CenturaPro™

Visit us at www.ProPublishing.com! Hot Ideas for Centura® Developers

Searching the Shell

Joachim Meyer

Several standard dialogs are exposed by the Windows shell, but Microsoft doesn't share all of its secrets with developers. Some built-in dialogs aren't exposed, and from time to time a secret is revealed. In this article the author sheds light on several of the built-in dialogs, be they documented or not.

From the time Microsoft started making operating systems, each version has had a documented Application Programming Interface or API. The API exposes all the functions necessary to maintain and extend the system and provides hooks for setting and retrieving information. Microsoft also provides documentation for these APIs.

So far, so good. But from the beginning Microsoft has also reserved some functions for internal use and has declined to document them for public consumption. This is usually no problem, nor can it be called unfair behavior. However, from time to time a programmer needs some special functionality that he or she knows is already present in the OS that is neither documented nor exposed.

SQLWindows 32

Generations of programmers have explored these OSs, and sometimes they stumble over little jewels. Some of these are the dialogs that prompt the user before shutting down or restarting Windows. Others include displaying file properties, connecting to network resources, and displaying the find files dialog. I wrote a class to encapsulate their functionality. As a bonus I included a documented dialog to make the class more complete.

The CITFCShell class

My new class is part of the Ice Tea Foundation Classes (ITFC). The purpose and origins of ITFC are covered in detail in my other article in this issue, "Ice Tea Foundation Classes". This article on shell dialogs is the first in a series of articles that will expand, enhance, and explain ITFC.

I have put the undocumented functions into a class called CITFCShell (see Listing 1). I used this name because almost the entire functionality depends on the SHELL32.DLL which comes with Windows. The class is derived from CITFCBase, which is the root in the ITFC class hierarchy. All other classes should be derived from it to inherit the basic capabilities CITFCBase offers: basically, error handling.

Listing 1. The CITFCShell class.

```
◆ Functional Class: CITFCShell
  ◇ Description:
  ◆ Derived From
    ◇ Class: CITFCBase
  ◇ Class Variables
  ◇ Instance Variables
  ◆ Functions
    ◇ ! public
    ◇ Function: AboutDialog
    ◇ Function: DiskFullDialog
    ◇ Function: EmptyRecycleBin
    ◇ Function: EnumSpecialFolders
    ◇ Function: ExitWindowsDialog
    ◇ Function: FindFilesDialog
```

Continues on page 3

November 1999

Volume 4, Number 11

PRO Publishing

Something for Nothing, Something for Sale

Mark Hunter

It's a crowded issue this month, so I must be brief. Lots of interesting things going on, though. Remember the Complementary Software Partner Program described in my September editorial? Its new manager is . . . David Burke! Great choice—someone who has "been there and done that" in the developer world. Congratulations, David!

Software vendors have been busy, too. In November, Ice Tea Group will reveal an e-commerce "shopping cart" application on its own web site, www.iceteagroup.com, for easier ordering of ITG products. They also intend to sell the shopping cart application itself for reuse by their customers. It's based on Centura Team Developer and Ice Tea Active Pages. They are also starting a fascinating new service, enabling strong encryption of e-mail messages between any two addresses, using any e-mail service, even when no encryption software is installed on either of the two machines. And it's free! Take a look; you'll be intrigued.

CAM Data Systems (www.camdata.com) also sells point-of-sale and e-commerce systems based on SQLBase, net.db, and other Centura products. In fact, their low-end point-of-sale and inventory control package, Retail ICE, is absolutely free for the download. CAM Data expects to capture sales from these entry-level customers as they graduate from the single-user, single-store system to multi-user systems.

Much of the issue this month is devoted to Ice Tea Foundation Classes, a group of classes that will be free for downloading. The technical articles deal so much with the "what" and "how" of ITFC, there wasn't room to explain the "why". But look for that next month. Centura Pro has published hundreds of cool source code files over the years. ITFC is a step toward organizing those cool ideas into a more powerful and coherent collection. **CP**

Editor Mark Hansel Hunter, **Publisher** Dian Thumbelina Schaffhauser, **Business Manager** Shelley Rapunzel Doyle, **Production Editor** Paul Horner Gould, **Hickory Dickory Dog** Mocha

Centura Pro (ISSN: 1093-2100) is published monthly (12 times per year) by Pro Publishing, PO Box 2399, Nevada City, CA 95959.

POSTMASTER: Send address changes to *Centura Pro*, PO Box 2399, Nevada City, CA 95959.

Subscription information: To order, call Pro Publishing at 530-265-4082. Cost of domestic subscriptions: 12 issues, $119; Canada: 12 issues, $129. Other countries: 12 issues, $139. Ask about source code disk pricing. Individual issues cost $15. All funds must be in U.S. currency.

Call Centura Software Corp. at 650-596-3400.

If you have questions, ideas for bribing authors, or would just love to chat about what you're doing with Centura products, contact us via one of the means at right.

Contact Us

Centura Pro on the Web
http://www.ProPublishing.com

Editorial Department
Phone: 818-249-1364
Fax: 818-246-0487
E-mail: mhunter@sprintmail.com

Subscription Services
Phone: 530-265-4082
Fax: 530-265-0368
E-mail: shelley@propublishing.com

Mail
Pro Publishing
PO Box 2399
Nevada City, CA 95959

Searching the Shell …

Continued from page 1

```
◇ Function: GetSpecialFolderPath
◇ Function: NetConnectionDialog
◇ Function: ObjectPropertiesDialog
◇ Function: OutOfMemoryDialog
◇ Function: PathToPIDL
◇ Function: PIDLFree
◇ Function: PIDLToPath
◇ Function: RestartDialog
◇ ! private
◇ Function: __Error
◇ ! virtual
◇ Function: CITFCShell_Error
```

Virtual interface

The class provides one virtual function called CITFCShell_Error( ). The function simply sets the error code and error class and returns one of the values ITE_Ok or ITE_Error to the caller.

Why make it virtual? The reason is to provide a hook for a derived class where it can implement error handling. Because all error handling within CITFCShell is done by a call to Error( ), which in turn calls CITFCShell_Error( ) in a late-bound fashion, the developer can override the function and implement his or her own error handling. This overridden function will then be called instead of the built-in error handling. (For more detail on error handling in ITFC, refer to my other article in this issue.)

Item ID Lists

I added a few helper functions that deal with the so-called Item ID Lists (IDL). These are frequently used in the SHELL32 DLL as a substitute for pathnames. So I had to include functions for creating, converting, and releasing IDL pointers. The functions are:

- *PathToPIDL.* Converts a UNC pathname to an IDL pointer.
- *PIDLFree.* Frees the memory previously allocated by PathToPIDL or other shell APIs.
- *PIDLToPath.* Converts an IDL pointer to a UNC pathname.

You don't have to understand what IDLs really are and how they work. All the IDL handling is done internally inside the class functions so you don't have to deal with it. Nevertheless, I declared the three functions in the public section; if you ever use a SHELL32 API that isn't included here and that incorporates IDL pointers, then you have the handling routines ready for use.

About Windows Dialog

The first dialog to be described is the About Windows dialog (see **Listing 2**). This one is documented on Microsoft's developer knowledgebase, MSDN, so it doesn't involve much magic.

The function to display this dialog is customizable to a limited extent, which is exposed by the list of parameters (see Listing 2).

Listing 2. The About Dialog.

```
◆ Function: AboutDialog
  ◇ Returns
  ◆ Parameters
    ◇ Window Handle: p_hWndParent
    ◇ String: p_sCaption
    ◇ String: p_sInfo
    ◇ String: p_sText
    ◇ Number: p_hIcon
```

The first parameter is similar in all dialog functions. It expects a window handle to be the owner (or parent) of the dialog. Usually you pass hWndForm here, but you can also pass hWndNULL.

The second parameter specifies the text to be displayed on the title bar of the dialog.

The third parameter specifies the text to be displayed right beside the Microsoft company name. If you specify only one text and set the other to an empty string (or STRING_Null), the same text is displayed on the title bar and next to the Microsoft company name. The fourth parameter is optional. The text you pass here is displayed as a custom text below the copyright message. I found that this text can display two lines at most. Line breaks will be displayed accordingly.

The last parameter is a handle to an optional icon. If you specify an icon handle, the built-in Windows logo is replaced by the icon. But the question is, how will a 32x32 icon look when displayed here? I think it will be too ugly.

The function (as well as the API) doesn't have a return value.

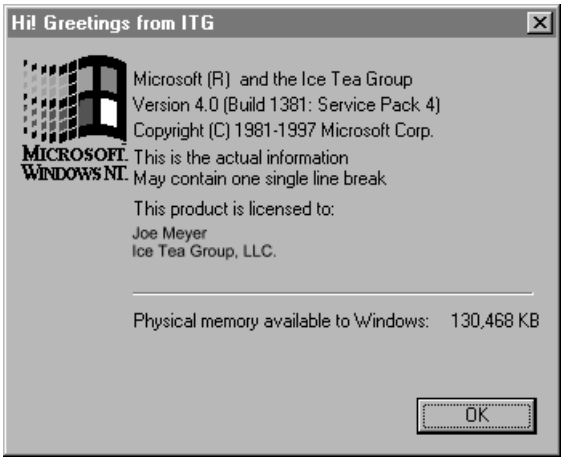

Figure 1. The About Windows dialog.

Disk Full Dialog

The Disk Full dialog (see **Figure 2**) is undocumented but not undiscovered.

Of course, this dialog can't handle a full disk, but it can at least do something that helps a bit. The dialog gives the user the ability to open or empty the Recycle Bin. Unfortunately, this doesn't always apply, because the dialog follows a strange logic (some say that Microsoft does as well). If the specified drive lacks a Recycle Bin or the Recycle Bin is empty, this function does absolutely nothing. The dialog doesn't even appear. To be honest, it's not absolutely clear to me how to use this API. Using it as a standard error box doesn't make sense because in some cases it might not appear. Additionally, there's no return value from the API so the application has no information about what the user has done.

Listing 3. The Disk Full dialog.

```
◆ Function: DiskFullDialog
  ◇ Returns
  ◆ Parameters
    ◇ Window Handle: p_hWndParent
    ◇ Number: p_nDriveNumber
```

As with the other functions, the first parameter expects a window handle. Pass hWndForm or hWndNULL. The second parameter is used to determine which drive is actually affected. The information is passed in numeric format with 0 meaning drive A:, 1 meaning drive B:, and so on.

Empty Recycle Bin

The documented Empty Recycle Bin dialog prompts the user and empties the Recycle Bin. The function for this dialog (see **Listing 4**) is included in the CITFCShell class.

Listing 4. The Empty Recycle Bin dialog.

```
◆ Function: EmptyRecycleBin
  ◆ Returns
    ◇ Boolean:
  ◆ Parameters
    ◇ Window Handle: p_hWndOwner
    ◇ String: p_sRootPath
    ◇ Number: p_nFlags
```

The function takes three parameters. The first one is the window handle of the owner. Pass hWndForm or hWndNULL here.

The second parameter specifies the root path for the Recycle Bin. This is usually the C: drive, and you pass "C:\\" here. If you configured the Recycle Bin to be separate for each drive, it could also be "D:\\". Passing STRING_Null in this parameter causes the shell to empty all available Recycle Bins.

The third parameter expects a flag mask, for which I have defined the necessary constants:

- *ITC_SHERB_NOCONFIRMATION*. Don't prompt the user.
- *ITC_SHERB_NOPROGRESSUI*. Don't display the progress dialog.
- *ITC_SHERB_NOSOUND*. Don't play a sound.

Exit Windows Dialog

After clicking the Start button on the task bar and selecting the menu item for exiting Windows, the desktop turns gray and a small dialog pops up (see **Figure 3**).

In this dialog you find the options for shutting down the system, logoff or restarting. As far as I know, Microsoft doesn't provide an API to invoke this dialog directly, and the declaration of this function isn't documented. Nevertheless, it exists, and here's the function declaration:

Listing 5. The Exit Windows dialog.

```
◆ Function: ExitWindowsDialog
  ◆ Returns
  ◆ Parameters
    ◆ Window Handle: HWND
```

Just pass the window handle of the calling application to the function and it works. You may pass hWndForm or hWndNULL.

The function doesn't have a return value. This wouldn't make sense; what code could possibly be executed after Windows has been shut down?

Figure 2. The Disk Full dialog.

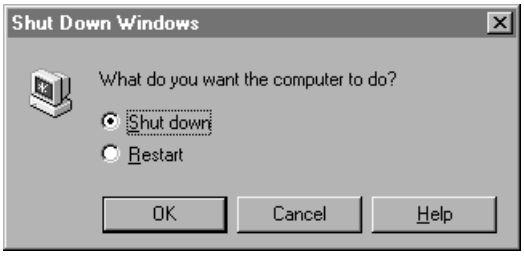

Figure 3. The Exit Windows dialog.

Find Files Dialog

When you right-click on a directory in the Windows explorer and select "Find...", an application starts (see **Figure 4**) that lets you define search criteria and find matching files. As you might expect, this is another built-in dialog.

The function to invoke the undocumented API is very easy (see **Listing 6**). It expects just a single parameter to define the location from which to start the search.

Listing 6. The Find Files Dialog.

```
◆ Function: FindFilesDialog
  ◆ Returns
    ◇ Boolean:
  ◆ Parameters
    ◇ String: p_sSearchRoot
```

The function doesn't have a return value; as soon as the dialog appears, the function returns control to the calling application. The Find Files dialog itself is started in a separate process, which you can see on the taskbar.

Net Connection Dialog

Windows Explorer has a menu item (and, if configured so, a button on the toolbar) that lets the user associate drive letters with network resources. When selected, a standard dialog (see **Figure 5**) appears that can have one of two different styles depending on whether the network resource should be predefined or variable.

Listing 7 shows the function to display this dialog.

Listing 7. The Net Connection Dialog.

```
◆ Function: NetConnectionDialog
  ◆ Returns
    ◇ Boolean:
  ◆ Parameters
    ◇ Window Handle: p_hWndOwner
    ◇ String: p_sResourceName
```

Again, it expects the window handle of the owner, which can be hWndForm or hWndNULL.

The second parameter decides if the user will be able to browse the network and select any available network resource. If you specify a valid network resource (UNC path) the user can only select the drive letter. In this case the Network Neighborhood isn't displayed and the resource can't be changed.

If you pass STRING_Null, the user is free to browse and select any available network resource.

Object Properties Dialog

Another feature of Windows Explorer is the dialog that displays information for files and directories (see **Figure 6** on page 6). The number of tabs depends on whether the associated application has installed any shell extensions. Microsoft Office, for example, installs this type of extension. Right-click on any Word document, select the Properties menu item, and you can see several items of information specific to Word documents.

The function that displays this dialog is called ObjectPropertiesDialog (see **Listing 8**). It takes four parameters from which the first is our obligatory window handle of the owner. Pass hWndForm or hWndNULL here.

The second parameter specifies whether the object is a printer name or pathname (see **Listing 9** for the declared constants).

The third parameter expects the complete UNC path to the file whose properties are to be displayed. If you pass a constant or literal for this parameter, don't forget to use double backslashes. SQLWindows interprets a backslash as a special character; if you want the string to contain the backslash character, you must precede it with another backslash.

The fourth and last parameter specifies what tab of the properties dialog is to be displayed immediately after the dialog appeared. If this parameter contains an invalid tab name or is empty, the first tab is displayed.

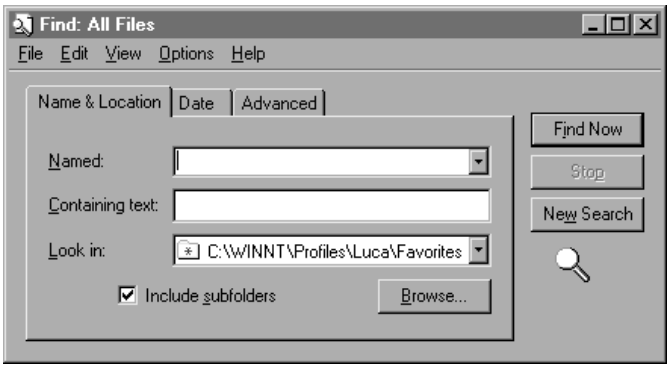

Figure 4. The Find Files Dialog.

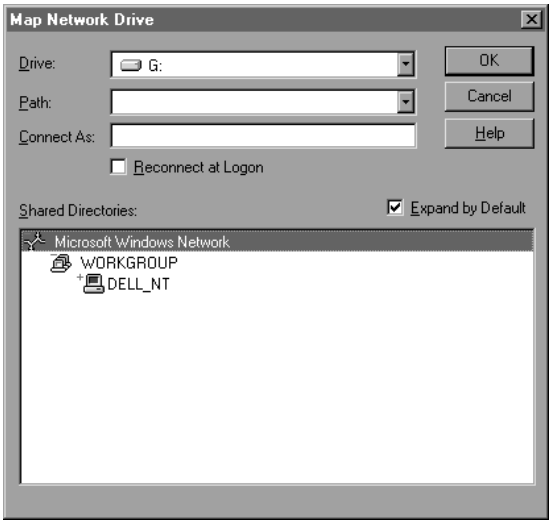

Figure 5. The Net Connection Dialog.

Listing 8. The Object Properties Dialog.

```
◆ Function: ObjectPropertiesDialog
  ◆ Returns
    ◇ Boolean:
  ◆ Parameters
    ◇ Window Handle: p_hWndOwner
    ◇ Number: p_nFlags
    ◇ String: p_sObject
    ◇ String: p_sInitialTabName
```

Listing 9. The type of object constants.

```
Number: ITC_OPF_PRINTERNAME = 1
Number: ITC_OPF_PATHNAME = 2
```

Out of Memory Dialog

The Out of Memory dialog is undocumented as well. The API uses the Windows message box with a localized error message.

Listing 10. The Out of Memory Dialog.

```
◆ Function: OutOfMemoryDialog
  ◆ Returns
    ◇ Number:
  ◆ Parameters
    ◇ Window Handle: p_hWndOwner
    ◇ String: p_sCaption
    ◇ Number: p_nFlags
```

The first parameter is again the window handle of the caller where you pass hWndForm or hWndNULL.

The second parameter is used as the title text of the message box. If you pass hWndForm in the first parameter and STRING_Null in the second, the title of the calling application is used.

The third parameter expects the standard message box flags like MB_Ok and MB_IconStop. The actual message dialog appears on top of the application. If this fails, the call is tried again with the SYSTEMMODAL flag. This should work regardless of the application's available memory but in fact a bug has been reported that the second call never happens. So if the application is really running out of memory, this function will most likely be incapable of displaying anything.

The return value is the same as with SalMessageBox( ). If the function returns a value other than 0, the message box must have been visible and the user must have been able to click on a button. A value of 0 indicates that it didn't appear at all.

I don't really see a practical use for this function, but for the sake of completeness, I included it in the class.

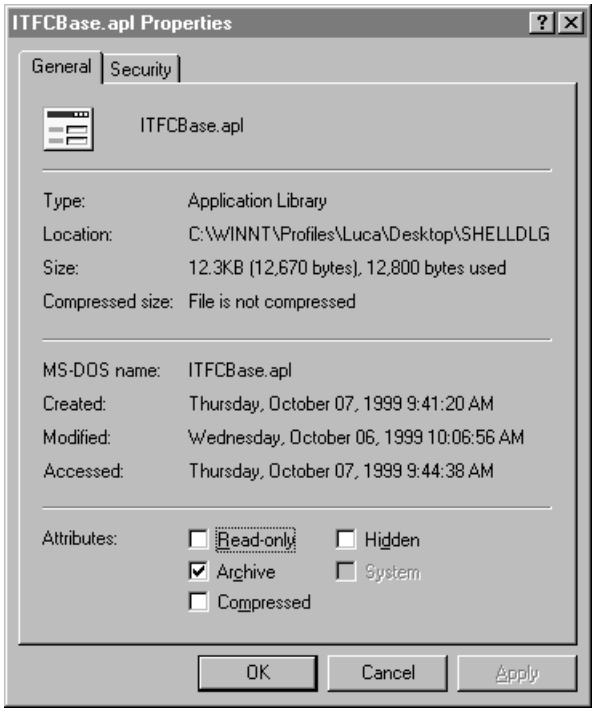

Figure 6. The Object Properties Dialog.

System Restart

Have you ever changed anything with your driver configuration, such as installing a network protocol or changing IP configuration? If so, then you know the dialog shown in **Figure 7**:

This built-in dialog can be executed through a simple API.

Listing 11. The Restart Dialog.

```
◆ Function: RestartDialog
  ◆ Returns
    ◇ Boolean:
  ◆ Parameters
    ◇ Window Handle: p_hWndOwner
    ◇ String: p_sMessage
    ◇ Number: p_nFlags
```

The API accepts three parameters. The first one is the window handle of the owner. This can be hWndForm or hWndNULL.

The second parameter is a pointer to a string displayed on the dialog before the standard text. It can also be STRING_Null. I recommend that you append a carriage return/linefeed to the string in order to separate it from the standard text. Then it looks much better. Unfortunately, I ran into a small problem with this

Figure 7. The built-in dialog to reboot after modifying system options.

function. When run on Windows 98, everything works fine; but when run on Windows NT 4.0, the user-defined string is displayed as garbage characters. On NT the API expects a wide-char string; on Windows 95/98 it expects an ANSI string. So I wrote a small function that checks whether the application is running on NT. If it is, I convert the string to wide-char format. Windows 95/98 and NT have an API for this: MultiByteToWideChar, which is also declared in the source code. To determine which platform the application is running on, I use the GetVersionEx API.

The last parameter is a flag that tells Windows what action to execute after the user clicks the yes button.

Listing 12. The Restart Dialog.

```
◇ Number: ITC_RESTART_LOGOFF = 0x00
◇ Number: ITC_RESTART_SHUTDOWN = 0x01
◇ Number: ITC_RESTART_REBOOT = 0x02
```

Pass one of these (self-explanatory) constants in **Listing 12** to the function in order to make Windows execute the desired action.

Special Folders

Windows introduced special folders beginning with Windows 95 and NT 4.0 that the OS relies on. Examples are folders for recent files, send to, profiles, and desktop. The location of the folders is predefined and Windows provides an API to retrieve the actual locations. The function declaration is shown in **Listing 13**.

Listing 13. For retrieving a special folder location.

```
◆ Function: GetSpecialFolderLocation
  ◆ Returns
    ◇ Boolean:
  ◆ Parameters
    ◇ Window Handle: p_hWndOwner
    ◇ Number: p_nFolderID
    ◇ Boolean: p_bCreate
    ◇ Receive String: r_sPath
```

As usual, the function takes the window handle of the owner as the first parameter, Pass hWndForm or hWndNULL.

The second parameter expects a folder ID. Several folder IDs exist, and I declared constants for them (see **Listing 14**).

Listing 14. Special folder IDs.

```
◇ Number: ITC_CSIDL_DESKTOP  = 0x0000
◇ Number: ITC_CSIDL_INTERNET = 0x0001
◇ Number: ITC_CSIDL_PROGRAMS = 0x0002
◇ Number: ITC_CSIDL_CONTROLS = 0x0003
◇ Number: ITC_CSIDL_PRINTERS = 0x0004
◇ Number: ITC_CSIDL_PERSONAL = 0x0005
◇ Number: ITC_CSIDL_FAVORITES = 0x0006
◇ Number: ITC_CSIDL_STARTUP = 0x0007
◇ Number: ITC_CSIDL_RECENT = 0x0008
◇ Number: ITC_CSIDL_SENDTO = 0x0009
◇ Number: ITC_CSIDL_BITBUCKET = 0x000a
◇ Number: ITC_CSIDL_STARTMENU = 0x000b
◇ Number: ITC_CSIDL_DESKTOPDIRECTORY = 0x0010
◇ Number: ITC_CSIDL_DRIVES = 0x0011
◇ Number: ITC_CSIDL_NETWORK = 0x0012
◇ Number: ITC_CSIDL_NETHOOD = 0x0013
◇ Number: ITC_CSIDL_FONTS = 0x0014
◇ Number: ITC_CSIDL_TEMPLATES = 0x0015
◇ Number: ITC_CSIDL_COMMON_STARTMENU = 0x0016
◇ Number: ITC_CSIDL_COMMON_PROGRAMS = 0x0017
◇ Number: ITC_CSIDL_COMMON_STARTUP = 0x0018
◇ Number: ITC_CSIDL_COMMON_DESKTOPDIR = 0x0019
◇ Number: ITC_CSIDL_APPDATA = 0x001a
◇ Number: ITC_CSIDL_PRINTHOOD = 0x001b
◇ Number: ITC_CSIDL_ALTSTARTUP = 0x001d
◇ Number: ITC_CSIDL_COMMON_ALTSTARTUP = 0x001e
◇ Number: ITC_CSIDL_COMMON_FAVORITES = 0x001f
◇ Number: ITC_CSIDL_INTERNET_CACHE = 0x0020
◇ Number: ITC_CSIDL_COOKIES = 0x0021
◇ Number: ITC_CSIDL_HISTORY = 0x0022
```

The third parameter tells the function to create the folder if it doesn't already exist.

The fourth parameter returns the pathname where the folder can be found. If the folder doesn't exist, the return value may be empty.

The function returns ITE_Ok on success.

I also implemented a function that fills a string array with all the defined special folders: EnumSpecialFolders( ). It expects a dynamic string array as the only parameter in which it will return all the folder pathnames. The return value of the function is the number of folders that have been found.

No guarantees

As I mentioned earlier, some of the functions aren't documented by Microsoft. They're not officially supported and may be subject to change in future versions of Windows. This could result in problems when your applications don't work in a future rev of Windows.

However, in my opinion, the functions will remain intact for quite awhile. Microsoft itself is using them and that's a pretty good guarantee for an API to survive the next version update. Additionally, Microsoft usually continues to include older APIs for a long time. I even expect that the company will continue to implement these dialogs when they don't need them any more.

But, after all, there's no guarantee . . .**CP**

Download Dialogs.zip from this issue's table of contents at www.ProPublishing.com or find it on this month's Companion Disk.

Joachim Meyer is one of the owners of BASYS EDV-Systeme, a networking company in Bremen (Germany). Being a Novell and Microsoft systems specialist he is responsible for software development and controlling. He has been programming since 1982 and has worked with SQLWindows, CTD, Delphi, and C. He's a member of the Team Assist Organization and one of the founding members of the Ice Tea Group, LLC..

Open to the Public: Ice Tea Foundation Classes

Joachim Meyer

Last year the Ice Tea Group (ITG) started a project called "Ice Tea Foundation Classes" (ITFC). Now this class library has become freely available to all Centura developers. Here's an introduction.

Originally, ITG planned to build a class and function library under the Ice Tea Foundation Class name and sell it; later we decided to release it to the public domain. We do this in order to serve the developer community and show the great potential that exists in Centura Team Developer. Our hope is to increase the number of developers using CTD and to persuade some not to switch to another development environment. Consider this article (and others that will follow) as our commitment to Centura's development products. In this first article, I'll discuss the basics of ITFC.

Naming conventions

As ITFC articles are published here, the libraries will be extended with needed constants, functions, and classes. In order to make your reading and learning easier, the ITG has agreed on basic naming conventions:

- Constants are prefixed with three letters, depending on their meaning. The prefix ITM stands for messages, ITC for general-purpose constants, and ITE for error messages.

Listing 1. Examples of naming conventions.

```
◆ Function: Test
  ◆ Parameters
    ◇ Number: p_nNumeric
    ◇ String: p_sString
    ◇ Receive Date/Time: r_dtDate
    ◇ cITFCBuffer: p_Buffer
  ◆ Local variables
    ◇ Number: nNumber
    ◇ String: sString
```

- Types will be prefixed as well. We adopt the naming conventions suggested by Centura, which means b=Boolean, n=Number, s=String, dt=Date/Time, hWnd=Window Handle, and so forth. When variables are passed as parameters, the parameter itself is prefixed with p_ followed by type prefix and variable name (see **Listing 1**). If it's a receive parameter, the prefix changes to r_.

- When variables are used inside classes, the prefix changes to m=public instance member, c=public class member, and __(double underscore)=private member. Using the double underscore has the additional advantage that you can suppress these member variables in the popup dialog of IntelliSal (the intelligent coding helper published by ITG) and they don't appear in the Coding Assistant.

- The names of internal functions start with the capital letters ITFC followed by a mixed-case descriptive name. There is no underscore between prefix and function name.

- The names of classes start with the uppercase letters CITFC followed by a mixed-case descriptive name.

- Generally, coding global functions will be avoided as much as possible, and necessary functions are put into a functional class instead. Because IntelliSal is able to recognize qualified function calls, there's no limitation as there is with the Coding Assistant. Using IntelliSal almost eliminates the need for the Coding Assistant.

- If it's necessary to initialize a class, I will implement initialization code in a function with a special name: Construct( ). This function is meant to be a constructor, comparable to other OOP languages like C++ or Pascal. The counterpart of the Construct( ) function is Destruct( ), which cleans up the class.

Because SAL doesn't automatically call constructors/destructors, it's the responsibility of the developer to call the constructor before using any function of the class, and to call the destructor at the end as well.

Initial structure

The structure at this point in the development of ITFC is very easy:

- *ITFCBase* is the generic library, which will most likely be included into any other library. It contains the CITFCBase class.
- *ITFCMemory* includes ITFCBase because its CITFCMemory class is derived from CITFCBase.
- *ITFCWinSDK* includes ITFCMemory directly because the ITFCWinGetPlatform( ) function uses a memory buffer from ITFCMemory. ITFCBase is included indirectly through ITFCMemory but not needed currently.

ITFCBase

ITFCBase collects functions and constants that are generic and don't logically fit into any other library. Currently, only the error handling is implemented.

The CITFCBase class is the "mother of all classes." All other classes will be derived from it, so it's the perfect place to implement functions common to all classes. For the time being, CITFCBase has four functions to handle the error code and error class.

ITG members have had long discussions about which error handling mechanism is the best. Several flavors came up immediately and were dropped just as quickly. We've finally concluded that all approaches to handling errors have their pros and cons, and we can't agree on more than a generic list of error constants. This will leave the class library open to all kind of errors that we might face in the future.

The same error number can have different meanings depending on the place where the error occurred. The Windows API might raise the same error number as the SQLBase API. To get rid of this double meaning, I introduce the "error class" to distinguish between the different types of errors. This means that an error is defined by two numbers: the actual error code and its error class. Currently three error classes are defined:

- *ITC_ERRCLASS_ITFC* for errors that have been raised by the Foundation Classes.
- *ITC_ERRCLASS_Windows* for errors being raised by Windows APIs.
- *ITC_ERRCLASS_SQL* for database-related errors.

The CITFCBase class has four functions to deal with errors:

- *ClearError( )* resets the internal error variables to "no error."
- *GetErrorClass( )* returns the class of the most recent error.
- *GetErrorCode( )* returns the number of the most recent error.
- *SetLastError( )* sets the internal error variables to the passed values and returns whether it was an error (ITE_Error) or not (ITE_Ok).

Every class derived from CITFCBase inherits the error functions and is ready to use the error system. The functions of derived classes should always return a Boolean value indication whether the function succeeded (return value=ITE_Ok) or failed (return value=ITE_Error). Because ITE_Ok is declared as TRUE and ITE_Error as FALSE, you can write easy-to-read code:

```
◆ If MyClass.Afunction()
  ◇ ! go on with next action
◆ Else
  ◇ ! display error message
```

ITFCMem

ITFCMem is a library that deals with buffers and blocks of memory.

Basically this library is just a wrapper that encapsulates the CSTRUCT DLL that comes with SQLWindows. The library contains a single class, CITFCMemory, which is derived from our base class CITFCBase, of course.

The class inherits the error handling capabilities from CITFCBase and it exposes a virtual interface hook into the error handling. Every function that needs to set or clear errors, calls the **__Error( )** function. This, in turn, calls the **CITFCMemory_Error( )** function in a late-bound fashion. This means that you can implement the CITFCMemory_Error( ) function in every derived class, and it will be called whenever an error condition is set or cleared. This is a classic way to implement hooks into a class library. Virtual functions are called in a late-bound fashion (with the two dots in front of the name), which makes it easy to implement different or extended behavior in your derived class.

The class is able to set three different error conditions:

- **ITE_InvalidBufferPos**. Offset and size specified would result in a read or write beyond the boundaries of the memory block.

- **ITE_CstructError**. An error occurred with a call to the Cstruct functions.

- **ITE_NoAllocatedMemory**. A function was called before any memory was allocated or after the destructor has been executed.

CITFCMemory deals with memory blocks. It has functions to allocate and free memory, as well as functions to set values in memory and retrieve them from memory.

Construct( ) allocates a block of memory and stores the pointer and size internally. In order to free the memory after use, you must call the **Destruct( )** function. Calling the Construct( ) function twice without having freed the memory doesn't hurt because Construct( ) first checks to see if there is any allocated memory; if there is, it simply calls Destruct( ) for you. But be careful: If you've filled the memory with structures that are pointers themselves, those pointers aren't handled. If you have to use such, you must first free these embedded memory pointers and afterwards call Destruct( ).

You also have the ability to construct the memory block by passing a pointer or a string. **ConstructFromMem( )** requires three parameters. The first one is a pointer to a memory block, and the internal pointer will be set to the same value. The second parameter is the size of the passed memory block, and the internal size variable will be set to the same value. The third parameter tells the class whether the memory shall be released in the destructor or not. By setting this parameter to TRUE, the class automatically frees the memory when the destructor is called. When set to FALSE, the destructor doesn't free the memory but simply sets the internal variables to zero. Be careful in your application design, because there's the danger of freeing memory twice or not freeing it at all. Most likely either mistake will sooner or later lead to an access violation.

Setting and retrieving values is done through several functions in CITFCMemory. Currently the class handles only numbers and strings. For numbers, there's the universal function **GetNumber( ).** It's able to handle several number types within a single function. The first parameter is a Receive Number variable that returns the value being read. The second parameter tells the function to begin reading at a specified offset. This offset is zero-based. The last parameter is one of the ITC_TYPE constants (see **Listing 2**). These constants actually contain two pieces of information. First, each datatype is represented through a unique number, which is encoded in the high byte. The low byte carries the information about the size in bytes that a data type occupies in memory.

The **PutNumber( )** function is similar to the GetNumber function( ) except that it writes the passed value instead of reading it.

As a bonus, the functions that read and write values insure that memory isn't accessed beyond the end of the memory block (which would also lead to an access violation). The private function **__CheckPosition( )** is responsible for this.

For string handling the GetString( ) and PutString( ) functions are provided.

GetString( ) expects three parameters. The first one is a string variable in which the value will be returned. The second one defines the offset in the memory block from where the function should start reading. The last one tells the function how big the string value would be at maximum.

PutString( ) is quite similar except that the first parameter is the value to be written to the memory block. The second parameter defines the offset where writing should begin, and the last parameter defines the size of the value in bytes. If you pass –1 for the third parameter, the function assumes that the value is a normal string rather than some binary data and uses SalStrLength( ) to determine the size. You can't pass –1 if the value contains one or more zero bytes as actual data because SQLWindows interprets this as an end-of-string byte.

Last, **CopyFromMem( )** is a wrapper around the CstructCopyBuffer( ) function, which is able to copy data from one memory block to another memory block. My function expects a pointer to an external memory block in the first parameter. The offset where copying should begin goes into the second parameter. The third parameter defines the offset in the internal buffer (the target) to which the data is copied. The fourth and last parameter tells the function how many bytes are to be copied.

Listing 2. Data type constants.

```
◇ ! // Number types
◇ Number: ITC_TYPE_BYTE  = 0x0101
◇ Number: ITC_TYPE_WORD  = 0x0202
◇ Number: ITC_TYPE_SHORT = 0x0202
◇ Number: ITC_TYPE_LONG  = 0x0304
◇ Number: ITC_TYPE_INT = 0x0304
◇ Number: ITC_TYPE_FLOAT = 0x0404
◇ Number: ITC_TYPE_DOUBLE  = 0x0508
◇ Number: ITC_TYPE_BFLOAT  = 0x0608
```

ITFCWinSDK

ITFCWinSDK is a library that declares the functions of the Windows API and provides constants and wrapper functions. As more ITFC classes are implemented, this library will be extended dramatically, because Windows has so many APIs, and they all go into this library.

Currently, only three APIs are declared: ITFCWinGetPlatform, ITFCWinStrToWideStr, and GetLastError.

As you might have noticed, GetLastError( ) doesn't have the ITFCWin prefix. It's declared as an external

Continues on page 12

Do You Make This Mistake with DELETE CASCADE?

R.J. David Burke

With a self-referencing table, CASCADE is the only DELETE rule allowed, when you use SQLBase's declarative referential integrity (RI). For example, consider the code in **Listing 1**, which creates a table with self-referencing referential integrity.

Using a BEFORE DELETE trigger to implement an DELETE RESTRICT and DELETE SET NULL rules.

Centura

Listing 1. A SQLTalk script to create a typical self-referencing table with declarative referential integrity.

```
CREATE TABLE EMP
(
  EMP_ID INTEGER NOT NULL,
  NAME   VARCHAR(20) NOT NULL,
  SALARY DECIMAL(8,2) NOT NULL,
  MGR_ID INTEGER
)
/

INSERT INTO EMP (EMP_ID, NAME, SALARY, MGR_ID)
  VALUES (:1, :2, :3, :4)
\
1,Albert,100000,,
2,Barry,90000,1
3,Charles,90000,1
4,Diane,91000,1
5,Edward,80000,2
6,Fred,80000,3
7,George,70000,3
8,Heidi,81000,4
9,Isabelle,71000,6
10,Jim,70000,6
11,Kathy,71000,8
12,Larry,70000,8
13,Mary,61000,10
14,Ned,60000,10
/

CREATE UNIQUE INDEX EMP_PK
  ON EMP (EMP_ID)
/

ALTER TABLE EMP
  PRIMARY KEY (EMP_ID)
/

ALTER TABLE EMP
  FOREIGN KEY MANAGER (MGR_ID) REFERENCES EMP
  ON DELETE CASCADE
/
```

With the example in Listing 1, the MGR_ID column is a self-referencing column to the same table, EMP, identifying the manager of a given employee. However, the DELETE CASCADE rule probably *isn't* what you want. If you were to delete Charles, for example, you'd also end up deleting Fred, George, Isabelle, Jim, Mary, and Ned. SQLBase's enforcement of the DELETE CASCADE rule ensures that RI is maintained. However, deleting one employee shouldn't propagate to deleting their subordinates. Just because a manager quits doesn't mean that all of his or her staff quits as well.

To prevent this from happening, you can use a BEFORE DELETE trigger, as shown in **Listings 2** and **3**. Listing 2 shows an implementation of DELETE RESTRICT and Listing 3 shows an implementation of DELETE SET NULL.

Listing 2. A BEFORE DELETE trigger to implement DELETE RESTRICT.

```
CREATE TRIGGER T_DEL_EMP
  BEFORE DELETE ON EMP
(EXECUTE INLINE(EMP_ID)
Procedure T_DEL_EMP Static
Parameters
  Number: p_EmpID
Local Variables
  Boolean: bExists
  Number: nRetVal
Actions
  Call SqlExists( 'SELECT * FROM EMP WHERE MGR_ID =
  ':p_EmpID', bExists )
  If bExists
    Set nRetVal = 8429
  Else
    Set nRetVal = 0
  Return nRetVal
)
  FOR EACH ROW
/
```

The BEFORE DELETE trigger in Listing 2 returns error code 8429 if dependent rows exist; this is the same error returned when dependent rows exist in a non-self-

referencing relationship, so it seems appropriate. By returning a non-zero value from a trigger, you effectively cancel the DML operation.

Listing 3. A BEFORE DELETE trigger that implements the DELETE SET NULL rule.

```
CREATE TRIGGER T_DEL_EMP
  BEFORE DELETE ON EMP
(EXECUTE INLINE(EMP_ID)
Procedure T_DEL_EMP Static
Parameters
  Number: p_EmpID
Local Variables
Actions
  Call SqlImmediate( 'UPDATE EMP SET MGR_ID = NULL WHERE
MGR_ID = :p_EmpID' )
  Return 0
)
  FOR EACH ROW
/
```

Although use of SqlImmediate and SqlExists in SQLWindows and CTD applications has been frowned upon, they're safe to use in SQLBase SAL, as they're an entirely different implementation and don't have any code dependencies on their counterparts in the development tools.

The downside of using these triggers is that you first have to remove the self-referencing foreign key, with a command as shown in **Listing 4**. This is disappointing because the self-referencing relationship is no longer documented in the database system catalog tables.

Listing 4. Removing the self-referencing foreign key.

```
ALTER TABLE EMP
  DROP FOREIGN KEY MANAGER
/
```

This doesn't have any effect on your other RI declarations. You can still declare the table's primary key and any non-self-referencing foreign keys. **CP**

R. J. David Burke is Manager of the Complementary Software Partner Program at Centura. Contact him through Pro Publishing.

Open to the Public...

Continued from page 10

function with an ordinal of 0.

If an ordinal is set to 0, SQLWindows searches the DLL library for a function with the specified name. If an ordinal is specified, SQLWindows ignores the name and searches the DLL for a function with that ordinal. In the first case the name of the External Function must be written exactly as it's compiled into the DLL; it's case-sensitive. If the ordinal is specified, the name of the External Function is ignored, which would make it much easier to avoid name conflicts. But (and there's always a but) Microsoft isn't well known for being consistent, so the same APIs have similar function names, but different ordinal numbers, in Windows 9x vs. Windows NT. This discrepancy makes it impossible to use ordinal numbers in applications that are expected to run on both systems. So we have to ignore a great feature of SQLWindows and use function binding by name in every case. I truly hope that someday Centura will change this and let the developer specify an alternative name for the External Function. Then several libraries can declare the same APIs without causing duplicate-definition conflicts.

Two wrapper functions are included in ITFCWinSDK: ITFCWinGetPlatform( ) and ITFCWinStrToWideStr( ). ITFCWinStrToWideStr( ) expects a standard string and converts it to a Widestring. Widestrings are needed on Windows NT because some of the APIs expect them instead of normal strings. And it gets worse: The same API function expects a *normal* string under Windows 9x. This made it necessary to check the current platform with ITFCWinGetPlatform( ) before converting. So the result of ITFCWinStrToWideStr( ) is a normal string on Windows 9x and a Widestring on Windows NT.

Don't worry if you don't see the use of these two functions now. The cover article "Searching the Shell," needs them, and for the sake of completeness I explain them here.

The last function is GetLastError( ), which is declared in the External Function section. Because this API is very easy to access and doesn't involve special techniques, I didn't implement a special function for it.

It's most likely that this library will be subject to change in the future. As far as possible Internal Functions should be avoided and classes developed instead. Because this library doesn't contain much at present, a CITFCWinSDK class isn't provided yet. **CP**

Special thanks go to Gianluca Pivato, Thomas Althammer, and Frank Böttcher (all members of the ITG) who helped with solving problems and inspiring me. Without them, the ITFC project would never have been started.

Download ITFC_Part1.zip from the November 1999 table of contents at www.ProPublishing.com or find it on this month's Companion Disk.

Joachim Meyer is one of the owners of BASYS EDV-Systeme, a networking company in Bremen (Germany). Being a Novell and Microsoft systems specialist he is responsible for software development and controlling. He has been programming since 1982, using such development environments as SQLWindows, CTD, Delphi, and C. He's a member of the Team Assist Organization and one of the founding members of the Ice Tea Group, LLC.