Chapter 16
Using Windows API Functions

Objective

In this lesson, you will learn how to call Windows API functions directly from CA-Visual Objects. It will demonstrate how to expand the functionality of your applications, as well as illustrate how Windows programs work on a low level.

When you have completed this lesson, you will be able to use the Windows API functions in your CA-Visual Objects applications to provide functionality not found in the native CA-Visual Objects language classes. You will also have a better understanding of how Windows programs work.

Overview

What Is the Windows API?

Windows provides more than just the graphical shell in which our applications run. It also consists of over 600 built-in functions available to any Windows program at runtime. These functions range from creating and manipulating windows and menus, to simple network functions. The Windows API also provides predefined data structures and message definitions that your programs can use.

CA-Visual Objects provides the capability to use as little or as much of the Windows API as we wish. Indeed, we can write entire CA-Visual Objects programs using only Windows API calls. However, we would lose the ability to use many of the object classes provided. If a function is provided by both CA-Visual Objects and the Windows API, it is preferable to use the CA-Visual Objects function.

Calling Windows API Functions

Calling Windows API functions is as simple as calling any other function. The only stipulation is that the function's prototype must be provided. The Windows API library does precisely this, as well as defining Windows data structures and messages. For example, this is the prototype for the MessageBox() function, which provides a simple modal dialog box:

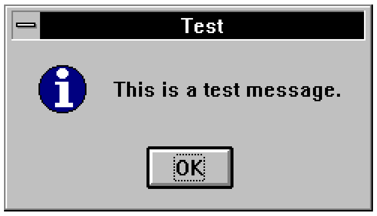
```
_DLL FUNC MessageBox(hParent AS WORD, ;
      pszText AS PSZ, ;
      pszCapt AS PSZ, ;
      siFlags AS SHORTINT) AS SHORTINT
PASCAL:USER.1
```

The _DLL FUNC statement indicates that the MessageBox() function resides in an external DLL. The parameters for MessageBox() are the parent window's handle (a unique identifier within Windows), the message text and caption as zero-terminated strings, and the flags indicating the type of message box desired. The function returns a value which is defined as a short integer, in this case a number corresponding to the user's selection from the message box. The function follows the Pascal calling convention and resides in the USER library. In this particular case, USER does not refer to a DLL but to USER.EXE.

Calling this function is as simple as:

```
MessageBox(SELF:Handle(), ;
   String2Psz("This is a test message."), ;
   String2Psz("Test"), _OR( MB_ICONINFORMATION, ;
   MB_OK))
```

This function would display the following dialog box:


Test

This is a test message.

The following table is a list of functions that will be examined in this lesson:

Function Name	**Purpose**
DispatchMessage()	Dispatches a message to the appropriate window within an application
GetFreeSpace()	Retrieves the amount of global memory available (i.e., physical memory and virtual memory)
GetFreeSystemResources()	Retrieves the free system resource space as a percentage
GetMessage()	Waits for a message to enter the queue until a WM_QUIT message is encountered
GetSystemMetrics()	Retrieves various system settings
GetWindowsDirectory()	Retrieves the path, including the drive, in which Windows is installed
GetProfileInt()	Retrieves the value of an integer from within the specified section of the Windows initialization file, WIN.INI
GetProfileString()	Retrieves a string from within the specified section of WIN.INI
GetPrivateProfileInt()	Retrieves the value of an integer from within the specified section of the specified initialization (.INI) file

Continued

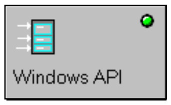
Continued

Function Name	**Purpose**
GetPrivateProfileString()	Retrieves a string from within the specified section of the specified initialization file
PeekMessage()	Checks for a message in the queue, allowing Windows to process messages for other applications if none are present for this application
TranslateMessage()	Translates a message into a character message and posts the message to the application's message queue
WritePrivateProfileString()	Copies a string into the specified section of the specified initialization file
WriteProfileString()	Copies a string into the specified section of WIN.INI

Exercise

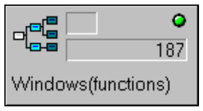
Let's begin this lesson by examining some of the Windows API functions:

1. Open the Windows API library by double-clicking on its button in the Application Browser:

   The module browser appears.

2. Open the Windows (functions) module by double-clicking on its module button:

   The list of Windows API functions appears in an Entity Browser.

3. Scroll down the list of entities until the GetFreeSpace() function is highlighted.

4. Open GetFreeSpace() by double-clicking on this entity.

   Its prototype is displayed in the Source Code Editor:

   ```
   _DLL FUNC GetFreeSpace(x AS WORD) AS DWORD
      PASCAL:KERNEL.169
   ```

   This shows that GetFreeSpace() is from an external DLL, has a parameter which is of the word data type, returns a double word value, follows the Pascal calling convention, and resides in the KERNEL library. In this particular case, KERNEL does not refer to a DLL, but to KRNLx.EXE, where x is either 286 or 386. This is one of the resident programs loaded when Windows is started. The final part of the prototype (.169) identifies the number of the function within the library.

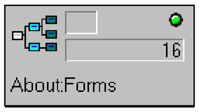
5. Close the Source Code Editor, the Window (functions) Entity Browser, and the Modules of Windows API Module Browser.

Windows Memory Information

To begin, let’s examine the use of GetFreeSpace() and GetFreeSystemResources() in the South Seas Adventures application. As you will see, this is a very simple process.

Most Windows applications have a dialog called the About box, which shows information such as the system’s revision level and perhaps displays credits for the development team. You may also have noticed that some applications display Windows memory information. The About dialog box in the South Seas Adventures application does precisely this by calling GetFreeSpace() and GetFreeSystemResources().

1. Open the South Seas Adventures application by double-clicking on its button in the Application Browser.
2. Open the About:Forms module by double-clicking on its button in the Module Browser:

   The base class for the About box is called _AboutDialog. However, we have subclassed _AboutDialog into AboutDialog so that we can add the resource and memory information while maintaining the capability to modify the _AboutDialog using the Window Editor.
3. Scroll down the list of entities until AboutDialog:Init() is highlighted.
4. Open the AboutDialog:Init() method by double-clicking on it.
5. Note the code assigning the caption values to *oDCFreeMemoryText* and *oDCFreeResourcesText*:

   ```
   SELF:oDCFreeResourcesText:Caption:=FreeResources(
   )
   SELF:oDCFreeMemoryText:Caption:=FreeMemory()
   ```

FreeMemory() and FreeResources() are defined by the South Seas Adventures application in the App:Misc module. If you like, access the Entity Browser for this module and load the source code for each of these functions into the Source Code Editor. You will see that FreeMemory() consists solely of a call to GetFreeSpace(), and the appropriate formatting:

```
FUNCTION FreeMemory() AS STRING

   LOCAL cRetString AS STRING,;
         dwFreeMem AS DWORD

   dwFreeMem := GetFreeSpace(0) / 1024

   cRetString := AllTrim(Str(dwFreeMem)) + ;
         " Kb Free"

   RETURN cRetString
```

Similarly, FreeResources() formats the return value from GetFreeSystemResources():

```
FUNCTION FreeResources() AS STRING

   LOCAL cRetString AS STRING,      ;
         wResources AS WORD

   wResources := GetFreeSystemResources(0)

   cRetString := AllTrim(Str(wResources)) +;
         "% Free"

   RETURN cRetString
```

Note: The FreeMemory() and FreeResources() functions that are being used to call underlying Windows API functions are referred to as *wrapper functions*. This is a common programming technique that allows you to perform the conversions between the CA-Visual Objects and Windows data types in one location, freeing you from performing these conversions repeatedly in your code.

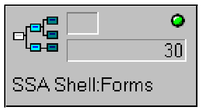
6. Close all open Source Code Editors and Entity Browsers by double-clicking on the appropriate system menus.

These two examples have allowed you to view information about the current Windows environment. Now you will examine how to use Windows API calls to modify the appearance of the South Seas Adventures application.

Windows Metric Information

The GetSystemMetrics() function retrieves information about various system metrics (for example, the heights and widths of various elements displayed by Windows). You will now examine the use of this function to set the size of a window to the full size of the screen without maximizing the window.

1. Open the SSA Shell:Forms module from the Module Browser by double-clicking on its button:



2. Open the SSAWindow:Init() method by double-clicking on its button. Note the function call assigning the return value from the function FullWinSize() to the window's size attribute. This code makes the window occupy the entire screen, without having to be maximized:

```
SELF:Size := FullWinSize()
```

FullWinSize(), defined in the App:Misc module, determines the actual maximum size of the window by calling the Windows function GetSystemMetrics():

```
FUNCTION FullWinSize() AS Dimension

   LOCAL oDim AS Dimension, ;
         nHeight AS SHORTINT, ;
         nWidth AS SHORTINT

   nHeight := GetSystemMetrics(SM_CYSCREEN)
   nWidth  := GetSystemMetrics(SM_CXSCREEN)

   oDim := Dimension{nWidth, nHeight}

   RETURN oDim
```

The parameter passed to GetSystemMetrics() indicates which system value is to be returned. In this case SM_CXSCREEN instructs GetSystemMetrics() to return the width of the screen, and SM_CYSCREEN returns the height of the screen. In FullWinsize(), these two values are combined as a Dimension object, which is used to set the size of a window.

3. Close all open Source Code Editors and Entity Browsers by double-clicking on the appropriate system menus.

Profile Functions

Many Windows applications, including Windows itself, use initialization (.INI) files to contain configuration information. The Windows API provides several functions for using these files. In the South Seas Adventures application, we have defined two classes for manipulating .INI files. These are IniFileSpec and a subclass called WinIniFileSpec. Both of these classes use the Windows profile functions for dealing with .INI files.

IniFileSpec Class

The IniFileSpec class is used for reading and writing any .INI file, and is a subclass of the FileSpec class.

1. Open the Class Browser for the South Seas Adventures application by choosing the Class Browser command from the Tools menu.
2. Highlight the IniFileSpec Class in the left pane of the Class Browser.

3. Examine the IniFileSpec:GetString() method by double-clicking on its button in the right pane of the Class Browser. GetString() is used to retrieve a string value from the specified .INI file:

```
METHOD GetString(sSection, sEntry) ;
   CLASS IniFileSpec

   LOCAL sValue AS STRING
   LOCAL ptrBuffer AS PTR

   ptrBuffer := MemAlloc(INI_STRING_LEN)

   GetPrivateProfileString(Psz(sSection),;
         Psz(sEntry), Psz(""),;
         Psz(_CAST, ptrBuffer),;
         INI_STRING_LEN, Psz(SELF:FullPath))

   sValue := Psz2String(Psz(_CAST, ptrBuffer))

   MemFree(ptrBuffer)

   RETURN Trim(sValue)
```

   The GetString() method is passed two parameters: the name of the section and the name of the specific entry to be retrieved. GetPrivateProfileString() is the Windows API function used to retrieve the string value from an .INI file other than WIN.INI. It expects six parameters:

   - .INI file section name
   - Name of the entry to be retrieved
   - Default value for the string if no entry exists in the file
   - Pointer to the return string
   - Size of the return string
   - Name of the .INI file.

   Note that the strings passed to GetPrivateProfileString() are all converted to PSZ, or zero-terminated strings, which is the standard for Windows.

   GetPrivateProfileString() returns the requested string by assigning it to the pointer passed by *ptrBuffer*. This is the same concept as passing a parameter by reference—you are passing the memory location of the string.

To do so, you must first allocate memory for the return string by using MemAlloc(), which is passed the amount of memory to be allocated. In this case the value is stored in the constant INI_STRING_LEN, which is defined as 1024 bytes.

Note: Any memory allocated using MemAlloc() must be freed by calling the MemFree() function when the space is no longer needed.

Finally, the string retrieved from the .INI file is converted from a pointer to a CA-Visual Objects string, and this value is returned by the method.

4. Close the Source Code Editor by double-clicking on its system menu.

WinIniFileSpec Class

The WinIniFileSpec class differs from the IniFileSpec class in that it is used to read from and write to WIN.INI. The IniFileSpec class is used for manipulating any .INI file.

WinIniFileSpec inherits from the IniFileSpec class and, thus, shares the attributes and methods of IniFileSpec. However, some changes are necessary in order to restrict the .INI file access to WIN.INI, as these classes use different Windows API functions. Where the IniFileSpec class uses “private” profile functions, such as GetPrivateProfileString(), WinIniFileSpec uses functions such as GetProfileString(). These functions assume that the file to be read/written is WIN.INI.

When instantiating the IniFileSpec class, the .INI file name is passed. The Init() method of the IniFileSpec class includes the code that appends the drive and path, if they are not available, and assigns it to SELF:FullPath. This code will be activated, as well as the code in the Init() method of its ancestor class, the FileSpec class. However WinIniFileSpec always uses WIN.INI, so a file name is not required.

There are other differences between the Init() methods for the WinIniFileSpec and IniFileSpec classes which you will now see:

1. Click the Expand button to the left of the IniFileSpec class in the left pane of the Class Browser.

   Its subclass, WinIniFileSpec, appears in a tree-like structure beneath IniFileSpec.

2. Highlight WinIniFileSpec in the left pane, and double-click its Init() method in the right pane to view its code in the Source Code Editor, as shown below:

   ```
   METHOD Init() CLASS WinIniFileSpec

       LOCAL sValue AS STRING
       LOCAL ptrBuffer AS PTR
       LOCAL wLen AS WORD

       ptrBuffer := MemAlloc(WIN_DIR_LEN)

       wLen := GetWindowsDirectory( ;
              Psz(_CAST, ptrBuffer), WIN_DIR_LEN )

       sValue := Psz2String(Psz(_CAST, ptrBuffer))

       MemFree( ptrBuffer )

       SUPER:Init(sValue + "\WIN.INI")

       RETURN SELF
   ```

   In order to instantiate the ancestor, FileSpec, you need the file name. Since you cannot assume that Windows is always installed in C:\WINDOWS, you must determine the path in which it is installed. The GetWindowsDirectory() function does precisely this.

   GetWindowsDirectory() is passed a pointer to the string which will be assigned the full path in which Windows is installed. It is also passed the length of the string.

   With the Windows directory in hand, you can now initialize WinIniFileSpec’s ancestor, FileSpec, with the Windows directory and WIN.INI.

3. Close the Source Code Editor by double-clicking on its system menu.

Note: Several other functions, not discussed in this section, allow you to read integer values from .INI files, as well as write values to .INI files. These are GetPrivateProfileInt(), GetProfileInt(), WritePrivateProfileString(), and WriteProfileString().

Loop Processing

In business applications, often it is required to process data in loops. In DOS applications, there is usually nothing else happening in the system since there is no multitasking, and therefore, this processing is fine. It's a different story in Windows.

Windows is a multitasking operating environment. It is quite conceivable that you will have multiple applications running at the same time. When your program goes into a loop, it essentially freezes all other activity in Windows. That's because of the type of multitasking that Windows implements.

Windows offers a *non-preemptive* type of multitasking, which means that Windows will not interrupt code in progress unless you let it. It will only *task-switch* during calls to the GetMessage(), PeekMessage(), and WaitMessage() functions.

The Message System

Windows maintains a message queue for each running application. When an event occurs (such as keyboard or mouse input), Windows translates it into a message which is placed in the appropriate application's message queue. It is then up to the application to fetch the message and act on it.

In other development environments, a GetMessage() loop is used to fetch and dispatch messages, which would typically be found in the application's WinMain() function. Here is the CA-Visual Objects syntax equivalent of this loop:

```
DO WHILE GetMessage(@msg, hwnd, 0, 0)
   TranslateMessage(@msg)
   DispatchMessage(@msg)
ENDDO
```

The GetMessage() function waits for a message to enter its queue. All messages, except for WM_QUIT, will then be translated and dispatched to the appropriate window procedure within the application.

GetMessage() will return a 0 if it encounters the WM_QUIT message in its queue, which in turn exits the loop.

CA-Visual Objects provides us with a framework where this type of "manual coding" of the message loop is unnecessary, although still available. In making use of the CA-Visual Objects rich GUI Classes library, this GetMessage() loop is replaced by a single method call to class App:

```
METHOD Start() CLASS App

       // Open windows etc...

       SELF:Exec()  // Start receiving events
```

The Exec() method is essentially a GetMessage() loop with another layer of functionality. CA-Visual Objects retrieves the messages, then dispatches them, as Event objects, to the appropriate event handlers in your program. See the "Customizing Window Event Handlers" chapter in this guide for more on event handlers.

Not only does the GetMessage() function allow your programs to receive messages, it is also the key to Windows multitasking. During the GetMessage() call, if no messages are present in the application's queue, Windows switches to another running application that has messages waiting. Once it has serviced those messages, control will come back to your program. This is sort of an optimistic type of multitasking. If any one application in the system does not allow others to receive messages, then that's it—no other application will run during this period.

If, for example, you have the Windows Clock program up and you start into a loop that will take a while to execute, the clock, which usually runs in the background, will stop running until the loop has done and control returns to Windows:

```
DO WHILE (lALongTime)
   // Do some data processing
ENDDO
```

You must, therefore, get a call into the GetMessage() function while in your loop.

The CA-Visual Objects App:Exec() method accepts a parameter, <*kConstant*>, which can be one of the following:

- EXECNORMAL, the default, executes until the application is closed
- EXECWHILEEVENT executes while there are events queued

The EXECNORMAL form of the call is what you would expect to find in the Start() method of your application. This form of the call actually waits for events to occur and exits only once the application is closed. This is not what you would want to use in your loop. Although messages would be processed, your loop would stop at the Exec() call:

```
DO WHILE (lALongTime)
       // Do some data processing

       // Code will stop here...waiting till
       // application closes
       SELF:Exec(EXECNORMAL)
ENDDO
```

Instead you would pass the EXECWHILEEVENT constant to the Exec() method. Each time around the loop, the Exec() method allows Windows to service other message queues:

```
DO WHILE (lALongTime)
       // Do some data processing

       // SELF refers to the current App object
       SELF:Exec(EXECWHILEEVENT)

ENDDO
```

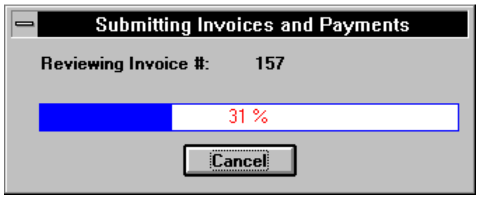
Just as App:Exec(EXECNORMAL) is like the GetMessage() loop, App:Exec(EXECWHILEEVENT) is like the following PeekMessage() loop:

```
DO WHILE PeekMessage(@msg, 0, 0, 0, PM_REMOVE)
       TranslateMessage(@msg)
       DispatchMessage(@msg)
ENDDO
```

PeekMessage() is similar to GetMessage(). It looks into the message queue; and, if no messages are present, Windows will switch to another running application that has messages waiting. Once it has serviced those messages, control will come back to your application; however, instead of waiting for messages like GetMessage(), PeekMessage() returns 0, thus exiting the loop.

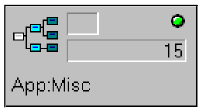
Selecting Which Messages Are Processed

In the South Seas Adventures application, the Submit Invoices and Payments command from the Options menu uses a progress bar to give a visual status as it loops through the invoices and payments in the system:



While the operation is occurring, we want only the progress bar to be updated, while the rest of the application remains still. We also want other applications in the environment to continue running. The App:Exec(EXECWHILEEVENT) method processes messages for *any* window in the application, so we cannot use it in this case and must rely on Windows API functions to accomplish the task.

1. Open the App:Misc module by double-clicking on its button:



2. Find and open the YieldMessageLoop() function by double-clicking on its button:

   ```
   FUNCTION YieldMessageLoop(oWindow)

      LOCAL msg AS _WINMSG
      LOCAL hwnd AS WORD

      IF oWindow == NIL
             // Allows all windows to receive events
             hwnd := 0
      ELSE
             // Allows a specific window to
             // receive events
             hwnd := WORD(oWindow:Handle())
      ENDIF

      DO WHILE (PeekMessage(@msg, hwnd, 0, 0, ;
             PM_REMOVE))

             TranslateMessage(@msg)
             DispatchMessage(@msg)

      ENDDO

      RETURN NIL
   ```

   The YieldMessageLoop() function implements its own PeekMessage() loop. The second parameter to the PeekMessage() function, *hwnd*, allows you to specify the handle of the window for which you want to process messages. YieldMessageLoop() accepts one parameter, a window object, from which we can extract that handle: oWindow:Handle(). When the *hwnd* parameter is 0, behavior is similar to that exhibited by App:Exec(EXECWHILEEVENT) (that is, messages to any window in the current application are processed).

3. Close the Source Code Editor by double-clicking on its system menu. Similarly, close the Entity Browser for App:Misc.

4. To see where this function is used, open the Progress:Forms module by double-clicking on its module button.

5. Select the Edit All Source in Module command from the Edit menu.

6. Find the ProgressDialog:Advance() method. Because this is the method that actually updates the progress bar in the dialog window, it is the ideal place for calling the message loop:

```
METHOD Advance(sUpdateText);
   CLASS ProgressDialog

   // Update the bar properties
   ...

   // Give other processes a shot!
   YieldMessageLoop(SELF)

   RETURN NIL
```

7. This method, which is in the App:Misc module, calls the YieldMessageLoop() function, passing SELF (the ProgressDislog object) as a parameter:

```
FUNCTION YieldMessageLoop(oWindow)
   LOCAL msg IS _WINMSG
   LOCAL hwnd AS WORD

   IF oWindow == NIL
         //Allow all windows to receive events
         hwnd := 0
   ELSE
         //Allow a a specific window to receive
         events
         hwnd := WORD(oWindow:Handle())
   ENDIF

   DO WHILE
(PeekMessage(@msg,hwnd,0,0,PM_REMOVE)
         TRANSLATE Message (@msg)
         DISPATCH Message (@msg)
   ENDDO

RETURN NIL
```

8. This concludes our exercise. Return to the Application Browser by closing the Source Code Editor and all other open browsers.

Summary

In this chapter, you have learned about the Windows API functions and how to use direct calls to the Windows API in your CA-Visual Objects applications.

In the next lesson, you will be able to create a library and a DLL in order to share code among your applications.