WinSock 2 Debug and Trace Facilities

1Introduction

Developers of WinSock 2 applications and service providers need to be able to isolate bugs to one of:

- The client application,
- The WinSock 2 DLL, or
- The service provider (transport or name space)

The WinSock 2 debug/trace layer address this problem. It allows all procedure calls across the WinSock 2 API or SPI to be monitored, and to some extent controlled.

Developers can use this mechanism to trace the procedure calls, procedure returns, parameter values, and return values. Parameter values and returns can be altered on procedure-call or procedure-return. If desired, a procedure-call can even be prevented or redirected. With access to this level of information and control, it should be easy for a developer to isolate any problem to the application, WinSock 2 DLL or service provider.

The debug/trace layer is supported only by a specially instrumented version of the WinSock 2 DLL. The SDK license terms do not allow re-distribution of this instrumented WinSock 2 DLL in order to prevent inappropriate use of the debug hooks in production systems.

This document serves two purposes. First, it defines the interface between the WinSock2 DLL and an auxiliary debug/trace DLL. Second, it describes the functionality, design, and implementation of the default debug/trace DLL (named dt_dll.dll), which is supplied in source-code form with the WinSock 2 SDK. Developers may find that the default debug/trace DLL supplies all the necessary functionality for their purposes. However, they are also free to modify the code to provide additional debugging functionality, or they may start from scratch and build a new debug/trace DLL. In the latter case, only the interface definition portion of this document may be relevant, and the reader can skip the section describing the default DLL. The default DLL does contain hooks, however, which allow it to be extended for all but the most complex types of debugging and tracing.

2The Interface

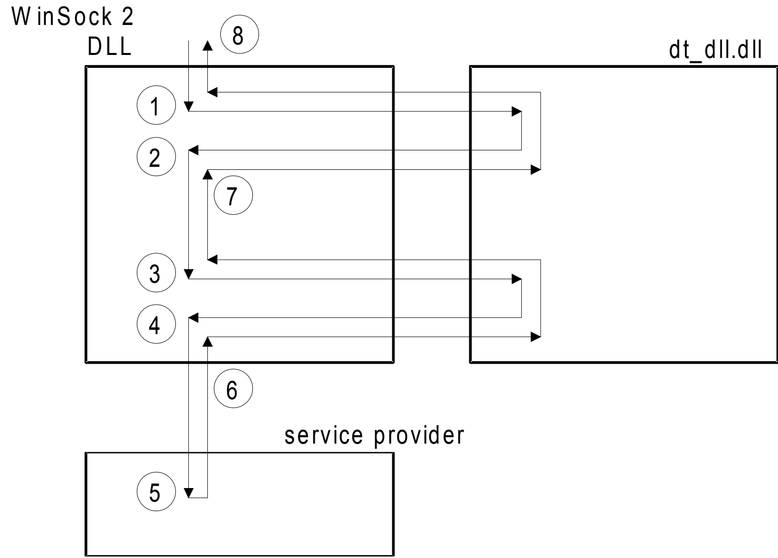
2.1Overview

As mentioned above, the debug/trace layer is composed of a specially-instrumented WinSock 2 DLL, combined with an auxiliary debug/trace DLL. The interface between these two modules is defined by two functions which must be exported by dt_dll.dll and called *WSAPreApiNotify* and *WSAPostApiNotify*. When the instrumented WinSock 2 DLL loads, it attempts to load dt_dll.dll and retrieve pointers to these functions (the header file dt_dll.h in the WinSock 2 sources gives typedefs for these pointers, as well as prototypes of the exported functions). Later, the application will call an API function. However, the instrumented WinSock 2 DLL has API function names bound to special hook functions. These hook functions wrap calls to WSAPre/PostApiNotify around the call to the actual API (and SPI) functions, as follows.

- Before the hook function does anything, it calls the WSAPreApiNotify function, passing along all the information about the call and parameters. WSAPreApiNotify, implemented in the debug/trace DLL, returns a boolean indicating whether the original procedure call should proceed or not. If WSAPreApiNotify returns a “should proceed” indication, the instrumented WinSock 2 DLL goes ahead and proceeds with the function execution. When the execution is ready to return to the application, the instrumented WinSock 2 DLL calls the WSAPostApiNotify function in dt_dll.dll. When WSAPostApiNotify returns, the WinSock 2 DLL returns to the application..

- When the instrumented WinSock 2 DLL is about to call a SPI function, the WinSock 2 DLL first calls WSAPreApiNotify in the auxiliary DLL. WSAPreApiNotify returns an indication of whether or not the call should proceed. If the call should proceed, the instrumented WinSock 2 DLL calls the SPI function. After the SPI function returns, the WinSock 2 DLL calls WSAPostApiNotify and eventually returns to the caller.

The execution path through these debug hooks for a typical function such as WSAConnect is illustrated in the figure below:



A typical execution path for WSAConnect involving the debug/trace layer is shown. (1) A client application calls WSAConnect. The WinSock 2 DLL calls the WSAPreApiNotify entry point in dt_dll.dll passing WSAConnect call information. The dt_dll.dll decodes the call info and returns. (2) The WinSock 2 DLL proceeds until it is about to call the provider. (3) The WinSock 2 DLL calls WSAPreApiNotify with WSPConnect call information. The dt_dll.dll decodes the information and returns. (4) The WinSock 2 DLL calls the service provider's WSPConnect function, which (5) performs the connect and returns.

(6) The WinSock 2 DLL calls WSAPostApiNotify with WSPConnect call information. The dt_dll.dll decodes and returns. The WinSock 2 DLL completes its processing. (7) the WinSock 2 DLL calls WSAPostApiNotify with WSAConnect call information. The dt_dll.dll decodes and returns. (8) The WinSock 2 DLL returns the final result to the application.

Typical execution through the debug/trace layer

It is important to note the sequence of events as viewed by the dt_dll.dll:

1. WSAPreApiNotify announces a call into the WinSock 2 DLL at the WSAConnect function
2. WSAPreApiNotify announces a call out of the WinSock 2 DLL to the WSPConnect function
3. WSAPostApiNotify announces a return back into the WinSock 2 DLL from the WSPConnect function.
4. WSAPostApiNotify announces a return back from the WinSock 2 DLL from the WSAConnect function.

Since the invocation of the service provider's WSPConnect function is nested within the invocation of the API-level WSAConnect function, the Pre/Post notification pair announcing the SPI activity is nested between the Pre/Post notification pair announcing the API activity. The default dt_dll.dll supplied with the SDK (see below) includes fairly straightforward logic to match "post" notifications with their corresponding "pre" notifications for each thread in spite of any intervening activity.

2.2 Procedure Specifications

2.2.1 WSAPreApiNotify

WSAPreApiNotify announces the start of processing for the WinSock 2 DLL's implementation of an API function or the start of processing for a service provider's SPI function. A notification code passed to this procedure indicates the specific function invocation that is being announced. The dt_dll.dll must implement WSAPreApiNotify. Note that WSAPreApiNotify has "C" calling sequence with a variable argument list.

```
BOOL WINAPIV
WSAPreApiNotify(
    IN  INT    NotificationCode,
    OUT LPVOID ReturnCode,
    IN  LPSTR  LibraryName,
    ...);
```

NotificationCode — Supplies a value indicating the specific function invocation that is being announced. Values for this parameter corresponding to each API and SPI function are defined in the header file. Additional values may be defined in the future to announce other WinSock 2 processing events of interest.

ReturnCode — Returns a value that should replace the return value of the announced function if the dt_dll.dll elects to skip the processing that would otherwise occur. The actual type of the value depends on the specific function being announced. This value is ignored if WSAPreApiNotify returns FALSE.

LibraryName — Supplies the name of the library invoked. For API function announcements, this is "winsock2.dll". For SPI function announcements, this is the path of the service provider DLL.

... — Supplies a variable argument list of pointers to each of the parameters of the specific function being announced. For example, for a function with three INT parameters, the variable argument list has three "pointer-to-INT" values.

Return Value If WSAPreApiNotify returns TRUE, the caller (WinSock 2 DLL) will skip the processing that otherwise would have been done and return from the original announced function. The value used as the return value of the announced function is the current value in the location pointed to by ReturnCode. If WSAPreApiNotify returns FALSE, the ReturnCode value is ignored and processing proceeds as usual.

The variable argument list is filled with pointers to each of the formal parameters that were passed to the original announced function. So for announcing a function such as connect(SOCKET s, const struct sockaddr FAR *name, int namelen), the call sequence looks something like the following:

```
BOOL  should_skip;
int   connect_return;

should_skip = (* lpWSAPreApiNotify) (
    DTCODE_connect,             // NotificationCode
    (LPVOID) & connect_return,  // ReturnCode
    "winsock2.dll",             // LibraryName
    & s,
    & name,
    & namelen);

if (should_skip) {
    return(connect_return);
}
```

This gives the WSAPreApiNotify function full access to the parameters of the function being announced *but with an extra level of indirection*. The extra level of indirection is not needed in most cases, since a typical WSAPreApiNotify function will simply log or display parameter values and return. However, since WSAPreApiNotify has pointers to the original announced function's formal parameters, WSAPreApiNotify has the ability to modify the actual parameter values before any processing begins. This gives the dt_dll.dll a very powerful tool to use in more complex debugging, diagnostic, and testing scenarios.

Similarly, the WSAPreApiNotify function has a pointer to a variable where the original announced function's return value will be stored. The dt_dll.dll can modify this variable and return TRUE to indicate that processing should be skipped for this API or SPI function. The modified return value is used as the return value from the announced function as if the value had been determined from the actual processing. Together with read/write access to the formal parameters this gives the dt_dll.dll the ability to completely replace the announced function if desired. Note that a typical dt_dll.dll that only does logging or display returns FALSE and the ReturnCode parameter is ignored.

2.2.2WSAPostApiNotify

WSAPostApiNotify announces the completion of processing for the WinSock 2 DLL's implementation of an API function or announces a return from processing by a service provider's SPI function. A notification code passed to this procedure indicates the specific function completion or return that is being announced. The dt_dll.dll must implement WSAPostApiNotify. Note that WSAPostApiNotify has "C" calling sequence with a variable argument list.

```
BOOL WINAPIV
WSAPostApiNotify(
    IN     INT    NotificationCode,
    IN OUT LPVOID ReturnCode,
    IN     LPSTR  LibraryName,
    ...);
```

NotificationCode	Supplies a value indicating the specific function completion or return that is being announced. Values for this parameter corresponding to each API and SPI function are defined in the header file. Additional values may be defined in the future to announce other WinSock 2 processing events of interest.
ReturnCode	Supplies the return value determined by function processing or returned from the service provider. Returns a value that should replace the return value from the function processing or the value returned from the service provider. The actual type of the value depends on the specific function completion or return being announced.
LibraryName	Supplies the name of the library whose processing has just completed. For API function announcements, this is "winsock2.dll". For SPI function announcements, this is the path of the service provider DLL.
...	Supplies a variable argument list of pointers to each of the parameters of the specific function being announced. For example, for a function with three INT parameters, the variable argument list has three "pointer-to-INT" values.

Return Value For announcements of API processing completion or SPI function return, the return value is ignored. The dt_dll.dll's implementation should return FALSE. The interpretation of the return value is reserved for possible future use with new NotificationCode values.

The variable argument list is filled with pointers to each of the formal parameters that were passed to the original announced function. So for announcing completion of processing for a function such as connect(SOCKET s, const struct sockaddr FAR *name, int namelen), the call sequence looks something like the following:

```
BOOL  dont_care;
int   connect_return;

connect_return = actual_connect_processing(s, name, namelen);

dont_care = (* lpWSAPostApiNotify) (
    DTCODE_connect,             // NotificationCode
    (LPVOID) & connect_return,  // ReturnCode
    "winsock2.dll",             // LibraryName
    & s,
    & name,
    & namelen);

return(connect_return);
```

As with WSAPreApiNotify above, the WSAPostApiNotify function has full access to the parameters of the function being announced but with an extra level of indirection (we realize this is not particularly useful). WSAPostApiNotify can modify any buffers or variables supplied as out parameters, but beware that it must go through an extra level of indirection to get to them.

Similarly, the WSAPostApiNotify function has a pointer to a variable where the original announced function's return value was stored. The dt_dll.dll can modify this variable to alter the return value determined from processing or returned from the service provider. Note that a typical dt_dll.dll that only does logging or display leaves ReturnCode value unmodified.

3Typical Implementation

The following pseudo-code example illustrates one way a fragment of a WSAPreApiNotify function could be implemented. The actual dt_dll.dll source code is somewhat more elaborate and differently organized since processing common to all notifications is "factored out" to reduce the source-code size. The dt_dll.dll implementation also contains additional logic not shown here to match pre/post notifications, generate log output, synchronize multiple-thread access to the log output etc. See the section on the default debug/trace DLL, below, for more details.

```
BOOL WINAPIV
WSAPreApiNotify(
    IN  INT    NotificationCode,
    OUT LPVOID ReturnCode,
    IN  LPSTR  LibraryName,
    ...)
{
    va_list  vl;     // used for variable arg-list parsing

    // Prepare to parse variable argument list
    va_start(vl, LibraryName);

    // Determine which function is being announced
    switch (NotificationCode) {

        case DCODE_connect:
            // handle a "connect" announcement
            {
                // declare some local variables for parameters
                // and return value
                int                    * RetVal = (int *) ReturnCode;
                SOCKET                 * s;
                struct sockaddr FAR * * name;
                int                    * namelen;

                // parse parameters out of variable argument list
                s       = va_arg(vl, SOCKET *);
                name    = va_arg(vl, struct sockaddr FAR * *);
                namelen = va_arg(vl, int *);

                // construct a log entry including one of the
                // parameter values.  Note the "extra" level
                // of indirection when accessing the "namelen"
                // parameter value.
                wsprintf(
                    Buffer,
                    "Library: %s, Function: connect, namelen=%d\n",
                    LibraryName,
                    * namelen);

#if defined(DEMO_PARAM_MODIFICATION)
                // illustrate parameter value modification
                * namelen = sizeof(sockaddr);
#endif

#if defined(DEMO_EXECUTION_SKIPPING)
                // illustrate return-value substitution and
                // execution skipping
                * RetVal = SOCKET_ERROR;
                return(TRUE);  // skip further execution
#endif
            }
            break;

    } // switch (NotificationCode)
```

```
        return(FALSE);  // allow execution to proceed normally

    }  // WSAPreApiNotify
```

4DT_DLL.H header file (excerpt)

The following is an excerpt from the dt_dll.h header file that defines the syntax of the interface between the WinSock 2 DLL and the auxiliary dt_dll.dll. The long list of manifest constants for the function announcement codes is mostly omitted for brevity.

```
/*++

Copyright (c) 1995 Intel Corp

File Name:

    dt_dll.h

Abstract:

    Definitions, constants, and data structures for the Debug/Trace
    DLL for the WinSock2 DLL.  Please refer to the design spec for
    more information.

Author:

    Michael A. Grafton

Revision History:

    10-August-1995 mike_grafton@ccm.jf.intel.com
       -- first working draft

--*/

#ifndef _DT_DLL_H
#define _DT_DLL_H


BOOL WINAPIV
WSAPreApiNotify(
    IN  INT    NotificationCode,
    OUT LPVOID ReturnCode,
    IN  LPSTR  LibraryName,
    ...);

BOOL WINAPIV
WSAPostApiNotify(
    IN     INT    NotificationCode,
    IN OUT LPVOID ReturnCode,
    IN     LPSTR  LibraryName,
    ...);



// API function codes for Pre/PostApiNotify functions

#define DTCODE_accept 1
#define DTCODE_bind 2

// skipping ...

#define DTCODE_WSAAccept 22
#define DTCODE_WSAAsyncSelect 23
#define DTCODE_WSACancelBlockingCall 24

// skipping ...
```

```
#define DTCODE_WSPAccept 73
#define DTCODE_WSPAsyncSelect 74
#define DTCODE_WSPBind 75
#define DTCODE_WSPCancelBlockingCall 76
#define DTCODE_WSPCleanup 77

// skipping...

#define DTCODE_WPUCloseEvent 103
#define DTCODE_WPUCloseSocketHandle 104

// skipping...

#endif _DT_DLL_H
```

5The Default Debug/Trace DLL

5.1Overview

The default debug/trace DLL supplied with the WinSock2 SDK provides a simple logging mechanism for API and SPI boundary crossings, as well as a base implementation from which developers can create more complicated debugging and tracing schemes. This section of the document describes both the functionality of the default DLL -- i.e. what you can expect from it "out of the box" -- as well as the design and implementation of the module. Developers can use this information to quickly and easily modify the DLL to suit their particular debugging needs.

5.2Functionality

The default debug/trace DLL, when loaded by the specially instrumented WinSock 2 DLL, simply logs some key information about each API/SPI call or return event, as reported by WSAPre/PostApiNotify. Each line of output corresponds to one such event, and each contains the following information:

- *line number* of the output.
- *thread ID* of the application thread that made the call.
- *function call ID* to keep track of nested calls (see below for further explanation).
- *function name* of the API or SPI.
- *event type*, i.e. whether this was a call event or a return event.

The *function call ID* field of the debugging output helps the reader of the listing quickly identify the matching *xxxx() called* and *xxxx() returned* lines. This will be helpful when API functions may be nested several levels deep and the matching lines have several lines of intervening WinSock 2 activity in between them. As you can see below, API and SPI functions are treated exactly the same, and thus all SPI calls are nested within a corresponding API call. It should also be noted that the function call IDs are local to the calling thread; as described below in the implementation notes, each thread has it's own separate stack of function call numbers to keep track of this nesting.

Here is the first several lines of a listing created by the WS2 Chat sample application:

```
(0) Log initiated: 10-28-1995, 17:6:5
(1) Process ID: 0xFFFEFEB5   Thread ID: 0xFFFEFA15
(2) TID: 0xFFFEFA15   Function call: 0   WSAStartup() called.
(3) TID: 0xFFFEFA15   Function Call: 0   WSAStartup() returned.
(4) TID: 0xFFFEFA15   Function call: 1   WSAEnumProtocols() called.
(5) TID: 0xFFFEFA15   Function Call: 1   WSAEnumProtocols() returned.
(6) TID: 0xFFFEFA15   Function call: 2   WSAEnumProtocols() called.
(7) TID: 0xFFFEFA15   Function Call: 2   WSAEnumProtocols() returned.
(8) TID: 0xFFFEFA15   Function call: 3   WSASocket() called.
(9) TID: 0xFFFEFA15   Function call: 4   WSPSocket() called.
(10) TID: 0xFFFEFA15   Function Call: 4   WSPSocket() returned.
(11) TID: 0xFFFEFA15   Function Call: 3   WSASocket() returned.
(12) TID: 0xFFFEFA15   Function call: 5   bind() called.
```

```
(13) TID: 0xFFFEFA15   Function call: 6   WSPBind() called.
(14) TID: 0xFFFEFA15   Function Call: 6   WSPBind() returned.
(15) TID: 0xFFFEFA15   Function Call: 5   bind() returned.
(16) TID: 0xFFFEFA15   Function call: 7   WSAAsyncSelect() called.
(17) TID: 0xFFFEFA15   Function call: 8   WSPAsyncSelect() called.
(18) TID: 0xFFFEFA15   Function Call: 8   WSPAsyncSelect() returned.
(19) TID: 0xFFFEFA15   Function Call: 7   WSAAsyncSelect() returned.
(20) TID: 0xFFFEFA15   Function call: 9   listen() called.
(21) TID: 0xFFFEFA15   Function call: 10   WSPListen() called.
(22) TID: 0xFFFEFA15   Function Call: 10   WSPListen() returned.
(23) TID: 0xFFFEFA15   Function Call: 9   listen() returned.
```

When the debug/trace DLL is loaded, it brings up a dialog box asking the user to choose what type of output he would like. The choices include file, debug window, or debugger output. It should be noted that the debug window is exceptionally slow. If using the window, a workaround to this large performance penalty is to minimize the window and pop it up after the activity in question has occurred.

5.3Implementation

The following is a brief description of the implementation of the default debug/trace DLL. It is intended as a “roadmap” to the code. Please refer to the code itself for more detailed comments.

5.3.1User Interface

As noted above, the default debug/trace DLL, when first loaded, pops up a dialog box prompting the user to choose an output method. The choices are window only, window and file, file only, and debugger output. The user must choose one such method; choosing one of the middle two methods brings up a file selection dialog box to specify a filename.

The debug output window, if used, is a simple subclassed edit control. It disallows editing, cutting or pasting, but will allow users to copy text to the clipboard (we added that feature solely to paste the above listing). If the debug window option is chosen, the debug/trace DLL launches a separate thread to create and service the messages sent to the window.

5.3.2Thread Stacks

The debug/trace DLL uses a fairly straightforward stack class to keep track of function call ID numbers. When the DLL is notified of the application creating a new thread, it creates an instance of a cstack object and stores a pointer to the object in thread local storage (TLS). This is a classic stack with one small difference; the *push()* method, rather than taking a pointer to a piece of data, takes no parameters. Instead, it simply pushes a private variable called *counter* onto the stack, and increments it’s value (*counter* is initialized to zero). Pop simply pulls the integer off the stack.

When an API function is called, WSAPreApiNotify retrieves it’s the stack object, records the counter, and then performs a push operation. The output is then displayed. Any number of nested, intervening API or SPI functions can now take place, pushing and popping their own function call IDs, but they will all have returned by the time WSAPostApiNotify is called for this function in this thread. We then pop the same counter off the stack and print that number in the output. Note that if WSAPreApiNotify determines that it wants to short-circuit the API or SPI function by returning FALSE, it pops the counter off the stack, but no output is displayed. Thus, it is possible for there to be no matching *return* notification for some *call* events.

5.3.3Handler Functions

Both WSAPre- and WSAPostApiNotify can be called with over 100 different notification codes. To handle all the different possibilities, the debug/trace DLL uses a slew of *handler functions*, one for each possible notification code. These functions are contained in handlers.cpp. Rather than using a giant switch statement to call the appropriate handler function, we use a table of function pointers which is initialized during the DLL_PROCESS_ATTACH segment of DllMain(). WSAPre/PostApiNotify both use the same method to call the handler functions -- the notification code is used to index into the pointer table and that pointer is dereferenced with a common set of parameters. These parameters mostly duplicate the information passed to WSAPre/PostApiNotify (i.e. a pointer to the API function’s return

value, and a pointer to the variable-length parameter list). A few additional parameters give other information to the handler functions, such as a boolean to distinguish the calling functions from each other, and a buffer into which the handler may dump the rest of the (function-specific) output line.

Here is a typical handler function:

```
BOOL CALLBACK
DTHandler_accept(
    IN     va_list vl,
    IN OUT LPVOID  ReturnValue,
    IN     LPSTR   LibraryName,
    OUT    char    *Buffer,
    IN     int     Index,
    IN     int     BufLen,
    IN     BOOL    PreOrPost)
{
    SOCKET *RetVal = (SOCKET *)ReturnValue;
    SOCKET *s = va_arg(vl, SOCKET *);
    struct sockaddr FAR **addr = va_arg(vl, struct sockaddr FAR **);
    int FAR **addrlen = va_arg(vl, int FAR **);

    wsprintf(Buffer + Index, "accept() %s.\r\n",
             PreOrPost ? "called" : "returned");
    DTTextOut(DebugWindow, LogFileHandle, Buffer, OutputStyle);
    return(FALSE);
}
```

It doesn't do much; basically, it uses the va_arg() macro to strip off pointers to the original API function's parameters, and puts these pointers into local variables. If the developer wants to examine the parameters passed into any particular API/SPI call, he is encouraged to set breakpoints in these handler functions; at that point, he can use the debugger to dereference the local variables that point to the parameters (don't forget about that extra level of indirection!). After stripping off the parameters, the handler functions print some output into the given buffer and return FALSE, indicating to WSAPreApiNotify to continue with execution of the function (WSAPostApiNotify ignores this return value, of course). It is here, in these handler functions, that the more ambitious developer could add her code. Type-specific formatting procedures could be written to output the API parameters in a readable way. Or some processing may need to occur if indeed the API function is to be short-circuited. All these possibilities are left as exercises for the reader.

5.3.4Thread Synchronization

There are a few minor thread synchronization issues addressed in the default implementation of dt_dll.dll. First, the bodies of WSAPre/PostApiNotify are protected from simultaneous use of the global text buffer by a Critical Section variable. Another solution would have been to give each thread its own buffer in thread local storage.

Secondly, an event is used to ensure that the two line of header information printed at startup are not printed after (or in the middle of) text from API function calls and returns. This was happening because the debug window is created by another thread, and it is entirely possible for the original, intializing thread to return to the application, call an API function, and print some output well before the debug window is created by the new thread and the initialization information printed out.

5.3.5Error Preservation

Because WSAPre/PostApiNotify have system calls in them, it is possible for them to change the value that would be returned by WSAGetLastError() or GetLastError(). This could be a very bad thing -- for instance, an error set by the real API function would be lost after WSAPostApiNotify writes over it. To counter this, these functions have some code in them to preserve the error code as it exists when they are invoked. However, it also possible that one of the handler functions may wish to change the error using SetLastError(), especially if a short-circuit is taking place. WSAPre/PostApiNotify both exit with the

error code being either the same value it was upon invocation, or, if the handler function changed the value, the new value.