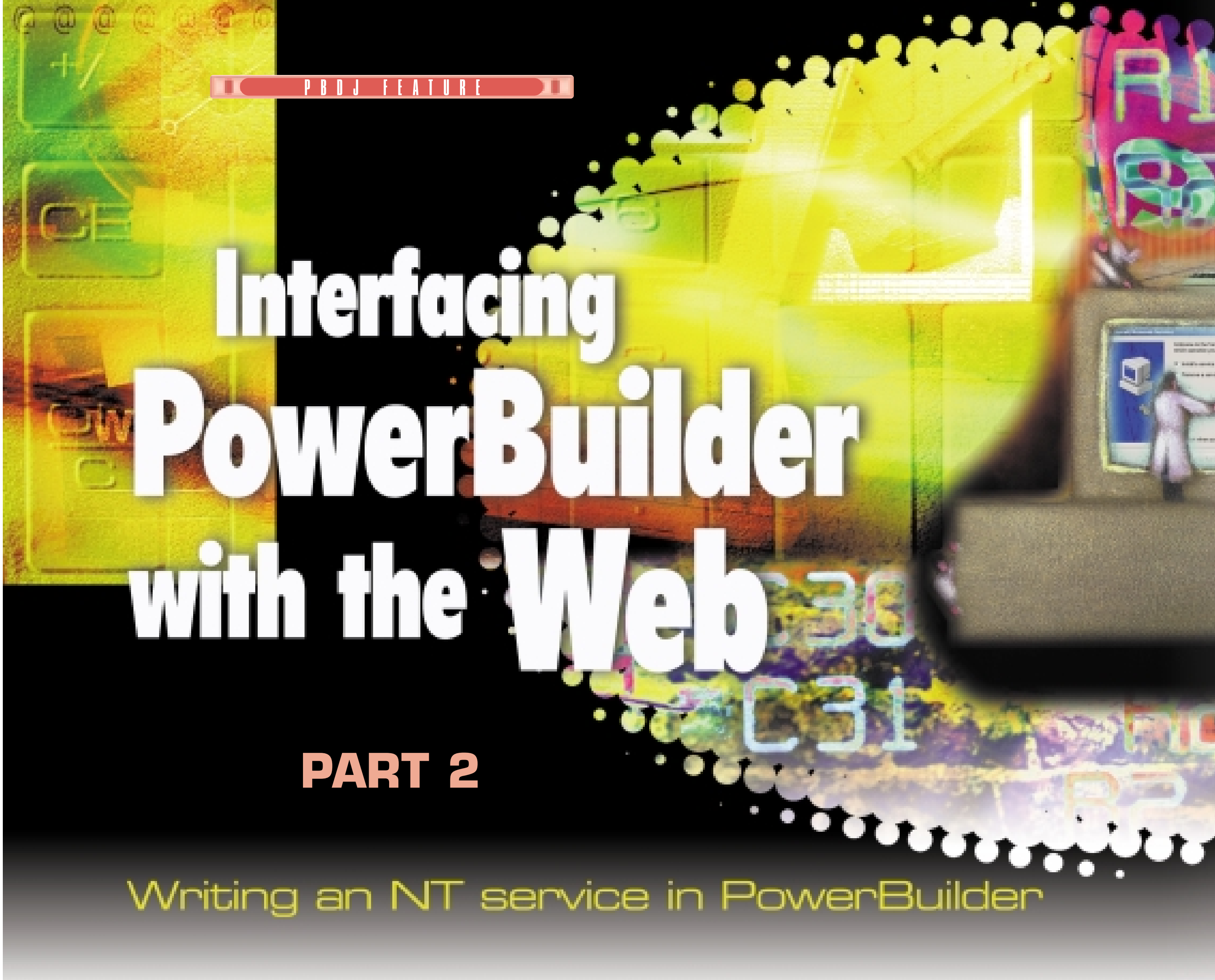
PBDJ FEATURE



Interfacing PowerBuilder with the Web

PART 2

Writing an NT service in PowerBuilder

This is the second of two articles describing how JP Morgan in London developed an XML interface between a Web-based bond trading system and one of its back-office systems. Part 1 (Vol. 8, issue 8) focused on parsing the XML file; Part 2 shows how to write an NT service in PowerBuilder.

Why Use an NT Service?

JP Morgan's XML interface has to perform three core functions:

1. Process incoming XML messages.
2. Watch for state changes in the database.
3. Process outgoing XML messages.

These tasks had to be performed 24/7, without human intervention, in a secure and reliable manner. The ideal solution was an NT service. This could be installed on a server and configured to start as soon as the machine booted up without the need for anyone to log on. We knew that Microsoft had a utility, SRVANY, that would let any EXE be deployed as a service. A bit of research showed that it would work with a PowerBuilder application.

Writing an NT service is different from a traditional PowerBuilder application. Services are essentially batch jobs. They have no user interface, so if you need to get a message to the outside world you can't just pop up a message box, and they are time-driven rather than waiting for user interaction such as mouse clicks. There are three things you need to know about to write a service in PowerBuilder: how to use the timer object in a nonvisual application, how to write to the NT event log, and how to deploy your EXE as a service.

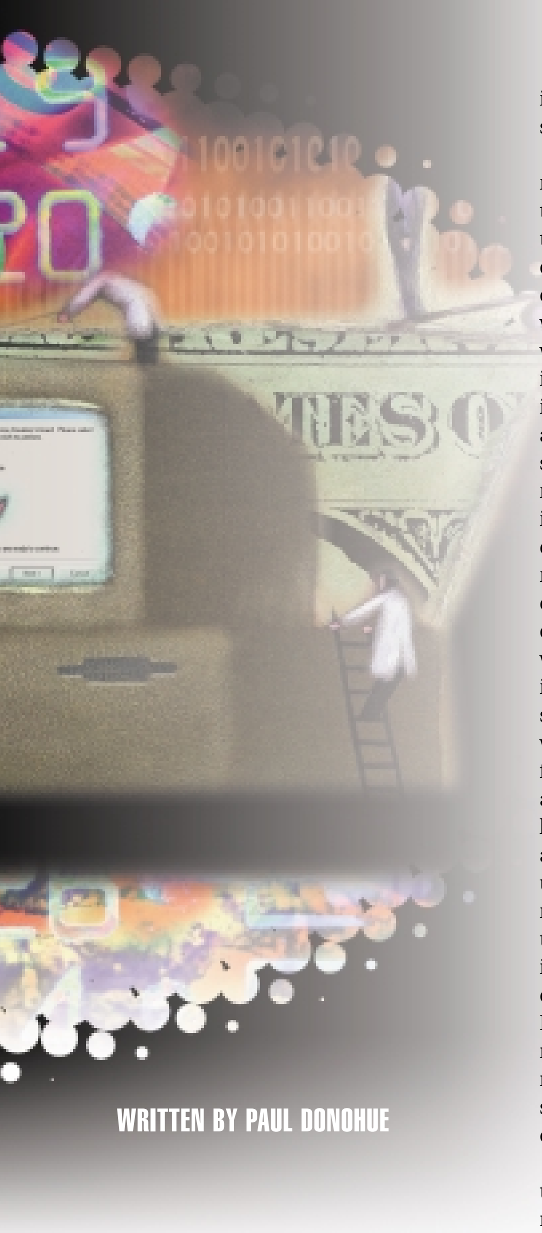
Creating the Timer Object

The basic design of an NT service is an application that loops continuously, waiting for certain actions to occur. This sounds like a perfect use for PowerBuilder's timer object. There are four steps to creating a timing object for an NT service:

1. Create a timer object that is a standard class inherited from the timing object.
2. Add a function to initialize the service.
3. Add a function to finalize the service.
4. Add code to the timer event.

The Initialize Function

It's good practice for an NT service to record the fact that it started successfully. The preferred way of doing this is to write an entry to the NT event log. (I'll discuss how to do this later. For now just be aware that



WRITTEN BY PAUL DONOHUE

it's the first thing you should do.)

PowerBuilder terminates an application when the last window closes or the application's Open event finishes – whichever comes first. This works well in most applications where at least one window is open while the system is in use. However, with a nonvisual application, such as a service, there are no windows. If you initialize a timer when the application starts, PowerBuilder may close down the application before the timer event is triggered. As a workaround, open an invisible window before starting the timer. This will prevent PowerBuilder from terminating your application. In addition to keeping the application alive, the window is useful during development. Check the application's handle to determine if you're running as an executable or from the PowerBuilder IDE. If you're running in development, make the window visible so that you can display debugging messages on it.

The only other task that the initialize function needs to do is to start the timer. You should retrieve the timer interval from the registry, INI file, or database rather than using a hard-coded value.

The Finalize Function

It's important that an NT service tidy up after itself. The finalize function should do standard shutdown processing such as:

- Disconnecting from the XML parser
- Logging off from the database
- Destroying OLE objects

It's also good practice to write an entry to the NT event log recording that the service has finished.

The Timer Event

The timer event is the heart of the service. It's triggered every *x* seconds and each time it performs four functions:

1. ***It stops the timer:*** The timer is stopped just in case a single cycle takes longer than the timer interval. Although this is unlikely under normal processing conditions, it's quite likely if you're using the debugger. If you don't stop the timer, overlapping timer events may be triggered, which is confusing.
2. ***It performs a single cycle of work:*** To keep the timer event code nice and clean, call a function that performs a single iteration of what the service is supposed to do. (More about this later.)
3. ***It performs garbage collection:*** Even though PowerBuilder should tidy up any orphaned objects, I prefer to leave nothing to chance. The service may have to run 24/7 so it's important that it's robust with no memory leaks.
4. ***It restarts the timer:*** If you don't restart the timer, no further timer events will be triggered.

A Cycle of Work

A cycle of work is a discrete unit of processing that should be small enough to start and stop during the timer period. Ideally, each cycle would be stateless and wouldn't rely on events that occurred in previous cycles, although in practice this may not be possible. For example, you might want each cycle to connect to the database, do its processing, and disconnect. Although this would start and stop the cycle in the same state, your database administrator may not be happy with performing expensive operations like connect and disconnect every few seconds.

To make matters worse, if you use Sybase 11.5 there's a memory leak in the Open Client driver, so if you do connect and disconnect each cycle you'll have to reboot the server on a regular basis. It's more efficient to maintain a transaction that's connected when the service starts and disconnected when it finishes. If you decide on a permanent transaction, it's important to tend to lost database connections. At the start of each cycle check that the transaction is still alive and reconnect if necessary.

A typical cycle of work for an XML parsing service might be:

- Check that the database connection is alive and reconnect if necessary.
- Check an "in box" directory for incoming XML files. (More about this later.)
- Parse the XML files using the XML parsing methods discussed in Part 1 of this article.
- Process each XML file. This probably involves updating a database, calling a stored procedure, sending an e-mail, or invoking a business rule nonvisual object.
- Generate any outgoing XML files that are required either as a result of the incoming messages or in response to state changes in the database.

There are some things to remember when designing your cycle of work. Services can't access network drives, so you may need to configure your server accordingly. At JP Morgan our Sybase drivers were installed on network drives so we had to install local copies before the service could connect to the database. Obviously, services can't interact with the user because they may be started when nobody is logged on. This means you can't use message boxes or ask the user for any sort of input. You'll have to record any application settings in the registry, an INI file, or the database.

The NT Event Log

It's probably time to explain how to use the NT event log. Table 1 shows the three Win32 API calls you'll use to write to the event log. Listing 1 shows how to declare them as external functions; Listing 2 is a code snippet that will write "Hello" to the event log. (All code can be found at www.PowerBuilderJournal.com.)

After you run the code in Listing 2, open the event log viewer and find the message in the application log. As you can see from Figure 1, the message has been prefixed by a warning and appears as:"The description for event ID (1) in source (NT Event Log Demo) cannot be found.

WIN32 API Function	Description
RegisterEventSource()	Establishes a connection to the event log
ReportEvent()	Writes an entry to the event log
DeregisterEventSource()	Closes the event log handle

TABLE 1 Win32 event log API functions

The local computer may not have the necessary registry information or DLL files to display messages from a remote computer. The following information is part of the event: Hello."

Windows NT has inserted the warning because you don't have a message file. The event log doesn't normally store the wording of every message in the log. Instead, the text of each message is stored in a message file and given a unique identifier. Messages can have placeholders such as "Error number %1 occurred during the %2 process." When you connect to the event log you specify an event source that relates to a message file. When you log an event you supply the identifier of the message along with values for the placeholders. If there's no message file for the event source, NT will add the "description for event cannot be found" warning to the start of your message.

How do you make a message file? Unfortunately, these are DLLs and PowerBuilder cannot compile a DLL of the required format. If you have a C++ compiler you can make your own message file. You can either make a file with one entry for each message your service requires or you can make a generic message file DLL that has only one message consisting of just a placeholder. The generic approach uses more event log resources, as the text of each message is stored each time, but it lets you write any message to the log, and you can reuse the message file for all your applications.

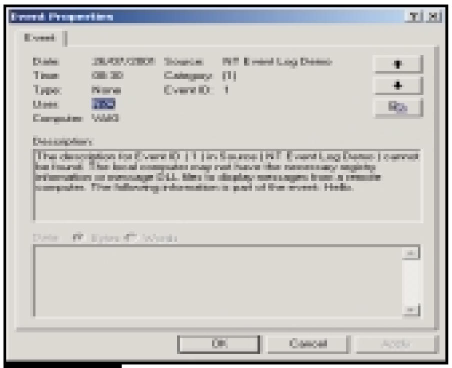
I won't go into the details of compiling a message file, but if you're interested, refer to Kevin Miller's book, mentioned in the Resource section at the end of this article. I use a generic message file that you can download, along with the example source code, from the ***PBDJ*** URL mentioned earlier.

You have to let the event log service know about your message file. This is achieved with the following registry entries. Add your service as a new key under "HKEY_LOCAL_MACHINE \ SYSTEM \ ControlSet001 \ Services \ Eventlog \ Application \ MyService" where the "MyService" is the event log source you register in your application. Add a string value called "EventMessageFile" whose value is the fully qualified name of your message file, for example, "C:\SERVICE\MESSAGE.DLL". Finally, add a DWORD value called "TypesSupported" with a value of 7. Why 7? You'll have to read Kevin Miller's book.

Processing Files

Unless you're using a queuing system such as IBM's MQSeries, your NT service will have to access XML files. Although PowerBuilder's 10 built-in file functions are useful for manipulating individual files, they provide no way to identify all the files in a directory or move files – both of which are essential for a service that processes XML files. Table 2 shows three Win32 API functions that provide this functionality. Listing 3 shows how to declare the functions and Listing 4 gives an example of how to use them.

Both FindFirstFileA and FindNextFileA use a structure passed by reference to hold the file information. Actually, you have to create two structures because the "file_data" structure contains the "file_datetime" structure. See Tables 3 and 4 for details of what's required. The third API, MoveFileA, is straightforward. It has two arguments, which are a "from" file and a "to" file. When called, it will move a file from one location to the other.



Date: [illegible] Source: NT Event Log Demo
Time: [illegible] Category: (1)
Type: None Event ID: 1
User: [illegible]
Computer: [illegible]

Description:
The description for Event ID (1) in Source (NT Event Log Demo) cannot be found. The local computer may not have the necessary registry information or message DLL files to display messages from a remote computer. The following information is part of the event: Hello.

FIGURE 1 Entry in NT event log showing missing message file error

WIN32 API Function	Description
FindFirstFileA	Find the first file in a directory
FindNextFileA	Find the next file in a director.
MoveFileA	Move a file

TABLE 2 Win32 file manipulation API functions

Data Type	Variable
unsignedlong	ul_fileattributes
s_file_datetime	str_creationtime
s_file_datetime	str_lastaccesstime
s_file_datetime	str_lastwritetime
unsignedlong	ul_filesizehigh
unsignedlong	ul_filesizelow
unsignedlong	ul_reserved0
unsignedlong	ul_reserved1
character	ch_filename[260]
character	ch_alternatefilename[14]

TABLE 3 The "file_data" structure used by the FindFirstFileA and FindNextFileA API functions

Data Type	Variable
unsignedlong	ul_lowdatetime
unsignedlong	ul_highdatetime

TABLE 4 "file_datetime" structure used by "file_data" structure

Using the Timer Object

Using the event log and file processing API calls, it's simple to write your timer object's cycle of work. It might scan an "in box," parse any XML files it finds, then move the processed files to a "processed box." Once the timer object has been developed, you have to add code to the application object to use it.

First, declare a global variable of the same type as your timer object. If the timer object is called "n_cst_service", the declaration might look like this:

```
n_cst_service      gnv_service
```

Second, in the application's open event create a new instance of the timer object and call the initialize function to start it running. Remember, this function will open an invisible window to keep the application running after the open event script finishes.

```
gnv_service = CREATE n_cst_service
gnv_service.of_initialize()
```

Third, in the application's close event call the finalize function and destroy the timer. If you don't destroy the timer, then strange things will start to happen – particularly in the development environment.

```
gnv_service.of_finalize()
DESTROY gnv_service
```

Running as a Service

So far, all you've done is develop a nonvisual PowerBuilder application that acts like a service. To deploy your application as a service, you'll use two utilities that come with the WindowsNT4 Resource Kit – SRVANY and SRVINSTW. If you don't have the resource kit, you can download these utilities from Microsoft's Web site. By the way, I've tested these on Windows 2000 and they work fine.

SRVANY is a wrapper that can run any executable as a service. The first

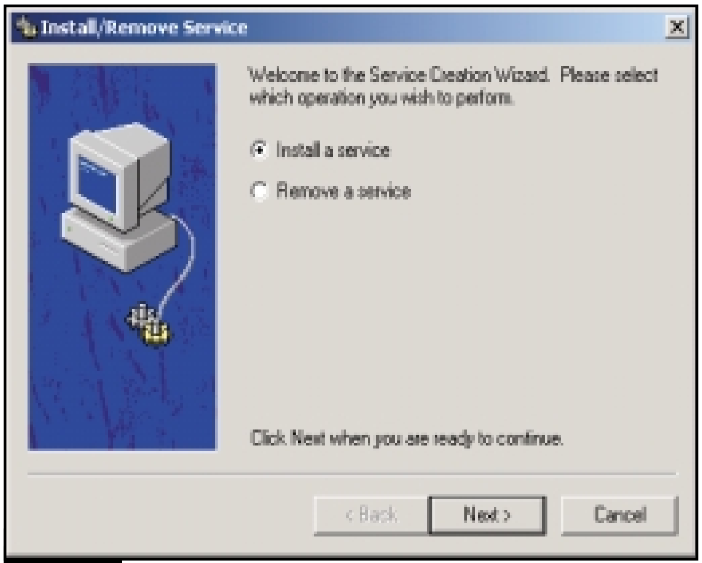

Install/Remove Service
Welcome to the Service Creation Wizard. Please select which operation you wish to perform.
Install a service
Remove a service
Click Next when you are ready to continue.
< Back
Next >
Cancel

FIGURE 2 SRVINSTW, the service installation wizard

step in using this is to compile your application into an EXE. You can use machine code or interpreted, but it's worth the extra compile time to make machine code executable.

SRVINSTW is a service installation wizard. You'll use it to configure SRVANY as a service that will run our executable. Using SRVINSTW is very straightforward – see Figure 2 for a sample screen print. The only thing to remember is that the executable file for the service will be SRVANY.EXE, not your PowerBuilder executable. The wizard will ask you a number of questions. The correct answers are as follows:

- Choose "install a service".
- Select local machine.
- Give your service an appropriate name.
- The EXE to run is SRVANY.EXE.
- Run your service as its own process.
- Use the system account as it doesn't require a user ID or password. As soon as the server is booted, the system account is available.
- Check the "interact with desktop" box. Your service may not interact with the desktop, but SRVANY needs this enabled.
- Set the start-up option to automatic.

At this point SRVANY is set up as a service, but it hasn't been configured to run your application. Some registry entries are required to associate your EXE with SRVANY. The installation wizard will have created a registry entry for your service under the key "HKEY_LOCAL_MACHINE\SYSTEM\ControlSet001\Services". You have to add two new keys – "parameters" and "enum". Add three string values under the "parameters" key:

1. ***Application:*** Your service's path and EXE
2. ***AppDirectory:*** Your service's working directory
3. ***AppParameters:*** Any command-line parameters that your service requires

You need to add any values under the "enum" key when the service is first run. If everything has been set up correctly, reboot your machine and your service should start working.

Controlling the Service

An NT service doesn't have any way of interacting directly with a user. There's no window on which to place controls such as "Stop" or "Start" buttons, and even if there was, the service would start running when the machine boots rather than waiting for a user to log on.

So how do you control it? The easiest way is to store control information in the registry. The bare minimum would be a flag to indicate whether the service should be running. On each cycle check the value of this flag and, if it's set to "N", stop the service. The only thing that keeps the application running is the invisible window so closing this when the flag changes to "N" will allow the application's close event to execute, which will call the finalize function.

To make your service easier to support, it's worth developing a simple administrator utility so you don't have to edit the registry directly. This utility can control the stop/start process and record database connection details and the location of in, out, and processed directories for your XML messages. At JP Morgan our administration tool takes the start/stop idea one step further by defining daily processing start and stop times to ensure that the service will shut down during the nightly backup.

If you stop your service from Control Panel, SRVANY will use the TerminateProcess() function to stop your EXE. This is a drastic way of stopping executables as it sends no application or window close events. Your application gets no warning that SRVANY wants it to close down and it will be stopped immediately. The correct way to stop your service is to set the start/stop flag in the registry using the administration utility. This will stop the PowerBuilder executable cleanly and execute the finalize function. Remember that your service will log an event when it shuts down cleanly, so check the log to make sure it's finished. Once you see the shutdown message, it's safe to stop the service from Control Panel, which will terminate SRVANY.

Final Thoughts

A service should be designed to run continuously – especially one that interfaces with the Internet. Make sure your application is bulletproof by checking every return status and planning for every possibility. Time spent adding self-healing features like automatically reconnecting lost database connections will pay off.

Make good use of the event log. A service is like a black box in that you can't really tell what's going on inside. The event log is your window into the service, so log progress messages on a regular basis. If an error does occur, record everything that might be useful in tracking down the problem.

Before you put the service into production, spend some time monitoring its resource usage with NT Performance Monitor. If you find a memory leak, test each component in isolation if possible to determine whether the problem is with the parsing, file management, or something else.

"A service should be designed to run continuously – especially one that interfaces with the Internet"

Once you get an NT service working properly, it can be a lot easier to support than a traditional Windows application. Because it operates below the level of user interaction, there's much less that can go wrong with it. Short of turning off the power, it should run continuously, and if the worst happens and it does crash, all it takes to recover is to reboot the server.

Resource

Miller, K. (1998). *Professional NT Services*. Wrox Press. This book covers everything about NT services although, unless you intend writing one in C++, there isn't much else you need to know. ▼

AUTHOR BIO
Paul Donohue has 15 years' experience as a solution provider. He's worked with PowerBuilder since version 2 and is a certified PowerBuilder developer.

pbdj@pauldonohue.com

Listing 1

```
Declaring the Win32 event log API functions
function long RegisterEventSource( string pszServerName,
string pszSourceName ) LIBRARY "ADVAPI32.DLL" alias for
"RegisterEventSourceA"

function boolean ReportEvent( long hEventLog, integer
wType, integer wCategory, long dwEventID, long pUserSID,
integer wNumStrings, long dwDataSize, string
pStringArray[], long pRawData ) LIBRARY "ADVAPI32.DLL"
alias for "ReportEventA"

function boolean DeregisterEventSource( long hEventLog )
LIBRARY "ADVAPI32.DLL" alias for "DeregisterEventSource"
```

Listing 2

```
Example of writing to the event log
integer li_icon  // The icon id (e.g. 1 = stop sign)
long ll_event_source_handle  // The event log system's
handle
string ls_message_array[]  // The event log expects an
array
string ls_server  // The server whose log you want to
write to
Boolean lb_result  // Success or failure?

// Place "hello" into an array because the API
// call does not accept a simple string
ls_message_array[1] = "hello"

// Register the event source and get a handle
// to the WIN32 event logging system
SetNull(ls_server) // A null server means write locally
ll_event_source_handle = RegisterEventSource(ls_server, &
                          "NT Event Log Demo" )

// Log the message
lb_result = ReportEvent ( &
             ll_event_source_handle, /* Handle for the
event log */ &
             li_icon,                /* Icon type */ &
             1,                      /* Category */ &
             1,                      /* Event ID */ &
             0,                      /* User ID */ &
             1,                      /* Strings in the
array */ &
             0,                      /* Amount of raw
data */ &
             ls_message_array,       /* Message array */ &
             0                       /* Raw data */ &
             )

// Deregister the event source
DeregisterEventSource(ll_event_source_handle)
```

Listing 3

```
Declaring the Win32 file manipulation API functions
function long FindFirstFile (ref string sPathName, ref
s_file_data fd) LIBRARY "KERNEL32.DLL" alias for
"FindFirstFileA"

function long FindNextFile (long lFileHandle, ref
s_file_data fd) LIBRARY "KERNEL32.DLL" alias for
"FindNextFileA"

function long MoveFile (String lpExistingFileName, String
lpNewFileName) LIBRARY "KERNEL32.DLL" Alias for "MoveFileA"
```

Listing 4

```
Example of finding all files in a directory
s_file_data lstr_filedata // An array to hold file info
string ls_file_path // Path of the directory to process
string ls_file_name_this // Current file name
string ls_file_name_all // A list of all files
long ll_file_handle // Current file handle
long ll_more_files // Any more files?

// Find the first file in the temp directory
ls_file_path = "C:\TEMP\*.XML"
ll_file_handle = FindFirstFile (ls_file_path, lstr_fileda-
ta)

// Process any files in the temp directory
IF   ll_file_handle = -1 THEN // -1 = no files found
     MessageBox ("No files to process", &
                 "No XML files were found.", &
                 Exclamation!, &
                 OK!)
ELSE

     DO
          // Get the current file name from the
          // structure and add it to the list
          ls_file_name_this = lstr_filedata.ch_filename
          ls_file_name_all = ls_file_name_all &
                             + ls_file_name_this &
                             + "~r~n"

          // Get the next file
          ll_more_files = FindNextFile (ll_file_handle,
lstr_filedata)

          LOOP WHILE ll_more_files = 1 // 0 = no more files

          MessageBox ("XML Files", &
                      "These XML files were found.~r~n~r~n"
&
                      + ls_file_name_all, &
                      Information!, &
                      OK!)

END IF
```

Download the Code!

The code listing for this article can also be located at
www.PowerBuilderJournal.com