerQueue.qControlHwnd = lParam
  GET(HandlerQueue, HandlerQueue.qhWnd, HandlerQueue.qControlHwnd)
  IF ~errorcode()
    ! An object is there? Call its WndProc
    RETURN(HandlerQueue.qControl.WndProc(MyHwnd, usMsg, |
        WParam, lParam))
  END
  ! Else return the default window procedure
  RETURN(CallWindowProc(mWndProc, MyHwnd, usMsg, WParam, lParam))
```

The last element is the class itself. This defines the interface that the programmer will deal with. In this first draft of that class, there is a base class and a derived class. The base class defines different properties and a do-nothing WndProc method. All derived classes will need to define that virtual method.

Subclassing the window procedure

As I said, it is necessary to subclass the Clarion internal window procedure in order to trap the `WM_HSCROLL` and `WM_VSCROLL` send by the trackbar to its parent window. Subclassing the window is achieved using the Windows API function call `SetWindowLong`. This function accepts as parameters the handle of the window to subclass, the index representing the value to be changed and the new value itself.

The handle of the window is available using the `{PROP:ClientHandle}` property. The others parameters for the call of `SetWindowLong` will be `GWL_WNDPROC` and the address of the new window procedure.

```
SetWindowLong(Window{prop:ClientHandle}, GWL_WNDPROC, ↵
        ADDRESS(NewWndProc))
```

The call to this function will return the address of the original window procedure. This returned value is extremely important to keep because it will be needed in order to call the original procedure for those messages not processed by the code. In that case, the `CallWindowProc` API call will be used and the first parameter of that function is the address of the original procedure. The others parameters are the same as what is passed by Window to the callback function. See the generated code in the example application for the details.

I must add that there is another way to set the window procedure if you really want to avoid a Windows API call. The address of the internal window procedure for a given window can be obtained by using the {PROP:WndProc} property. This property is read/write. You will need first to read the address of the actual window procedure and use another line of code to set the new one. The advantage of using the `SetWindowLong` is everything is done in only one line of code.

When the user interacts with the trackbar a `WM_HSCROLL` or `WM_VSCROLL` message is sent to the parent window. At this point the window procedure needs to know exactly what the user did with the control. This is the job of a notification code, which will usually be paired with the `WM_HSCROLL/VSCROLL` message. The notification code will be in the low word of the `Wparam` parameter. For the `TB_THUMPOSITION` and `TB_THUMBTRACK`, the high-order word of the `Wparam` specifies the position of the slider. For all other notification codes, the high-order word is zero.

There are different messages that can be sent to the trackbar control. If you want to set the slider position, you will send a `TBM_SETPOS` message. To retrieve the slider position, you send a `TBM_GETPOS` message.

The class presented with this article is basically a wrapper around those different messages. For example, to retrieve the position of the slider, you will call the `GetPos` method.

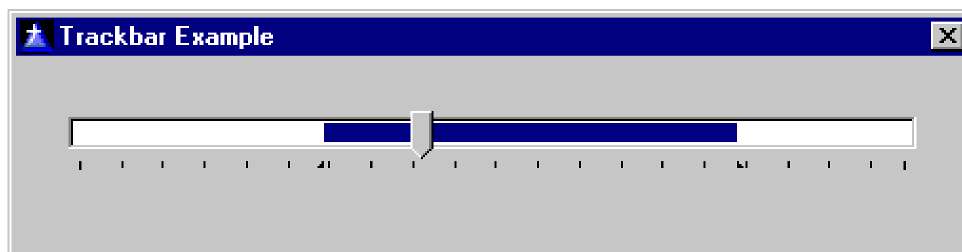
```
Var = cTrack1.GetPos()
```

The class has also a virtual method (`cTrack1.TakeNotification`) to let you place your own source for a particular notification code. In fact, the template (see below) generates this method if you set the checkbox to limit the slider thumb inside of the selected range. To track a particular notification code, you can use the `cTrack1.Notification()` method which returns the current notification code under process.

```
CASE SELF.Notification()
Of TB_BOTTOM
  Message('User pressed END key')
END
```

Figure 1 shows the trackbar in action. Note that in this example the slider thumb has been restricted to the area in the middle of the trackbar shown by the solid bar and the extra tick marks. See Figures 2 and 3 for the template settings.

Figure 1. The trackbar control.



The template

The template supplied with this article (as well as the source code) will take care of almost any aspect of trackbar use. The only errors that the template will check for are that the application must be 32 bits and a variable should be assigned to keep the value of the slider thumb position.

The use of the template is pretty straightforward. You'll need to copy TRACKBAR.TPL to your \TEMPLATE directory and register it. You'll also need to copy the TRACKBAR.INC and TRACKBAR.CLW files to your \LIBSRC directory. You may also need to refresh the ABC class list so that the AppGen can see the trackbar classes. To do this go to any classes tab (such as on global properties or the trackbar control template) and click Refresh Application Builder Class Information.

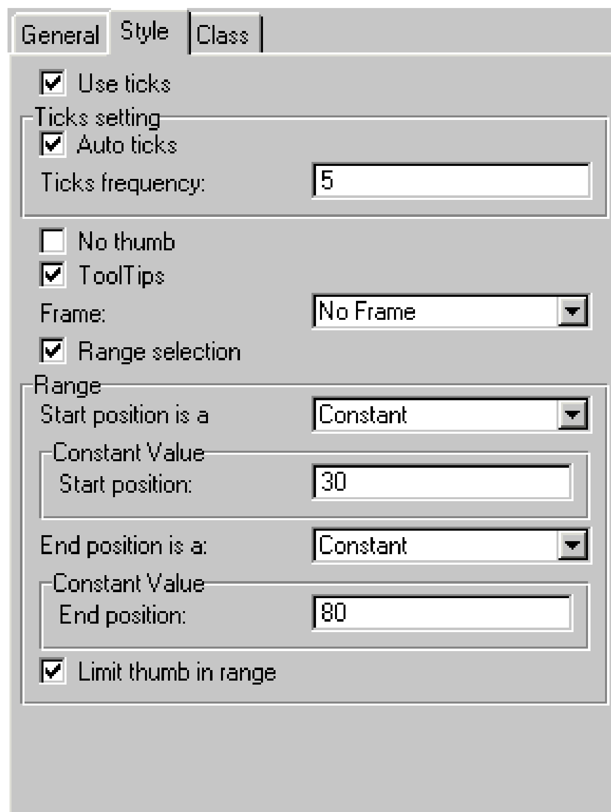
If your main concern is simply to have a trackbar on the screen and have your application be aware of the position value of the thumb, you don't need any embed code. Just populate the control template on the window and set the Slider Variable property on the template's General tab. The variable will be updated each time the thumb is moved.

Figures 2 and 3 show the template settings used to create the trackbar shown in Figure 1.

Figure 2. The trackbar template General settings.

General | Style | Class
Object Name: cTrack1
Slider variable: SliderVar
Position Slider with current value
Oriention: Horizontal
Ticks location
Ticks are: Bottom
Page Size
Page Size is a: Constant
Constant Value
Page size: 10
Limit Boundary
Lower limit is a: Constant
Constant Value
Lower limit: 1
Upper limit is a : Constant
Constant Value
Upper limit: 100

Figure 3. The trackbar template Style settings.



It is possible to have the control drawn with a border around it. In that case, the tooltip, which shows the current value of the thumb position, will be displayed at the edge of the control rather then moving with the thumb as the user drags it. Setting the checkbox on the style tab in the template extension dialog sets the tooltip. Without a border, the tooltip will follow the slider when the user drags it.

This control template is using a region control to visually place the control on the screen. Once the trackbar is created on the screen, the region control is destroyed.

There are some special cases where you need to use embed points in order to ensure the application behaves properly. One of those cases is when the control is placed on a tab sheet. Fortunately, I was lucky enough to have someone pointing out to me (thanks Robert!) that in this situation, the control will stay displayed no matter which tab is selected.

This ended up with the creation of two new methods called Hide and Unhide. You need to place the code in the `EVENT:NewSelection` of the sheet control and hide/unhide the trackbar control.

Listing 3. Hiding/unhiding the trackbar control on a tab.

Listing 3. Hiding/unhiding the trackbar control on a tab.

```
CASE EVENT()
OF EVENT:NewSelection
  CASE choice(?)
  OF 1
    CTrack1.Unhide()
    CTrack2.Hide()
  OF 2
    CTrack1.Hide()
    Ctrack2.Unhide()
  END
END
```

Hide and Unhide use the Window API function `ShowWindow` and set the second parameter to true (Unhide) or false (Hide).

Vertical trackbars can also be a problem. It seems that Windows places the smallest value of the range at the top of the trackbar. The simplest, best, and quickest workaround is to set the range using a negative value as the lower limit. So instead of calling SetLimit(1,50), you will call SetLimit(-50, -1) and you will have the trackbar displayed in a more natural way.

Of course, the template is also ABC compliant. Each trackbar object used in the procedure will appear under the Local Objects tree in the embeditor

In conclusion, this trackbar was a very enjoyable small project. I am also happy to donate the source to the Clarion Open Source Project.

Download the source

Pierre Tremblay has worked in the programming and corporate world for the last 16 years, and has been as an independent contractor for TopSpeed Consulting Division since April 1998. He is also a member of Team TopSpeed.

Copyright © 1999 by CoveComm Inc. All Rights Reserved. Reproduction in any form without the express written consent of CoveComm Inc., except as described in the subscription agreement, is prohibited. If you find this page on a site other than www.clarionmag.com, email covecomm@mbnet.mb.ca.

Vol 1, No 5
June 1999

Clarion magazine

Main Page
Log In
Subscribe
Open Source
Links
Mailing Lists
Advertising
Submissions
Contact Us
Site Index
ClarionMag FAQ
Download PDFs
Search ClarionMag

Call For Photos

Coming Soon: The ClarionMag Photo Album

One of the things I like best about Clarion conferences is the opportunity to put faces to names of people I've come to know through email or the newsgroups (or in a bygone era, Compuserve). Photographic evidence of the worldwide Clarion community has already appeared in the pages of Clarion magazine with conference reports from Australia and Argentina.

That said, there's no particular reason to wait for the next conference to see more ugly mugs. If you have a picture of yourself, your user group, or if you're a really lonely programmer, your favourite PC (just kidding – we really don't want any PC pictures), send them along. Photos from past conferences are also welcome. JPEGs or GIFs are preferable, and please include a caption or other descriptive material with each picture.

Send your favourite Clarion-related pictures to editor@clarionmag.com. Please indicate if you wish a copyright notice attached to those selected for publication.

Dave Harms, Publisher

BKO Enterprises, Inc.
Great Opportunity!
Salary to $125,000+
Pre-IPO Stock Opts
Sunny Boca Raton

etc 2000
If you're interested take our poll & let us know!

Copyright © 1999 by CoveComm Inc. All Rights Reserved. Reproduction in any form without the express written consent of CoveComm Inc., except as described in the subscription agreement, is prohibited. If you find this page on a site other than www.clarionmag.com, email covecomm@mbnet.mb.ca.

Vol 1, No 5
June 1999

Clarion magazine

Main Page
Log In
Subscribe
Open Source
Links
Mailing Lists
Advertising
Submissions
Contact Us
Site Index
ClarionMag FAQ
Download PDFs
Search ClarionMag

Feature Article

The Novice's Corner

Handling Many-To-Many Relationships

by David Harms

BKO Enterprises, Inc.
Great Opportunity!
Salary to $125,000+
Pre-IPO Stock Opts
Sunny Boca Raton

etc 2000
If you're interested take our poll & let us know!

NOTE: To avoid confusion with ABC terminology the file called Classes in the previous article in this series has been renamed to Courses. The prefix for this class has been changed from `CLS` to `CRS`.

In the previous article in this series I began developing a data dictionary and application to track information about students attending a university or college. I began by defining a student file as well as an address file, on the premise that students may have one or more addresses. This dictionary design reflects a one-to-many relationship between students and addresses, or, if you look at it from the other side, a many-to-one relationship between addresses and students.

This example relationship is obvious and easy to understand. Real data, however, is often a bit more complex. Sometimes any number of records from one file can be related to any number of records from another file. These many-to-many relationships are quite common in databases, and most likely you'll have to know how to handle them. In this article I'll examine such a relationship and outline how it can be defined in the data dictionary.

Adding Courses

It's now time to expand the demonstration application to handle not just students and addresses, but the courses for which students can register. You can probably guess at some of the fields required to describe courses:

- ID (keeping in mind last article's discussion about unique IDs)
- course title
- start date
- end date
- number of registrants
- instructor (suggests a many-to-one link to an instructor file)

As you create the data file keep in mind any fields which are defined in pool data and consider whether any other fields should be added to the pool. In the case of the Courses file the ID can come from the pool data (where it has the Do No Populate flag set). There are two date fields in the file and you will want them to have a standard date format, and you may wish to make them spin boxes as well. In either case a pool Date field is a good idea. Figure 1 shows the Courses fields in the data dictionary, and Figure 2 shows the Courses keys.

Figure 1. Fields for the Courses file.

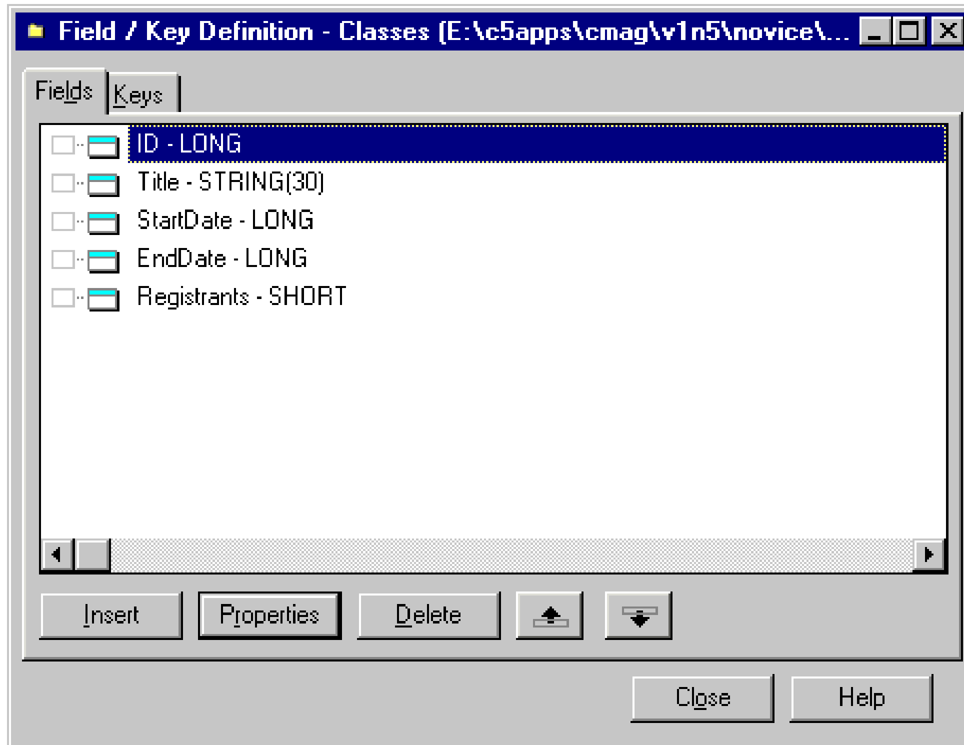
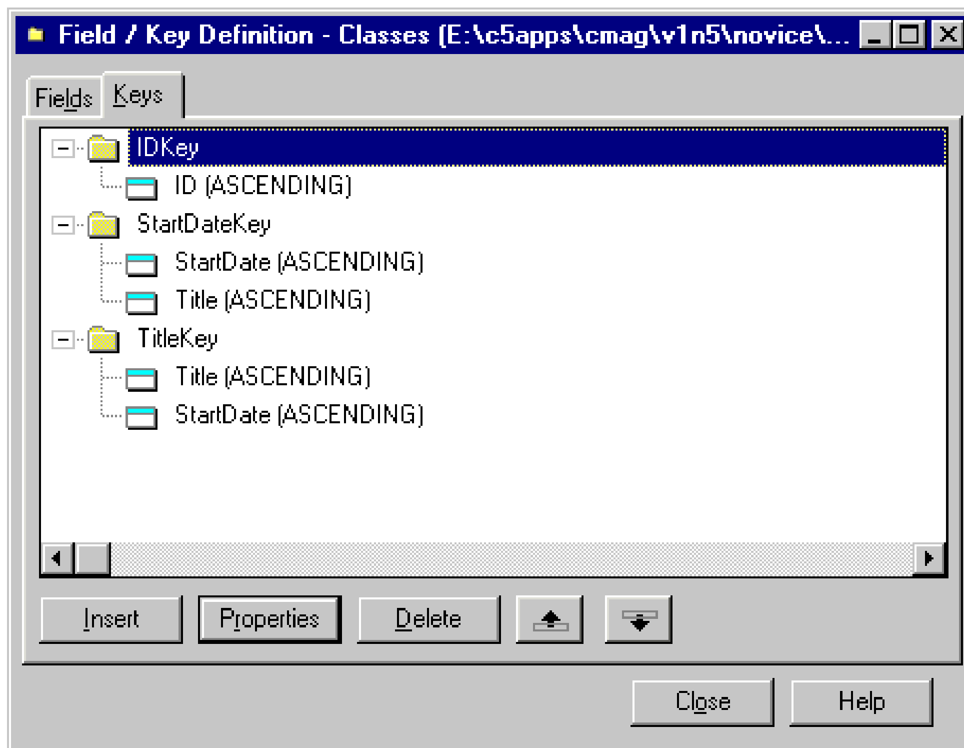


Figure 2. Keys for the Courses file.



Linking Courses To Students

Now you have a way to store courses and a way to store students. How do you link students to courses? The problem is that one student can take a number of courses, and any course can be taken by a number of students. The relationship is many-to-many, diagramed in Figure 3. The two triangles indicate the "many" aspect of the relationship.

Figure 3. A many-to-many relationship.

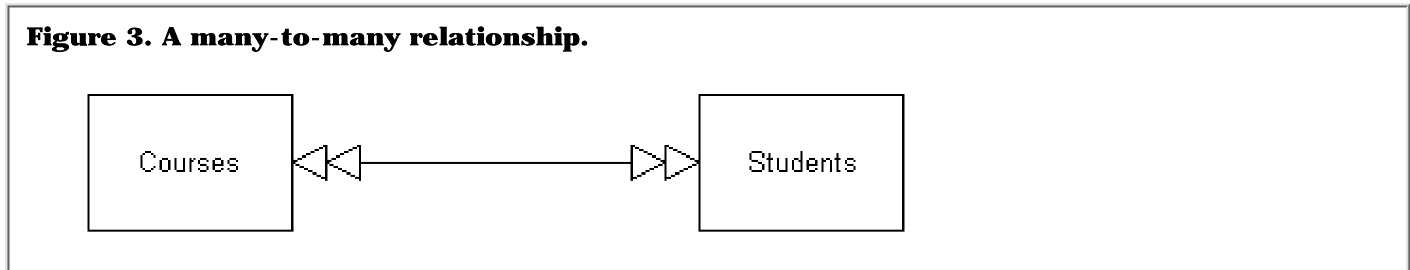


Figure 3. A many-to-many relationship.

If you examine the data dictionary's relationship editor you won't find any options for creating a many-to-many relationship, because it's almost impossible to create using just two files.

The problem is that from a student perspective you need to store an unknown number of course IDs, and from the course perspective you need to store an unknown number of student IDs. Some developers approach this the same way they do many-to-one relationships: they use arrays.

As I indicated in the previous article, arrays are generally a bad idea for linking files, and they're even worse in a many-to-many situation. Arrays by definition limited in size (at least in Clarion), which means you have to make the array size as large as the highest possible value, thereby wasting a lot of space. Furthermore you cannot use arrayed fields in keys. That's not a problem in a many-to-one where only one side needs to be keyed, but in a many-to-many both sides need to be keyed. You need to see which courses a given student takes, and which students are in a given course.

The answer is to use a file as an intermediary between Students and Courses, as shown in Figure 4.

Figure 4. An intermediary for managing the many-to-many relationship.

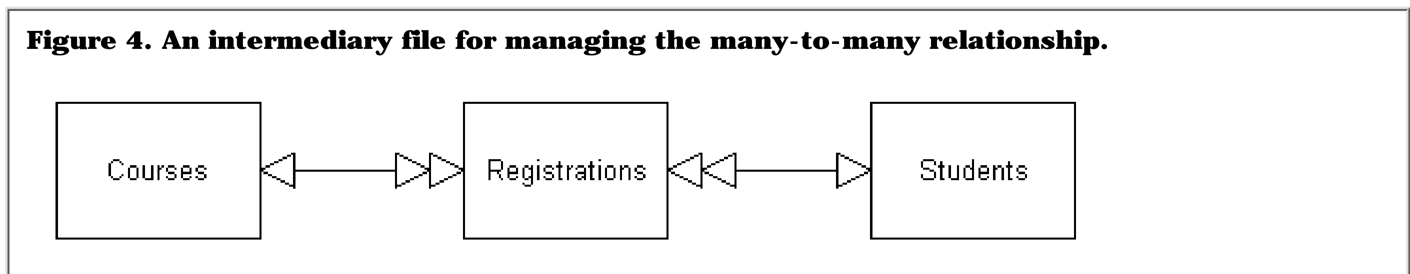


Figure 4. An intermediary file for managing the many-to-many relationship.

In Figure 4 the single triangles indicate the "one" side of the relationship and the double triangles indicate the "many" side. As this diagram shows, the many-to-many has been broken down into two many-to-one relationships. This is the standard approach to handling many-to-many situations.

The linking Registrations file is simplicity itself – it needs to contain only three fields: a unique autonumbered Registration ID, a student ID and a course ID. (You might want to add several additional fields, however, including the date the registration was taken.)

> **NOTE**: In a TPS or other flat-file (i.e. non-SQL) database you can get away without the unique autoinc key for this record, but it's a good idea to have it anyway as you may wish to link other records to the registration record.

Figure 5 shows the fields used in the linking file, and Figure 6 shows the keys.

Figure 5. The Registrations file fields.

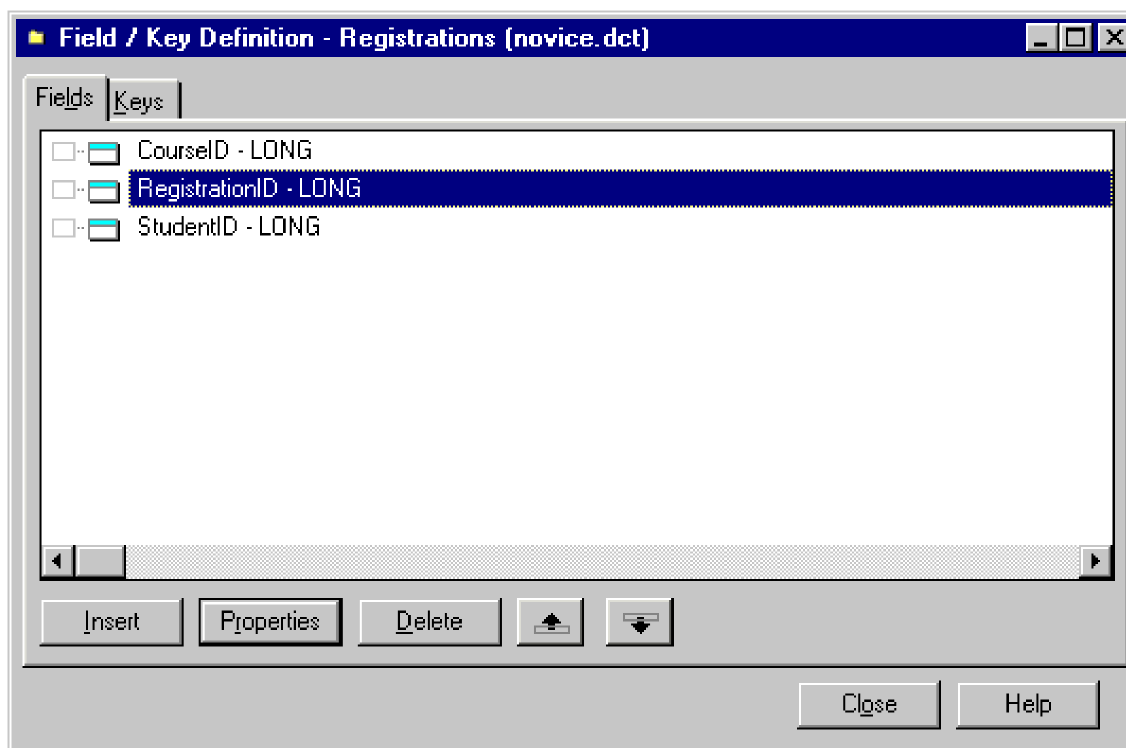
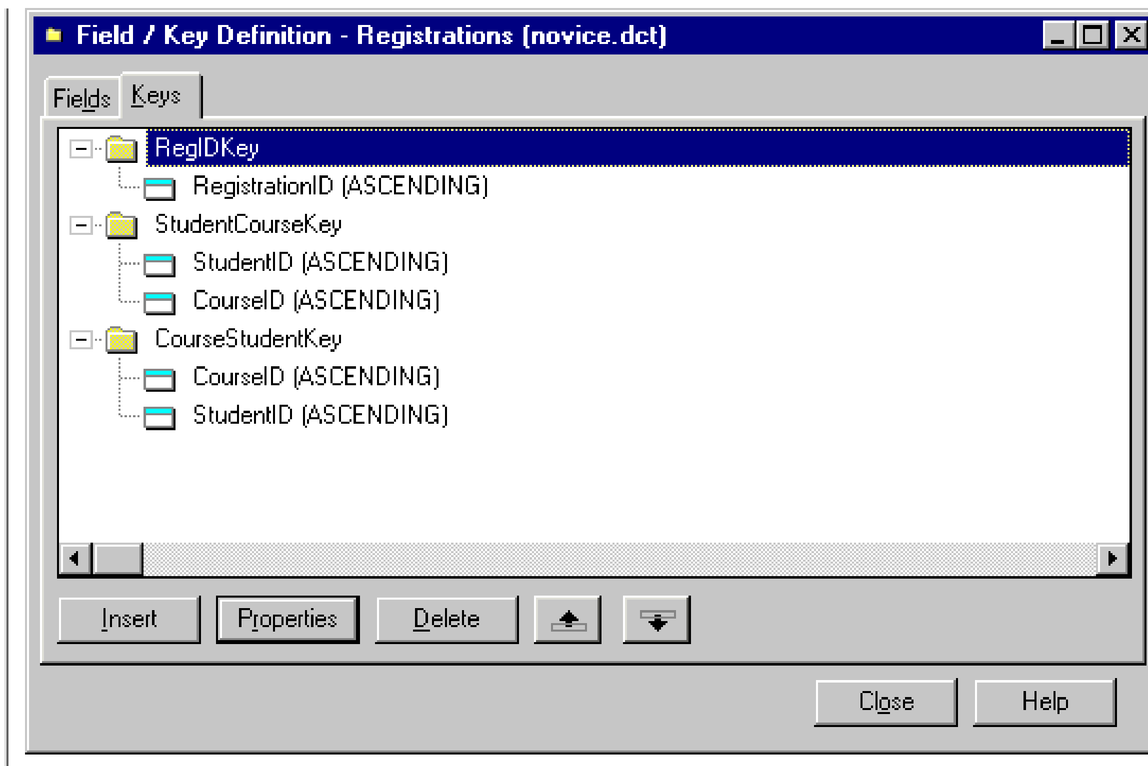


Figure 6. The Registrations file keys.

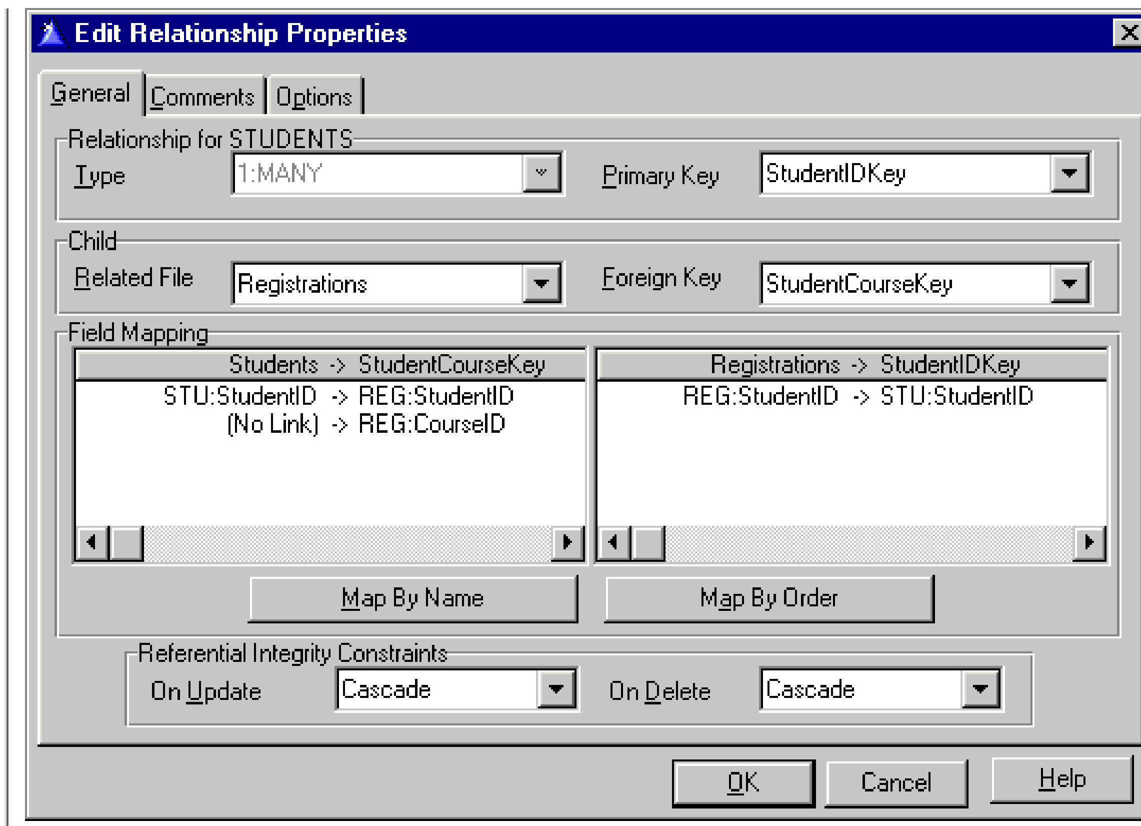


StudentCourseKey and CourseStudentKey both have the course and student IDs as elements, and if you look at the data dictionary you'll see that they have the unique attribute as well. This prevents a student from being registered for the same course twice.

Now define the relationships in the dictionary editor. Although both the keys in Registrations have two elements, you only want to make the links on the first elements in the key in each case. Figure 7 shows the relationship between Students and Registrations.

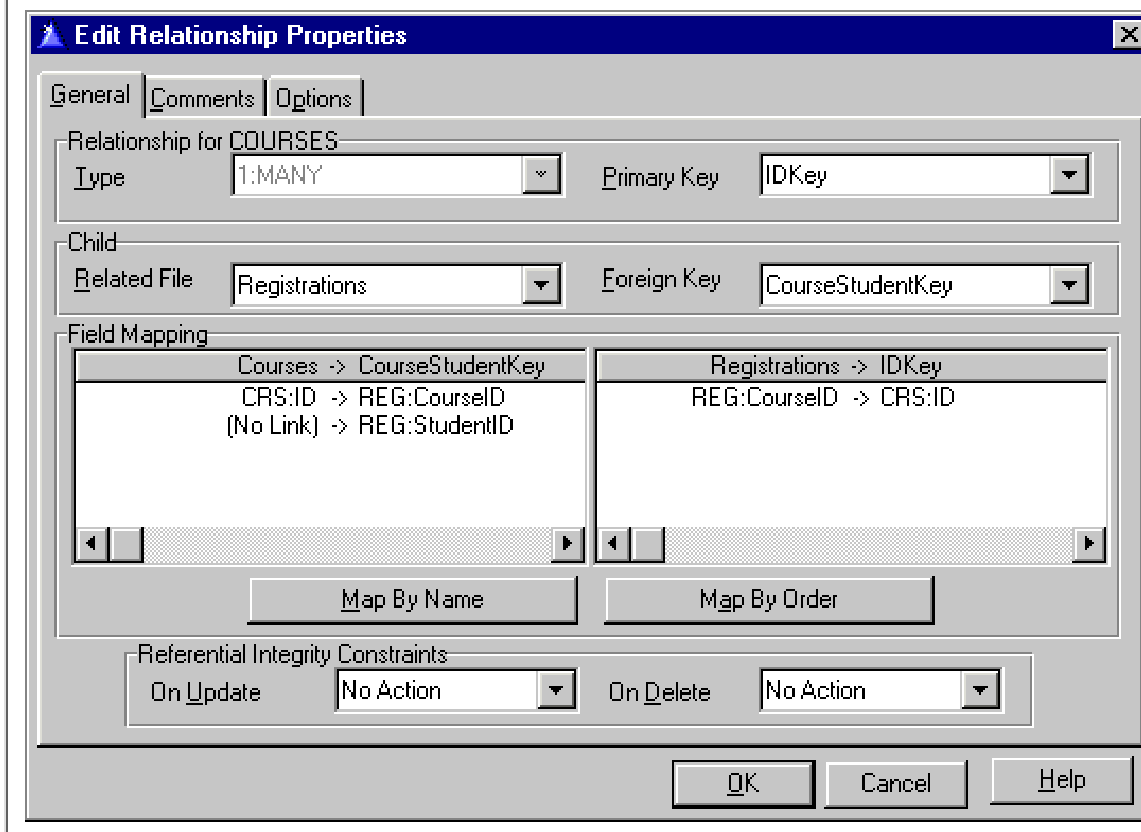
Figure 7. The relationship between Students and Registrations.

Figure 7. The relationship between Students and Registrations.



A similar relationship exists between Courses and Registrations, as shown in Figure 8.

Figure 8. The relationship between Courses and Registrations



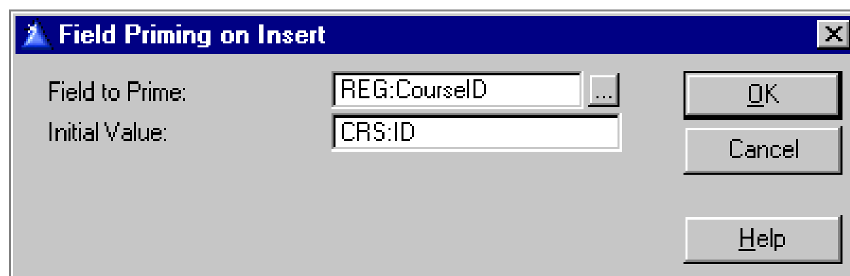
If at this point you use the Browse wizard to create a browse and update form for Courses, you'll find a browse of Registrations records on the Course update form. That may not be the final place you'd like to have it, but it will show you how the link works. (Another option is to populate a browse of Registrations records below the Courses browse, and restrict its records based on the relationship with Courses.)

You'll want to display the student name on the Registrations browse rather than just the ID. Highlight Registrations in the file schematic and click Insert. Select the Students file from the related files list. This will ensure that the related Students record is retrieved for each Registrations record (see the accompanying application for an example).

Since the only fields in the Registrations file are two `LONG` `ID`s, you'll need to set these values when you add a registration. Whether you're updating Registrations records from a browse on the Courses update, or from a child browse you've placed on the Courses browse, you know that the current Courses record is in memory. Use Field Priming on Insert on the Registrations update form to preload `REG:CourseID` with `CRS:ID`. Figure 9 shows the appropriate Field Priming on Insert setting.

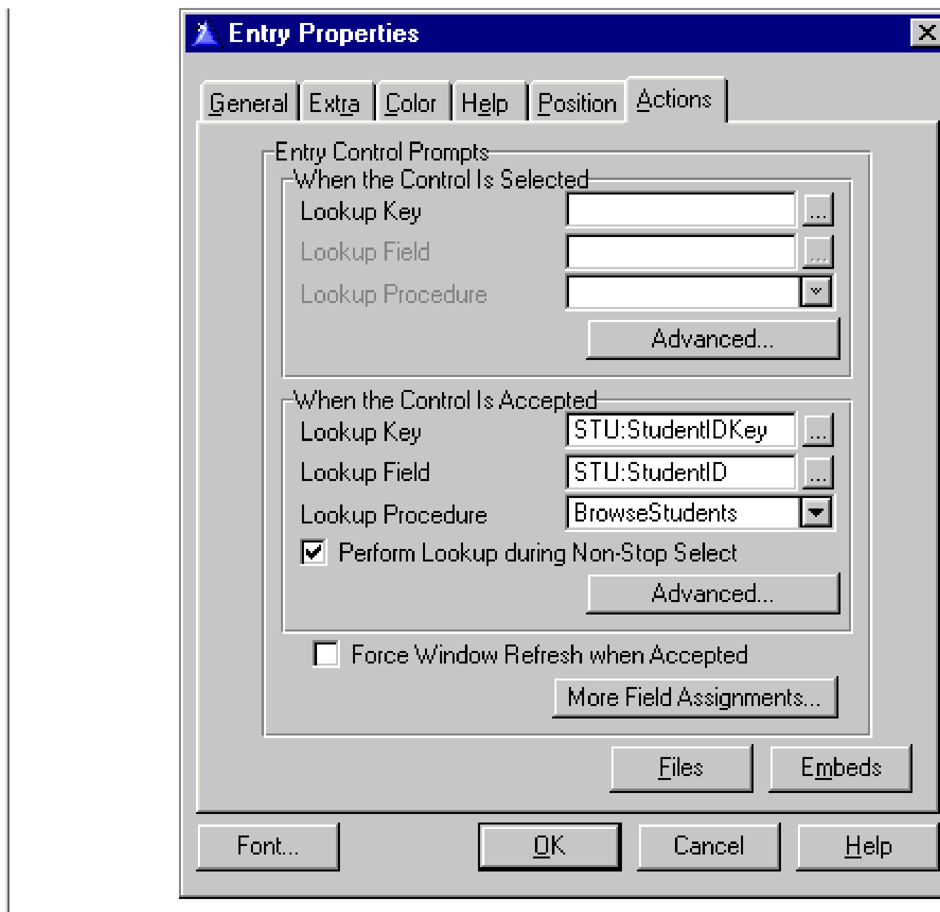
Figure 9. Priming `REG:CourseID` on insert.

Figure 9. Priming REG:CourseID on insert.



Since you already have `REG:CourseID` there's no need to display it on the window: delete the entry field. Now all the user has to do is choose a student. You probably don't want the user entering in the student's ID directly, so you should provide a lookup on a list of students. One way to do this is to set up the `REG:StudentID` field's actions to do a lookup, as shown in Figure 10.

Figure 10. Lookup settings for REG:StudentID.



After entering the settings in Figure 10 and saving the changes, populate the FieldLookupButton template on the window and on its Actions tab select `?REG:StudentID`. Now you have a button you can use to force a lookup on the Students browse.

Next bring up the property window for `REG:StudentID`. Check the Hide box so the ID won't be visible. In the form's file schematic, click on the Registrations file and choose Students from the Related Files tab. Populate two string fields on the form and on the string Properties window, Use field, enter `REG:FirstName` for one field and `REG:LastName` for the other (or use the field lookup button to the right of the Use prompt). The relation manager will take care of looking up the student record from the ID and will display the name of the student associated with the registration.

> **NOTE**: This is just one approach to selecting records. Another would be to use the FileDrop template (though this code is due for major revision and may not be a safe choice). You may also want to consider a third party solution like ProDomus' highly-regarded PDLookup templates which add Quicken-style incremental lookups.

Bending The Rules

If you look at the data dictionary in the accompanying example you'll see that the Registrations file has a Registrants field which is defined as a SHORT variable. Assuming that a successful link is created between courses and students it should be possible to calculate this value, so it isn't strictly necessary to keep it on the course record.

In fact, good relational design suggests that this is unnecessarily duplicated data and should not be stored. In reality there are often tradeoffs involved in designing a database. Your aim should be to avoid situations where conflicting data can exist, but sometimes performance requirements must also be addressed. If one of the requirements is to display the number of registrants on a browse of courses, and you don't keep a total on the Courses record, then you'll have to loop through the

registrant information for each record.

One easy way to do this is to use a hidden browse with the child records and a total (count) field. If you take this approach make sure that you set the child browse's `ActiveInvisible` property to True as the default behaviour for browse objects is to become idle when hidden.

If there are a large number of child records or you frequently run reports or other processes where the number of registrants is required then it may be to your advantage to count the registrations whenever one is added or deleted and update a Registrants field with this value.

You may think at first glance that a `BYTE` value would suffice for Registrants, but for a particularly large lecture course there might be more than 255 students. Should that happen with a `BYTE` variable the numbers would wrap around and the displayed count would be incorrect. If there's any possibility that you'll overrun the variable's capacity then use the next larger variable. A `SHORT` has a maximum value of 32767, and while course overcrowding is often a problem, its not likely to get that bad.

Many-To-Many Redux

In my initial design of the course file I made the assumption that there would be only one instructor per course. This isn't necessarily the case. There may be multiple instructors, or an instructor and teaching/lab assistants who should also be associated with the course. A simple many-to-one relationship would be too restrictive. To handle this I'll need to create a linking file just as I did for the course registrations. As with the Registration file, this linking at a minimum would need only two IDs.

The only difference between this many-to-many and the registrations many-to-many is that this one is a bit more difficult to recognize. In the software requirements (if a formal document existed) you would probably come across the term "Course registration," which might have led you to think about storing this information in its own file.

What if a suitable term hasn't been defined for the link between instructors and courses? You can discover the need for such links by examining each file relationship carefully and asking yourself if the relationship will satisfy all the needs of the application.

By now you should be starting to see some other possible links between data. If you're storing information about instructors, then perhaps there's some possible duplication of the kind of information you keep about students. Perhaps a generalized contacts file would be better, with a `Type` field to differentiate between students and instructors (and perhaps staff as well). This way you can use the same approach for instructor addresses as for student addresses.

Telephone numbers (and email addresses and the like) are another potential area for generalization. Rather than having two or three telephone number fields in the names file you might want to go with a Phones or other kind of contacts file. Remember that you can assign field pictures at runtime, so you could even use a simple string field and use various pictures to format the string for phone numbers, email addresses, and so forth. You might want to keep this kind of configuration information in an INI file (not a great idea) or a single-record control file in your database (a better option as it allows for encryption).

Paranoid Anticipation

When I analyze or create a database design I'm constantly asking myself how the data will be used given the current requirements and what I anticipate the future requirements will become. Few of us ever actually finish a software project. We may leave it or hand it off to someone else, but there's always work to be done.

I find that the better I am at anticipating what the user will want to do with the program, the more robust my database design is likely to be. In

the next article in this series I'll look at "use case" analysis, a text-only tool that I find helpful in fleshing out a design and discovering hidden requirements.

Download the source code

David Harms is an independent software developer and the co-author with Ross Santos of Developing Clarion for Windows Applications, published by SAMS (1995). His company, CoveComm Inc, publishes Clarion Magazine.

Copyright © 1999 by CoveComm Inc. All Rights Reserved. Reproduction in any form without the express written consent of CoveComm Inc., except as described in the subscription agreement, is prohibited. If you find this page on a site other than www.clarionmag.com, email covecomm@mbnet.mb.ca.

Vol 1, No 5
June 1999

Clarion magazine

Main Page
Log In
Subscribe
Open Source
Links
Mailing Lists
Advertising
Submissions
Contact Us
Site Index
ClarionMag FAQ
Download PDFs
Search ClarionMag

Feature Article

FileManager II – ABC Design

by David Bayliss

BKO Enterprises, Inc.
Great Opportunity!
Salary to $125,000+
Pre-IPO Stock Opts
Sunny Boca Raton

etc 2000
If you're interested take our poll & let us know!

This article is part two of the ABC design document on the FileManager class, and you really should read the first part first as this article simply continues the FileManager code overview. I hope I have also provided enough hyperlinks that people returning to the article for reference will be able to dig out the information they require. In this article I'll look at some of the administration functions, the error handling and the snapshot mechanism.

Whilst some of the aims of the class can be gleaned simply from reading this article, most fruit is available to those that actually settle down and read the ABC code along with the corresponding comments. This is actually something I would always encourage you to do. The backbone of ABC amounts to around 4,000 lines of code, so if you aim to master 100 lines a day you will understand the basic ABC paradigm completely within eight weeks! The FileClass amounts to 25% of that work.

Administration

This section details those methods provided almost entirely as wrappers upon internal information for the benefit of higher level methods and/or methods outside of the FileManager.

```
ClearKey PROCEDURE(KEY K,BYTE LowComp,BYTE HighComp,BYTE High)
```

This method is there to provide a shortcut for a piece of template code that occurred very frequently. Essentially it handles the problem of a multi-component key where you wish to perform a `SET(KEY,KEY)` but you only know the major components. In order to ensure the `SET(KEY,KEY)` gets you to the start of all the records you require you need to clear the low order key components. Clarion does not have a `CLEAR(KEY)` so the FileManager provides one for you.

Rather than clear the whole key the routine allows you to specify the low (majormost) and high (minormost) components you want cleared. This is to allow minor component clearing to happen when the major components have already been filled in.

The method works by first performing a `SetKey` so that the current record of the `FileKeyQueue` holds information for the current key. Then the key components are stepped through from low to high and the `KeyFieldQueue` is fetched to retrieve the information for the current component (see `AddKey` in the previous article). The `GET` is error trapped with a simple return if the component doesn't exist. This is to allow the `HighComponent` to be specified as 255, meaning "to the end."

The XOR logic illustrates a useful trick (and hides a complexity!). Remember that as you are trying to get all the records in a `SET(KEY,KEY)`, you might assume that means you just `CLEAR` all the key

values low. But wait a minute; suppose you are about to do a PREVIOUS rather than a NEXT. Then you need to clear all the values high. This works in the common case of ascending key components, but remember it is possible to have descending key components. Worse yet you can mix ascending and descending in the same key. If you sit down with pencil and paper (you may be able to do this in your head, but I needed pencil and paper) you will find you need to clear a component low if it is ascending and you are clearing low, or if it descending and you are clearing high. This can be expressed using a disjunction (OR) of conjunctions (AND) but the XOR operator wraps it up perfectly and goes down to one machine instruction. I could have coded this more tightly still as:

```
CLEAR(SELF.Keys.Fields.Field,CHOOSE(~(SELF.Keys.Fields.Ascend XOR High)))
```

But I thought that might be just a little too scarey.

```
GetComponents PROCEDURE(KEY K),BYTE
```

This simple little method simply returns the number of components in a key. It uses SetKey and the fact that there is one KeyFieldQueue record for each component of the key.

```
GetEOF PROCEDURE,BYTE
```

The FileManager has a very specific meaning for EndOfFile: it means the last attempt to NEXT or PREVIOUS a record failed because the end of file has been reached. Specifically, if you have a file with 10 records EOF is true after the 11th NEXT, not the 10th. As such GetEOF is really just a short hand to detect a specific error condition.

The functionality could almost certainly be achieved by looking at the return code from NEXT/PREVIOUS and then delving to see what the error identifier was. Again this is a situation where the FileManager does work simply to reduce the amount of coding required by users of the object.

```
GetField PROCEDURE(KEY K,BYTE Component),*?
```

This method is used to return an ANY variable corresponding to a given component of a key. I didn't want to have to protect the rest of my code against GetField returning a null so the procedure ASSERTs that the incoming component will be found. In other words, GetField gracelessly handles out of range components.

This does illustrate another agenda within ABC: offensive programming. Defensively I would have coded so that an out of range value returned a null, which would take two lines of code. Then on the receiving end nulls would have been handled, presumably in some "see if we can still keep going" fashion.

There are four calls to GetField in abfile (i.e., this method is relatively underused). Each would have had to temporarily store the GetField result, test for the null and do something smart with it. This might have taken five lines of code each (one for the declaration, one for the extra assign, two for the null test, one to handle the null case). In total I would now need 22 lines of code to handle something that should never happen as opposed to the one line of code used in ABC. Doing that throughout a heavily integrated file like ABFILE could turn 2000 lines of code into 40,000 lines of code 95% of which would be rarely executed and thus minimally tested. QED.

```
GetFieldName PROCEDURE(KEY K,BYTE Component),STRING
```

This is really there for the benefit of methods using the `PROP:Filter` technology on a view. It provides the `BIND` name of a give key component.

```
GetName PROCEDURE,STRING
```

The FileManager has to cope with two possibilities for the name of a file. It may either be a constant or a variable (the latter corresponds to the case where the `NAME` attribute on a file contains a string variable). `GetName` is there to encapsulate this dilemma from the rest of the class. If a variable file name has been assigned then it returns that, otherwise it assigns the constant provided to it by the driver itself.

```
KeyToOrder PROCEDURE(KEY K,BYTE MajorComp),STRING
```

This method really takes `GetFieldName` one logical step further. Rather than just return a field name corresponding to a key component, this method returns an `ORDER` clause (in Clarion syntax) that is equivalent to this key starting at component `MajorComp`. A value of one thus gives the whole key as an order clause, two skips the leading component etc.

Note that the null key case is defended against. This is because it is totally reasonable to have a null key specified as the sort key of an object (corresponding to not specifying a key in the file schematic).

The only real complexity is in the `RetVal` assignment. The first `CHOOSE` is there to prepend the field name with a comma only if the string being built up is non-null. The second `CHOOSE` is there to place a leading '-' before a descending key component (the view driver treats –string as a descending string, it does not convert it to a number as the language would).

```
SetKey PROCEDURE(KEY K),PROTECTED
```

`SetKey` is used to fetch the correct record within the FileManager key queue for the usage of a key passed in to it. You cannot sort a queue on a reference field so the method has to loop through the queue finding a match. Files don't have that many keys so this should not be too onerous. I could start the method with a check to see if the current record value already matches as a kind of first level cache, but the downside is that this would hide a raft of bugs where people had not done a `PUT` after modifying the key information.

The loop illustrates an interesting and occasionally useful quirk of Clarion. You can have loop head and loop tail conditions (`WHILE` and `UNTIL`) in the same loop. The conditions are tested (and code body executed) in the order they appear lexically.

Again note the assert. A failure to set the key throws an error; see the discussion in `GetField`.

```
SetName PROCEDURE(STRING Text)
```

This method is a counterpart to GetName; it only allows the name to be assigned if there is an underlying variable for the NAME attribute of the file. By having the GET/SET in the FileManager the burden of tracking the global variable name disappears (it simply becomes the province of the dictionary). This makes it far easier to have an automated path assignment system built in.

Error Handling

The action of the [ErrorClass] has already been covered, however each FileManager re-vectors the error manager calls through its Errors property. This serves one main purpose: it allows a global object to be referenced from within base class code. The secondary purpose is to make the error handler used by the FileManager re-assignable. This is useful as the file system is one of the major generators of errors and the file calls are usually out of the direct control of the programmer. The ability to intercept errors on a file by file basis allows fine grain recovery mechanisms to be written. In addition to having a single vector point, the FileManager has a small suite of routines through which all FileManager/ErrorClass interaction is managed. Again the purpose is to make errors and recovery mechanisms overridable with a minimum of effort.

```
GetError PROCEDURE,SIGNED
```

The FileManager stores the last file error thrown within it. The number is the ErrorClass number, and it has nothing to do with ErrorCode or Error. It should be noted that ErrorCode et al are not valid upon return from FileManager methods. In particular it is quite probable that the FileClass (coming in a future major release) will not utilise ErrorCode and Error in normal operation and thus the FileManager will not even have error codes available. The error suite is one of the instances of the FileManager trying to smother an encapsulation leakage coming from underneath.

```
SetError PROCEDURE(USHORT Number)
```

This method separates out the recording of an error condition from the Throw (or exception) that the error could raise. Occasionally this is used to simplify internal coding, but more usually it is used in the Try*Action* methods so that they can return an error signal and leave the ErrorClass able to Throw the error if the caller requires.

```
Throw PROCEDURE(USHORT ErrorNumber),BYTE,PROC,VIRTUAL
```

This function is purely a syntactic convenience. It is equivalent to a SetError followed by a Throw.

```
Throw PROCEDURE,BYTE,PROC,VIRTUAL
```

This routine takes the last error number (as recorded by SetError) and simply forwards it to the ErrorClass stored in SELF.Errors. The main purpose of this routine is simply to provide a common focus point (and thus override point) for the FileManager error handling. The return value comes from the ErrorClass and denotes the severity level as attached by the error class. This could be used to provide a sophisticated error recovery mechanism, by default most Throws are considered fatal and this facility is not used.

It is worth nothing that although Throw does not pass on the file label at this point, the ErrorClass does have access to the file name as this has been set up by the [SetThread] method as detailed in the [previous article].

```
ThrowMessage PROCEDURE(USHORT ErrorNumber,STRING Text),BYTE,PROC,VIRTUAL
```

This is a simple extension to `Throw` to allow an extra message to be passed on to the `ErrorClass`.

Snapshots

The snapshot interface's purpose is to allow file state and buffer contents to be saved and restored by anyone without them having to know the structure of the file. The routines all use a handle to denote a particular state. This handle is undefined (presently it is an ID number within a queue.) Eventually these routines will become vectors for fresh instances of the file class to be created and destroyed.

The words buffer and file have specific meanings. Buffer means the contents of the current record; that is the record buffer but also the memo contents. Blob contents are not stored as the overhead is potentially too onerous. File means buffer plus additional file state information such as Held, Watched, auto-increment done etc. For this reason all of the Buffer methods are fairly cheap involving only memory copies, while the File methods also involve disk access.

```
EqualBuffer PROCEDURE(*USHORT Handle),BYTE,VIRTUAL
```

This method is used to check if the current record contents differ from those when the snap shot (denoted by the parameter) was taken. For example, this might be used to see if a cancel on a form should be allowed to happen without user intervention.

First the `Handle` is looked up in the buffer queue; this gives the previous contents of the record buffer which can be compared byte for byte against the present values (this function is boolean - it doesn't say how the two buffers differ). If the two record buffers are the same the routine steps through the memos of the file seeing if they differ. The stored memo buffers are (by convention) stored consecutively in the queue following the record buffer. The present contents of the memos are retrieved by using `MyFile{PROP:Value,-memonumber}` (the negative number indicates this is a memo). This was necessary as it is not possible to store `ANY` references to memos as memos are created on the heap at file open time (on each thread) and are thus highly treacherous when involved with references.

```
RestoreBuffer PROCEDURE(*USHORT Handle,BYTE DoRestore=1)
```

This routine is used to restore the contents of the file buffer to the point they were when the `SaveBuffer` was called. If you pass in a zero as the second parameter then no restoration is done but the memory is freed. Commencing with C5EEA this routine actually becomes a shell that calls into `RestoreBuffer(handle,filemanager,byte)`.

```
RestoreBuffer PROCEDURE(*USHORT Handle,FileManager FM,BYTE DoRestore = 1),PRIVATE
```

This routine allows the contents of a buffer to be restored to the present file buffer from contents snapshotted by the passed in FileManager. Now in general restoring to a file other than your own is a dangerous, unmaintainable and generally very stupid thing to do (this is why the only public interface to RestoreBuffer passes in SELF). However in the particular case where the "other" file is absolutely identical structurally to your own, and is guaranteed to be so, it does give an extra degree of flexibility. We use this facility when dealing with aliases. However when reading this code you should generally assume that Frm and SELF are the same thing (if you're writing it, the distinction is vital of course!) Other than that, this code is essentially analogous with EqualBuffer, the only extra being KillBuffer which first frees them memory used for the buffer contents and then kills the queue record.

```
RestoreFile PROCEDURE(*USHORT Handle)
```

This is used to restore a file to the state it was in when the snap shot was taken. The current file position, sort sequence, held and watch state are all recorded along (since C5EEA) with the auto-increment state. Note that additionally the record contents are restored after the file position. This is to allow for instances where the current record had begun to be modified at the point the snap-shot was taken.

As with RestoreBuffer, RestoreFile has been split out to aid the use of aliases, or more specifically, to allow FileManagers of aliased files to re-vector their methods through the Filemanager of the actual file without corrupting the current state of the actual file.

```
RestoreFile PROCEDURE(*USHORT Handle,FileManager FM),PRIVATE
```

The file state (as opposed to record contents) is retrieved from the Saved queue. The Saved.Key element is the key number of the key active when the snapshot was taken. If this is non-zero then the key reference is found from the file driver and used in the RESET (otherwise the File is used). Because Watched and Held are read-only properties in the file driver they have to be restored by re-arming them and applying a NEXT. Having performed the NEXT (and thus "corrupted" the buffers) the buffers are restored. The auto-increment state is then put in place. Note the PUT on the SELF.Info to store that information for the current thread.

Actually this raises a slight cheat. Many of the file methods need to start with a SetThread for reasons previously described in FileManager I. Many then needed a UseFile to prime the lazy open. UseFile also needed to do a SetThread, so SetThread was often called twice. This is clearly inefficient so we cheated and allowed an information leakage that stated that UseFile does, and will always, perform an implicit SetThread. Again we find that ABC is not just about science, it is also about engineering. We allowed for one assumption and removed 15 lines of code and an efficiency drag on most of our core functions, and also lost some conceptual purity.

```
SaveBuffer PROCEDURE,USHORT
```

This code snapshots the current contents of the record buffer; most of the code is analogous to `EqualBuffer`. The interesting piece is the allocation of the `Id` to act as a handle to the outside world. At first sight you can simply get the number of records in the queue, add on one and you have your new `Id`. Better yet, you don't need to store the `Id` in the queue; you simply use the `Id` as a record number.

Further, in just about all the testing you ever do, it will work beautifully. But sometimes, somehow, it will corrupt when the users use it. The reason is that simply counting the records only works if Save/Restore pairs are performed in a stack-wise manner. If a deletion from the queue has happened in the middle then the next `RECORDS` will return a value lower than the current highest `Id`. Actually it will even work if the restores are not done stack-wise provided the result has been stack-wise by the time you do the next `Save`. If you do the `Save/Restores` in an unpaired way you will actually get the identifiers duplicated in the queue and havoc ensues. The solution is that you get the final record in sorted order and then add one on to whatever you receive back. `DupString` is a private member function used to allocate heap for, and copy the value into, a temporary string (like strdup in C++).

```
SaveFile PROCEDURE,USHORT
```

This method is the mirror of `RestoreFile`. Note that rather than replicating the buffer storage code, `SaveFile` simply calls on to `SaveBuffer` and stores the result. It is worth mentioning that the handles returned from `SaveFile` have no relation to those returned by `SaveBuffer`. You cannot `SaveFile` / `RestoreBuffer` or vice versa.

One slight tweak is the storage of the current key. You cannot simply save a key reference as that will not work when you are restoring to a different FileManager. Instead you have to store an ordinal number corresponding to the declaration order of the key. That number is computed using the loop. Note too the usage of a cast from `CK` (which is a long) to a key reference:

```
K &= (CK)
```

The rule is that a numeric value can be assigned in place of a valid reference of the right type. `CK` in itself is not a value (it is a variable) so the parenthesis is used to form a value. This form of casting (which can be used in conjunction with `ADDRESS` and references) allows all of the (horrible) type conversion common to C++. It should be used extremely sparingly, but when needed it is brilliant.

A Thought

If you have been following this article in the source code, reading and understanding as you went, it is quite probable that by this point you are thinking. Hey! This stuff is all obvious; what is all the fuss about? If so, this article has worked. If not it may be worth your while backtracking to see where the confusion enters. Object systems are hierarchical, one layer builds upon another. Therefore, comprehension of object systems tends to be hierarchical. If one layer doesn't make sense it typically means you didn't quite catch hold of the layer underneath. Happy hunting...

Read Part III

David Bayliss is a Software Development Manager for Topspeed Corporation. He is also Topspeed's compiler writer and the chief architect of the Application Builder Classes.

Copyright © 1999 by CoveComm Inc. All Rights Reserved. Reproduction in any form without the express written consent of CoveComm Inc., except as described in the subscription agreement, is prohibited. If you find this page on a site other than www.clarionmag.com, email covecomm@mbnet.mb.ca.

Vol 1, No 5
June 1999

Clarion magazine

Main Page
Log In
Subscribe
Open Source
Links
Mailing Lists
Advertising
Submissions
Contact Us
Site Index
ClarionMag FAQ
Download PDFs
Search ClarionMag

The Clarion Challenge

Clarion Challenge: A String Parser Revisited

by Dave Harms

Last month's Clarion Challenge was a bit more complex than the previous challenge, so there were correspondingly fewer entries. It was also a much more difficult to evaluate since the specification was somewhat open-ended.

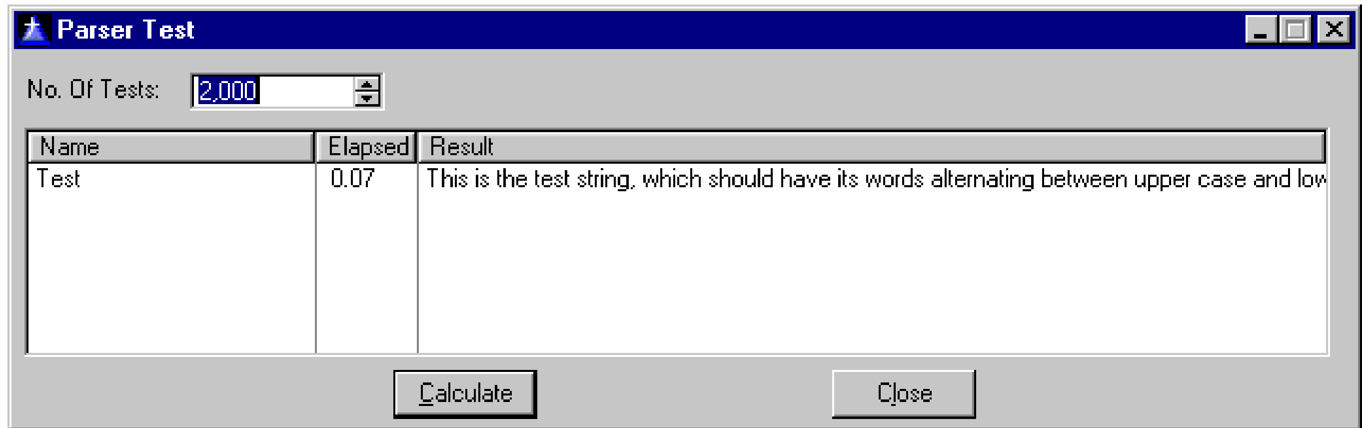
In fact, it became clear that it would not be possible to fairly compare the various entries given how differently they had been implemented. Accordingly, I've written the challenge and have asked the entrants to adapt their code to a very specific code framework. This will also make it a lot easier for anyone who's still interested to submit some code, since it's much more a case now of plugging and playing your code within a defined test environment. This same approach will be used for all future challenges, although I expect they'll be considerably smaller in scope.

This revised challenge includes a test application adapted from that used for the previous Clarion Challenge, and shown in Figure 1.

BKO Enterprises, Inc.
Great Opportunity!
Salary to $125,000+
Pre-IPO Stock Opts
Sunny Boca Raton

etc 2000
If you're interested take our poll & let us know!

Figure 1. The test application.



The test application isn't an APP file – instead it's made up of one source file (parser.clw) and one project file (parser.prj). To run the application use Project|Set and choose a File Type of Project File (*.prj). Select parser.prj. You can then compile and run as you would an APP (but you don't have to wait for code to generate).

The parser application uses two classes to implement the code. There is a base class called ParserBaseClass. This class is NOT to be modified. All changes are to be made to a derived class called MyParser. I've split the challenge into two classes for exactly the same reason that your ABC applications consist of base classes and derived classes. The base class contains code and data that must be the same for all implementations, and the derived class contains entrant-specific code.

The Test method shown in Listing 1 is declared in the base class.

Listing 1. The Test method.

```
ParserBaseClass.Test                    procedure
!--------------------------------------------------------------!
! DO NOT MODIFY THIS METHOD!!! And do NOT derive it.
!--------------------------------------------------------------!
TempString  string(200)
x           long
   code
   self.Reset()
   self.AddDelimiter(' ')
   self.SetString('This is the test string, which should '|
      & 'have its words alternating between '|
      & 'upper case and lower case. The actual test will '|
      & 'parse Clarion code and capitalize keywords.')
   self.BeforeTest()
   loop x = 1 to self.GetTokenCount()
      TempString = self.GetToken(x)
      if x % 2
         TempString = upper(TempString)
      else
         TempString = lower(TempString)
      end
      self.PutToken(x,TempString)
   end
   return
```

The Test method is the only method you may not derive. All the rest are fair game, although in the example I've only derived those methods I think you're likely to want to use. Whatever you do with the derived class keep in mind that you cannot change how the test is executed.

All of the derived methods (in MyParser class) are virtual methods, which essentially means they function like embeds and will be called automatically by the Test method, even though that method is declared in the base class. These methods are shown in Listing 2 (they're in the source file – you don't need to copy from the listing).

Listing 2. The derived methods.

```
!---------------------------------------------------------------!
! You may not modify anything above this point except for
! the MyParser class declaration at the top of the file.
!---------------------------------------------------------------!

MyParser.AddDelimiter         procedure(string Delim)
!---------------------------------------------------------------!
! You don't need to modify this method but you may wish to
! if you have a better idea for how to store delimiters.
! By default this method adds records to the DelimQ. To get
! the delimiters that have been added you can query the queue
! as you would any other queue.
!---------------------------------------------------------------!
   code
   self.DelimQ.Delim = Delim
   add(self.DelimQ)

MyParser.BeforeTest           procedure
!---------------------------------------------------------------!
! Write code here to prepare for the test, if necessary.
! This method is called automatically by the test procedure
! before the test is run. You will most likely want to use
! this method if you parse your string before calling any of
! the string manipulation methods. If you do everything
! on the fly then you probably won't need this method
!---------------------------------------------------------------!
   code

MyParser.GetTokenCount        procedure
!---------------------------------------------------------------!
! Write code here to count the number of tokens. By default
! you have a queue (self.DelimQ) of delimiters which you
! can use to examine the text string (self.Text).
!---------------------------------------------------------------!
   code
   return(0)  ! Replace this code!

MyParser.GetToken             procedure(long Index)
!---------------------------------------------------------------!
! Write code here to get a specific token. In the example
! text a token is an individual words, but if the string
! were some Clarion code and the period is a delimiter then
! if the string were "MyParser.GetToken" GetToken(1) would
! return "MyParser" and GetToken(2) would return "GetToken"
!---------------------------------------------------------------!
   code
   return('')  ! Replace this code!

MyParser.PutToken             procedure(long Index,string Text)
!---------------------------------------------------------------!
! Write code here to set a specified token in the string to
! a new value. For instance, if the string is
! "MyParser.GetToken" and the period is a delimiter then
! calling PutToken(1,'SomeOtherParser') would change the
! string to "SomeOtherParser.GetToken".
!---------------------------------------------------------------!
```

```
code
return(False)  ! Replace this code!
```

All of these methods are implemented at the end of parser.clw. If you want to derive another method just follow the example of how these methods are declared in the class declaration at the top of parser.clw, and implemented at the end of parser.clw. Most likely you'll be able to complete the challenge just by writing code for the methods in Listing 2. You may also want to add some variables to MyParser so that you can share them between methods. See the declaration for ParserBaseClass for an example, and email advisor@clarionmag.com if you have questions.

Also please note that although this way of declaring and implementing a class in a single source file clearly works, it isn't recommended for most work. You're generally better off keeping base class declarations in INC files and code in CLW files as ABC does.

Click here to download the parser shell application and project file.

Send your entries to advisor@clarionmag.com.

In This Issue

June 1999 News
Posted on June 21, 1999

New Improved Clarion Challenge!
Posted on June 21, 1999

How ABC Handles Multiple Sort Orders (Part III)
Posted on June 21, 1999

Developers' Open Source Public License Version 1.0
Posted on June 21, 1999

Copyright © 1999 by CoveComm Inc. All Rights Reserved. Reproduction in any form without the express written consent of CoveComm Inc., except as described in the subscription agreement, is prohibited. If you find this page on a site other than www.clarionmag.com, email covecomm@mbnet.mb.ca.

Vol 1, No 5
June 1999

Clarion magazine

Main Page
Log In
Subscribe
Open Source
Links
Mailing Lists
Advertising
Submissions
Contact Us
Site Index
ClarionMag FAQ
Download PDFs
Search ClarionMag

Feature Article

Associating RunTime Tabs With Added "Keys"

Sorting On The Run (Part III)

by Steve Parker

Read Part I

Read Part II

BKO Enterprises, Inc.
Great Opportunity!
Salary to $125,000+
Pre-IPO Stock Opts
Sunny Boca Raton

etc 2000
If you're interested take our poll & let us know!

When I first undertook the project to allow users to store additional sort orders in a file and automatically provide those orders on browses and reports, there were several things I knew.

I knew that additional orders could be created and I also knew that additional tab controls could be created. I had only a vague idea of how to do either of these tasks, never having had to before, but I did know these things could be done.

Having looked at the code generated by the templates for different orders on different tabs and the Solodex example for runtime tab creation, I knew that implementing each feature reduced to only a few lines of code each, very few as it turns out. I also "knew" that understanding what was happening, especially in adding sort orders, would mean tracing method calls and that would surely not be a matter of just "a few lines of code."

> While I will certainly settle for code that works, I've always preferred understanding how and why it works. Understanding allows me to not only extend solutions into strategies but helps avoid problems in the first place.

I was right on both counts.

From the beginning, it seemed to me that the really hard part was going to be associating the additional sort orders with the additional (runtime) tabs. That is, it would be hard to make sure that the sort order records in the file, the selected tab and the applied sort actual mapped one-to-one. Not simply one-to-one but predictably and without variation. Everything has to come together at runtime in just the right way.

I do most of my programming with a pencil and paper. And as I write this I am looking at my original notes. They are exclusively concerned with setting up the tabs so that each links to a single, predictable sort order. My notes indicate that I finally decided to loop through the file containing the sort orders and read the needed fields to a queue. At the same time, I determined how many additional tabs had been created, the ordinal position of each new tab relative to the current record and adding a number indicating that ordinal position to the queue (mine, not the browse object's). When the user changed tabs, I was planning on using the tab position (`Choice(?CurrentTab)`) to retrieve the matching queue record and ... you get the idea.

A typical programmer's solution.

I could not have been more wrong. Ensuring that the selected, additional tab and the additional sort order match could not have been easier. (Ok, it could have been easier but no one has a template available that does what I need.)

Basic Considerations

In the last installment I demonstrated that user defined sort orders need to be added to the `Sort` property after the sort orders supplied by the templates. Where n is the number of tabs created within the IDE, adding custom orders after the template's ensures that the first n queue entries and the first n tabs match in the standard way, as shown in Listing 1.

Listing 1. Sorting by tabs.

```
IF CHOICE(?CurrentTab) = 2
  RETURN SELF.SetSort(1,Force)
ELSIF CHOICE(?CurrentTab) = 3
  RETURN SELF.SetSort(2,Force)
ELSIF CHOICE(?CurrentTab) = 4
  RETURN SELF.SetSort(3,Force)
ELSE
  RETURN SELF.SetSort(4,Force)
END
```

The key to this kingdom is that, as the developer of the application, I know the value of n. I know how many tabs were created at design time and, therefore, I know the initial number of entries in the sort queue. More important, I know the ordinal position of each.

That also means that I know that any supplemental orders and any runtime tabs start at n + 1. To ensure that queue entries and tabs are mapped in the same order, to ensure the mandatory correspondence, they only need to be created in the same order, using the same Key.

Too simple. (Proper prior planning ....)

That leaves only:

Activating Sorts on Tab Change

For all of this work to pay off, the additional tabs must actually activate the corresponding additional order.

The code in Listing 1 above is the standard template generated tab-changing code for a four-tabbed browse. It contains all the clues needed to finish this project.

The logical core of this code is the use of `CHOICE(?CurrentTab)` to calculate the first parameter of the `SetSort()` method. `SetSort()` then does the actual work of re-setting the sort order.

For the sake of example, suppose you have a sheet with two tabs. This means that the first custom order created at runtime will be queue entry number three and the first runtime-created tab will be `CHOICE(?CurrentTab) = 3`.

This means that for all dynamically created tabs:

Listing 2.

```
RETURN SELF.SetSort( Choice(?CurrentTab) , Force )
```

ought to retrieve the correct sort queue record. In turn, this means that the desired order will be correctly set. I'm just "requisitioning" the standard code, after all.

Of course, it cannot possibly be quite so simple. It isn't.

The statement in Listing 2 will give incorrect sorts for the default tabs, tabs created in the IDE. Remember that in the case of two tabs, the queue entry for tab two is added first then the default order (tab one) is added to the queue. So, what you really need to do is use this code for runtime tabs only, leaving the generated code to handle "static" tabs, something more like:

Listing 2.

```
IF CHOICE(?CurrentTab) > n
  RETURN SELF.SetSort( Choice(?CurrentTab) , Force )
End
```

The question now is:

Where is this code placed?

And the answer is: "It depends."

It does depend. If the underlying browse has only one key and, therefore, only one tab, no tab switching code will be generated by the templates. There will be no check on `Choice(?CurrentTab)` because `?CurrentTab` only has one position.

If there is more than one tab, code like that above will be generated. In this case, a structure like that in Listing 1 will be generated by the templates.

One Tab: In this case, since the templates supply nothing, you need to supply everything. The code in Listing 2 does everything necessary and it seems pretty clear that it should go in the `?CurrentTab … NewSelection` embed.

Unfortunately, the code in Listing 2 will not work. It will fail to compile.

It will not compile because the browse object is not in scope here (the `NewSelection` embed is not a derived virtual method of the browse object, it is an event embed). But `SetSort` is a property of the browse object and the procedure knows about that, so:

```
RETURN BRW1.SetSort( Choice(?CurrentTab), 1 )
```

will do the job.

In the sample app that accompanies this article, check the `WithNewTabs_Corrected` procedure ("corrected" for supplementary orders being incorrectly activated when the browse is first opened; this was discussed in the previous installment). This placement of the revised code does just what we want.

Multiple Tabs: When there is more than one tab in the sheet before the ad hoc tabs, you want to use the code in Listing 3 as is.

Where?-Check the template code (Listing 1) and you will see that there is an `ELSE` clause.

Runtime tabs will always return a value greater than n and will, therefore, trigger the `ELSE` clause.

This means that code handling the created tabs must come before the template generated code. As this code tests only for values greater than the number of tabs created in the window formatter, the standard generated code will still do its job correctly.

Opening the Embeditor and finding the standard tab change code shows that the embed immediately preceding is `Before Refresh Window for Browse Box` (see Figure 1).

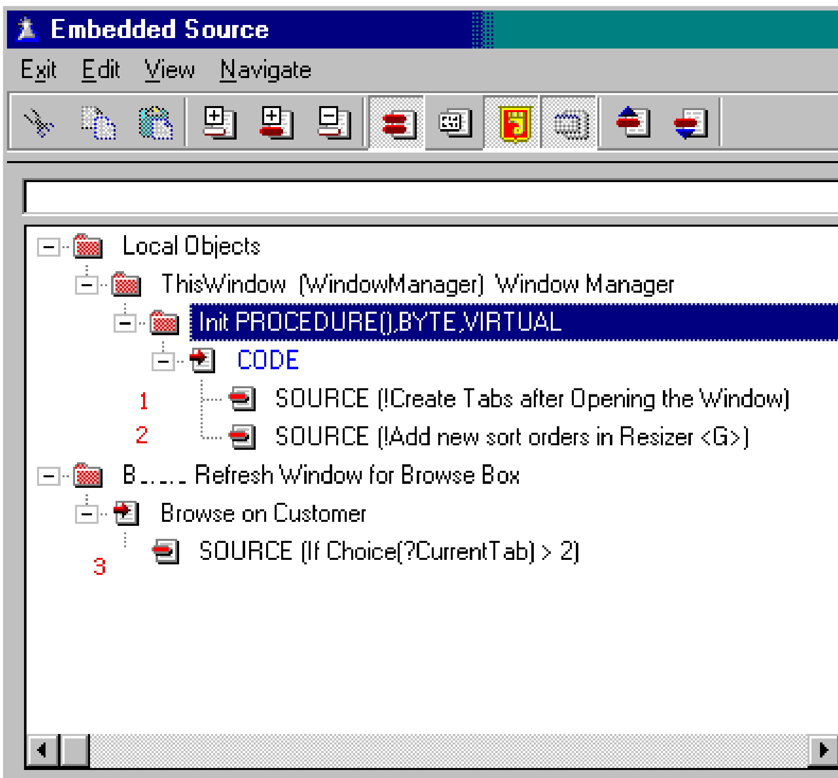
Figure 1. Embedding the tab change code.

```
! Start of "Browser Method Executable Code Section"
! [Priority 1300]
                                    ABC embed
! Select sort order based on browse conditions
! Start of "Before Refresh Window for Browse Box"
! [Priority 5000]
If Choice(?CurrentTab) > 2            Legacy embed
  Return Self.SetSort( Choice(?CurrentTab), Force )
End
! End of "Before Refresh Window for Browse Box"
IF CHOICE(?CurrentTab) = 2
  RETURN SELF.SetSort(1,Force)
ELSE                                    Default code
  RETURN SELF.SetSort(2,Force)
END
! Start of "After Refresh Window for Browse Box"
! [Priority 5000]
```

If you check the Embed tree, this is a Legacy embed. I can't say that I much care but if you're an OOP purist, `Local Objects … BRW1 … ResetSort` is the corresponding ABC virtual method.

So, there you have it, easy as 1-2-3:

Figure 2. The embed tree showing the embed point.



1. create your tabs After the window is opened
2. add your new sort orders after any existing key are added
3. add the code to act on a tab change.

One caveat: unless you limit the number of records in the sort file, you have neither any idea of how many tabs will be created nor any control. Created tabs could well end up stacked on top of each other in a most unattractive fashion. Play it safe and select a scrolling option for the tabs (on the second tab on the Sheet's Property Worksheet) whether you think you need it or not.

An interesting consequence of this solution is that it will continue to work even if Topspeed changes how it adds sort orders to the queue. If the templates are changed so that tab one is added to the queue first or so that there is no `ELSE` clause, this solution will still do what is expected of it. The only thing that Topspeed must do is maintain the `AddSortOrder` and `AppendOrder` methods.

Reports

I mentioned reports as requiring runtime sorts.

This particular requirement would not seem to be solved by the technique developed here. Indeed, it is not. However, the People example provides a variation applicable to reports. See the `PrintPEO:KeyID` report, `ThisWindow (ReportManager) … OpenReport` (before Parent Call) embed.

Summary

Once you understand how the ABC templates implement multiple sort orders, duplicating and adapting that behavior to whole new program functionalities is not at all hard. The total code required is 22 lines or less.

What is difficult is tracing and understanding the ABC method calls. However, it is not necessary to master all the complexities of the browse object, just enough to comprehend the logic.

Download the source code

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

Copyright © 1999 by CoveComm Inc. All Rights Reserved. Reproduction in any form without the express written consent of CoveComm Inc., except as described in the subscription agreement, is prohibited. If you find this page on a site other than www.clarionmag.com, email covecomm@mbnet.mb.ca.

In This Issue

June 1999 News
Posted on June 21, 1999

New Improved Clarion Challenge!
Posted on June 21, 1999

How ABC Handles Multiple Sort Orders (Part III)
Posted on June 21, 1999

Developers' Open Source Public License Version 1.0
Posted on June 21, 1999

Main Page
Log In
Subscribe
Open Source
Links
Mailing Lists
Advertising
Submissions
Contact Us
Site Index
ClarionMag FAQ
Download PDFs
Search ClarionMag

DEVELOPERS' OPEN SOURCE PUBLIC LICENSE

Version 1.0, June, 1999

Copyright @ 1999 Cove Comm Inc.

PREAMBLE

This license is designed to protect the rights of software developers who wish to release software libraries and software development tools under an open source arrangement.

Most open source licenses are either written with complete software packages in mind or are intended to cover code which is created using open source software tools. This license is specifically designed to accommodate developers who use open source software in the context of proprietary software development, or who work with proprietary software development tools.

In brief, this agreement means the following:

- The code you place under this license remains freely available to others.
- Any modifications to the code "inherit" this license agreement, allowing you to obtain the modifications others make to your code.
- Although anyone can profit from code covered under this license, the code remains freely available. This places the emphasis on implementation and support, since there is no monetary value associated with the code itself.

TERMS AND CONDITIONS

1. Definitions.

1.1. "Contributor" means each entity that creates or contributes to the creation of Modifications.

1.2. "Contributor Version" means the combination of the Original Code, prior Modifications made by a Contributor, and the Modifications made by that particular Contributor.

1.3. "Covered Code" means the Original Code or Modifications or combination of the Original Code and Modifications, in each case including portions thereof.

1.4. "Electronic Distribution Mechanism" means a mechanism accepted in the software development community for the electronic distribution of data.

1.5. "Executable" means Covered Code in any form other than Source Code.

1.6. "Initial Developer" means the individual or entity identified as the Initial Developer in the Source Code notice required by Exhibit A.

1.7. "Larger Work" means a work which combines Covered Code or portions thereof with code not governed by the

terms of this License.

1.8. "License" means this document.

1.9. "Modifications" means any addition to or deletion from the substance or structure of either the Original Code or any previous Modifications. When Covered Code is released as a series of files, a Modification is:

A. Any addition to or deletion from the contents of a file containing Original Code or previous Modifications.

B. Any new file that contains any part of the Original Code or previous Modifications. In the case of new files, the differentiating factor is whether the file simply makes use of, or extends, the Original Code or previous Modifications. For example, if Covered Code is object-oriented software, a class derived from a Covered Code class is a Modification, while an instance of a Covered Code class is not a Modification.

1.10. "Original Code" means Source Code of computer software code which is described in the Source Code notice required by Exhibit A as Original Code, and which, at the time of its release under this License is not already Covered Code governed by this License.

1.11. "Source Code" means the preferred form of the Covered Code for making modifications to it, including all modules it contains, plus any associated interface definition files, scripts used to control compilation and installation of an Executable, or a list of source code differential comparisons against either the Original Code or another well known, available Covered Code of the Contributor's choice. The Source Code can be in a compressed or archival form, provided the appropriate decompression or de-archiving software is widely available for no charge.

1.12. "You" means an individual or a legal entity exercising rights under, and complying with all of the terms of, this License or a future version of this License issued under Section 6.1. For legal entities, "You'' includes any entity which controls, is controlled by, or is under common control with You. For purposes of this definition, "control'' means (a) the power, direct or indirect, to cause the direction or management of such entity, whether by contract or otherwise, or (b) ownership of fifty percent (50%) or more of the outstanding shares or beneficial ownership of such entity.

2. Source Code License.

2.1. The Initial Developer Grant.

The Initial Developer hereby grants You a world-wide, royalty-free, non-exclusive license, subject to third party intellectual property claims:

(a) to use, reproduce, modify, display, perform, sublicense and distribute the Original Code (or portions thereof) with or without Modifications, or as part of a Larger Work; and

(b) under patents now or hereafter owned or controlled by Initial Developer, to make, have made, use and sell ("Utilize'') the Original Code (or portions thereof), but solely to the extent that any such patent is reasonably necessary to enable You to Utilize the Original Code (or portions thereof) and not to any greater extent that may be necessary to Utilize further Modifications or combinations.

2.2. Contributor Grant.

Each Contributor hereby grants You a world-wide, royalty-free, non-exclusive license, subject to third party

intellectual property claims:

(a) to use, reproduce, modify, display, perform, sub license and distribute the Modifications created by such Contributor (or portions thereof) either on an unmodified basis, with other Modifications, as Covered Code or as part of a Larger Work; and

(b) under patents now or hereafter owned or controlled by Contributor, to Utilize the Contributor Version (or portions thereof), but solely to the extent that any such patent is reasonably necessary to enable You to Utilize the Contributor Version (or portions thereof), and not to any greater extent that may be necessary to Utilize further Modifications or combinations.

3. Distribution Obligations.

3.1. Application of License.

The Modifications which You create or to which You contribute are governed by the terms of this License, including without limitation Section 2.2. The Source Code version of Covered Code may be distributed only under the terms of this License or a future version of this License released under Section 6.1, and You must include a copy of this License with every copy of the Source Code You distribute. You may not offer or impose any terms on any Source Code version that alters or restricts the applicable version of this License or the recipients' rights hereunder. However, You may include an additional document offering the additional rights described in Section 3.5.

3.2. Availability of Source Code.

Any Modification which You create or to which You contribute must be made available in Source Code form under the terms of this License either on the same media as an Executable version or via an accepted Electronic Distribution Mechanism to anyone to whom you made an Executable version available; and if made available via Electronic Distribution Mechanism, must remain available for at least twelve (12) months after the date it initially became available, or at least six (6) months after a subsequent version of that particular Modification has been made available to such recipients. You are responsible for ensuring that the Source Code version remains available even if the Electronic Distribution Mechanism is maintained by a third party.

3.3. Description of Modifications.

You must cause all Covered Code to which you contribute to contain information documenting the changes You made to create that Covered Code and the date of any change. You must include a prominent statement that the Modification is derived, directly or indirectly, from Original Code provided by the Initial Developer and including the name of the Initial Developer in (a) the Source Code, and (b) in any notice in an Executable version or related documentation in which You describe the origin or ownership of the Covered Code.

3.4. Intellectual Property Matters

(a) Third Party Claims.

If You have knowledge that a party claims an intellectual property right in particular functionality or code (or its utilization under this License), you must include a text file with the source code distribution titled "LEGAL" which describes the claim and the party making the claim in sufficient detail that a recipient will know whom to contact. If you obtain such knowledge after You make Your Modification

available as described in Section 3.2, You shall promptly modify the LEGAL file in all copies You make available thereafter and shall take other steps (such as notifying appropriate mailing lists or newsgroups) reasonably calculated to inform those who received the Covered Code that new knowledge has been obtained.

(b) Contributor APIs.

If Your Modification is an application programming interface and You own or control patents which are reasonably necessary to implement that API, you must also include this information in the LEGAL file.

3.5. Required Notices.

You must duplicate the notice in **Exhibit A** (with appropriately modified names and dates) in each file of the Source Code, and this License in any documentation for the Source Code, where You describe recipients' rights relating to Covered Code. If You created one or more Modification(s), You may add your name as a Contributor to the notice described in **Exhibit A**. If it is not possible to put such notice in a particular Source Code file due to its structure, then you must include such notice in a location (such as a relevant directory file) where a user would be likely to look for such a notice. You may choose to offer, and to charge a fee for, warranty, support, indemnity or liability obligations to one or more recipients of Covered Code. However, You may do so only on Your own behalf, and not on behalf of the Initial Developer or any Contributor. You must make it absolutely clear that any such warranty, support, indemnity or liability obligation is offered by You alone, and You hereby agree to indemnify and save harmless the Initial Developer and every Contributor for any liability incurred by the Initial Developer or such Contributor as a result of warranty, support, indemnity or liability terms You offer.

3.6. Distribution of Executable Versions.

You may distribute Covered Code in Executable form only if the requirements of Section 3.1-3.5 have been met for that Covered Code, and if You include a notice stating that the Source Code version of the Covered Code is available under the terms of this License, including a description of how and where You have fulfilled the obligations of Section 3.2. The notice must be conspicuously included in any notice in an Executable version, related documentation or collateral in which You describe recipients' rights relating to the Covered Code. You may distribute the Executable version of Covered Code under a license of Your choice, which may contain terms different from this License, provided that You are in compliance with the terms of this License and that the license for the Executable version does not attempt to limit or alter the recipient's rights in the Source Code version from the rights set forth in this License. If You distribute the Executable version under a different license, You must make it absolutely clear that any terms which differ from this License are offered by You alone, not by the Initial Developer or any Contributor. You hereby agree to indemnify and save harmless the Initial Developer and every Contributor for any liability incurred by the Initial Developer or such Contributor as a result of any such terms You offer.

3.7. Larger Works.

You may create a Larger Work by combining Covered Code with other code not governed by the terms of this License and distribute the Larger Work as a single product. In such a case, You must make sure the requirements of this License are fulfilled for the Covered Code.

4. Inability to Comply Due to Statute or Regulation.

If it is impossible for You to comply with any of the terms of this License with respect to some or all of the Covered Code due to statute or regulation, then You must: (a) comply with the terms of this License to the maximum extent possible; and (b) describe the limitations and the code they affect. Such description must be included in the LEGAL file described in Section **3.4** and must be included with all distributions of the Source Code. Except to the extent prohibited by statute or regulation, such description must be sufficiently detailed for a recipient of ordinary skill to be able to understand it.

5. Application of this License. This License applies to code to which the Initial Developer has attached the notice in **Exhibit A**, and to related Covered Code.

6. Versions of the License.

6.1. New Versions. CoveComm Inc. may publish revised and/or new versions of the License from time to time. Each version will be given a distinguishing version number.

6.2. Effect of New Versions. Once Covered Code has been published under a particular version of the License, You may always continue to use it under the terms of that version. You may also choose to use such Covered Code under the terms of any subsequent version of the License published by CoveComm Inc.. No one other than CoveComm Inc. has the right to modify the terms applicable to Covered Code created under this License.

6.3. Derivative Works. If you create or use a modified version of this License (which you may only do in order to apply it to code which is not already Covered Code governed by this License), you must (a) rename Your license so that the phrase "Cove Comm" or any confusingly similar phrase does not appear anywhere in your license and (b) otherwise make it clear that your version of the license contains terms which differ from the CoveComm Inc. Public License. (Filling in the name of the Initial Developer, Original Code or Contributor(s) in the notice described in Exhibit A shall not of themselves be deemed to be modifications of this License.)

7. DISCLAIMER OF WARRANTY.

COVERED CODE IS PROVIDED UNDER THIS LICENSE ON AN "AS IS'' BASIS, WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, WITHOUT LIMITATION, WARRANTIES THAT THE COVERED CODE IS FREE OF DEFECTS, MERCHANTABLE, FIT FOR A PARTICULAR PURPOSE OR NON-INFRINGING. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE COVERED CODE IS WITH YOU. SHOULD ANY COVERED CODE PROVE DEFECTIVE IN ANY RESPECT, YOU (NOT THE INITIAL DEVELOPER OR ANY OTHER CONTRIBUTOR) ASSUME THE COST OF ANY NECESSARY SERVICING, REPAIR OR CORRECTION. THIS DISCLAIMER OF WARRANTY CONSTITUTES AN ESSENTIAL PART OF THIS LICENSE. NO USE OF ANY COVERED CODE IS AUTHORIZED HEREUNDER EXCEPT UNDER THIS DISCLAIMER.

8. TERMINATION. This License and the rights granted hereunder will terminate automatically if You fail to comply with terms herein and fail to cure such breach within 30 days of becoming aware of the breach. All sublicenses to the Covered Code which are properly granted shall survive any termination of this License. Provisions which, by their nature, must remain in effect beyond the termination of this License shall survive.

9. LIMITATION OF LIABILITY.

UNDER NO CIRCUMSTANCES AND UNDER NO LEGAL THEORY, WHETHER TORT (INCLUDING NEGLIGENCE), CONTRACT, OR OTHERWISE, SHALL

THE INITIAL DEVELOPER, ANY OTHER CONTRIBUTOR, OR ANY DISTRIBUTOR OF COVERED CODE, OR ANY SUPPLIER OF ANY OF SUCH PARTIES, BE LIABLE TO YOU OR ANY OTHER PERSON FOR ANY INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES OF ANY CHARACTER INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF GOODWILL, WORK STOPPAGE, COMPUTER FAILURE OR MALFUNCTION, OR ANY AND ALL OTHER COMMERCIAL DAMAGES OR LOSSES, EVEN IF SUCH PARTY SHALL HAVE BEEN INFORMED OF THE POSSIBILITY OF SUCH DAMAGES.

10. MISCELLANEOUS. (a) Severability This License represents the complete agreement concerning the subject matter hereof. The provisions of this License are severable one from the other and if any of its provisions is declared void, the decisions so holding shall not be construed as impairing any other provisions of this License. If any provision of this License is held to be unenforceable, such provision shall be reformed only to the extent necessary to make it enforceable. **(b) Governing Law** This License shall be governed by such laws of Manitoba and Canada as may be applicable. **(c) Arbitration** In any dispute concerning any matter arising out of or in connection with this License, every such dispute shall be referred to arbitration within five (5) days of written notice of a dispute being personally delivered or mailed by registered mail, postage prepaid, by the disputing party to the other party. The disputing parties, unless otherwise agreed, are each to respectively appoint one arbitrator within twelve (12) days of the mailing of such notice of dispute and the two arbitrators shall appoint a third arbitrator within eighteen (18) days of the mailing of the notice of dispute. If the arbitrators fail to agree upon the appointment of a third arbitrator within the time limit set out above, either party may, upon two (2) days notice to the other party, apply to a Judge at the Court of Queen's Bench for the Province of Manitoba to appoint a third arbitrator. If any party fails to appoint an arbitrator within the time stipulated, then the arbitrator so appointed shall act as a sole arbitrator to settle the dispute. The decision of any two of the three arbitrators or of the sole arbitrator shall be final and binding. Each of the parties hereto shall pay one-half (1/2) of the expenses of such reference.

11. RESPONSIBILITY FOR CLAIMS. Except in cases where another Contributor has failed to comply with Section 3.4, You are responsible for damages arising, directly or indirectly, out of Your utilization of rights under this License, based on the number of copies of Covered Code you made available, the revenues you received from utilizing such rights, and other relevant factors. You agree to work with affected parties to distribute responsibility on an equitable basis.

EXHIBIT A. PLACING CODE UNDER THIS LICENSE.

To place eligible code under this license place the following notice, appropriately modified, in the covered code. If you are distributing an executable program containing covered code (see Sections 3.6 and 3.7) you must include a notice with the program in compliance with Section 3.6.

Form Of License Notice.

The contents of this file are subject to the CoveComm Inc. Developers Open Source Public License Version 1.0 (the "License"); you may not use this product except in compliance with the License. You may obtain a copy of the License at http://www.clarionmag. com/common/dosl.html. Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License. The Original Code is copyright by ____________________ (name of copyright owner)., released ____________ (date of release). The Initial Developer of the Original Code is ___________________ (name of initial developer) . Portions created _______________________________ (name of contributor(s), if any) are Copyright © __________________________ (year and name of copyright holder(s)). All Rights Reserved.

Vol 1, No 5
June, 1999

Clarion magazine

Main Page
Log In
Subscribe
Open Source
Links
Mailing Lists
Advertising
Submissions
Contact Us
Site Index
ClarionMag FAQ
Download PDFs
Search ClarionMag

Clarion News

June 28, 1999

CWICWEB's New Look

Steve Parker has given his CWICWEB site a new look. Among the links on this page are the three Clarion knowledge bases and Steve's index to every article published in CTJ, CWJ, COL, and Clarion Magazine.

New Handy Tools Build Available

A new Handy Tools build (L) is now available. New features include ruler, slider, and drive selector controls. There's also a filter control that translates plain language statements into Clarion filter strings, and a template/class which adds a number of features such as tagging and sorting to non-ABC browses and list boxes.

Special Agent Special Price Ends June 30

The Special Agent Introductory price of $149 ($50 off the regular price) expires June 30, 1999. A demo is available from the web site.

Asher To Keynote DevCon '99

Hank Asher, Founder and Chief Executive Officer of eData.com (formerly Indar), will present the Keynote Address at DevCon '99. Asher is also the founder of Database Technologies, a $500 million company built on Clarion technology.

Pervasive Discontinues Btrieve 6.15

Pervasive Software Inc. will discontinue general availability of Btrieve on June 30, 1999. Technical support for 6.15 will continue through September 1, 1999.

June 21, 1999

Report & Presentation Manager v5b (Interim Release)

RPM5b is now available for download by all registered users of RPM5. This release is for C5PE and C5EE with at least the SR1 patch. This update fixes a problem with empty reports and CW timers, large reports and "print & remain" and a fairly rare ASCII export problem. If you encounter any problems during installation please contact Lodestar Software at support@LodestarSoftware.com. A full install of RPM5B will be available shortly.

Zip Code Library for Clarion 5 Now Available

The Zip Code Library for Clarion 5 is now available from Topspeed Corporation. Contact TopSpeed Sales at (800) 354-5444 or your local distributor to acquire updates.

Data Modeler Updated

There is a new version of Data Modeller available for both Clarion 5 Enterprise (DM 5.2000 b) and Clarion 5 Professional (DM 5.0001 b). This version requires Clarion 5a (SP1) to run, and will run with Clarion 5 b (SP2). Clarion 5 Professional users can use the same password as the previous update.

ABC Free Templates and Tools Updated

BKO Enterprises, Inc.
Great Opportunity!
Salary to $125,000+
Pre-IPO Stock Opts
Sunny Boca Raton

etc 2000
If you're interested take our poll & let us know!

In This Issue

June 1999 News
Posted on June 28, 1999

Clarion Advisor: Debugging Tricks
Posted on June 28, 1999

The ABCs Of OOP - Part 3
Posted on June 28, 1999

Clarion Magazine Best Read With Verdana
Posted on June 28, 1999

Detecting Crashes With DDE
Posted on June 28, 1999

The ABC Free Templates and Tools have been updated with bug fixes and new features including the ability to exclude ABC classes and include files.

June 14, 1999

etc Activity Schedule Layout Poll
Lee White has posted a new online survey for the East Tennessee Clarion Developer's Conference. This one is for your preferred schedule of activities. The conference will be bookended by classes, most likely ABC/OOP and SQL/ODBC.

CPCS Report Emailer Addon Now Available
CPCS now has an addon product which allows for fast and easy emailing of CPCS reports and the optional report viewer/printer. Purchase includes versions for Clarion5 (Legacy & ABC), Clarion4 (Legacy & ABC), and CW2003 (Legacy).

June 7, 1999

etc 2000 Scheduled For May 22
It's official – there will be another East Tennessee Clarion Developer's Conference. etc 2000 will be held in at the Edgewater Hotel in Gatlinburg, TN during the week of May 22.

Imaging Templates Support ABC And Legacy
The latest version of the Imaging Templates is now available for download. This release contains support for both ABC and legacy apps for $149.

Clarion Profile Exchange
The Clarion Profile Exchange has been updated. This release includes a number of freeware templates and download links for many demos and templates. You must have the newly released Version 3.5 of Product Scope 32 Bookmarks to view these data files.

Wholesale Distribution Accounting Package Available To Developers
KV Enterprises, Inc. has a complete royalty-free Wholesale Distributor Accounting Package coded in C5. Modules include Accounts Payable, Accounts Receivable, General Ledger, Inventory Control, Purchasing, Order Entry, and Payroll. Source is included. Online help and security is built in.

Help On Creating Help
Following his DevCon and EuroDevCon talks on writing help, James Fortune has created a web page containing links to help creation resources.

HTTP Server Edition Version 1.5 Released
Mike Pasley has released version 1.5 of his "Nothing But Clarion, Nothing But 'Net" HTTP Server/Internet Framework Template which allows for fast creation and testing of all-Clarion web servers. Some knowledge of HTML is required. Price is $99. Free beta of PowerMerge Mail Merge Templates included.

ABC Free Templates and Tools Updated
The ABC Free Templates and Tools have been updated with bug fixes and new features. The uninstall no longer uninstalls Clarion 5, and new templates features include class exporting and thread limiting.

DevCon '99 Presentation Survey
Going to DevCon in September? Topspeed has an on-line survey page where you can indicate which sessions you will/may attend. The survey will expire on Wednesday, June 23, 1999.

Read the May 1999 News

Do you have a news story or press release we should know about? Send

it to editor@clarionmag.com

Copyright © 1999 by CoveComm Inc. All Rights Reserved. Reproduction in any form without the express written consent of CoveComm Inc., except as described in the subscription agreement, is prohibited. If you find this page on a site other than www.clarionmag.com, email covecomm@mbnet.mb.ca.

Vol 1, No 5
June 1999

Clarion magazine

The Clarion Advisor

More Debugging Tricks

Clarion Magazine has featured several articles on debugging applications (go to the Search page and search for "debug"). But you can never have too many tricks up your sleeve, and Clarion developers get as wily as anyone when it comes to using alternative debugging techniques.

Many developers use STOP() and MESSAGE() for quickie debugging, but as Russ Eggen recently pointed out in his excellent article on the debugger, these statements interfere with the normal operation of your program, and the behaviour with the use of these statements may be quite different from the behaviour without these statements.

One non-intrusive technique is to put messages in the window's title bar. Just use the following line of code any time after the window has opened:

```
0{PROP:Text} = 'Some debugging text'
```

The zero stands for the current window and is more convenient than having to know the actual window label (which may be `Window`, `QuickWindow`, or something else). Another bonus is that you can use this statement anywhere a window is open – the label of the window doesn't have to be in scope. So if the procedure with the window calls a source code function, you can still use 0{prop:text} inside the function.

If you're testing for a particular condition in your code which happens more than once, but you're always using the same message, you'll probably want to know that you're looking at a redisplay of the message and not just the original message. Add a timestamp to the message:

```
0{PROP:Text} = CLOCK() & ' Some debugging text'
```

Now you'll be able to see when the message changes.

Do you have a favourite debugging technique? Send it to advisor@clarionmag.com.

BKO Enterprises, Inc.
Great Opportunity!
Salary to $125,000+
Pre-IPO Stock Opts
Sunny Boca Raton

etc 2000
If you're interested take our poll & let us know!

June 1999 News
Posted on June 28, 1999

Clarion Advisor: Debugging Tricks
Posted on June 28, 1999

The ABCs Of OOP - Part 3
Posted on June 28, 1999

Clarion Magazine Best Read With Verdana
Posted on June 28, 1999

Detecting Crashes With DDE
Posted on June 28, 1999

Copyright © 1999 by CoveComm Inc. All Rights Reserved. Reproduction in any form without the express written consent of CoveComm Inc., except as described in the subscription agreement, is prohibited. If you find this page on a site other than www.clarionmag.com, email covecomm@mbnet.mb.ca.

Vol 1, No 5
June 1999

Clarion magazine

Feature Article

The ABCs of OOP

Understanding Virtual Methods

by Dave Harms

BKO Enterprises, Inc.
Great Opportunity!
Salary to $125,000+
Pre-IPO Stock Opts
Sunny Boca Raton

etc 2000
If you're interested take our poll & let us know!

Virtual methods are one of the most useful and powerful features of object-oriented programming. I don't think it's understating the case to say that understanding virtual methods opens up whole new vistas of software development. In fact, without virtual methods, the ABC templates and class library simply couldn't exist, at least not in their present form.

I've had several opportunities to present object-oriented programming basics in articles and in seminars, and I remain convinced that while OOP concepts often look intimidating, they're really not that difficult to grasp. If anything, it's the appearance of difficulty that is the real difficulty. But if there is an aspect of OOP that can be a bit tricky to get, it's virtual methods.

In this installment I'll build on the information I presented in the previous articles in this series. As you'll recall, the first article described some basic OOP concepts, and the second article elaborated on inheritance and encapsulation.

The DebugClass Example

Both of those articles discuss a small example class which can be used to store debugging messages in a log. Listing 1 shows the declaration for the class, and Listing 2 shows the implementation.

Listing 1. The DebugClass declaration.

```
TraceQueue      QUEUE,TYPE
Text               STRING(200)
                END


DebugClass      CLASS,TYPE,MODULE('DEBUG.CLW')
NextLineToWrite    long(1)
TraceQ             &TraceQueue
Construct          PROCEDURE
Destruct           PROCEDURE
ShowTrace          PROCEDURE
Trace              PROCEDURE(STRING Text)
WriteTrace         PROCEDURE
                END
```

Listing 2. The DebugClass implementation.

```
   MEMBER

   MAP
   END

   INCLUDE('DEBUG.INC')

TraceLog          FILE,DRIVER('ASCII'),NAME('TRACE.LOG')↵
        ,CREATE,PRE(TRACE)
Record               RECORD,PRE()
Text                     STRING(1000)
                     END
                  END

DebugClass.Construct         PROCEDURE
   CODE
   SELF.TraceQ &= NEW(TraceQueue)


DebugClass.Destruct          PROCEDURE
   CODE
   FREE(SELF.TraceQ)
   DISPOSE(SELF.TraceQ)

DebugClass.ShowTrace         PROCEDURE

window WINDOW('Debug Messages'),AT(,,331,206),|
         FONT('MS Sans Serif',8,,,CHARSET:ANSI),|
         SYSTEM,GRAY,DOUBLE
       LIST,AT(5,5,320,180),USE(?List1),HVSCROLL,|
         FONT('Courier New',8,,FONT:regular,CHARSET:ANSI)|
         ,FROM(self.TraceQ)
       BUTTON('Close'),AT(150,190,,14),USE(?Close)
     END

   CODE
   OPEN(WINDOW)
   ACCEPT
      IF FIELD() = ?Close AND EVENT() = EVENT:Accepted
         BREAK
      END
   END

DebugClass.Trace             PROCEDURE(STRING Text)
   CODE
   SELF.TraceQ.Text = Text
   ADD(SELF.TraceQ)
   IF RECORDS(SELF.TraceQ) % 10 = 0 THEN SELF.WriteTrace().

DebugClass.WriteTrace        PROCEDURE
X                 LONG
   CODE
   OPEN(TraceLog)
   IF ERRORCODE()
      CREATE(TraceLog)
      OPEN(TraceLog)
```

```
      IF ERRORCODE()
        MESSAGE('Unable to open error log: ' & ERROR())
        RETURN
      END
   END
   LOOP X = SELF.NextLineToWrite TO RECORDS(SELF.TraceQ)
      GET(SELF.TraceQ,X)
      TraceLog.Text = self.TraceQ.Text
      ADD(TraceLog)
   END
   SELF.NextLineToWrite = RECORDS(self.TraceQ) + 1
   CLOSE(TraceLog)
```

You will notice at least one difference between the class discussed in the previous article and the one shown in Listings 1 and 2. This version of the class has a WriteTrace method which figures heavily in this discussion of virtual methods.

The WriteTrace Method

The WriteTrace method isn't a virtual method – it's just a normal method like the others. The purpose of WriteTrace is to write the debug messages out to a text file. You could call this method periodically yourself (as when exiting a procedure) but it's much easier to have the debug class handle the call automatically.

The Trace method (see Listing 3) accomplishes this by calling the WriteTrace method on every tenth trace call. The modulus (%) operator simply tests for the remainder of division, in this case by 10. This effectively creates a cache of up to ten records, which is a much more efficient approach than opening the file, writing out one record, and closing the file again. (You could use a property in place of "10" and have an adjustable cache size, if you wished.)

Listing 3. The Trace method calls WriteTrace.

```
DebugClass.Trace              PROCEDURE(STRING Text)
   CODE
   SELF.TraceQ.Text = Text
   ADD(SELF.TraceQ)
   IF RECORDS(SELF.TraceQ) % 10 = 0 THEN SELF.WriteTrace().
```

Accept for the moment that you've decided you'd like DebugClass to store its log of information in one of your application's data files rather than in a text file. Remembering the principle that it's usually better to derive a new class than modify an existing class, you decide to create a derived class that will work with your application. Can you simply rewrite the WriteTrace method so it uses your application's data file?

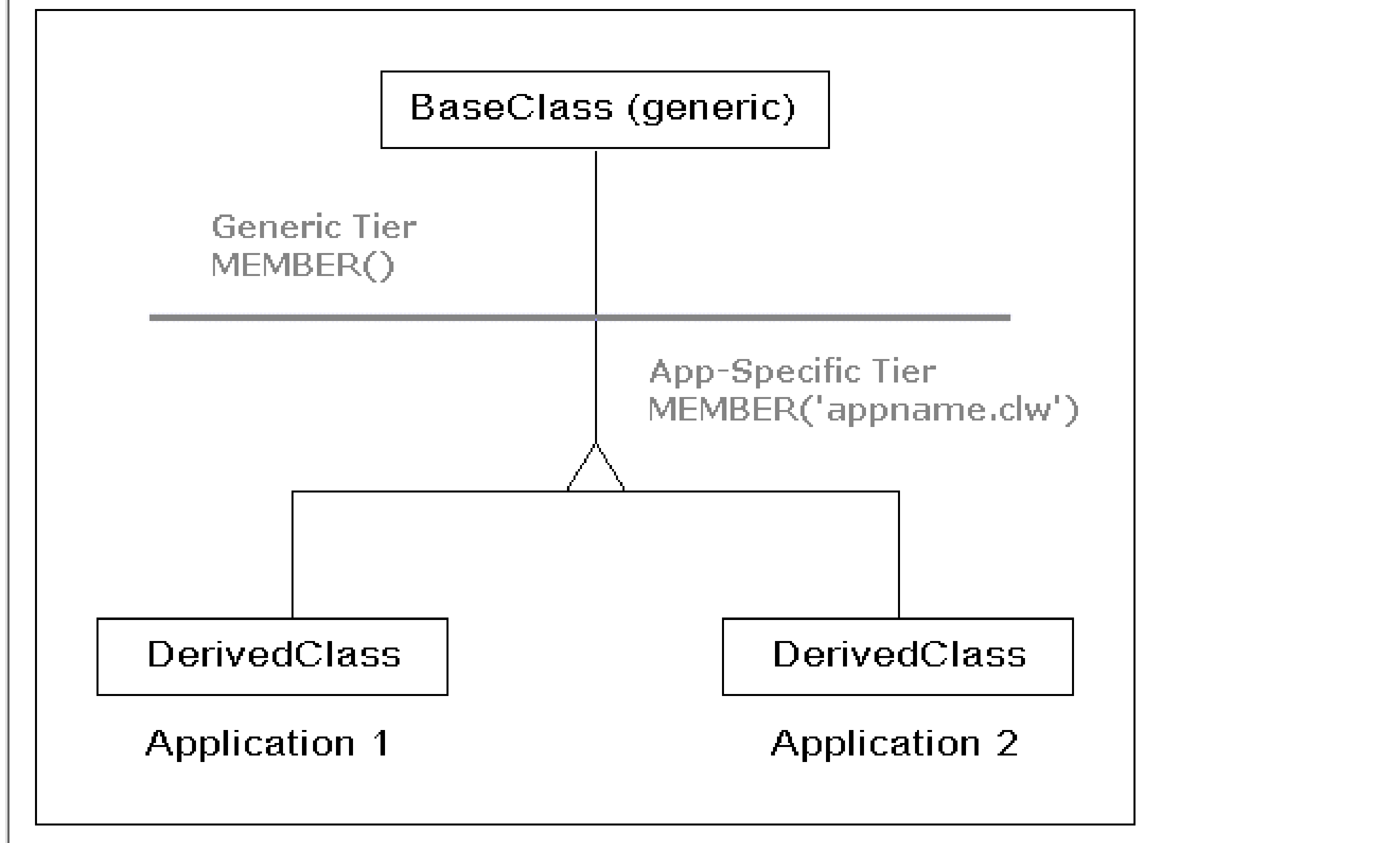
You can try, but the compiler won't be impressed. As you can tell by the empty `MEMBER` statement in Listing 2, this class is generic, which means it can be compiled with any application. Now in Clarion 2.1, modules with empty `MEMBER` statements automatically gained access to the global data of the application in which they were compiled. That's not the case any more. Modules can now only "see" declarations to which they're given explicit access, either by a member statement that points to an application source file, or by use of `INCLUDE` statements.

DebugClass doesn't have any knowledge of your application's data, and that's the way you want it, because this class needs to work with any application. If you use an application-specific `INCLUDE` or `MEMBER` statement that points to an application, you've effectively prevented yourself from using that class in any other application.

Two-Tiered Development

The answer to the problem is to use a two-tiered development strategy, which is what ABC does. In the top tier are generic classes with empty `MEMBER` statements. In the bottom tier are the application-specific classes. All the common code (or as much of it as possible) goes in the top tier, thereby maximizing code reuse. Figure 1 shows the two tiers.

Figure 1. A two-tiered class design.



This two-tiered approach is an ideal way to solve the problem of making DebugClass work with a specific application's data (and code).

To test this, create a TPS file in the application's data dictionary (or follow along with the example application). This file, called Trace, only needs one field (`Text STRING(1000)`) and doesn't need any keys, although if you wanted to jazz it up you could add the date and time the trace record was added.

In the application, create a main menu item to call a browse procedure which will display the trace log records. Again, you don't have to specify a key, since if no key exists the browse will display the records in the order they were added. You don't need to bother with an update procedure for this browse since all the records will be created by the derived debug class. Use the Browse wizard to create a procedure to browse the Trace TPS file.

The Derived Debug Class

When I create generic classes I usually put them in my `LIBSRC` directory along with all of the ABC classes. When I create an application-specific class I normally put it in the application directory, as I won't be wanting to use it with any other apps.

The derived `AppDebugClass` declaration is shown in Listing 4. There are several differences between the way this class is declared and the way its parent class (`DebugClass`) is declared.

Listing 4. The derived AppDebugClass declaration (AppDebug.INC).

```
OMIT('_EndOfInclude_',_APPDEBUGPRESENT_)
_APPDEBUGPRESENT_   EQUATE(1)


   INCLUDE('DEBUG.INC')

AppDebugClass        CLASS(DebugClass),TYPE,↵
        MODULE('appdebug.clw'),LINK('appdebug.clw')
WriteTrace              PROCEDURE
                     END
 _EndOfInclude_
```

As with last article's DebugTimedClass AppDebugClass has a CLASS(DebugClass) attribute which tells the compiler which class to use it comes across methods and properties which aren't declared within AppDebugClass. The INCLUDE statement points to the source file which contains the parent class's declaration.

The LINK attribute tells the compiler which source file contains the class implementation. If you have a LINK attribute on the class you don't need to explicitly add the source file to the project.

> **NOTE:** Although it's usually best to use the LINK attribute, you may on occasion wish to add the source to the project manually, as this lets you set compiler pragmas on just that file. You will also need the DLL attribute if using classes exported from a DLL. See the ABC class declarations for examples.

Another curious aspect of this include file is the use of an OMIT statement. The OMIT prevents the declaration from being referenced by the compiler more than once. This is a way of preventing duplicate symbol warnings which can happen if you have a number of nested includes. The first time the compiler references this code the AppDebugPresent flag is false, by default, and immediately after the OMIT it's set to true, so on all subsequent passes the compiler ignores the OMITted code.

There is only one method declared in AppDebugClass, and that is WriteTrace. The implementation, shown in Listing 5, replaces the original WriteTrace method with code that puts the trace statements in the application's Trace file.

Listing 5. The AppDebugClass implementation (APPDEBUG.CLW).

```
MEMBER('TEST.CLW')

   MAP
   END


AppDebugClass.WriteTrace              PROCEDURE
X        LONG
   CODE
   ACCESS:Trace.OPEN()
   ACCESS:Trace.UseFile()
   LOOP X = SELF.NextLineToWrite TO RECORDS(self.TraceQ)
      GET(SELF.TraceQ,x)
      TRA:Text = SELF.TraceQ.Text
      ACCESS:Trace.Insert()
   END
```

```
SELF.NextLineToWrite = RECORDS(self.TraceQ) + 1
ACCESS:Trace.CLOSE()
```

At the top of the source listing is the `MEMBER('TEST.CLW')` statement which, following the two-tier model, ties this class to the application's data and allows the use of all global data, classes, and procedures. If you compare this code with the base class code, you'll see that one of the benefits of using ABC is a lot of tedious code is taken care of in the libraries. Instead of the original's somewhat hacked attempt to make sure that the trace file can be opened, you have a single call to `ACCESS:Trace.Open` which takes care of all of the error checking. Similarly `ACCESS:Trace.Insert` checks for any problems with the add. Other than this, the replacement method simply takes the same approach as the original but adds the records to the application's TPS Trace file rather than to the ASCII trace file.

To implement the class put the following in a global embed point, such as After Global Includes:

```
   INCLUDE('APPDEBUG.INC')
db AppDebugClass
```

If you've been following along with the previous articles you'll be working with an application that already has a `db` object, but which is declared as an instance of `DebugClass` not `AppDebugClass`. You will also have an `INCLUDE` statement that points to `'DEBUG.INC'` rather than `'APPDEBUG.INC'`. Because `AppDebugClass` is derived from `DebugClass`, you can leave all of the calls to the `db` object's methods as they were, and simply make `db` an instance of the derived class. Derived classes don't lose any data or methods, although they may change the implementation of some of those methods.

Testing AppDebugClass

The apps from the previous articles, and the example application for this article demonstrate a simple use of the debug classes. The each have the code:

```
db.Trace('Event ' & EVENT())
```

in the TakeEvent method of the Names browse object in the BrowseNames procedure. If you call BrowseNames, the Trace method records all of the events that the browse receives.

This application uses AppDebugClass which has a replacement WriteTrace method which writes messages to the TPS Trace file. So what happens if you run the application using AppDebugClass?

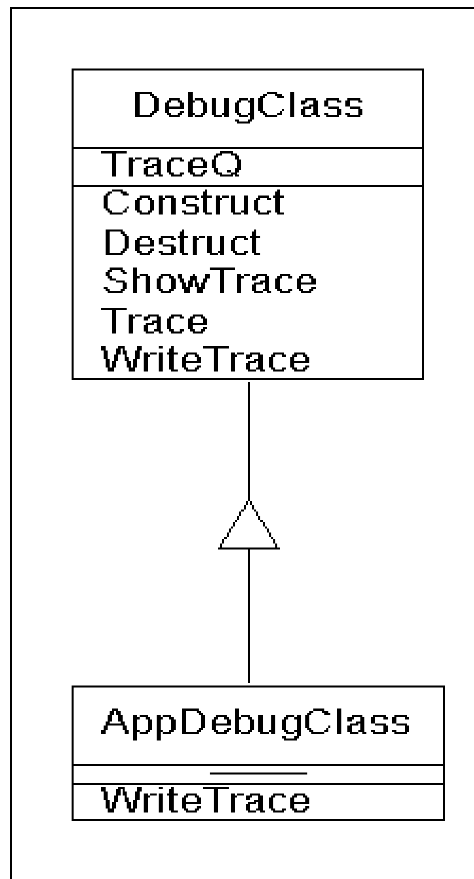
Well, not what you might expect. If you call the BrowseNames procedure, it turns out that AppDebugClass functions exactly the same way as DebugClass. The trace messages are still logged to the ASCII file, and the TPS Trace file remains empty.

What went wrong?

As Figure 2 shows, only the WriteTrace method is declared in both classes. The actual object or instance of the class which is used in your application contains both methods, and there are rules which govern which of the two methods will be called.

Figure 2. The DebugClass/AppDebugClass class diagram.

Figure 2. The DebugClass/AppDebugClass class diagram.



The problem is that although the object is an instance of `AppDebugClass`, in the example application code the `WriteTrace` method is never directly called by the application, as is the Trace method. `WriteTrace` is always called by the `Trace` method itself, which is declared only in `DebugClass`.

To understand what's happening you need to be clear on several concepts. One is that the db object is a combination of `AppDebugClass` and `DebugClass`. The other is that within that object are (in this case) two levels at which code exists: the upper (parent) and lower (child) levels.

When your code calls the `Trace` method it's making a call to the upper parent part of the object. The rule is that by default, an upper level cannot call down to a lower level. This makes sense, since the upper part (`DebugClass`) doesn't have an INCLUDE statement that points to the lower level (`AppDebugClass`), although the reverse is true. Another way to say this is that the child class always knows about the parent class, but the parent doesn't normally know about the child.

If you call a lower level method, that method can call a higher level method, implicitly (if the method doesn't exist at the lower level) or explicitly (by using the `PARENT` keyword). But under normal circumstances a higher level method can never call a lower level method. So once control passes to the higher level, that's where it's going to stay.

It would be possible to get around this problem by copying the Trace method to AppDebugClass since that would bring it down to the same level as the replacement WriteTrace method, but that would defeat the whole purpose of reusing code. What you really want is a way to tell the upper part of the object that you've created a replacement for one of its methods, and you want it to call your replacement instead of its own method.

Thankfully there is a way to do this, and it's called the Virtual Method.

Virtual Methods

Virtual methods are ridiculously easy to implement. Simply add the VIRTUAL attribute to the WriteTrace method in both classes, as shown in Listings 6 and 7.

Listing 6. DebugClass with the VIRTUAL attribute on WriteTrace.

```
DebugClass       CLASS,TYPE,MODULE('DEBUG.CLW')
NextLineToWrite    long(1)
TraceQ             &TraceQueue
Construct          PROCEDURE
Destruct           PROCEDURE
ShowTrace          PROCEDURE
Trace              PROCEDURE(STRING Text)
WriteTrace         PROCEDURE,VIRTUAL
                 END
```

Listing 7. AppDebugClass with the VIRTUAL attribute on WriteTrace.

```
AppDebugClass        CLASS(DebugClass),TYPE,↵
        MODULE('APPDEBUG.CLW'),LINK('APPDEBUG.CLW')
WriteTrace              PROCEDURE,VIRTUAL
                     END
```

The `VIRTUAL` attribute reverses the natural direction in which the program, at runtime, looks for methods. By default, methods are always checked for on the current level. If they can't be found, the program looks in the parent, and then that class's parent if present, and so on up until it finds a method.

Virtual methods work the other way around. If a virtual method exists at a particular level in an object, the program looks in the object to see if a replacement virtual method has been declared at a lower level.

Now if you compile the test application, and you create more than ten trace messages (or whatever the cache size has been set to) and bring up the Trace browse, you'll see that trace records have in fact been added to the TPS file.

> **NOTE:** You need to put the `VIRTUAL` attribute on both the parent and child `WriteTrace` methods.

This is an incredibly powerful feature of object-oriented programming. With virtual methods you can plug and play with the methods of a class without having to know how or when those particular methods are called. In fact, once you understand how virtual methods work, you understand how ABC works!

Virtual Methods and ABC

It's no understatement to say that without virtual methods there would be no ABC templates and class library. It's not that difficult to create a block of code, class or procedure, that can display and browse records. But how do you allow someone to plug their own code into that code? The legacy templates handle this problem by generating all of the code and allowing you to insert your code in as needed, but this results in excessive code generation and mixes templates and source code.

In ABC, almost all of the logic is in top-tier generic classes, which makes testing and debugging (by Topspeed) much easier. In most cases when you put code in an embed point, the templates create a derived method which is automatically called by the parent class (such as BrowseClass) at the appropriate time.

The derived classes are located in your modules, so they have access to all of your application's information. By means of virtual methods, the base classes "call down" into your created code whenever appropriate.

If you wish to format a browse box field for display, for instance, your code is generated into the virtual SetQueueRecord method which is called whenever the browse object needs to copy the data from the file to the queue used for display. (And this derived, generated SetQueueRecord also contains a call to PARENT.SetQueueRecord to ensure that the default behaviour still happens as well.)

I'm sometimes asked if, since virtual methods are so powerful, all methods should be made virtual. There is a small performance penalty associated with using virtual methods, since the system must maintain a virtual method table (VMT), or list of methods, so it knows at runtime which methods to call under which situation (this is an example of "late" binding). VMT lookups do take some time, though it's debatable whether in most situations you'd notice.

Under The Hood

At the start of this series of articles I commented that most ABC procedures begin with one line of code:

```
GlobalResponse = ThisWindow.Run()
```

and that this code appeared to not call any other code generated as part of the procedure. As you can now see, the answer to this puzzle is (drum roll) virtual methods. The call to `ThisWindow.Run` passes control up to the top tier base class, and the code rambles about at that level for a while until it comes across a virtual method (such as `ThisWindow.Init`), at which time control passes down to the lower derived level.

Virtual methods really do make working with the ABC class library "plug and play." You can also use this mechanism in your own classes. It's a good exercise to look at your class designs and ensure they're structured in such a way that someone else could selectively replace methods to change the behaviour without breaking the code. And by going to a two-tiered approach you help to maximize code reuse and maintainability, two key benefits of object-oriented programming.

Download the source code

David Harms is an independent software developer and the co-author with Ross Santos of Developing Clarion for Windows Applications, published by SAMS (1995). His company, CoveComm Inc, publishes Clarion Magazine.

Copyright © 1999 by CoveComm Inc. All Rights Reserved. Reproduction in any form without the express written consent of CoveComm Inc., except as described in the subscription agreement, is prohibited. If you find this page on a site other than www.clarionmag.com, email covecomm@mbnet.mb.ca.

June 1999 News
Posted on June 28, 1999

Clarion Advisor: Debugging Tricks
Posted on June 28, 1999

The ABCs Of OOP - Part 3
Posted on June 28, 1999

Clarion Magazine Best Read With Verdana
Posted on June 28, 1999

Detecting Crashes With DDE
Posted on June 28, 1999

Vol 1, No 5
June 1999

Clarion magazine

Main Page
Log In
Subscribe
Open Source
Links
Mailing Lists
Advertising
Submissions
Contact Us
Site Index
ClarionMag FAQ
Download PDFs
Search ClarionMag

Feature Article

Detecting Application Crashes

by David Podger

How do you know when your system is recovering from a crash or power failure? If you knew, you could check for any damage and run any necessary recovery procedures. This is not, however, an article about what to do when you crash; it is just concerned with figuring out that a crash happened.

If your application cannot work out that it is being restarted after a crash, then it has to rely on a user telling it that this is the case. But there are crashes and there are crashes, and your user may not always realise that a recovery is required. And even if they know, they may not always tell.

So, how to know? Here is a method that uses the Clarion DDE commands. I use it in an accounting app. It is intended to be used in combination with a log file (called EVENTS) which records each logon plus other significant events occurring within the app.

In a multi-user setting, when a user logs on the log file of current events will often show that other users logged on earlier and that they are presently doing various things using their respective instances of the app. That is, there will be unacquitted events (begun, but not ended) for each of these users. This contrasts with a single-user situation, where the simple presence of unacquitted events in the EVENTS file when a solitary user logs on is sufficient to tell the application that crash recovery is needed. In a multi-user situation, you need a detection method which is independent of the log file. Here's how you do it.

First you create a DDE server as a very small, separate program. It is not going to be much of a server, since its only purpose in life is to say: "I'm here, I'm here." Listing 1 shows the entire source code for such a server (called NoDelete.CLW to dissuade accidental deletion). (You can also download the source.)

BKO Enterprises, Inc.
Great Opportunity!
Salary to $125,000+
Pre-IPO Stock Opts
Sunny Boca Raton

etc 2000
If you're interested take our poll & let us know!

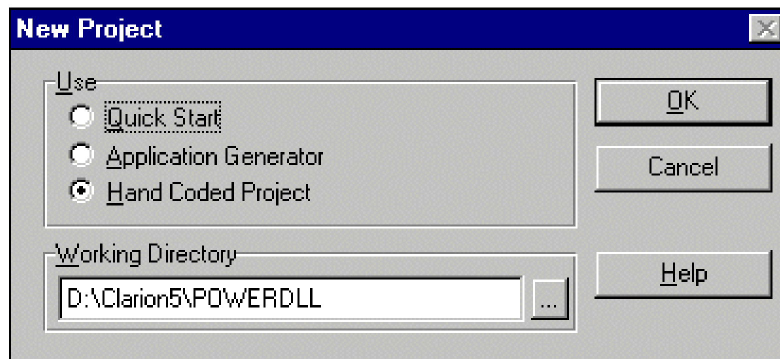
Listing 1. The DDE server (NoDelete.CLW)

```
    PROGRAM
    MAP
      INCLUDE('DDE.CLW')
    END
DDERetVal   STRING(20)

Window WINDOW,AT(,,95,13),COLOR(0FFFF80H)
         ENTRY(@s20),AT(0,0,5,12),USE(DDERetVal),HIDE
         PROMPT('PowerBooks Sentinel'),AT(12,2),USE(?Prompt1)
        END

MyServer LONG
```

```
    CODE
    OPEN(Window)
    ! Get the channel number
    MyServer = DDESERVER('RunControl','CheckAlive')
    ACCEPT
        CASE EVENT()
        OF EVENT:DDEexecute
            RETURN    ! blows away on any event
        END
    END
```

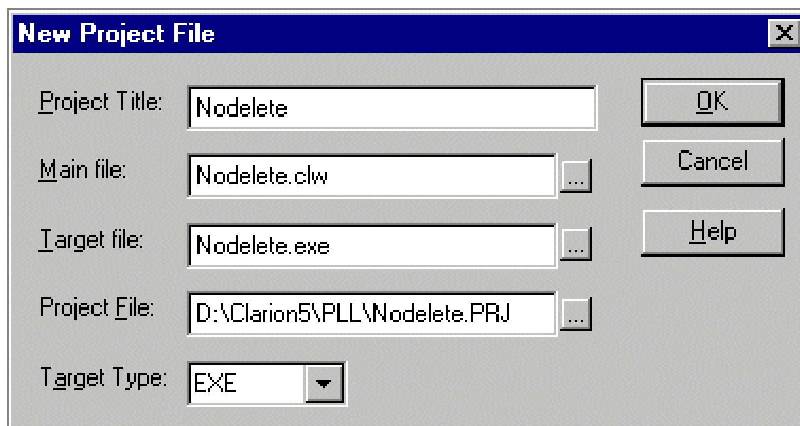
To create this program, click on Project in the main menu. Choose New and you will see the window in Figure 1.

Figure 1. Creating a new project.



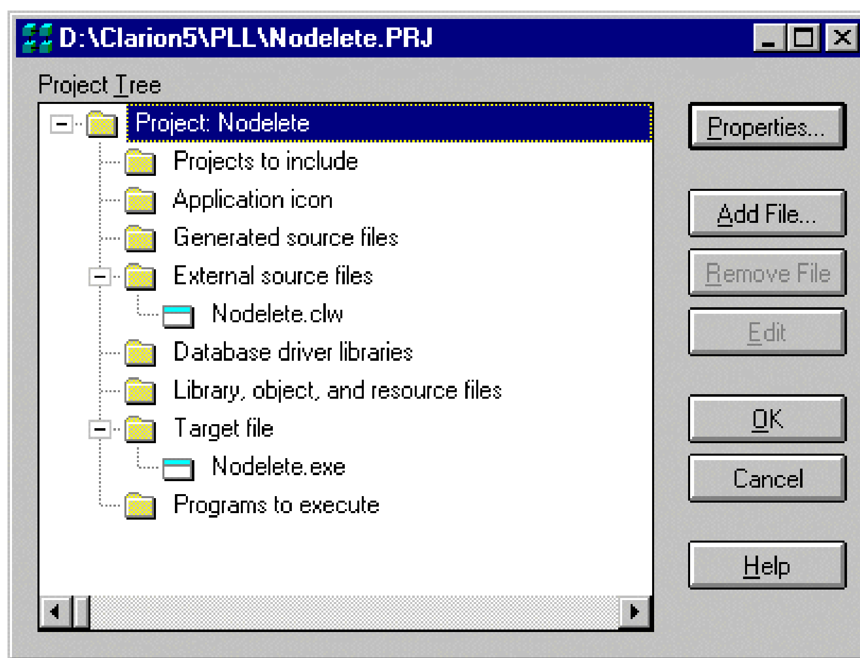
Choose Hand Coded Project and enter your Working Directory. Click the OK button and fill in the following form as shown in Figure 2.

Figure 2. The project properties window.



Click the OK button and the window in Figure 3 appears:

Figure 3. The project tree.



Now, double-click on "Nodelete.clw" and enter the code shown in Listing 1.

The business part of the code is the `DDESERVER` function. This registers NoDelete.exe as a DDE server with Windows, giving it an application name of "RunControl" and a topic of "CheckAlive." This is a server that doesn't have to respond to clients. In fact, the first time it detects any **command** `(via OF EVENT:DDEexecute)` it will terminate.

You could probably make the server even smaller, with a bit of experimenting.

When the main app is run it checks to see if NoDelete.exe is running. If it is not, and there are log file records that are unacquitted then very likely the application is recovering from a crash. The only other thing that can have happened is that NoDelete.exe has been accidentally deleted. I will discuss that eventuality later.

Listing 2 shows the code to run as you come into the app and process a request to log on from a user.

Listing 2. Checking for the existence of the DDE server.

```
ServerString = DDEQUERY('RunControl','CheckAlive')! Attempt link
IF UPPER(ServerString) <> 'RUNCONTROL:CHECKALIVE'    ! Not up
  RUN('NoDelete.exe')    ! Run it and return immediately
  ServerBegun = ''       ! Blank passed to returned parameter
ELSE                     ! Says server just started, non-blank
  ServerBegun = ServerString  ! Says server already running
END
```

When NoDelete.exe runs, it displays the window shown in Figure 4.

Figure 4. The DDE server window.

PowerBooks Sentinel

DDE communications depend on there being a window and an ACCEPT loop in the code that makes up the server. Without these, DDE won't work.

The DDEQUERY function does not communicate with the server. Rather, it addresses Windows and it asks only if a server with the specified application and topic name is registered. If it is, Windows returns the same information slightly differently formatted. That is all you want to know: that is, is your little sentinel program alive and well?

Well, there is one more thing you have to do at start up, but you cannot do it now because you need to give NoDelete.exe time to get up and running, if it isn't already. So, quite a ways into the Main Frame procedure put one line of code:

```
GLO:Server = DDECLIENT('RunControl','CheckAlive')
```

This code defines your app as a client of the NoDelete server and gets a channel number for the connection between this instance of the app and the server. Every time the app is run, each instance of it gets a distinct channel number. You need a channel number for one reason only: when you are shutting down the last instance of your app, you want to terminate the server. You need a channel number to be able to do this.

Just before your app shuts down, embed the code shown in Listing 3.

A convenient Legacy embed point is "End of Procedure before Closing files".

Listing 3. Shutting down the DDE server.

```
IF RECORDS(EVE:Key_Begun) = 0    ! nothing left in LOG file
  SETCURSOR(CURSOR:Wait)
  OPEN(ClosingWindow)
  ClosingString = ' Please wait a few seconds'
  DISPLAY(?ClosingString)
  DDEEXECUTE(GLO:Server,'[ShutDown]')  ! so shut down server
  SETCURSOR()
END
```

The DDE action here is to send a command to the server. The command DDEEXECUTE requires a channel number as a parameter. It doesn't matter which instance of your app is the last to close. It doesn't matter that each instance will have a different channel number. The effect on the server is the same. It terminates. Trouble is, it takes a while to do so, so that's why you see a "Please wait" message in the above code.

What if a user accidentally deletes the little server? It's not that easy to do. NoDelete.exe does not appear in the toolbar (in a 16 bit version running under windows 95) and it has no **[X]** box to tick. A user would have to press Ctrl Alt Del and deliberately do a deletion from the list of running programs. But, in the spirit of free and independent enquiry, some do, don't they.

One method of protecting against this is simple redundancy. Have two servers with different names and don't recognise a crash unless both are not there. If one is there, restart the other. Give the second one a name similar to those always in the list of running programs. Make sure its window displays right on top of the first one, with exactly the same text so the user has no clue there are two. The amount of overhead goes up slightly, particularly on shut down. It takes around 5 seconds to terminate one server on a 300Mhz Pentium. You decide.

Download the source

David Podger has been an independent Clarion developer in Australia for a decade. He presently lives in Katherine in the Northern Territory, where he sells a specialised accounting application for remote

June 1999 News
Posted on June 28, 1999

Clarion Advisor: Debugging Tricks
Posted on June 28, 1999

The ABCs Of OOP - Part 3
Posted on June 28, 1999

Clarion Magazine Best Read With Verdana
Posted on June 28, 1999

Detecting Crashes With DDE
Posted on June 28,

communities. 1999

Copyright © 1999 by CoveComm Inc. All Rights Reserved. Reproduction in any form without the express written consent of CoveComm Inc., except as described in the subscription agreement, is prohibited. If you find this page on a site other than www.clarionmag.com, email covecomm@mbnet.mb.ca.