CA-Visual Objects 2.5
Putting The Pieces Together

Gary Stark

CA-World 2000
eBusiness Solutions in Internet Time
JP065SN

Introduction

CA-Visual Objects. It's here and now, and it's ready for prime-time development. As you know, CA-Visual Objects is a Windows 95/98/NT/2000-based development tool. As a fully object-oriented development environment, it provides us, as developers, with the tools we need to step into the brave new world of software development.

For those of us familiar with traditional DOS-based programming techniques, there are many new methods to learn and concepts to understand. For those of us familiar with CA-Clipper, we will also find that the richness of the CA-Visual Objects language will also increase the challenge in developing our applications. There are so many new tools, so many new features that we need to assimilate and become familiar with, that the task seems daunting.

Those of us who have become comfortable with the concepts of object-oriented programming and design will readily appreciate the power and flexibility afforded to us by CA-Visual Objects. Leveraging CA-Visual Objects' power through the use of an object-oriented database tool, such as CA's Jasmine ODB, really adds to the flexibility, power, and ease of use offered to us in our 21st century development efforts.

What now is needed is a road map to CA-Visual Objects and Jasmine – a guidebook, if you like – and today I shall attempt to provide you but a small portion of that guidebook. We shall have a brief look at some of the various components that comprise CA-Visual Objects and Jasmine, and we shall see how those components all fit together for us in our job of creating applications for our clients.

Subsystems

CA-Visual Objects is built from a number of different subsystems. These subsystems include, but are not restricted to, the various editors provided within the IDE, the Database Server, DBF classes, GUI Classes, Jasmine classes, and so on. These subsystems may be used independently on their own or in conjunction with each other. The way that you will specify and use them will be dependent upon the specifics of how you will want to build your application.

It is now many years since the Windows environment became the de-facto standard for desktop systems. I can recall hearing – many times, over the years – that DOS applications will die and be replaced by their Windows replacements.

Just like COBOL

Although this is happening, it is happening very slowly. There are still many Clipper applications in regular day-to-day production environments, and these legacy systems continue to be deployed on brand new high-performance Windows-based systems. I'm aware of one major international financial institution that today continues to run its primary business systems, written in Clipper, on NT 4.0 workstation systems. If your initial interest is in converting your legacy CA-Clipper code into CA-Visual Objects, then some of your primary points of focus will most likely be the DBF and GUI classes.

If what you need to build is a fully object-oriented application, using all of the bells and whistles that Windows can provide, then there are a number of different areas to look at. These may include the Jasmine, DBF, and GUI classes and perhaps the Windows API.

Perhaps, if you're into communications or need some other facilities, then you might be looking at some third party DLLs, OCXs, or perhaps some of the Microsoft APIs such as Telephony or MAPI as well.

If you need to quickly prototype something for a user, then using the Menu and Window Editors will be of great assistance to you. Selecting the GUI classes when you're creating a brand new application will cause CA-Visual Objects to automatically build a default application. You can then take this default application and using the Menu and Window Editors begin to customize this application - add data elements, buttons, scroll bars, or whatever - to build your prototype application in a very short time.

Using some of the predefined applications from the application Gallery will make your task even easier. Creation of a Jasmine application begins, of course, here, with your selection of either the Jasmine SDI or Jasmine MDI application. Of course, having the appropriate Jasmine servers installed and available to your application is a prerequisite at this time.

Depending upon your needs and requirements – or more specifically, your application's needs and requirements - you can choose to use virtually any combination of these pieces in pretty well any way that you wish to build your application in the way that you feel is most appropriate.

Let's now look at some of these pieces.

The First Piece

To begin with, all CA-Visual Objects applications need to have a starting point. In C applications, and in Clipper, this point was the function called "MAIN." In CA-Visual Objects, this is called "START", and although it may be a function of your application, if you're using the GUI classes, your most likely point of entry will see this as a method of the Class App.

Here is the code from the Start method of the standard MDI application that CA-Visual Objects provides in the application gallery.

```
METHOD Start() CLASS App
LOCAL oMainWindow AS StandardShellWindow

   SELF:Initialize()

   oMainWindow := StandardShellWindow{SELF}

   oMainWindow:Show(SHOWCENTERED)

   SELF:Exec()
```

As you can see, this is quite a small amount of code, but within it is quite a lot of power, brought about by the use of classes within CA-Visual Objects.

We are initially declaring a local variable, oMainWindow, and strong typing it as belonging to class StandardShellWindow. This class has a number of properties, such as menus and methods, which although predefined, are also able to be extended and modified by you. It is by your modification of these properties and provision of additional methods that you can provide your application with the functionality that your users require.

We could start by assigning a different caption to our window, so that it now shows a meaningful name for our application.

Here is the modified code for the method Start of the class App:

```
METHOD Start() CLASS App
LOCAL oMainWindow AS StandardShellWindow

   SELF:Initialize()

   oMainWindow := StandardShellWindow{SELF}

   oMainWindow:Caption := "CA World 2000 Demonstration"

   oMainWindow:Show(SHOWCENTERED)

   SELF:Exec()
```

One of the features of object-oriented programming techniques is the ability to create classes and bestow upon those classes properties and methods that define the characteristics and behavior of those classes. We need to understand and accept that within CA-Visual Objects, windows inherit from a common base class and therefore possess many common properties; the *Caption* property is one such item. As the class StandardShellWindow inherits from this base window class, it therefore inherits the properties and methods of its derivative class.

As we can see, all that was required in order to change the text on the caption line was a simple assignment of a text string to the Caption property of the oMainWindow object.

And for much of what is required within CA-Visual Objects, this is all that is needed: Determine the property that is affected and assign the desired value to it.

But before we move on, consider also the effect of this code:

```
Assign Caption( cCaption ) CLASS StandardShellWindow

   SUPER:Caption := cCaption + " – Visual Objects with Jasmine"
```

Let us now draw a contrast between the standard application's startup code and that of a CA-Visual Objects Jasmine application.

```
METHOD Start() CLASS App
       LOCAL oMainWindow AS StandardShellWindow
       LOCAL i, iLen AS INT
       LOCAL sCName AS STRING

       SELF:Initialize()

       CAPaintShowErrors(TRUE)
       goSession := GetDefaultSession("", "", "", "", "test.log", TRUE)
       IF !goSession:StatusOK
                MessageBox(0, "Unable to create session", "Jasmine Standard
       Application", _Or(MB_OK, ;
        MB_ICONEXCLAMATION))
           BREAK
       ENDIF
       goSession:Debug := TRUE

       oMainWindow := StandardShellWindow{SELF}

       oMainWindow:Show(SHOWCENTERED)

       SELF:Exec()
```

The significant changes in this code are the addition of lines that create and establish a Jasmine session within which the application will conduct its business. This code establishes the connection to the Jasmine server, where the data source for the application resides.

Menus

Within the world of DOS programming I have always found that menus were a contentious point with me. There have always been just so many different ways of doing these tasks, all of which worked, and many of which worked very well. I think that it's a relief to come into the Windows world and have them standardized, and the fact is that even by hand coding this, it's just so easy to do.

But the easiest way to create menus in CA-Visual Objects is via the Menu Editor. By simply pressing the Enter key with the cursor on an option, a new item will be created. Type in the caption you wish to see, using a leading ampersand character to make a letter a hot key. Then, add the name of the event to be called when the selection is made, and that's about it.

You can add toolbar buttons and ToolTips by assigning the appropriate values in the Menu Item Properties dialog.

Windows

Windows are of course where the application does its stuff. Within the CA-Visual Objects IDE there are five basic types of window available. These are the Data window, the Dialog window, the DataDialog window, the Shell window, and the OleData window. It is likely that for most of the work that you are likely to do, the Data or DataDialog windows will be where you spend much of your development time.

To enable access to a window from the menu event, name the window for the event that you have specified in the Menu Item Properties dialog. For example, if the menu item is called "My Window" in the caption, and the Menu Item Event property has a value of "MyWindow" assigned to it, you should name the window "MyWindow" also. Binding of the menu event and its associated window will occur automatically if you conform to this procedure.

While this procedure is powerful and automatic, thus saving us, as developers, much time and work, you may also choose to override the default behavior and create a specific method that will load the window from the Menu Click event. This might be done when, for instance, you wish to load some specific values into properties of the window prior to exposing the window. Setting a custom caption to a window might be a good example of when this might need to be done.

Once your window has a name, you may then assign some controls, such as Single- or Multi-Line edits, Buttons, List Boxes, and Tree- or List-Views to it and modify the window properties if you wish.

To alter the basic behavior of a window, you should be careful to avoid modifying its Init() method. Instead, use the PreInit() or PostInit() method, as this will preserve your source code but still provide for the use of the code that CA-Visual Objects generates for you from the Window Editor.

Customizing Your Desktop

The way that the CA-Visual Objects desktop appears and operates on your system is configurable.

The default fonts and colors used, and such items as the default compiler switches, subdirectories used for export of your AEFs, MEFs and source code files, may all be set. Many of the basic properties, such as the size of the windows or the width of the list view columns, may be altered and saved according to your preferences.

To set the basic desktop sizes and appearances, set them the way that you wish to see them, and then select the Files/Save Desktop menu option.

To modify the fonts, colors, or default compiler options, select the Files/Setup menu option.

Many of the options may also be set at the application or module level by accessing the properties for that application or module. For instance, typically you would set the debug option to auto, but for one particular piece of code in one module you might wish to specifically turn debugging on.

The Windows API

This is where the fun really begins.

With CA-Visual Objects you have access to all of the Windows API calls. This includes the GDI, Sound, Animation, Multimedia, Telephony and more. Additionally, the support that has been provided for calling DLLs and OCXs enhances the power available to you as a programmer.

What exactly can you do?

What exactly can you think of?

Let's start with something very simple. We'll open up the Windows Notepad application. Remember to include the Win32 API Library when you create the application.

```
FUNCTION Start
   WinExec( PSZ( "NotePad" ), SW_SHOW)
RETURN NIL
```

While this is not the sort of application you're likely to make lots of money with as a vertical applications developer, it graphically demonstrates the ease with which you can start to become very sophisticated with your selection and use of the components which get included in your application.

Let's get a little more daring. We'll presume that we have a Windows sound driver loaded, and hopefully a sound card installed in the system.

We'll also presume that the file "Ding.Wav" resides in the C:\Windows subdirectory.

```
FUNCTION Start
   SndPlaySound( PSZ( "C:\Windows\Ding.Wav" ), SND_ASYNC)
RETURN NIL
```

Now we've got the means of providing some quite usable sounds to accompany our application. With the right sound effect to accompany various windows and dialog boxes, we can provide a very friendly - or hostile - environment for our users to work in.

Imagine if, after having entered data that fails a validation check, your user gets a massage that says something like "I'm sorry, that data is not valid."

Or perhaps when your user invokes the runtime error handler with an unrecoverable error you might like to supply a message along the lines of "I've fallen and I can't get up!"

There is much to look at and digest within the Windows API.

C Without Semicolons?

Let's look a little more closely at the Windows API, and some of the things that we can do with it.

Calling of functions within this interface is performed using the standard Windows conventions, which of course to many programmers remain uncharted waters.

Much of the work that we do has a certain C-like structure to it. As we move into the world of CA-Visual Objects you will see that this scenario becomes even more the case. Many of the functions that we may call will be C functions in the API. If you have the opportunity to delve into the Visual Objects SDK, you might even notice that much of what is there is simply a CA-Visual Objects wrapper to the Windows API. This is not a bad thing; quite the contrary in fact, as it has simplified the way in which we can address and utilize the elements and power of Windows.

One of the really nice things – to me – is the fact that we can even have the Windows Operating System institute a callback to our application. That is, the operating system will actually place a call to our application to tell it to do something that we want it to do. We call it and it calls us. To give you some idea of what's happening, consider for a moment MS-DOS calling a CA-Clipper function in one of your apps. In the MS-DOS environment this does not happen.

In Windows, however, this is very typical behavior, and we can use this to our advantage when we need to. Consider that we might need one application to notify a second when a task has been completed, so that it may continue its processing. The first application merely sends a message to Windows, which then acts as a traffic cop, forwarding that message on to its intended recipient.

Oh yes, the code we write will now begin to look very much like C code, but it's not, it's CA-Visual Objects.

But getting things done doesn't really take much effort at all. Let's start by writing a simple program that tracks the position of the mouse in our application's client window.

Although this code doesn't really do very much, it will illustrate very graphically just how much happens under the Windows operating system, and at the same time it will show you methods to paint windows and determine where your mouse is, both very important tasks within the Windows world.

I need to point out here that we're working at a very low level of code, and we're not using the CA-Visual Object IDE for code generation. But we're going to go places that the IDE doesn't take you.

Creating a Window

The creation of a window requires the use of a new data type, the structure. For Windows applications we need to create the window class and tell Windows all about it. This is called registering the class, and it's done by the Windows function called RegisterClass. Note that we are dealing with a Windows class here, not a CA-Visual Objects class.

Before we call this class we should assign certain characteristics, called properties, to the window.

We only need concern ourselves with just a few of these properties for today. We shall presume we have a structure for the storage of the class data called wc. This will have been created through the declaration

```
LOCAL wc Is _WINWNDCLASS
```

We can then assign some properties to this class:

```
wc.style  := _Or(CS_VREDRAW, CS_HREDRAW )

wc.lpfnWndProc  := @MainWndProc()
wc.cbClsExtra   := 0
wc.cbWndExtra   := 0
wc.hInstance    := _GetInst()

wc.hIcon        := LoadIcon( 0, _MakePtr(0, 32512))

wc.hCursor      := LoadCursor( 0, _MakePtr(0, 32512))

wc.hbrBackground := GetStockObject( WHITE_BRUSH )

wc.lpszMenuName := NULL

wc.lpszClassName := Psz( "MouseWinClass" )
```

The first property that we shall look at is the style property. We might wish to make this window a child window, or perhaps a button, for example. This is all controlled by our assignment of the style properties. For now it will be a simple window that has sides that may be resized, and the window will be repainted whenever either the vertical or horizontal borders are altered.

A most important property is the WndProc member of the structure. This defines the callback function that will be used for this window. In other words, this is the function in our application that will be called by Windows whenever it determines that our application needs to do something.

We shall load the default icon and cursor provided by Windows and paint a white background on our window. At this time we can do without a menu, but we must assign a name to the class.

Finally, we register the class with Windows.

```
If !RegisterClass( @wc ) < 1
   RETURN FALSE
EndIf
```

Having done all of that, what do have to show for our work? Nothing. Not until we instruct Windows to actually create the window. Until we do this, it doesn't even know where to start to draw the window.

```
hwnd := CreateWindow(;
  psz( "MouseWinClass" ), ;
  psz( "Current Mouse Window"),;
  WS_OVERLAPPEDWINDOW,;
  100,;
  100,;
  500,;
  250,;
  0,;
  0,;
  _GetInst(),;
  _MAKEPTR(0,0))

IF hwnd = 0
   RETURN FALSE
EndIf
```

We are now telling Windows to create an overlapped window of the class "MouseWinClass." Our window will also have a caption bar and a normal system menu box to the left of that caption box, with the caption "Current Mouse Window" placed into the caption bar.

We are then saying that the window should be placed at the y and x coordinates (left and top corners) of 100 and 100 pixels relative to its parent window, and it will initially be 500 pixels wide and 250 deep.

Next, we give Windows the command to display the window, using the handle to the window returned upon creation of it, together with an instruction to Windows indicating how we want the window to be initially displayed. We might have chosen to run the app as an icon or perhaps maximized, and we would specify this at this point. In this case we're saying we want it displayed as a normal window. and then we say we wish to update it, which essentially means that we want it painted.

```
   ShowWindow(hwnd, SW_SHOWNORMAL)
   UpdateWindow(hwnd)
```

Although this is not strictly associated with painting our window, a most important part of Windows programming is the message loop.

If you've ever written code that just sits in a loop waiting for something to happen, such as perhaps a keystroke, then you'll be somewhat familiar with what happens here.

With Windows programming, once everything is stabilized control passes through to the message loop, where the program then sits waiting for something to happen. Typically, that something will be the receipt of a message from Windows to go and do something.

When a message is received we need to translate the raw keystrokes into those which are usable by the application, and then we direct Windows to pass the message to the window procedure for processing.

```
While GetMessage( @msg, 0, 0, 0 )
   TranslateMessage( @msg )
   DispatchMessage( @msg )
EndDo
```

When a quit message is received from Windows, we exit the message loop and remove the class before shutting down the program.

```
  UnregisterClass( ;
    "MouseWinClass", _GetInst())

RETURN
```

What to Do Next

When a DispatchMessage() call is processed, Windows then relies upon our callback function to assume the processing role.

We are passed the window handle, the message, and the additional parameters wParam and lParam. These latter two will provide extra detail regarding the message received in certain cases.

```
FUNCTION MainWndProc ( ;
   hwnd As Word, ;
   Message As Word, ;
   wParam As Word, ;
   lParam As Long ) As Long _WinCall

   LOCAL ps Is _WINPAINTSTRUCT , ;
              hdc As Word

   Do Case

      Case message = WM_CREATE
         hEraseBrush := GetStockObject(WHITE_BRUSH)
         RETURN 0L

      Case message = WM_SIZE
         GetClientRect(hwnd, @rc)
         RETURN 0L

      Case message = WM_DESTROY
         DeleteObject(hEraseBrush)
         PostQuitMessage(0)
         RETURN 0L

   Case message = WM_MOUSEMOVE
         hdc := GetDC(hwnd)
         p.x := WORD(_CAST, lParam)
         p.y := lParam >> 16
         MyPainter( hdc )
         ReleaseDC(hwnd, hdc)
         RETURN 0L
```

```
   Case message = WM_PAINT
         hdc := BeginPaint(hwnd, @ps)
         MyPainter(hdc)
         EndPaint(hwnd, @ps)
         RETURN 0L

   EndCase

RETURN DefWindowProc( ;
   hwnd, message, wParam, lParam)
```

Here it should be obvious that we're simply determining what the content of the message received is, and we're then branching according to that content.

The functions GetStockObject(), GetClientRect(), PostQuitMessage(), BeginPaint(), EndPaint(), GetDC(), ReleaseDC(), and many more are all just standard Windows calls. It is not my job to document those here, but you need to be aware that we can and will call these, and many others, directly and frequently.

Consequently we need to become fully conversant with many of these functions and the calling conventions and programming constraints involved in their use. As much of this is based upon the C language, there are a great many traps for the unwary to fall into.

Suffice to say that, for this particular application, we are trying to display the current position of the mouse cursor while it is within our window. The way that I have done this is to look for any mouse movement messages (WM_MOUSEMOVE) that we receive, and then assign the current mouse position (which is received in lParam for this type of message) to the structure that we have created for storage of this data.

Where Am I?

With the example that I've given so far, the only thing now missing is the code that actually displays the location of the mouse cursor.

This is easily handled within our painter function, which may be called directly whenever the mouse is moved, or whenever a paint message is received.

```
FUNCTION MyPainter(hdc As Word) ;
    as void pascal

FillRect(hdc, @rc, hEraseBrush)
TextOut(hdc,  10, 10, ;
  "Column  : " + Str(p.x, 4), 14)

TextOut(hdc,  10, 30, ;
  "Row       : "  + Str(p.y, 4) ,16)

RETURN
```

Again, we're calling Windows functions rather than CA-Visual Objects functions, but the syntax is very familiar and friendly.

In this case, we're simply repainting the window background, and then displaying the column and row data within the window.

Icon Do It

Of course, much of the appeal to our users in the use of Windows applications is the perceived ease of use of these applications and the Windows environment.

Part of this stems from the abilities provided by the graphical interface that Windows presents to them. Included with this graphical interface is the ability for us to customize the appearance of the application through the use of various graphical devices. These include the use of the mouse as a pointing device and the ability to then simply point at a task and select that task for processing.

Closely coupled with this is the use of icons to depict certain tasks. We can create an icon that represents a particular task, and then, recognizing that icon as the picture that loads a particular task, we can point at the icon to select that task.

We can go further too. For a particular part of a task we may need (or would like) to implement a special cursor. Look at the Windows Desktop for a moment: as your cursor passes over a resizable window border the cursor's appearance changes, indicating the change in the facilities available.

Sometimes we might like to, or perhaps have a need to, embed a picture into our screen as part of our data. Consider for a moment a personnel system where we wish to retain a photograph of each employee as a part of the individual record and display this photo whenever the record is displayed on the screen.

Support for all of these services is again provided as a standard part of the Windows operating system. Let us now examine how we might wish to perform each of these tasks. None of them is very difficult; let's look at them all together.

To start with, we'll declare the resources that we wish to utilize. We do this by using the resource statement and assigning a name to the resource. We describe the type of the resource being used, and finally we specify the file which contains the actual resource. Each of these becomes a unique entity within CA-Visual Objects.

```
Resource MyCursor  Cursor c:\Icons\MyCursor.Cur

Resource MyIcon  Icon c:\Icons\MyIcon.Ico

Resource MyBitMap BitMap c:\BitMaps\MyBitMap.Bmp
```

We also need to provide handles for these resources, so that after they're installed we can refer to them. As we're going to install the new cursor and icon for the application, we can use the handles provided in the window structure that we use for registering the window class, but we need a separate one for the bitmap.

```
Local wc  Is _WNDWNDCLASS
Local hMyBitMap  As PTR
```

The next thing that we need to do is to actually install those resources into our application.

```
wc.hIcon  := Loadicon( ;
   _GetInst(), "MyIcon" )

wc.hCursor := LoadCursor( ;
   _GetInst(), "MyCursor" )

   ...
   ...

hMyBitMap  := LoadBitMap( _GetInst(), "MyBitMap" )
```

As far as the icon and cursor are concerned, that's all there is to it. Our application now has its own custom icon and cursor; these will be displayed instead of the default Windows icon and cursor.

With a bitmap there's still just a little more work to be done. With a cursor or icon we usually want them to be applicable throughout the whole application or window. A bitmap's life might be more restricted however. We might wish to display one only at certain times within the application, such as only for a certain data record, or perhaps only for a certain display window. Consequently our coding needs to be more selective.

```
LOCAL hdcBmp   As PTR , ;
      hdc      As PTR , ;
      hBmp     As PTR

...
hdc    := GetDc( hwnd )
hdcBmp := CreateCompatibleDc( hdc )
hBmp   := SelectObject( hdcBmp, hSomeBitMap )

SelectObject( hdc, hBmp )
SetMapMode( hdcBmp, ;
   GetMapMode( hdc ) )

BitBlt( hdc, 85, 10, 15, 8, hdcBmp, 0, 0, SRCCOPY )

DeleteDc( hdcBmp )

ReleaseDc( hwnd, hdc )

RETURN
```

Menus Again

I discussed earlier one method of providing a menu for your users. This used the CA-Visual Objects Menu Editor.

Let us now quickly revisit menus for a moment, and see how easy it is to create a Windows style menu using the resources that are available under Windows.

Let's start with the resource assignments: the Begin keyword identifies the start of a new set of menu labels; the Popup keyword identifies a sub-menu item, and the MenuItem keyword identifies, strangely enough, a menu item.

The ampersand sign "&" is used to indicate a hot-key assignment.

```
Resource Music MENU

Begin
   PopUp "&Music Tutor"
   Begin
      Popup "&Instrument"
      Begin
         MenuItem "&Guitar", IDM_GUITAR
         MenuItem "&Keyboard", IDM_KEYS
      End

      MenuItem SEPARATOR
      MenuItem "E&xit", IDM_EXIT
   End

   Popup "&Options"
   Begin
      MenuItem "&Sound", IDM_SOUND
      MenuItem "Show &Notes", IDM_NOTES
   End

   Popup "&Help"
   Begin
      MenuItem "&Help", IDM_HELP
      MenuItem SEPARATOR
      MenuItem "&About", IDM_ABOUT
   End

End
```

As you can see, there's nothing very unusual about what's happening here. The only thing to note is the specification of the key values to be returned upon the selection of a menu item. Here we're using constant values. These would have been defined to the application like this:

```
Define IDM_SOUND    := 7501
Define IDM_GUITAR   := 7502
Define IDM_KEYS     := 7503
Define IDM_NOTES    := 7504
Define IDM_INSTXIT  := 7505
Define IDM_NEXTNOTE := 7506
Define IDM_EXIT     :=   27
Define IDM_HELP     :=   99
Define IDM_ABOUT    :=   98
```

All that's left to do is to provide methods or functions to be executed upon the selection of a menu item. These will be handled within our callback procedure.

You will recall that this process is used to handle the message processing for our application. When the message received from Windows indicates that a menu item has been selected (WM_COMMAND), the value of wParam is set to the value that is associated with that menu item. Hence we can then use the following code to branch our application based upon the menu selection made.

```
...    // Other messages
...

Case message = WM_COMMAND

   Do Case

      Case wParam = IDM_GUITAR
         ...    // Some method or
                // function

         Case wParam = IDM_KEYS
         ...    // Some method or
                // function

         Case wParam = IDM_SOUND
         ...    // Some method or
                // function

         Case wParam = IDM_NOTES
         ...    // Some method or
                // function


   EndCase

RETURN 0L


   ...    // Other messages
   ...
```

We can now process anything that we wish conveniently and easily from our application's main menu.

Conclusion

Where Do I Go From Here

Firstly, I need to point out to you the richness of the CA-Visual Objects language. You could describe it as Clipper for Windows, and you'd be giving a fairly accurate description of just one very small facet of its capabilities. The language is just so feature rich that in a short session here I simply cannot do the language justice.

We need to understand what all the various data types, such as a pointer to a zero terminated string, are, and the differences between them and the more traditional data types that we may be used to. We also need to learn, if we're not yet there, about all of the facets of object-oriented development, and how the principles and techniques can best be utilized to help us in our day-to-day programming tasks.

I have deliberately avoided explaining such things as handles, device contexts, etc. We will need to learn and understand Windows memory-management techniques, DLLs, SQL, and much more. These require a complete session or more on their own, and should be studied long and hard to be understood; this is one area where much of the power and flexibility will be coming from.

The Clipper programmers amongst us are used to having a great deal of power, flexibility, and freedom in our programming tasks. Many of us believed that there was very little that we were not able to do with Clipper. In the world of CA-Visual Objects we will find that the power, the flexibility, the freedom that we've been used to for so long has just been given a massive boost. Add to that the benefits of the Jasmine engine, and you have a truly remarkable development platform.

Read, study, and play, and then read, study and play some more. Find some good Windows reference materials and learn and understand these. Become comfortable within the Windows environment and its benefits and limitations.

Have a good look at the sample code provided here this week. The Music Tutor application that I have provided has very little to do with data processing, but has everything to do with programming in Windows using CA-Visual Objects. It is a fairly simple application, but within it we are utilizing many of the fundamental concepts of programming with Windows: mouse control, windows, resources, menus, bitmaps, cursors, icons, sound, etc.

Use the techniques illustrated here to add to your skills, and make your programs more responsive to your users' needs.

The Sky Is the Limit

The richness of the language will ensure that we will be kept busy studying new techniques, and learning new tricks, for many years to come.

With the power and freedom to do what you wish comes the responsibility to ensure that you use this power carefully and wisely, as there are a great many traps for the unwary.

Go forward into the world of CA-Visual Objects, confident in the knowledge that the capabilities that we've become accustomed to within Clipper have been extended and expanded far beyond our imaginations.

Gary Stark is a qualified accountant from Australia who moved into the world of Data Processing in the early 1980s. He has been using PCs for most of that time, and xBase since 1984, CA-Clipper since 1985. He has been consulting in the PC world since 1986, with projects completed for major Australian insurance companies and banks as well as for government and small business, and has also conducted training in Australia for dBase III, dBase IV and Clipper.

In the US he has worked as a senior CA-Clipper analyst/programmer for the Gallo Winery, and for Dallas, Texas based Rent Roll Inc, a manufacturer of vertical market software for the real estate industry. He is currently based in Sydney, Australia acting as an independent software developer and consultant.

He is a co-author of the SAMS publication CA-Visual Objects Developer's Guide, and has spoken at Technicons and DevCons in New Orleans, Birmingham U.K., Cologne Germany and San Diego, as well as to users groups in the USA, Europe and Australia. He has had articles published in Clipper Advisor and VO Developer Magazines, as well as numerous user group publications.

He can be contacted on the Internet at gstark@RedbacksWeb.com. His home page URL is located at http://www.RedbacksWeb.com.