An Introduction to FiveWin

By James Bott

Getting Started

Windows is a whole new world for Clipper programmers, but FiveWin eases the pain. Once you get your feet wet you will be amazed how easy it is to create a Windows program using Clipper and FiveWin. Here's an example of the proverbial "Hello World" program:

```
#include "fivewin.ch"
function main()
msgInfo("Hello World")
return nil
```

Hey, you can handle that!

FiveWin

FiveWin is a Clipper add-on product that allows creation of true Windows programs using the familiar xBase syntax. FiveWin consists of some Windows API wrapper programs written in C, lots of classes to define Windows behaviors, and a number of header files to allow the use of xBase like syntax. Of course, you can still use most of the old Clipper syntax.

FiveWin insulates us from the Windows API with the C wrapper programs. FiveWin also contains a complete OOP language which was used to write the classes. If you prefer, you can write your code in the OOP syntax (similar to Classy) or you can use a very xBase-like syntax which is then converted to the OOP syntax by the preprocessor.

```
define window oWnd
activate window oWnd
```

Is translated by the preprocessor to:

```
oWnd:=twindow():new(...  )
oWnd:activate()
```

The xBase-like syntax makes it easy for Clipper programmers to get up to speed. The OOP capability provides power and flexibility for the advanced programmer. You get the best of both worlds.

In addition to FiveWin, you will also need a Windows capable linker (such as Blinker) and a resource editor (such as Borland's Resource Workshop or Microsoft's AppStudio) in order to create Windows programs. Many of you may already have such a linker, and resource editors

come with many other language compilers such as C so you may have one of those too.

And now for something completely different.

The Windows Way

The biggest problem that Clipper programmers have writing Windows applications is understanding the *Windows way*. This means non-modal and event driven applications. In typical Clipper applications, the programmer controlled the work-flow by presenting the user with a menu, edit screen, etc. and requiring the user to finish it before anything else could be done (a modal methodology). Windows applications are usually written mostly in a non-modal fashion allowing the user to be in control by allowing them to jump from one thing to another without finishing the first.

New FiveWin programmers typically spend a lot of time trying to figure out how to recreate DOS methodology in Windows. Partially this is because of their old DOS mindset and partly its because of pressure from users that are currently working with legacy DOS applications. They want the Windows version to work just the same. In the long run this is not good for either the user or the programmer. Users may want the Windows version like the DOS version at first, but soon they are going to be unhappy with it. Sometimes the programmer has to just do what is best in spite of user protests.

It is not possible to cover much Windows philosophy here. The user should get one of the many books on the topic and read it thoroughly. A very good one is Charles Petzold's "Programming Windows 3.1."

Resources

Resources are a new concept for DOS only programmers. Icons, bitmaps, dialog boxes, menus, strings, etc., are all called "resources" and can all be stored in a single resource file in either a RC or DLL format. They can then either be accessed from the resource file by the Windows application (DLL's) or alternately, they can be compiled right into the EXE using a special resource compiler (RC's).

Resources can be designed in a special resource editor or some can be designed in other editors (such as Paintbrush) and then imported into the resource file by a resource editor. The resource editor then saves them in the desired resource file format.

A resource editor does not come with FiveWin, you will have to purchase your own. Although, theoretically, you could write Windows applications in FiveWin without one, don't try it! Borland makes one called Resource Workshop which comes with all their C compilers. It is also available directly from Borland as an individual product, but it costs more than their Turbo C++ compiler and takes 4 weeks for delivery (they don't do rush). Microsoft also makes a resource editor called AppStudio which is included with some of their compilers. Get one.

Note that you will usually see the resource compiler referred to as RC.EXE but the Borland compiler is named BRC.EXE. If you have the Borland compiler, the FiveWin BUILD.BAT file will have to be modified since it calls RC.EXE. When you compile a program using the BUILD.BAT program (and assuming the PRG and RC files have the same name) the resource compiler will be called automatically and thus your resources will be placed in the EXE.

If you want to place your resources into a DLL you'll need to start with an empty DLL. FiveWin provides one called \DLL\SCREENS.DLL. Just make a copy of this file with a new name and load it into the resource editor and import all the resources into it. Then call it from your program using SET RESOURCES TO "myapp.dll".

If you use any of Borland's special resource types you will have to load Borland's BWCC.DLL also. If you don't, you will get some generic errors like "Can't load dialog box" that will leave you clueless.

To load the Borland library, in your code, near the top of your main function issue:

```
Local hBorland := LoadLibrary("BWCC.DLL")
```

And just before the return statement in your main function call:

```
FreeLibrary(hBorland)
```

Windows
We will use *Windows* in proper case to describe the Windows operating system and *windows* in lower case to describe a generic window. Windows themselves come in several types. There is a SDI window, a MDI window, a MDI child window, and a dialog window (also called dialog box or just dialog).

The SDI (single document interface) window is just, well, a window. When it is iconized it is placed on the desktop. The Windows Write program is an SDI application.

The MDI (multiple document interface) consists of a parent window which contains MDI child windows. MDI child windows always remain within the boundries of the MDI parent window, even when iconized. MDI child windows are non-modal. The Windows 3.1 Program Manger consists of a MDI parent window with each program group contained in a MDI child window. Most word processor's are also MDI applications.

A dialog window contains some type of dialog with the user. Dialogs are usually modal although some are non-modal such as a Find dialog in an editor.

Shell Window
Every application has a main or shell window. This is usually a MDI window but could be a SDI window and is sometimes a dialog window if it is a small single-purpose application.

Here's how to define a MDI shell window:

```
define window oWnd title "MDI Shell" MDI
activate window oWnd
```

Here's a SDI window:

```
define window oWnd title "SDI Shell"
activate window oWnd
```

Or, a dialog:

```
define dialog oDlg title "Dialog Shell"
activate dialog oDlg
```

Colors
FiveWin windows default to the colors black on dark gray. You can change the window colors by adding a **COLOR** clause to the window definition:

```
define window oWnd colors "N/W"
```

Or, you can use manifest constants (see \INCLUDE\COLORS.CH). You may find them easier to read.

```
define window oWnd colors CLR_BLACK, CLR_LIGHTGRAY
```

[Note: Remember that manifest constants must be all uppercase.]

However, you'll probably want to use the standard default Windows colors as defined by the Control Panel. These colors are stored in the WIN.INI file in the COLORS section. Here is a function to retrieve these colors:

```
function iniColor(cID)
local cSetting,nLoc1,nLoc2,nRed,nBlue,nGreen
cSetting:=alltrim(getProfString("colors",cID))
nLoc1:=at(" ",cSetting)
nLoc2:=rat(" ",cSetting)
nRed:=val(left(cSetting,nLoc1))
nGreen:=val(substr(cSetting,nLoc1+1,nLoc2-nLoc1))
nBlue:=val(right(cSetting,len(cSetting)-nLoc2))
return ( nRed + (nGreen * 256) +(nBlue * 65536) )
```

There are some colors sets you will find useful:

```
...colors iniColor("windowText"), iniColor("appWorkSpace")      // shell window colors

...colors iniColor("windowText"), iniColor("window")            // child window colors

...colors iniColor("graytext"), iniColor("window")              // disabled text
```

If you use these ini colors, users can change to whatever colors they like using the Control Panel. To get a better understanding of how this works, run the Windows 3.11 File Manager and change the colors using the Control Panel. The colors will be updated in the File Manager when you save them in the Control Panel. Note that, unlike the File Manager, FiveWin applications' colors will not be updated until the program is run again. But, this should not be a problem.

Icons

You will need to assign an icon to the shell window. You can assign an icon as a file:

```
define icon oIcon file "..\icons\myapp.ico"
define window oWnd title "Shell" icon oIcon
```

Or, use an icon as a resource:

```
set resources to "myapp.dll"
define icon oIcon resource "myapp"
define window oWnd title "Shell" icon oIcon
```

However, if you place this icon in a DLL it will not be visible to the Program Manager nor will it appear when the application is iconized on the desktop. It is best to place the shell window icon into a RC file using a resource editor, and then compile this icon into your application. Note that you can compile some resources and still call others from DLL's in the same application. When you refer to resources that have been compiled into the application there is no need to do a SET RESOURCES TO command, so all you need is:

```
define icon oIcon resource "myapp"
define window oWnd title "Shell" icon oIcon
```

If you are using multiple DLL's, then you need to specify them all early in your program.

```
set resources to "myapp.dll", "misc.dll"  // at beginning of program
```

Then before you call the desired resource, you must ensure that it's resource file is active.

```
set resources to "myapp.dll"
redefine ...
```

This syntax is somewhat like opening index files and then selecting the one to be active.

Brushes

Brushes are used to fill the background of a window. They can provide texture and/or color to an otherwise dull boring background. But be careful, you can get too carried away and make the background so "busy" that you get tired of looking at it. Also, be aware that brushes eat Windows memory resources. The BORLAND style is popular for the shell window:

```
define brush oBrush style BORLAND
define window oWnd brush oBrush
```

Here is an example of gray diagonal lines on a white background:

```
define brush oBrush style diagonal color CLR_LIGHTGRAY
```

You could define just a color brush, but it seems to be no different than using the **COLOR** clause for the window definition.

```
define brush oBrush color CLR_YELLOW
define window oWnd brush oBrush
```

Gives the same result as:

```
define window oWnd color CLR_BLACK,CLR_YELLOW
```

You can also use bitmaps, see BRUSH?.BMP and BRICK?.BMP. Play with all these options to see what you like. Personally, I find all but the BORLAND style a bit too busy.

Menus

The topic of menus can get complicated. We'll just cover the basics here. So far we have used the DEFINE command to create windows, icons, and brushes. Menus are created differently. One would think we would:

```
define menu oMenu
```

But since menus require complex multiline definitions, the syntax for menus is:

```
menu oMenu
...
endmenu
```

Since we cannot define a menu in one line like brushes and icons, we have to encapsulate the menu definition in a function which returns a menu object.

```
static function buildMenu()
local oMenu
menu oMenu
...
endmenu
return oMenu
```

Then we attach the menu to the window in the window definition statement just like icons and brushes:

```
define window oWnd menu buildMenu()
```

Menus can also be created using a resource editor, then **DEFINE**d in FiveWin:

```
define menu oMenu resource "mainmenu"
```

Sure, now we use the **DEFINE** clause with menu.

You will rarely see a Windows application that uses multiple level cascading menus as is common in DOS applications. This is due to the current popular belief that most users have a hard time grasping multilevel concepts. Windows provides the dialog box containing such controls as radio-buttons, listboxes, comboboxes, and checkboxes to handle collecting complex information from the user. This allows us to collect the information we need in just two levels--one pulldown menu and one dialog box.

Message Bars

If you want a status line on bottom of your window, you define a message bar. There are two ways you can do this. You can use the Clipper syntax with a minor variation to attach it to a window:

```
set message of oWnd to "Default message goes here."
```

Or you can use the FiveWin syntax:

```
define message oMsgBar of oWnd prompt "Default message goes here."
```

This default message will appear whenever other messages are not showing. Messages defined for menu choices will be displayed on the message bar when the menu choice is highlighted. Buttonbar messages also appear here. Just define the message as an empty string, "", if you don't want a default message.

There are also some great clauses; **KEYBOARD**, **DATE**, and **TIME**.

```
set message of oWnd to "FiveWin demo" keyboard date time
```

This message bar shows the keyboard status of numlock, capslock and insert keys, and the date and time, all in recessed boxes on the right side. Slick and simple!

Toolbars (Buttonbars)
Toolbars, or buttonbars as they are called in FiveWin, are defined in the conventional manner:

```
define buttonbar oBar of oWnd
```

Note that the **OF** clause attaches the buttonbar to the desired window. Buttonbars are not defined in the window definition statement like menus.

Buttonbars default to the same dark gray color as windows. You may prefer the more common light gray, but the buttonbar does not have a **COLOR** clause like a window, so we have to resort to OOP syntax and call the setcolor() method:

```
define buttonbar oBar of oWnd
oBar:setColor(CLR_BLACK,CLR_LIGHTGRAY)
```

If you use the three dimensional clause, **3D**, the default color is light gray and the default size is smaller.

```
define buttonbar oBar 3D of oWnd
```

Note that you will need bitmaps of a smaller size (26 by 26 pixels) than those included with FiveWin for this size bar. You can create these in Windows Paintbrush. Use the Option-Image Attributes dialog to set the image dimensions.

To create a buttonbar that looks like the Microsoft standard:

```
define buttonbar oBar OF oWnd size 24, 26 3D
```

You will need bitmaps 18 pixels high by 19 pixels wide for these buttons.

Next we attach buttons to the bar, and bitmaps, actions, and tooltips to the buttons:

```
define button of oBar;
  resource "new";
  action new();
  tooltip "New"
```

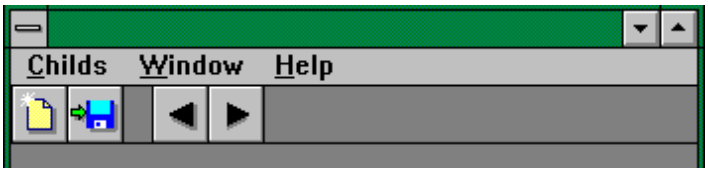
Here we have used the bitmap "new" from the current resource file, specified new() as the action to be performed, and "New" as the tooltip (a little yellow message that appears under the button when the cursor is left over the button for a second or so). You could use the **MESSAGE** clause instead of (or in addition to) the **TOOLTIP** clause, but in order to read a message on the message bar, the user has to move their eyes to the bottom of the screen while the tooltip appears right where they are already looking.

To separate the next button on the bar from the previous one, simply use the **GROUP** clause:

```
define button of oBar resource "prev" tooltip "Previous" group
```

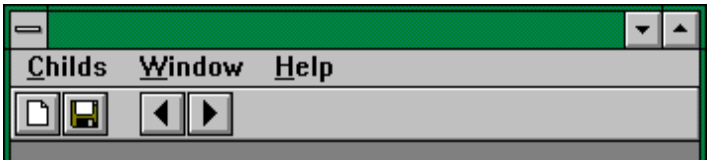
Here we have a FiveWin default style toolbar with two groups of buttons:

```
define buttonbar oBar of oWnd
define button of oBar resource "new"
define button of oBar resource "save"
define button of oBar resource "prev" group
define button of oBar resource "next"
```



Here's the same toolbar Microsoft style (and with Microsoft style bitmaps):

```
define buttonbar oBar of oWnd size 24, 26 3D
```



MDI Child Windows
These windows are moveable, resizable windows that stay entirely within the boundary of the MDI parent shell window even when maximized or iconized.

```
define window oWnd MDIchild of oWndShell
activate window oWnd
```

MDI child windows are great for database browses and text editors or viewers.

```
//--- database browser
function dataBrowse(cFile,oWndShell)
define window oWnd title "Data Browser" MDIChild of oWndShell
use (cFile) new shared
@ 0,0 listbox oLbx fields update
oWnd:setControl(oLbx)                    // attach the control to the window border
activate window oWnd
return nil

//--- text viewer
function textView(cText,oWndShell)
local oMemo,oWnd,oLbx
define window oWnd title "Text Viewer" MDIChild of oWndShell
@ 0,0 get oMemo var cText memo of oWnd update
oWnd:setControl(oMemo)
activate window oWnd
return nil
```

MDI Menus
Standard MDI child window menus are shown on the **MDI shell window** when the child window is in focus, **not** on the MDI child window itself as one might expect. This is a strange concept for Clipper programmers.

Thus, a MDI shell window might only have two pulldown menus, File and Help (although it might also have such menus as Tools and Options). When a MDI child window is open then, perhaps, the menu would display "File Edit View Window Help." The File and Help menus would be the same as they were when the shell window was in focus. Although the "Window" menu would be the same on all MDI child windows, it is of no use unless at least one child window is open, thus there is no reason to have it on the shell menu.

From a minimalist viewpoint, it would be best to use OOP to inherit the MDI shell menu and insert pulldown menus to the parent menu to create the MDI child menus. I do not know if this can be done in FiveWin since it uses an unusual combination of classes and functions to build menus. Otherwise you will have to recreate the shell pulldown menus in each MDI child window's menu.

IMPORTANT NOTE: Through FiveWin ver 1.9.1 you must explicitly END (destroy) the menu of any MDIChild window when you close the window or Windows memory resources will not be recovered, eventually causing a system crash. Adding this **VALID** clause to the **ACTIVATE** statement ensures that the window cannot be closed without **END**ing the menu. FiveWin will do this automatically in future versions.

```
activate window oWnd valid (oMenu:end())
```

MDI Message Bars
MDI child windows may have their own message bar or they may use the shell window's message bar.

Dialog Windows
Dialogs are a subclass of the window class, but they have some very different behaviors. This was dictated by Microsoft, not by FiveTech. Although dialogs are windows, they use different size units, cannot be defined as a MDIChild, and we have to resort to tricks to attach a buttonbar.

The size units used by dialogs are larger than those of a window. To get the same size object in dialog units divide window units by 2.05.

```
@ 1,1 button "&Cancel" size 80,25 of oWnd                    // on a window

@ 1/2.05,1/2.05 button "&Cancel" size 80/2.05,25/2.05 of oDlg    // on a dialog
```

Note that dialogs, once activated, become *windows*. This strange behavior allows you to do anything you can do to a window to a dialog box using the **ON INIT** clause. For example, using this technique we can define controls with the normal windows dimensions:

```
define dialog oDlg from 5,5 to 25, 35
activate dialog oDlg ON INIT initControls(oDlg)

static function initControls(oDlg)
@ 1,1 button "&Cancel" size 80,25 of oDlg          // normal window sizes
return nil
```

Normally, dialogs cannot have a buttonbar, but we can use **ON INIT** to attach a buttonbar to a dialog after it becomes a *window*:

```
#include "fivewin.ch"
static oDlg

function main()
define dialog oDlg from 5,5 TO 30,50
activate dialog oDlg ON INIT toolbar()   //  attach it to the dialog
return nil

static function toolbar()
```

```
local oBar
define buttonbar oBar of oDlg
define button of oBar resource "new"
return oBar
```

In this case, we define a buttonbar like a menu using a function, then we attach it with the **ACTIVATE** statement rather than the **DEFINE** statement that we use for menus.

Source Code vs Resources

You can create dialogs using source code as shown above, but this is when a visual designer, such as Workshop, really comes in handy. A resource editor allows you to create an entire dialog box using drag and drop. You drag the various controls onto the dialog box, position and resize them, and define their parameters. Those controls that are to be active during use are **REDEFINE**d in FiveWin to attach the proper variable (such as a fieldname) to them:

```
redefine get cust->name ID ID_NAME of oDlg
```

Static controls, such as labels (**SAY**s), don't have to be **REDEFINE**d. Each resource in a resource file has a unique ID number associated with it. These are written to a header file by the resource editor. This way you can use the manifest constants for the ID#'s. They're easier to write and read. Alternatively, you can eliminate the header file and use the numbers directly:

```
redefine get cust->name ID 102 of oDlg
```

But, we're getting ahead of ourselves. FiveWin comes with a great little utility to automatically generate a RC file from a DBF. First compile the \SAMPLES\DBF2RC.PRG program. Note that the 1.9.1 version has a couple of bugs. Find the generated "#INCLUDE" and "#DEFINE" statements in the program (lines 78, 79, & 89) and convert them to lower case. Also, change the generated remark notation from an asterisk to double slashes (line 61). Borland's Workshop (ver 4.5) will not be able to load the resource file if these are not fixed.

After you have this running, select a simple DBF such as one of the FiveWin sample files and run it through. You will get a RC file and a CH file of the same filename as the DBF. Now use a text editor to open up the RC file and take a peek. It is just an ASCII file. Check out the CH file in the text editor also.

Now, load this RC into your resource editor and select the dialog for editing. You will see all your fields defined as controls in the dialog box. All string fields are defined as GETs, all logical fields are defined as checkboxes, and all memo fields are defined as multiline GETs. Now you just have to move them around to your liking. Some controls you may wish to change into radio buttons, comboboxes or the like. Save the RC.

Now you need to write the dialog code. You will need to write functions for all the common navigation and editing capabilities in addition to **REDEFINE**ing the active controls.

```
//--- Customer Data Entry Screen
function custDES()
redefine dialog oDlg resource "customer"
redefine get cust->company id ID_COMPANY of oDlg
redefine get cust->address1 id ID_ADDRESS1 of oDlg
...
activate dialog oDlg on init buildBar()
return nil

static function buildBar(oDlg)
```

```
local oBar
define buttonbar oBar of oDlg
define button resource "prev" of oBar action previous() tooltip "Previous record"
define button resource "next" of oBar action next() tooltip "Next record"
...
return oBar

static function previous
skip -1
return nil

static function next
skip
return nil
```

This has been vastly oversimplified just to show the basics. One can readily see the need for better database access methodology and OOP programmers can see that this is a situation begging for a base class solution.

Database Class
Note that the above dialog is modal (by default) and thus we can get away with SKIP but if it was nonmodal we couldn't assume we were still in the correct workarea and we have to prevent skipping to the phantom record past the EOF. Also, we don't want to directly edit the database fields and we need to be concerned about multi-user issues. Luckily FiveWin's TDATABASE class handles most of this for us.

FIVEWIN ENHANCEMENTS

There are two enhancements that would greatly simplify creating a database application in FiveWin. First we need an enhanced database class that handles multi-user locking better. Second, we need a data window class that handles all the standard database navigation and editing functions. Such a data window class would reduce the code required for a new database dialog to just the **REDEFINE** statements. With the addition of another simple program to generate a PRG from the RC file, we can eliminate manual coding of data windows altogether!

The procedure to build a data dialog would then be to first create the DBF. Then generate a RC file from the DBF. Edit the RC dialog. Generate a PRG from the RC. Compile, link, and run!

DBF > DBF2RC.EXE > RC > WORKSHOP > RC2PRG.EXE > PRG > EXE

Database Class with Mulituser Locking
FiveWin comes with a simple database class, but, in order to develop multi-user applications you will either need to purchase a separate database class (FiveDB, available from Omicron) or you will need to write your own. The FiveWin TDATABASE class is a good start, but it doesn't provide any file/record busy messages or automatic retry capabilities which are essential for multi-user applications.

Also, you will have to address the record locking methodology issue. Even a single user Windows application is really multi-user since it is non-modal and many files can be acted upon at once by a single user. Even if we wrote a Windows application entirely modal, we can't prevent the user from accessing the same files from another Windows application. So, we really must write all Windows applications as multi-user.

Locking a record before editing as is usually done in Clipper apps by selecting "Edit," is not the Windows way. Windows users expect to be able to just point to a field, click, and begin typing. You could use an Edit button to trigger a record lock, but this would be inconsistent with other Windows applications and even other dialogs within the same application since most dialogs do not have Edit buttons.

FiveDB does a post-edit record lock and update. There is a risk that someone else has updated the record while the user was editing, so FiveDB checks and the user is allowed to determine which, if any, of the fields should be updated. The post edit locking method allows the record to remain unlocked except for a split second when it is updated after the edit.

An alternative locking method would be to detect when the user first entered a GET and to use that event to trigger a record lock just as if the user had pressed an Edit button. This way the interface acts in the Windows style, but the mechanism is the same as the common DOS method. The record is locked during the edit so there is no chance of another user making changes to the same record. The downside is that the record could be locked for a long time.

Note that other Windows applications such as editors and spreadsheets load the entire file into memory and lock the disk file, so they don't have this interactive locking issue to address. Lucky us!

Data Window Class

A data window class would greatly ease programming a FiveWin database application. A data window class would be fully able to handle the common record navigation and editing functions. It would have a menu and/or a toolbar built in. It would be multi-user capable and non-modal. It would also have built in browse and form views. All we would need to do is to pass the parent window object, and a data server object. In order to use a dialog resource, we would just subclass the data window and do the **REDEFINE**s in a new method.

```
create class CustWindow from tdataDialog
   method initControl
endclass

method initControl
redefine ...
...
return self
```

We could also add buttons and functionality in the new subclass by adding additional methods.

This dataWindow class would eliminate 95% of the coding each time we created a new database dialog form.

However, we could go a step further and write a program to read the RC file and generate that last 5% of code.

Summary

FiveWin provides Clipper programmers with a fast route to developing true Windows applications. The xBase language syntax is easy to learn and OOP language capability provides power and flexibility. Clipper programmers will probably have a harder time learning Windows methodology than learning the FiveWin language.

In addition to Clipper and FiveWin, programmers will need a resource editor and a Windows

capable linker (such as Blinker) to be able to generate Windows programs. With these tools one can create Windows programs that will run under Windows 3.x, Windows 95, and Windows NT.

I hope this introduction will get you up and running with a little less effort than it took me.

By James Bott, CompuServe 71706,551, Internet 71706.551@compuserve.com