CenturaPro™

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

I'll Call You Back

Christian Schubert

A callback procedure is a Windows mechanism to handle asynchronous events—events you trigger somehow (by calling a function) but which execute at some time unknown to you in the future. In some ways they're much like messages but with at least two differences:

Using Windows callback procedures in SQLWindows has always been difficult. There are several approaches to mastering them, including coding special DLLs for each purpose or using the object compiler. There was no global solution—until now!

- Callback procedures don't need window handles.
- Callback procedures can carry more than two parameters, an important advantage.

That advantage is the main reason callback procedures are used. The maximum number of parameters in sample callbacks that I researched was six; the minimum was just one.

SQLWindows 32

The problem with callbacks in SQLWindows

Some useful functions found in the Windows SDK require the address of a callback procedure. This is a problem with SQLWindows, because you can't determine the address of a SAL function at runtime. So the solution for the callback problem is going to require some C coding. However, I'll provide a "one size fits all" solution that can be used for almost any purpose and doesn't require you to dive into the depths of C programming or even own a C compiler.

The solution

The solution should have the following features:

- It must be written in C to be able to provide a pointer to a function.
- It must fit for any possible number of parameters of callback procedures.
- It must provide a way to write the code for the callback procedure in SAL.

The solution is a small DLL written in C. It contains its own procedure for each number of parameters required and relies on the fact that *every* parameter of *every* callback function uses four bytes. This is true for any type of pointer: handle, DWORD, long, and so on. (I've found no exception to this rule so far.)

The callback procedure hands over its parameters to a SAL function by calling SalCompileAndEvaluate. The name of the SAL function to be called can be set by the application.

Continues on page 3

December 1999

Volume 4, Number 12

PRO Publishing

I'll Call You Back . . .

Continued from page 1

How it works

Let's look at the simplest kind of callback procedure: one parameter. Let's say the procedure to handle the callback in our SAL application is named MyCallback. It's defined the from window frmTest and looks like this:

```
◆Function: MyCallback
  ◇Description:
  ◇Returns
  ◆Parameters
    ◇Number: nParam1
  ◇Static Variables
  ◇Local variables
  ◆Actions
    ◇! do something useful here...
```

We have to tell the DLL what SAL function should be called and how many parameters it uses. We do this by calling SetCallback:

```
◇Call SetCallback ( "frmTest.MyCallback", 1, TRUE,
  pCallbackProc, hModule )
◇If pCallbackProc = 0
  ◆! something went wrong, bail out
  ...
```

SetCallback takes the following parameters:

- *sProc*—Name of the SAL function to be called. It's always a good idea to fully qualify the functions name.
- *nNumParams*—Number of parameters to be used.
- *bReportErrors*—Report errors when calling SalCompileAndEvaluate.
- *pProc*—Receive parameter; contains the pointer to the callback procedure if everything was OK; otherwise, it's zero.
- *pModule*—Receive parameter; contains the module handle needed by some SDK functions.

The C callback procedure pointed to by pProc after calling SetCallback looks like this:

```
void CALLBACK CallbackProc1(DWORD dwParam1)
{
  TransferParams ( 1, dwParam1 );
}
```

TransferParams builds the following string and executes it via SalCompileAndEvaluate:

```
frmTest.MyCallback ( 12345 )
```

We assumed that dwParam1 contained the value 12345. The values of the parameters are converted to numeric constants when building the statement. The callback's parameters can now be used within our SAL function.

Samples

I include two sample applications to show how callbacks can be used.

The first one covers a topic that is frequently asked for in Centura's newsgroups and forums: the Windows folder-browsing dialog (see **Figure 1** on page 4). The SDK function used here is *SHBrowseForFolder*. It takes a pointer to a structure that contains information about the desired action (a description of the parameters follows):

```
typedef struct _browseinfo {
HWND hwndOwner;
LPCITEMIDLIST pidlRoot;
LPSTR pszDisplayName;
LPCSTR lpszTitle;
UINT ulFlags;
BFFCALLBACK lpfn;
LPARAM lParam;
int iImage; }
BROWSEINFO
```

This structure can be defined as an external function, as shown in **Listing 1**.

SHBrowseForFolder is embedded into a functional class that also contains the required variables and the callback function. You need the callback to be able to set the initial folder and do some validation. The callback function takes four parameters, as shown in **Listing 2**.

The callback is triggered for various reasons. Once it's sent right before the dialog is shown (nMsg =

Listing 1.

```
◆Function: SHBrowseForFolder
  ◇Description:
  ◇Export Ordinal: 0
  ◆Returns
    ◇Number: LONG
  ◆Parameters
    ◆structPointer
      ◇Window Handle: HWND  ! parent windows of browsing dialog
      ◇Number: LPVOID     ! pointer to start directory (not used here)
      ◇Receive String: LPSTR  ! buffer containing the folder the user chose
      ◇String: LPSTR    ! text to be shown in the dialog
      ◇Number: UINT     ! flags, control appearance and behavior
      ◇Number: LPVOID     ! pointer to callback function
      ◇Number: LPARAM     ! application defined value for callback
      ◇Receive Number: INT  ! image of selected folder (not used here)
```

Listing 2.

```
Number: nWnd     ! window handle of the dialog
Number: nMsg     ! type of callback message
Number: lParam     ! value dependant on the message type
Number: lpData     ! applications defined value (used to distinguish different callers)
```

BFFM_INITIALIZED) and lets us modify the initial directory, the title of the dialog, or the status text. It's also triggered when the user selects a folder (nMsg = BFFM_SELCHANGED) or clicks OK and the edit box contains an invalid folder (nMsg = BFFM_VALIDATEFAILEDA).

As you see there are many possibilities for influencing the dialog.

SHBrowseForFolder returns the pointer to a shell identifier list containing the folder. It must be extracted to a string by using the function *SHGetPathFromIDList*. Please note that the return value of *SHBrowseForFolder* points to a chunk of memory allocated by the shell. It would be the responsibility of the calling application to release this memory. However, the technique needed here can't be used with SQLWindows (at least I haven't found out how, but I'm still researching it). It would require us to deal with pointers to class interfaces (as returned by *SHGetMalloc*, just for those of you who are interested). So if you use *SHBrowseForFolder* the way we did here, you'll have a small (but probably acceptable) memory leakage.

The second sample (**Figure 2**) shows the definition of a keyboard system hook, which also requires a callback procedure. This procedure gets all keyboard events before the application gets them. This could be handy if you need special accelerators and don't want to (or can't) use the Visual Toolchest accelerator functions.

The callback function required here takes three parameters. We're only interested in two of them: If *nCode* is less than zero, we shouldn't process the event. *lParam* contains information about the keys that caused the event. We use *GetKeyNameText* to get a descriptive name of the key and display it.

Using more than one callback simultaneously

If you ever need to have more than one callback procedure at the same time, you can use the following trick to accomplish this:

1. Copy the file *callback*.dll to *callback2.dll* (or any other filename you like).

2. Add the following external function to your code:

```
◆Library name: Callback2.dll
  ◆Function: SetCallback
    ◇Description:
    ◇Export Ordinal: 1
    ◇Returns
    ◆Parameters
      ◇Number: INT
      ◇String: LPVOID
      ◇Number: INT
      ◇Boolean: BOOL
      ◇Receive Number: LPLPVOID
      ◇Receive Number: LPHANDLE
```

3. Now you have another callback available. You may repeat these steps if more simultaneous callbacks are needed.

Caveats

There are also some things to watch for when handling callback procedures.

- Be careful when writing hook procedures. You might get in trouble if your procedure generates events that trigger a new call of the hook procedure. This will almost certainly crash your application. Sometimes it's not even enough to provide a bypass mechanism to prevent this message stack overflow. I didn't manage to get a window message hook to work, which is probably due to the way SQLWindows handles its own function calls.

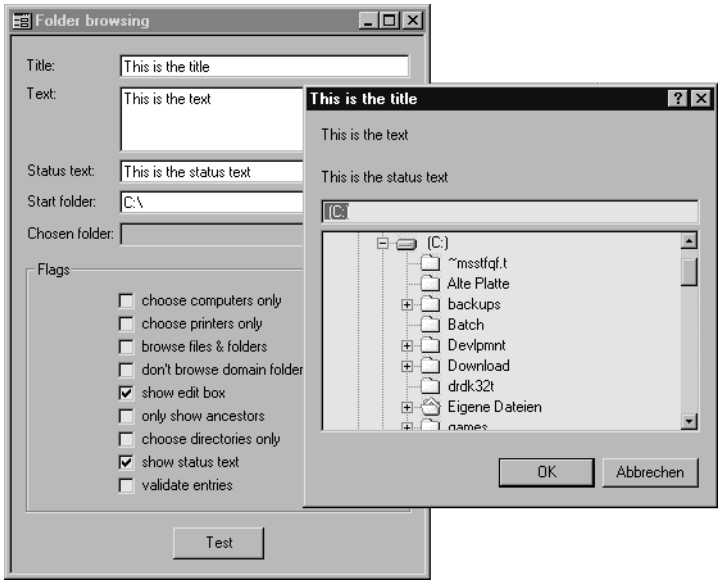

Figure 1. The shell folder browser.

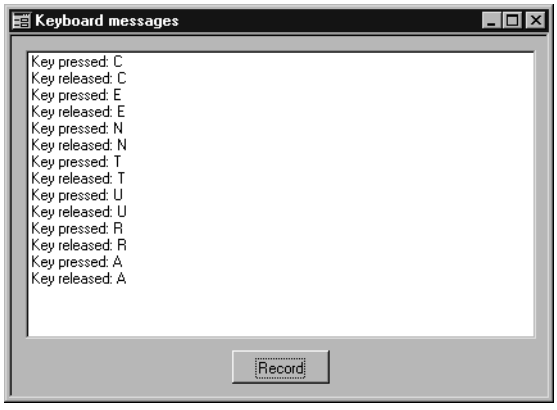

Figure 2. The output window of the keyboard hook.

- The method of using callback procedures as shown here only works with callback functions using four-byte parameters. Though I haven't found any callback procedures of a different kind, there's no proof that they don't exist. So have a close look at the definition of the callback procedure before using it.

- If you reuse a callback procedure by overwriting the definition of the SAL function, make sure that the old callback won't be triggered any longer, or the SAL function might be called with a wrong number of parameters.

- After calling *SetCallback*, check that *pProc* isn't zero. If you submit a NULL pointer to a function, you might get a GPF.

- I've seen applications that use fake callback procedures for communication between different programs. These procedures aren't defined as CALLBACK (in C) and have a totally different way of placing their parameters on the stack. Therefore you'll also get some nice GPFs if you try to use them. **CP**

Download Callback.zip from www.propublishing.com or find it on this month's Companion Disk.

Christian Schubert works as Senior Developer at GODEsys GmbH, Mainz, Germany. He has been using Centura products since 1993 and develops sales management applications. He can be reached at christian.schubert@mainz.netsurf.de.