Table Of Contents

Chapter 1

The Debugging Application Programming Interface

Randy Kath
Microsoft Developer Network Technology Group

November 5, 1992

Abstract

This article demonstrates how the debugging support in the Microsoft Windows Application Programming Interface (API) can be used by developers to create custom debugging applications that behave exactly the way they want, including any specific features they desire. Specifically, this article discusses the following topics:

- Exploring the built-in debugging support, including debug events and debug functions
- Looking at the relationship between a debugger and the process being debugged
- Representing information about a process being debugged
- Using event objects for communicating between debugger threads
- Managing the debugger's graphical user interface (GUI)
- Responding to user commands in debug threads
- Controlling the threads of a process being debugged
- Accessing thread context information from threads of a process being debugged
- Terminating and exiting a process being debugged
- Calling debug functions from a process being debugged
- Expanding on this debugger model

Each of the key concepts presented is supported with code segments extracted from a sample debugging application called DEBUGAPP.EXE, whose source is included with this article. Th sample application stands on its own as a multiprocess debugging application, or its source code can be used as the framework for a more elaborate custom debugger.

Introduction

Of the time a programmer spends developing an application, a large portion is usually spent debugging that application. Consequently, developers rely on third-party debuggers almost much as they do editors. Unlike editors, however, debuggers can rarely be customized much. If a debugger lacks an important feature or behaves in an unusual or irritating way, develop are simply forced to put up with it.

Windows appears ready to break this cumbersome debugging tradition with new, built-in debugging support included as part of the standard application programming interface (API). I developers have the flexibility to create their own personal debugger that behaves exactly the way they wish. And once that is complete, having the source code to that debugger makes all the more flexible. Developers can repeatedly add new features directly to the source code of the debugger as needed in the future.

The debugging architecture consists of a clean, relatively straightforward set of functions and events that make it useful to all developers, not just debugger builders. Simply being famili with Windows and, more importantly, the Windows API is enough to build an understanding of the debugging support. The debugger sample application described in this article require only about three weeks for implementation, including the time required to make sense of the API.

Exploring the Built-In Debugging Support

DebugApp, the sample application associated with this article, is a high-level debugger that meets a number of important requirements for a debugger. It can debug multiple applicatio simultaneously, controlling the execution of each process being debugged and presenting feedback about noteworthy events that occur in each of the processes. It can also be used to the 2 gigabyte (GB) heap space of each process for learning how memory allocations are organized. These are only some of the capabilities that could be added to a debugger. To get a better feel for what capabilities can be implemented in a debugger, you will need to gain some knowledge of how the debugging API works.

Debug Events

Debug events are the objects of interest to a debugger—they're noteworthy incidents that occur within the process being debugged, causing the kernel to notify the debugger when the occur. As defined by Windows, debug events are one of the following:

- CREATE_PROCESS_DEBUG_EVENT occurs before a new process being debugged initializes or at the time a debugger attaches to an active process.
- EXIT_PROCESS_DEBUG_EVENT occurs when the process being debugged exits.
- CREATE_THREAD_DEBUG_EVENT occurs when the process being debugged creates a new thread.
- EXIT_THREAD_DEBUG_EVENT occurs when a thread in the process being debugged exits.
- LOAD_DLL_DEBUG_EVENT occurs when the process being debugged loads a DLL (either explicitly or implicitly).
- UNLOAD_DLL_DEBUG_EVENT occurs when the process being debugged frees a DLL.
- EXCEPTION_DEBUG_EVENT occurs when an exception occurs in the process being debugged.
- OUTPUT_DEBUG_STRING_DEBUG_EVENT occurs when the process being debugged makes a call to the **OutputDebugString** function.

When a debug event is generated, it comes to the debugger packaged in a **DEBUG_EVENT** structure. The structure contains fields that represent an event code (listed above), the proces ID of the process that generated the debug event, the thread ID of the thread executing when the debug event occurred, and a union of eight structures, one for each of the different events. This structure provides information necessary for the debugger to distinguish between different debug events and process them individually based on their unique requirements The **DEBUG_EVENT** structure is:

```
typedef struct _DEBUG_EVENT {   /* de */
    DWORD dwDebugEventCode;
    DWORD dwProcessId;
    DWORD dwThreadId;
    union {
        EXCEPTION_DEBUG_INFO Exception;
        CREATE_THREAD_DEBUG_INFO CreateThread;
        CREATE_PROCESS_DEBUG_INFO CreateProcess;
        EXIT_THREAD_DEBUG_INFO ExitThread;
        EXIT_PROCESS_DEBUG_INFO ExitProcess;
        LOAD_DLL_DEBUG_INFO LoadDll;
        UNLOAD_DLL_DEBUG_INFO UnloadDll;
        OUTPUT_DEBUG_STRING_INFO DebugString;
    } u;
} DEBUG_EVENT, *LPDEBUG_EVENT;
```

Debug Functions

Two functions, **WaitForDebugEvent** and **ContinueDebugEvent**, are designed specifically for managing debug events as they occur in a process being debugged. These functions perm debugger to wait for a debug event to occur, suspend execution of the process being debugged, process each debug event, and resume execution of the process being debugged wher finished. Additionally, while the process being debugged is suspended, the debugger is able to change the thread context information of each of its threads. This ability provides a mechanism through which the debugger can alter normal execution of one or more threads in the process being debugged. It can, for example, change the instruction pointer for a thre to refer to an instruction at a new location. Then, when the thread resumes execution, it begins executing code at the new location. A discussion of this subject is presented later in the "Accessing Thread Context Information from Threads of a Process Being Debugged" section.

When called, the **WaitForDebugEvent** function does not return until a debug event occurs in the process being debugged or a time-out value is reached. The time-out value is set as or of the parameters in the function. The function returns TRUE if an event occurs, and FALSE if the function times out.

DEBUG.C

```
while (TRUE)
    {
    /* Wait for 1/10 second for a debug event. */
    if (WaitForDebugEvent (&de, (DWORD)100))
        {
        switch (de.dwDebugEventCode)
            {
            case EXCEPTION_DEBUG_EVENT:
                ProcessExceptionEvent (&de);
                break;

            case CREATE_PROCESS_DEBUG_EVENT:
                ProcessCreateProcessEvent (&de);
                break;

            .
            .
            .

            case default:
                ProcessUnknownDebugEvent (&de);
                break;
            }
        ContinueDebugEvent (de.dwProcessId,
                            de.dwThreadId,
                            DBG_CONTINUE);
        }
    else
        /* Perform periodic debugger responsibilities. */
    }
```

In the code fragment above, notice that **WaitForDebugEvent** returns a Boolean value where a value of TRUE indicates that a debug event occurred and FALSE means that the function timed out. This example waits for 1/10 second for a debug event to occur, but if an event does not occur in that amount of time, it uses the time-out indicator to perform some other periodic debugger responsibilities. Since time-outs only happen when there are no debug events, this time is analogous to the idle time a CPU observes. Specifically, the DEBUGAPP.EXE sample uses this time to communicate with the main debugger thread in order to process user commands.

When a debug event occurs, execution of that process is suspended until the debugger calls the **ContinueDebugEvent** function. Consequently, all threads in the process are suspended while the debugger is processing the debug event. A debugger needs to be mindful of the performance impact this will impose on the process being debugged. A good design, in this c is one that allows the process being debugged to continue as soon as possible after a debug event occurs. On the other hand, when the **WaitForDebugEvent** function times out, the process being debugged is able to run concurrently with the debugger process and no performance impact is observed. Any debug events that occur during the time-out period are queued until the debugger calls the **WaitForDebugEvent** function again. So, no need to worry—there is no possibility of missing a debug event because of this circumstance.

To call the **ContinueDebugEvent** function, the debugger must supply as parameters the thread ID and process ID of the process that generated the last debug event. Both the process a thread IDs are included as part of the **DEBUG_EVENT** structure with each debug event. They're also returned as part of the **PROCESS_INFORMATION** structure filled out by the **CreateProcess** function when starting a process for debugging. A debugger can attach to an active process for debugging, but an ID for that process is required prior to the attachment Once the debugger has attached, the thread ID is retrieved from the **DEBUG_EVENT** structure.

The Relationship Between a Debugger and the Process Being Debugged

For one application (process) to become the debugger of another, it must either create the process as a debug process or attach to an active process. In both cases, a parent/child relationship is established between the debugger and the process being debugged. If the debugger process ends without ending the process being debugged, the latter process is terminated by the system. If the process being debugged ends, the debugger process becomes a normal process, able to start or attach to another process to debug.

When the parent/child association is made, the debugger thread responsible for establishing this dependence—the thread that attaches or starts the process to be debugged—becomes the parent thread to the process being debugged. Only the parent thread of a process being debugged is capable of receiving debug events for that process. Consequently, the parent thread is the only thread able to call the **WaitForDebugEvent** and **ContinueDebugEvent** functions. If another thread calls these functions, they simply return FALSE. The basis for the design of the sample application, DEBUGAPP.EXE, is inherent in this requirement.

Creating a process to debug

To create a process for debugging, the debugger calls the **CreateProcess** function with the *fdwCreate* parameter set to either DEBUG_PROCESS or DEBUG_ONLY_THIS_PROCESS. DEBUG_PROCESS sets up the parent/child relationship so that the debugger will receive debug events from a process being debugged and any other processes created by that process. this case, processes created by the process being debugged are automatically debugged by the same debugger. Using DEBUG_ONLY_THIS_PROCESS restricts debugging to the immedia process being debugged only. Processes created by the process being debugged are normal processes that have no debugging relationship established with any other process.

An abbreviated definition of **CreateProcess** is found below. A complete definition of the **CreateProcess** function is in the Platform SDK.

```
BOOL CreateProcess(
   LPCTSTR lpszImageName,        /* address of image file name */
   LPCTSTR lpszCommandLine,      /* address of the command line */
   LPSECURITY_ATTRIBUTES lpsaProcess,  /* optional process attrs */
   LPSECURITY_ATTRIBUTES lpsaThread,   /* optional thread attrs */
   BOOL fInheritHandles,         /* new process inherits handles? */
   DWORD fdwCreate,                          /* creation flags */
   LPVOID lpvEnvironment,    /* address of optional environment */
   LPTSTR lpszCurDir,        /* address of new current directory */
   LPSTARTUPINFO lpsi,                /* address of STARTUPINFO */
   LPPROCESS_INFORMATION lppi);  /* address of PROCESSINFORMATION */
```

CreateProcess includes several parameters for establishing the environment of the process being debugged, passing command-line arguments to the process being debugged, specifyi security attributes about the process, and indicating how to start the application. The LPPROCESS_INFORMATION parameter is used for receiving information about the process being started. Specifically, it consists of the process and thread IDs of the process being debugged that are used in **ContinueDebugEvent** and handles to both the process being debugged ar its initial thread.

Attaching a debugger to an active process

A debugger can attach to any existing process in the system, providing that it has the ID of that process. Through the **DebugActiveProcess** function, a debugger can establish the same parent/child relationship described earlier with active processes. In theory, then, the debugger should be able to present a list of active processes to the user, allowing them to select wh one they would like to debug. Upon selection, the debugger could determine the ID of the selected process and begin debugging it by means of the **DebugActiveProcess** function. All is needed then is a mechanism for enumerating the handles of each active process in the system. Unfortunately, Windows provides no support for determining the ID of other processes the system. While this seems to render the function useless, it really just limits the way it can be used. On its own, a debugger process cannot determine the ID of other active processes, with some help from the system it can get the ID of a specific process in need of debugging.

Built into Windows is the ability for the system to start a debugger upon the occurrence of an unhandled exception in a process. When such an exception occurs, Windows starts the def debugger and passes it the ID of the process to be debugged as a command-line parameter. Windows also passes an event handle as a second command-line parameter. The debugger then calls the **DebugActiveProcess** function using the process ID passed as a command-line parameter. Once the debugger has established the debugging relationship with the offendi process, it signals that it is ready to begin debugging by calling the **SetEvent** function on the event handle. At that time the system releases control of the process to the debugger.

Note

The system administrator can change the default debugger in Windows to be any third-party debugger or any other debugger you choose. To do this, an entry must be added to the WIN.INI file as indicated:

```
[AeDebug]
  Debugger = ntsd -d -p %d -e %d -g
```

To change from the standard default debugger, ntsd, to one of your choosing, simply replace the name ntsd with the name of the debugger you want. Also, make sure that the debugger resides in a directory in the path.

An application can take advantage of this built-in behavior as a way of invoking a debugger to debug itself if, and only if, a special circumstance occurs. For example, an application coul be executing normally when a condition occurs that warrants debugging. The application could then start the debugger by calling the **CreateProcess** function similar to the way it was described above, only not as a process to debug. Also, the process needs to pass its ID as a single command-line parameter to the debugger process. The debugger is then started and passed an ID of the process to debug. One requirement of this technique is that the debugger must differentiate between the two ways that it can be created. The difference is that, whe the system starts the debugger, there is a second command-line parameter representing a valid wait event that the debugger must eventually signal. When the process wishing to be debugged creates the debugger, there is no second command-line parameter and no wait event to signal.

DEBUGAPP.EXE, A Sample Implementation

Combining a knowledge of the debugging API and some idea of the features a custom debugger should have is important for devising the architecture of a debugger application. For example, Figure 1 portrays the architecture that was used in implementing the sample custom debugger, DebugApp.

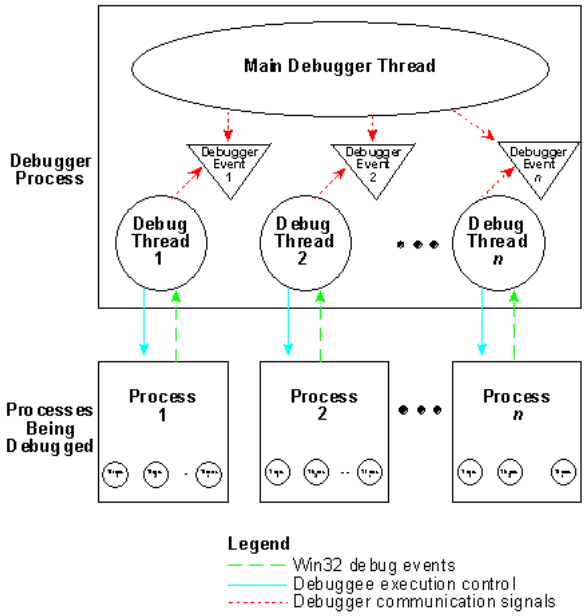

Figure 1. DebugApp's architecture. A main debugger thread manages the debugger interface, while one thread exists for each process being debugged.

DebugApp consists of one main thread and one or more debug threads. The main thread is responsible for handling the entire graphical user interface (GUI) of the debugger. Each of th other debug threads is created and destroyed as new debug sessions are started and ended. This structure serves to contain the debug-specific functionality in the debug threads and th GUI-specific functionality in the main thread. Embedded, then, is a layer of encapsulation that promotes maintenance and revision of the source code. Further, the source code for these types of threads is located in separate source modules, DEBUG.C and MAIN.C, making it easier to go back and add functionality to one part of the system without having an adverse imp on the other.

Once the underlying structure for the debugger is in place, the next issue is how to represent the debugger and, more specifically, each process being debugged in a single Windows® interface. This implementation uses multiple document interface (MDI) because MDI supports multiple process debugging simultaneously and offers basic multiple window managemen functionality for free. It turns out MDI is also a good selection because each MDI child window can be used as a separate object capable of maintaining its own private data structures. In that case, writing the interface code to support multiple processes for debugging is no more work than writing it for one.

A final consideration concerns the bells and whistles that should be added to the debugger. DEBUGAPP.EXE need only meet the basic requirements stated earlier in this article, but at thi point many more features and behaviors could easily be applied to the underlying debugger architecture. Specifically, DEBUGAPP.EXE implements support for controlling the execution individual threads in each of the processes being debugged. It also records all debug events chronologically and provides a mechanism for saving this log to a file for post-mortem revie

Representing Information About a Process Being Debugged

DEBUGAPP.EXE uses a single data structure for representing all of the information associated with a process being debugged and its threads. The structure is a simple, singly-linked list where the header (**DBGPROCESS** structure) represents the information of the process being debugged, and each node (**DBGTHREAD** structure) in the list represents information about each thread in the process. Figure 2 depicts this information and how it is organized.

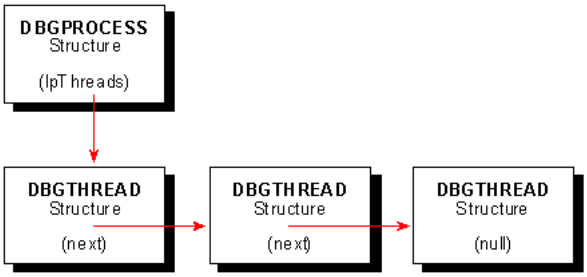

Figure 2. A linked list stores information about the process being debugged and its threads.

As the process being debugged creates and destroys new threads, the linked list grows and shrinks dynamically. Storage for this data structure is allocated within the debugger process the form of a *serialized heap*. Since all threads in the debugger process have access to a shared heap, their access must be serialized to allow one thread to finish accessing the heap bef another thread begins accessing it. Windows provides serialized heaps as a mechanism to prevent access contention between threads that share a heap.

In addition to information about the process being debugged, Windows uses the linked-list header as a place to store information that is communicated between the main thread and th debug thread. Such information includes the MDI child window handle representing a specific debug session, the module path and filename of the process being debugged, a thread number for control information, and a handle to the heap itself, which is used both for destroying the heap when the debug session ends and allocating additional linked-list nodes. Bot the linked-list header and node structures are presented in the code below.

DEBUG.H

```
// Define structures for debugging processes and threads.
typedef struct DBGTHREAD    *LPDBGTHREAD;
typedef struct tagDbgThread
    {
    HANDLE                  hThread;
```

```
    LPTHREAD_START_ROUTINE    lpStartAddress;
    BOOL                      bfState;
    LPDBGTHREAD               Next;
    }DBGTHREAD;

typedef struct tagDbgProcess
    {
    HANDLE       hDbgHeap;
    DWORD        dwProcessID;
    DWORD        dwThreadID;
    HANDLE       hProcess;
    HANDLE       hFile;
    LPVOID       lpImage;
    DWORD        dwDbgInfoOffset;
    DWORD        nDbgInfoSize;
    DBGTHREAD    *lpThreads;
    HWND         hWnd;
    int          ProcessPriority;
    HANDLE       hThread;
    int          ThreadPriority;
    char         szModule[MAX_PATH];
    }DBGPROCESS;
```

DebugApp's main thread creates a serialized heap for each debug thread prior to creating the thread, and a pointer to the heap is passed to the debug thread at creation time. Both the debug thread and the main thread, each keeping a separate copy of the pointer to the structure, maintain the heap independently. The main thread stores its heap pointer in the windov extra bytes of the MDI child window responsible for the process being debugged. This way each window can keep track of its own structures, independent of every other window.

Because the debug thread is implemented as a function (similar to a **WinMain** function) that returns only when the thread has completed, and because each thread has its own stack, ea debug thread can keep the pointer to the heap as an automatic variable on its stack. In this way, each debug thread can access its pointer as a local variable because it resides permaner on the stack for that thread. When the thread exits (returns from the function), its stack is deallocated and the pointer is automatically freed. Keep in mind that the heap itself is not freec this action; it must be explicitly freed through a call to **DestroyHeap**. When and where the heap gets deallocated is discussed in the "Terminating and Exiting a Process Being Debugged section later in this article.

Using Event Objects for Communicating Between Debugger Threads

Because, as described above, the debug thread and the main thread both share access to the same heap, some type of synchronization is necessary for at least creating and destroying t heap. Also, because both the debug thread and the main thread independently perform functions that occasionally must be coordinated, it stands to reason that a debugger needs a mechanism to communicate between threads. DEBUGAPP.EXE uses wait event objects for this purpose.

Windows uses wait events as a signaling mechanism much like a traffic signal, except there's no yellow light. A wait event represents either a signaled or unsignaled state. A thread can w for one or more of these events to become signaled and then perform some related action. While waiting for an event to become signaled, a thread is idle. Consequently, events are per synchronization objects that allow one thread to wait for a signal from another before performing a specific task. In DEBUGAPP.EXE, the main thread opens a set of event objects, one fo each debug thread, and stores the handles in a segment of global memory. The memory is treated as an array of object handles called *lpDbgEvents* and is kept in the window extra bytes the responsible child window, along with the linked-list structure mentioned in the previous section.

Threads cannot share handles to event objects, so the debug thread must open its own array of handles to access the same event objects. The strategy so far is reasonable, but a potenti problem lurks here. Since the debug thread must open handles to the same event objects, it has to do so by referring to these objects by name. Windows provides support for naming objects when they are created and referring to objects by name when they are opened for exactly this purpose. Consequently, the wait event objects must be named so that both the ma thread and the debug thread can refer to identical objects. The problem arises when you start a second debug session. The second debug session requires a unique set of wait event obj —objects that can be referred to by both the debug thread and the main thread, but that also must be distinguishable from the first debug session's objects.

To solve this problem, DEBUGAPP.EXE uses the process ID of each process being debugged as part of the name used to identify each object, as in the following example.

DEBUG.C

```
/* Local function creates debug event objects for thread */
/* synchronization. */
BOOL CreateDebugEvents (
    LPHANDLE    lpDbgEvents,
    DWORD       dwProcessID)
{
    char    szEvent[MAX_PATH];
    char    Buff[15];


    LoadString (GetModuleHandle (NULL),
                IDS_DBGEVNTACTIVE,
                szEvent,
                sizeof (szEvent));
    strcat (szEvent, itoa (dwProcessID, Buff, 10));
    if (!(lpDbgEvents[DEBUGACTIVE] = CreateEvent (NULL,
                                                  TRUE,
                                                  FALSE,
                                                  szEvent)))
        return FALSE;
.
.
.
```

Each wait event object is given a static name that is stored in a string resource table. Then, when creating a new event or opening a new handle to an existing event, the ID of the proces being debugged is appended to the end of the static string. Together the static string and the process ID uniquely identify an object belonging to a specific process.

Because each thread is responsible for storing its own array of wait event handles, each thread creates its own handles independently. Fortunately, Windows is robust enough that no synchronization is needed for this process. In fact, both threads can make the same call to **CreateEvent** using the same object name, but only the first call will actually create new object The second call will return a valid handle to the same object. For that reason, both threads use one function (**CreateDebugEvents**) to retrieve valid handles without regard to which one calls first. The debug thread stores its debug event handle array on the stack as an automatic variable.

Note

In an effort to make the code in DebugApp more readable, I defined constants to represent array indexes by name rather than number. Refer to DEBUG.H to find the array index value that corresponds to a specific wait event handle.

In addition to the array of wait events used for communication between the two threads, DebugApp uses two other wait events, one for synchronizing startup and one for shutdown of t debug thread. These two event objects need only be created and used for a relatively short duration, so no accommodation is needed for them. Instead, they are created, freed, and released, all within the context of a single window message in the main thread. In both cases they are used as in the following example.

DEBUG.C

```
/* Create initialize event. */
LoadString (GetModuleHandle (NULL),
            IDS_DBGEVNTINITACK,
            szEvent,
            MAX_PATH);
hEvent = CreateEvent (NULL, TRUE, FALSE, szEvent);

/* Create debug thread. */
if (!(CreateThread ((LPSECURITY_ATTRIBUTES)NULL,
                    4096,
                    (LPTHREAD_START_ROUTINE)DebugEventThread,
                    (LPVOID)lpDbgProcess,
                    0,
                    &TID)))
    return NULL;

/* Wait for debugger to complete initialization before opening */
/* debug events. */
WaitForSingleObject (hEvent, INFINITE);
CloseHandle (hEvent);
```

First, the main thread creates an event by name with an initial value of FALSE. Then, it starts the debug thread and waits for the object it created. At this point, the thread stops execution until the wait event becomes TRUE, its signaled state. Meanwhile, the debug thread starts execution at the same time. The following code fragment shows how the debug thread signals same wait event, identified by a common name, once it has completed its initialization.

DEBUG.C

```
/* Create process to be debugged. */
.
.
.
/* Signal completion of initialization to calling thread. */
LoadString (GetModuleHandle (NULL),
            IDS_DBGEVNTINITACK,
            szEvent,
            MAX_PATH);
hEvent = OpenEvent (EVENT_ALL_ACCESS, FALSE, szEvent);
SetEvent (hEvent);
CloseHandle (hEvent);
```

The main thread is able to continue execution after the event is signaled. Then, it is free to release the event object because the synchronization is complete. A similar wait event is used synchronization when the debug thread shuts down.

Managing the Debugger's Graphical User Interface

When DEBUGAPP.EXE begins, the first thread in the process gets started. This thread behaves exactly like a basic, single-threaded MDI Windows-based application. It registers window classes for the frame and debug windows, creates the frame and MDI client windows, and initializes application-specific data. When complete, the thread enters a continuous **GetMessa** loop, awaiting commands from the user.

When the command is sent to load a process for debugging, the main thread first calls the **GetOpenFileName** common dialog routine, validates the selected filename, and informs the client to create a new child window. The MDI client then creates the new child window, allowing it to perform its own window initialization.

The following initialization is performed during the WM_CREATE message of the debugger window:

1. The child window creates an edit control, used for recording debug information for this debug process, that completely fills its client area.
2. The child window allocates a segment of global memory for storing the array of wait event object handles.
3. The child window calls the **StartDebugger** function to create the debug thread and the process for debugging.

The segment of global memory is passed as a parameter and returned with the array filled with valid event handles. The **StartDebugger** function also returns a pointer to the serialized heap for this debugger. Both of these pointers are then placed in window extra bytes for this window. The new child window then returns—eventually back to the frame window where t command to load the process for debugging was originally sent—permitting the main thread to continue executing in support of the graphical user interface.

All subsequent file-loading commands work in exactly the same way, permitting the user to load simultaneously as many processes for debugging as the system can accommodate, give the amount of resources available. Other menu commands are distributed as appropriate by the frame window. Some of the commands are handled by the MDI client window, while oth are processed only by the frame window. Still other commands, like **View Thread** and **View Process**, are intended for the debug window that is currently active. The frame distributes th messages directly to the active debugger window.

Most commands intended for a specific debugger window involve communication between that window and the corresponding debug thread. In these cases, the debugger window sig a wait event for the debug thread. The debug thread can then act upon that event the next time it waits for it. Once the event has been signaled, the main thread simply returns back to message loop for the next user command. Since the debug thread has access to the window handle of the debug window in its **DBGPROCESS** data structure, it is able to submit data to edit control directly. This permits the user free access to commands without having to wait for any prolonged processing on the part of the debug thread.

Responding to User Commands in Debug Threads

Besides handling debug events in the process being debugged, the debug thread also handles all user commands once they have been signaled as wait events in the debug window. To accommodate user command events from the main thread and still be able to debug the process, the debug thread implements a multiple-object wait loop.

DEBUG.C

```
while (TRUE)
    {
    int    nIndex;

    /* Wait for debugger active. */
    switch (nIndex = WaitForMultipleObjects (nDEBUGEVENTS,
                                             hDbgEvent,
                                             FALSE,
                                             INFINITE))
        {
        case CLOSEDEBUGGER:
            {
            int    i;

            /* Terminate process being debugged. */
            TerminateProcess (lpDbgProcess->hProcess, 0);

            /* Signal close command acknowledged event. */
            LoadString (GetModuleHandle (NULL),
                        IDS_DBGEVNTCLOSEACK,
                        szEvent,
                        MAX_PATH);

            hEvent = OpenEvent (EVENT_ALL_ACCESS,
                                FALSE,
                                szEvent);
            SetEvent (hEvent);

            /* Close all debug events. */
            for (i=0; i<nDEBUGEVENTS; i++)
                CloseHandle (hDbgEvent[i]);
            CloseHandle (hEvent);

            /* Exit debugger now. */
            return TRUE;
            }
            break;

        case SUSPENDDEBUGGER:
            SuspendDebuggeeProcess (lpDbgProcess);
            ResetEvent (hDbgEvent[DEBUGACTIVE]);
            ResetEvent (hDbgEvent[SUSPENDDEBUGGER]);
            break;

        case RESUMEDEBUGGER:
            ResumeDebuggeeProcess (lpDbgProcess);
            SetEvent (hDbgEvent[DEBUGACTIVE]);
            ResetEvent (hDbgEvent[RESUMEDEBUGGER]);
            break;

        case DEBUGACTIVE:
            /* If debug active */
            if ((WaitForDebugEvent (&de, (DWORD)100)))
                {
                if (de.dwProcessId == lpDbgProcess->dwProcessID)
                    {
                    switch (de.dwDebugEventCode)
                        {
                        case EXCEPTION_DEBUG_EVENT:
                            ProcessExceptionEvent (&de);
                            break;
```

```
                        case CREATE_PROCESS_DEBUG_EVENT:
                            ProcessCreateProcessDebugEvent (&de);
                            break;

                        .
                        .
                        .

                        default:
                            ProcessDefaultDebugEvent (&de);
                            break;
                        }
                    }

                else
                    /* Notify of sibling process debug event. */
                    AppendEditText (lpDbgProcess->hWnd,
                                    de.dwDebugEventCode +
                                        IDS_SIBLING,
                                    NULL);

                ContinueDebugEvent (de.dwProcessId,
                                    de.dwThreadId,
                                    DBG_CONTINUE);
                }
            break;
        }

    }
```

In the example above, the debug thread begins a loop and immediately calls **WaitForMultipleObjects** to await the signaling of any debugger event. The debugger events are described below:

- CLOSEDEBUGGER signals the debug thread to abort debugging the current process. Highest priority event.
- SUSPENDDEBUGGER signals the debugger to suspend debugging the current process.
- RESUMEDEBUGGER signals the debugger to resume debugging the process.
- DEBUGACTIVE signals the debug thread to debug the process because there is nothing else to do. Lowest priority event.

The debug thread remains suspended upon this call until an event becomes signaled. Then, the **WaitForMultipleObjects** function returns the index of the event that was signaled. If mc than one debugger event is signaled at a time, the one with the highest priority is returned. Events are assigned priorities according to how they are ordered in the array of event handle where the lower the array position the higher the priority. The array of handles is passed as an argument to the **WaitForMultipleObjects** function. DEBUGAPP.EXE places the highest priority on exiting the debugger and the lowest priority on actual debugging. That means that debugging is performed only when the debug thread has nothing else to do. Really this means that the debugger is able to respond immediately to commands that do not occur frequently—such as exit.

Also, debug events are separate events from the events that are signaled by the main thread in a debug window. Consequently, another wait loop is embedded within the first for handli debug events alone. However, the debug thread must break from the debug event loop periodically to wait for debugger events. To facilitate this requirement, a time-out is used on the debug event loop to allow the debug thread a way of breaking out of the debug event loop when no debug events are occurring. When the debugger is no longer debugging, as it were is able to check for other events that may have become signaled in the interim. While doing so, the debugger does not need to suspend the process being debugged. Any debug events that occur while the debugger is not waiting for them are queued until the debugger resumes its call to **WaitForDebugEvent**.

Wait events, as used by the debugger, can be either automatic or manual reset types. DebugApp uses manual reset events for more control over when the events become acknowledgec This is necessary because, while the debugger is handling (or waiting for) debug events, more than one debugger event command might have become signaled. When the debugger returns to handle the events, it must handle them one at a time. Two automatic wait events would automatically become reset as soon as the debugger returned from the **WaitForMultipleObjects**. Yet the debugger prioritizes itself so that it responds only to the signaled event with lowest priority. When complete, it returns to handle any others that are st signaled. While handling each one, it resets the event manually to acknowledge the completion of the task for that event. This prevents events from slipping through the cracks while otl events are being processed.

Controlling the Threads of a Process Being Debugged

In preemptive, multithreaded operating systems, a mechanism often referred to as the *system scheduler* exists for scheduling each thread in the system. The system scheduler assigns ea thread a rank or priority that it uses to determine how much processing to attribute to a thread before switching to the next thread. Specifically in Windows, each thread has a base prio in the range 1–31, where the higher the priority, the more processing time is attributed to the thread. Windows establishes the base priority by combining the specific thread priority anc the priority class of its process, as shown in the diagram in Figure 3.

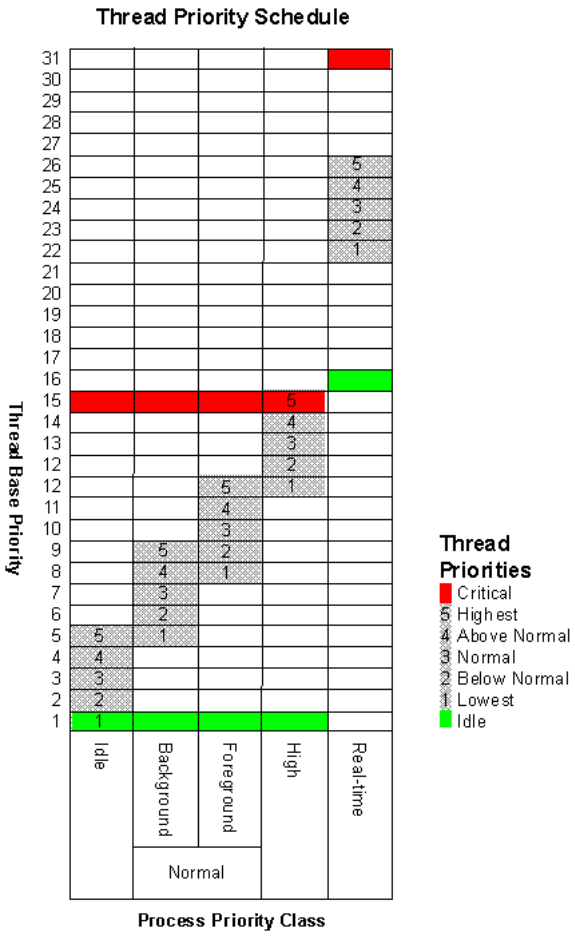

Figure 3. Base thread priorities for each of the process priority classes.

Windows provides processes with the capability of determining and adjusting their threads' base priorities. Windows provides this feature through the **SetPriorityClass**, **GetPriorityClas SetThreadPriority**, and **GetThreadPriority** functions. Using these functions, processes can adjust their threads' base priority values. To adjust the base priority of threads belonging to a process other than itself, a process needs special access rights.

By default, a debugger process has PROCESS_SET_INFORMATION access to the process it is debugging and THREAD_SET_INFORMATION access to all threads in that process. These accesses permit the debugger process to change both the priority class of the process being debugged and the thread priority value for each thread in the process. In addition, unlike debug events, any thread in the debugger process can adjust both the process priority class and thread priority values of the process it is debugging. This means that the main debugge thread is able to perform these functions directly without having to synchronize the procedure with the appropriate debug thread.

Because of this, the GUI thread of the debugger can perform execution control for all processes being debugged. In DEBUGAPP.EXE, execution control is handled by a single dialog box, Thread Execution Control. When the dialog box is invoked, the process being debugged is suspended until the dialog box is dismissed by the user. In the interim, the user is able to moc the priority value for each thread and the priority class for the process being debugged. The dialog box lists each of the threads in the process being debugged, showing their base priorities for comparison. After adjusting priorities for the threads and process, the user exits the dialog box by clicking either OK or Cancel. Clicking OK changes the priorities in the proc being debugged and resumes its execution. Clicking Cancel simply causes the process being debugged to resume with the priorities left unchanged.

In addition to adjusting the base priority of threads, the debugger process can also suspend and terminate the threads of the process being debugged. Specifically, suspending and resuming threads is only available to processes with THREAD_SUSPEND_RESUME access and termination to processes with THREAD_TERMINATE access, but again a debugger process h these access rights by default. The aforementioned Thread Execution Control dialog box in DEBUGAPP.EXE provides support for suspending and resuming threads, but not for terminatir threads. Windows provides support for controlling threads through the **SuspendThread**, **ResumeThread**, and **TerminateThread** functions.

Accessing Thread Context Information from Threads of a Process Being Debugged

By default, a debugger can change the context of any thread in the process being debugged by virtue of the fact that it has access to the handle for each thread of that process. In DEBUGAPP.EXE, the thread handles of the process being debugged are saved as the thread is created in the linked-list structure described earlier in "Representing Information About a Process Being Debugged." To change the context of a thread in the process being debugged, the debugger calls the **SetThreadContext** function. The arguments to this function are sir a handle to the thread to be affected and a pointer to a **CONTEXT** structure filled with information describing how the thread context will exist after making the call. Similarly, to view th state of a thread's context, the debugger calls **GetThreadContext**. Any process can call **Set/GetThreadContext** for any other thread, providing that it has a valid handle to that thread.

Note

A thread context is implementation-specific, differing from one hardware architecture to another. The fields of the **CONTEXT** structure vary, depending on whether you're running on an Intel platform or a MIPS platform. For details on implementing the **CONTEXT** structure for a specific platform, refer to the specific header file where the structure is defined.

Terminating and Exiting a Process Being Debugged

There is more than one way to end a process. Windows provides support for terminating a process, given the handle to that process, in the **TerminateProcess** function. Yet, using this function prevents the process from having the opportunity to clean up volatile data. **TerminateProcess** ends the process immediately without calling the **DllEntryPoint** function of any dynamic-link libraries (DLLs) that the application may have loaded. **TerminateProcess** does not send any last messages to window procedures (like WM_DESTROY); it simply terminates. Windows, however, is robust enough to clean up all system resources owned by the process and associated DLLs. Unlike Windows version 3.1, a process does not leave the system in an

unstable state solely by calling this function.

Terminating a process from the debugger may, in fact, be the appropriate way to end a process being debugged. If at any time a user of the debugger commands the process being debugged to exit from the debugger, it is understood that the process is closing abnormally. If the user wants to gracefully exit the process, the user can simply exit it normally directly fi the application interface. In fact, abruptly exiting a process—but only after it's allowed to save its changes—is, in itself, a contradiction.

Another method of exiting a process is to have a process call **ExitProcess** itself. This method is considered a "graceful" exit because all associated DLLs get a chance to clean up before being detached. In this case, the **DllEntryPoint** function gets called for each thread as it terminates and once for when the process goes away. This function permits the process to save volatile data before exiting. Yet, since **DllEntryPoint** does not include a parameter for a process handle, it cannot be called by one process in the hope of exiting another process. So, the debugger cannot command the process being debugged to exit gracefully by calling this function.

The final technique a debugger process could employ to command the process being debugged to exit gracefully does not use a straightforward API call. Instead, it involves manipulatir the context information of a thread in the process being debugged. Because **ExitProcess** can only be called from the process that intends to exit, the debugger can change the context information of the main thread in the process being debugged so that the next instruction it executes is a call to **ExitProcess**.

To do this, the debugger must:

1. Suspend the process being debugged.
2. Get the context information of a thread in the process being debugged. (It can be any thread in the process, as long as it is not a suspended thread.)
3. Replace the instruction-pointer contents with the address of the **ExitProcess** function as referenced by the process being debugged.
4. Set the altered context information structure back into the thread of the process being debugged.
5. Resume execution of the process.

The process being debugged behaves as though it made a call to **ExitProcess** itself. (Actually that is exactly what it does.) The following function illustrates this technique.

DEBUG.C

```
void ExitDebuggee (
    DBGPROCESS *lppr)
{
    CONTEXT    thContext;

    GetThreadContext (((DBGTHREAD *)lppr->lpThreads)->hThread,
                      &thContext);
    thContext.Eip = lppr->ExitProcess;
    SetThreadContext (((DBGTHREAD *)lppr->lpThreads)->hThread,
                      &thContext);
}
```

Obtaining the address of the **ExitProcess** function is problematic in the above procedure. While it is relatively easy to determine the whereabouts of the function in a process that is executing (it is a member of the system DLL, KERNEL32.DLL), finding the exact address of that function in the process being debugged is more difficult. Each application maps all of the D it uses into its own address space, based mostly on the order in which the DLLs are loaded. This means that, while more than one application may use a given DLL, two processes may or may not have loaded the same system DLL into the same location in their respective address spaces. Consequently, the same function called from a common DLL might be located at different virtual addresses in the two applications. It is tempting to draw the conclusion that system DLLs are loaded into the same base address in every application, for that would make this problem simply go away. **This conclusion, however, is invalid. It may work in some cases, but it cannot be considered a fail-safe assumption. Developers are wise not to dra this conclusion about the system.**

So, to make the **ExitDebuggee** function work properly (see code fragment above), the address of the **ExitProcess** function must be known in the context of the address space of the process being debugged.

Determining the location of ExitProcess

One safe assumption to make is that the location of a function in a DLL is always at the same offset from the DLL's base address. This assumption provides the necessary information to develop a technique that is fail-safe. The debugger is already notified when the process being debugged loads each of its DLLs, and at that time the base address of the DLL is provided the debugger. All the debugger needs to do at this point is determine which DLL being loaded contains the **ExitProcess** function and the offset of that function within the DLL.

To determine the offset of the function in the DLL, call **GetProcAddress** with the handle to the appropriate DLL and a string identifying the **ExitProcess** function. This can be done within the context of the debugger process since the offset is consistent across processes. The handle to this DLL can be obtained by making a call to **LoadLibrary**, specifying KERNEL32.DLL by name. Then, subtract the base address of the DLL from the address returned from **GetProcAddress**. The difference is the offset into the DLL. The base address of KERNEL32.DLL can be determined by calling the **VirtualQuery** function, supplying the address of the **ExitProcess** function as the base address for the region of memory. **VirtualQuery** returns a filled-out **MEMORY_BASIC_INFORMATION** structure. One field of that structure is the base address for the region of memory. In this case, that will be the base address of the DLL code region.

Even more difficult is the task of determining which DLL is being loaded in the process being debugged when LOAD_DLL_DEBUG_EVENT occurs. During this debug event, the debugger receives the base address for the DLL being loaded, but only a module file handle with which to identify the DLL. Fortunately, the file handle can be used to read information about the fi To identify the file as the correct DLL, the debugger must determine the name of the DLL by extracting the filename, assigned by the linker, from the executable image. The name is four after tracing through a maze of offsets and tables of data embedded within the executable file.

A limitation, though, exists in this technique. Since the name of the executable is embedded in the file during the link process, there is no way of knowing whether a user renamed the fil after linking. Unfortunately, there is no way around this limitation. No other way exists to determine the name of the DLL that is being loaded in the debugger—yet, one can always hope that this will be a feature included in a future release of Windows.

Once the name is extracted, it can be compared to see if, in fact, the DLL is the KERNEL32.DLL file. If so, the debugger saves this base address in the process structure for use in the **ExitDebuggee** function as shown in the code fragment in the previous section.

Debug Functions a Process Being Debugged Can Call

A few functions are provided as part of the Windows API for applications that are being debugged. Each of these functions generates a debug event in the debugger process:

- **DebugBreak** is provided simply to insert a break point in an application. This function generates the EXCEPTION_DEBUG_EVENT event and with it an **EXCEPTION_DEBUG_INFO** structure that includes an **EXCEPTION_RECORD** structure, which includes an EXCEPTION_BREAKPOINT exception code for this event.

- **OutputDebugString** provides the opportunity for the process being debugged to pass a string to the debugger application. This function can be extremely useful in a custom debugger application because it provides a mechanism for the process being debugged to pass information to the debugger. The debugger can then log these strings when they occur or respond according to their content. This function generates the OUTPUT_DEBUG_STRING_DEBUG_EVENT event and is accompanied by an **OUTPUT_DEBUG_STRING_IN** structure. This structure contains the address and length of the string in the process being debugged and a Unicode® flag, indicating the type of string it is. The debugger can ac the string by calling **ReadProcessMemory** and indicating the length and address of the string to read along with the process handle.
- **FatalExit** and **FatalAppExit** are functions provided for an application to exit immediately but pass control to the debugger before going away.

The debugger handles each of the above calls as any application calling them would when encountering any other debug event. The only distinction is the type of event itself. If it is desirable to have the debugger execute special processing after one of these types of events, the debugger simply treats each of these debug events uniquely. The debug event loop is already prepared to handle this eventuality.

Expanding on This Debugger Model

The debugger presented in this article falls short of a full-fledged, source-level debugger in several ways. It does not provide any source-level functionality, like single-step execution and break points. It also lacks any symbolic information support. Many features could easily be added to this debugger—many without too much effort and some that would require considerable effort. The purpose of DEBUGAPP.EXE is to provide a base upon which a complete debugging environment could be built while at the same time introducing the debuggin API. To that extent, this debugger is a solid debugging foundation, and it demonstrates extensive use of the debugging API. Don't be surprised to see future samples and technical articl based on this debugging sample!