

Le Portable Debugger

by Randy Charles Morin

I'm a fan of the portable debugger. These are small debuggers that can be quickly installed on production machines and used to view additional data flows available from an application in order to diagnose problems in the application. These were particularly useful at one company where they had many internal libraries that could not be compiled in release mode and had to release their application with debug mode on.

I was able to install a portable debugger on the production box and view all the debug messages that would normally be sent to a process debugger. This made debugging production servers easier than pie.

Don't get me wrong, I am not recommending releasing your software in debug mode in order to save a few moments of debugging. Software with debug enabled is software looking to be stolen. It also runs a few percentage points slower than software without the additional debug information overhead.

I'll start by showing you how these portable debuggers are built. Then I'll move onto how you can use the same technique to provide additional debug information in your applications, even when you compile in release mode.

1 DebugView

A really great website for any Windows Developer is the System Internet website at www.systeminternals.com. This site provides a great list of small utilities like RegMon and FileMon that can be very beneficial in analyzing an application. Mark Russinovich and Bryce Cogswell maintain the site. Mark Russinovich co-authored Inside Windows 2000 and is a contributing editor of DDJ and Windows 2000 Magazine.

The site has the best freely available portable debugger. It's called DebugView and is available at <http://www.systeminternals.com/ntw2k/freeware/debugview.shtml>. They have maintained the product for some years now and over time it has acquired a great deal of features that make it the best of the bread.

The product basically captures all messages sent to the OutputDebugString Win32 API function that would not otherwise be caught by a process debugger. You may be more familiar with TRACE macros. These TRACE macros usually have a definition that simply redirects the output to OutputDebugString when in debug mode and compiles out to nothing in release mode.

Listing Error! Bookmark not defined.: TRACE

```
#ifdef _DEBUG
#define TRACE(x) ::OutputDebugString(x)
#else
#define TRACE(x)
#endif
```

Now, let's test the DebugView utility. Write a small brief section of code that calls OutputDebugString.

Listing Error! Bookmark not defined.: Test

```
#include <windows.h>

int main( int argc, char ** argv )
{
    ::OutputDebugString("Hello\n");
    return 0;
};
```

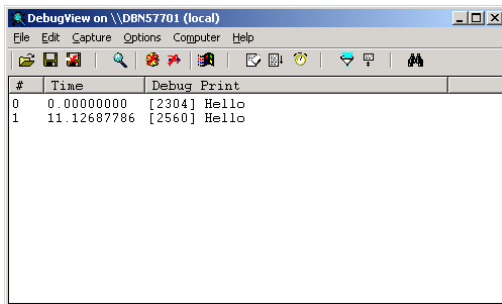
If you paste this code into a new command-line project in Visual C++, then you can create a small executable that sends a hello messages to the debug stream. Start the DebugView utility. Now compile the code in VC++ and run it from the IDE. Use Build | Start Debug from the menu bar, not the Build | Execute option. The later option actually runs the process outside the debugger. Now, look at the Visual C++ debug window.

Listing Error! Bookmark not defined.: Debug Window

```
Loaded 'C:\WINNT\System32\ntdll.dll', no matching symbolic information found.
Loaded 'C:\WINNT\SYSTEM32\KERNEL32.DLL', no matching symbolic information found.
Hello
The thread 0x984 has exited with code 0 (0x0).
The program 'C:\Documents and Settings\Administrator\My
Documents\Code\debugstream\Debug\debugstream.exe' has exited with code 0 (0x0).
```

You'll see that you printed the "Hello" message in the debug window. If you switch the DebugView utility, then you'll see that it is still empty. This is because the VC++ debugger was actively debugging the process and therefore intercepting the debug messages.

Now, run the code again, but use Build | Execute in order to run the test outside the VC++ debugger. Now switch to the DebugView utility and you'll be pleasantly surprised. The debug messages are now being captured by the DebugView utility.

Figure Error! Bookmark not defined.: DebugView

There's no magic here. Microsoft presented a small piece of code many years ago that served as the basics behind this little trick. Many groups took that code and expanded on it to provide much more functionality. This evolved into an array of portable debuggers of which DebugView is my current favorite. Next I'll present code similar to Microsoft's original piece of code that shows how to pull off this little trick.

2 Internals

The trick involves two named events, a named file mapping and a bit of protocol. The named events are "DBWIN_BUFFER_READY" and "DBWIN_DATA_READY". The named file mapping is "DBWIN_BUFFER".

Initially the buffer ready event is set to up and the data ready event is set to down. When a client wants to send data on the debug stream, then it checks the buffer ready event. If the buffer ready event is set to up, then it sets it to down, places the message in the buffer and sets the data ready event to up.

The debugger is waiting on the data ready event. When the data ready event is set to up, the event is set to down, the message is read from the buffer and finally the buffer ready event is set to up.

Listing Error! Bookmark not defined.: Portable Debugger

```
#include <windows.h>
#include <iostream>

int main( int argc, char ** argv )
{
    SECURITY_ATTRIBUTES sa;
    SECURITY_DESCRIPTOR sd;
    sa.nLength = sizeof(SECURITY_ATTRIBUTES);
    sa.bInheritHandle = TRUE;
    sa.lpSecurityDescriptor = &sd;

    std::cerr << "*****\n"
                "Debug Monitor Started\n"
                "*****\n";

    if (::InitializeSecurityDescriptor(&sd,
        SECURITY_DESCRIPTOR_REVISION) == FALSE)
    {
        std::cout << "InitializeSecurityDescriptor failed"
                  << std::endl;
        return 1;
    }

    if (::SetSecurityDescriptorDacl(&sd, TRUE, (PACL)NULL,
        FALSE) == FALSE)
    {
        std::cout << "SetSecurityDescriptorDacl failed"
                  << std::endl;
        return 1;
    }

    HANDLE bufferready = ::CreateEvent(&sa, FALSE, FALSE,
        "DBWIN_BUFFER_READY");
    if (bufferready == NULL)
    {
        std::cout << "CreateEvent failed" << std::endl;
        return 1;
    }

    if (GetLastError() == ERROR_ALREADY_EXISTS)
    {
        std::cout << "Debugger Already Running" << std::endl;
        return 1;
    }

    HANDLE dataready = ::CreateEvent(&sa, FALSE, FALSE,
        "DBWIN_DATA_READY");
    if (dataready == NULL)
    {
        std::cout << "CreateEvent failed" << std::endl;
        ::CloseHandle(bufferready);
        return 1;
    }

    HANDLE buffer = ::CreateFileMapping(INVALID_HANDLE_VALUE,
        &sa, PAGE_READWRITE, 0, 4096, "DBWIN_BUFFER");
    if (buffer == NULL)
```

```
{
    std::cout << "CreateFileMapping failed" << std::endl;
    ::CloseHandle(bufferready);
    ::CloseHandle(dataready);
    return 1;
}

void * str = ::MapViewOfFile(buffer, FILE_MAP_READ, 0, 0,
    4096);

if (str == NULL)
{
    std::cout << "MapViewOfFile failed" << std::endl;
    ::CloseHandle(bufferready);
    ::CloseHandle(dataready);
    ::CloseHandle(buffer);
    return 1;
}

char * string = (char *)str + sizeof(DWORD);
DWORD lastpid = 0xffffffff;
bool cr = true;

while (true)
{
    if (::SetEvent(bufferready) == FALSE)
    {
        std::cout << "SetEvent failed" << std::endl;
        ::CloseHandle(bufferready);
        ::CloseHandle(dataready);
        ::UnmapViewOfFile(str);
        ::CloseHandle(buffer);
        return 1;
    };

    if (::WaitForSingleObject(dataready, INFINITE)
        != WAIT_OBJECT_0)
    {
        break;
    }
    else
    {
        DWORD pid = *(DWORD *)str;
        if (lastpid != pid)
        {
            lastpid = pid;
            if (!cr)
            {
                std::cerr << std::endl;
                cr = true;
            }
        }

        if (cr)
        {
            std::cerr << lastpid << ":";
        }

        std::cerr << (char*)string;
        cr = (*string &&
            (string[::strlen(string) - 1] == '\n'));
    }
}

std::cout << "WaitForSingleObject failed" << std::endl;
::CloseHandle(bufferready);
::CloseHandle(dataready);
::UnmapViewOfFile(str);
::CloseHandle(buffer);
return 1;
};
```

This protocol creates a critical section that prevents clients and the portable debugger from accessing the buffer at the same time. It also triggers the debugger whenever messages are placed in the buffer to perform a task with the debug message.

3 My Viewer

One thing that I've considered doing is using the same technique to capture all sorts of messages from my application. Consider an application that sends and receives SOAP messages. You could create a data capture utility for the application that listens to the input and output from the application.

Instead of using the DBWIN_BUFFER_READY and DBWIN_DATA_READY named events and DBWIN_BUFFER named file mapping. You could create your own names, say SOAPMSG_BUFFER_READY and SOAPMSG_DATA_READY events and SOAPMSG_BUFFER file mapping.

Now all you need is an implementation of OutputDebugString for our SOAP messages. Next is sample code of what just such a function might look like.

Listing Error! Bookmark not defined.: OutputSoapMessage

```
VOID OutputSoapMessage(LPCTSTR output)
{
    HANDLE bufferready = ::OpenEvent(EVENT_MODIFY_STATE, FALSE,
        "SOAPMSG_BUFFER_READY");
    if (bufferready == NULL)
    {
        return;
    }

    HANDLE dataready = ::OpenEvent(EVENT_MODIFY_STATE, FALSE,
        "SOAPMSG_DATA_READY");
    if (dataready == NULL)
    {
        ::CloseHandle(bufferready);
        return;
    }

    HANDLE buffer = ::OpenFileMapping(FILE_MAP_ALL_ACCESS,
        TRUE, "SOAPMSG_BUFFER");
    if (buffer == NULL)
    {
        ::CloseHandle(bufferready);
        ::CloseHandle(dataready);
        return;
    }

    LPSTR memory = (LPSTR)::MapViewOfFile(buffer,
        FILE_MAP_WRITE, 0, 0, 4096);
    if (memory == NULL)
    {
        ::CloseHandle(buffer);
        ::CloseHandle(bufferready);
        ::CloseHandle(dataready);
        return;
    }

    ::WaitForSingleObject(bufferready, INFINITE);
    *((DWORD*) memory) = ::GetCurrentProcessId();
    ::wsprintf(memory + sizeof(DWORD), "%s", output);
    ::SetEvent(dataready);

    ::UnmapViewOfFile(memory);
    ::CloseHandle(buffer);
}
```

```
    ::CloseHandle(bufferready);  
    ::CloseHandle(dataready);  
    return;  
}
```

Now using a modified version of the Portable Debugger in Listing 4 and the previous OutputSoapMessage function, you can capture data from your application. In your application, wherever you frame a new SOAP message call the OutputSoapMessage function passing the SOAP envelop as the parameter.

When the portable debugger is not running the OutputSoapMessage function returns pretty quickly without doing anything. Then if you start the message capture utility, the OutputSoapMessage function begins sending the messages to your message capture utility.

About the Author

Randy Charles Morin is the Lead Architect of SportMarkets Development from Toronto, Ontario, Canada and lives with his wife and two kids in Brampton, Ontario. He is the author of the www.kbcafe.com website, author of Wiley's Programming Windows Services book and co-author of many other programming books and articles.