# Kernel and remote debuggers

28 Jun 2010 | by Albert Almeida | Filed in Comments PDF

*This article was originally published in VSJ, which is now part of Developer Fusion.*

Debuggers are exceptionally powerful tools used by developers to investigate the behaviour of working programs, and troubleshoot problems with failing ones. They help pinpoint program errors by stepping through the code instruction-by-instruction, examine and change the value of processor registers and program variables and set breakpoints in specific code locations to study certain sections of the program.

The approach to kernel-mode debugging taken by Microsoft's debugging tools typically requires a host computer and a target computer connected by a null-modem serial cable. Under this configuration, either WinDbg or KD run on the host computer acting as "remote debuggers", and they rely on kernel debugging support from the target OS to become aware of debugging events and to control the target system's operation.

In this article, I'll take you inside the Win2k kernel debugging support, focusing on the operation of the kernel routines that implement the core of the kernel debugger. I'll also present details of the debug protocol that a remote debugger such as KD and the kernel debugger use to send debugging data to each other.

### Kernel debugger fundamentals

The Win2k kernel debugging support is built right into the kernel. The Win2k OS defines a set of routines that cooperatively provide the kernel debugging support to a remote debugger such as WinDbg. Collectively, these routines implement the Win2k OS component called "the kernel debugger".

The basic operation of the kernel debugger is moderately simple. When the target system is normally running, the kernel debugger is quietly sleeping; it's only when certain events occur that it is brought into action. Specifically, it is activated whenever: an exception is raised (either by the processor or by calling ZwRaiseException), a breakpoint is hit, or a native debugging service is requested. Moreover, when the target detects a break-in request from the remote debugger, the kernel debugger is also activated. The kernel debugger's job is to construct a description of the event that occurred in the system and forward it to the remote debugger for the user to analyse.

Every time the kernel debugger is given control it freezes the operation of the system. Next, it informs the remote debugger (if one is present) about the event that caused the system to break into the kernel debugger. Depending on the event that occurred, the kernel debugger then initiates a loop to interact with the remote debugger to accept and process commands to perform basic operations upon the system such as: set breakpoints, read from memory, search the memory, and write to I/O ports, among others. This loop is repeated until the remote debugger orders the target to resume execution or to perform a reboot. It is important to note that most debugging commands are only available when the kernel debugger has initiated the command accept/process loop. The only exception is the special break-in command (CTRL+C), which may be issued anytime to cause a running target to break into the kernel debugger. If the kernel debugger was already active as a result of an earlier debugging action or because the system had already crashed, the break-in command does not affect the system.

### Setting up for kernel-mode debugging

For kernel debugging support to be activated in the target, the appropriate debugger options must be configured in Boot.ini (by specifying /DEBUG, for example).

When the target is booted from such a configuration, the kernel initialises several variables and sets up internal data structures required for the correct operation of the kernel debugger. One such variable is KdDebuggerEnabled, which is set to TRUE indicating that kernel debugging support is enabled in the system. The kernel also initialises data structures such as breakpoints (KdpBreakpointTable) and special calls (KdSpecialCalls) tables. Special calls (HalRequestSoftwareInterrupt and SwapContext, for example) are routines that, when called by some other routine, require special treatment when performing instruction counting and tracing. Another important debugger variable is KiDebugRoutine which is a pointer to a function and, points to KdpTrap if kernel debugging is enabled or KdpStub if not. The role of each one of these routines will be explained in the following sections. Another debugger variable is KdDebuggerNotPresent, which is often updated to reflect whether the remote debugger has recently been contacted. KdpControlCPending indicates that a break-in request has been detected on the serial line but the kernel debugger cannot be immediately broken into at this time.

### Notifying debugging events

To understand how debugging events are conveyed to the kernel debugger, you need to understand the Win2k exception-handling architecture because of its close relationship with the kernel debugging support.

In Win2k, almost all exceptions eventually resolve down into the kernel routine KiDispatchException. This routine is the system's exception dispatcher and it is at the heart of the structured exception-handling and debugging support provided by the system. The exception dispatcher's job is to find an appropriate exception handler to "dispose of" the exception. When the kernel invokes KiDispatchException to dispatch an exception,

this routine is passed a pointer to an EXEPTION_RECORD structure describing the exception, the trap frame created when the exception occurred, the previous execution mode and a Boolean indicating whether frame-based exception handlers should be given a chance to handle the exception. KiDispatchException then goes through a series of steps to find an appropriate handler for the exception. The steps that it takes are selected according to the previous execution mode. Specifically, if the exception occurred in kernel mode, the kernel debugger is given a first chance to handle the exception. Next, if the kernel debugger doesn't handle the exception, and frame-based exception-handling is allowed, the exception dispatcher invokes RtlDispatchException to search for and invoke a frame-based exception handler. If RtlDispatchException finds no handler for the exception or frame-based exception-handling is not allowed, the kernel debugger is given a second and last chance to handle the exception. Finally, if the exception is still not handled, KiDispatchException calls KeBugCheckEx to shut down the system with the bugcheck code KMODE_EXCEPTION_NOT_HANDLED.

Several more steps are taken by KiDispatchException if the exception originated in user mode. However, for the purposes of this article, I'll only mention the steps that are related to the kernel debugger. Specifically, if frame-based exception-handling is allowed for the exception, the kernel debugger is given a first chance to handle an exception if and only if, the process is not being debugged by a user-mode debugger.

The link between the exception dispatcher and the kernel debugger is the routine whose address is stored in KiDebugRoutine, which KiDispatchException invokes to give the kernel debugger a chance to deal with the exception. KiDebugRoutine points to KdpTrap if kernel debugging is enabled in the system. This routine returns TRUE if it successfully dealt with the exception, otherwise it returns FALSE. The kernel debugger functionality is fully implemented in KdpTrap.

Both KdpTrap and KdpStub are prototyped as follows:

```
typedef stdcall BOOL (*PKDEBUG_ROUTINE)
(IN PKTRAP_FRAME TrapFrame,
        IN PVOID Reserved,
        IN PEXCEPTION_RECORD ExceptionRecord,
        IN PCONTEXT Context,
        IN KPROCESSOR_MODE PreviousMode,
        IN UCHAR LastChance);
```

The Win2k operating system defines an interface through which programs can invoke several debugging services. These services can be classified in: informative services, which inform the kernel debugger of some event and, interactive services, which request input from the remote debugger's user.
The debugging interface is accessed by "int 2d". To convey the service request to the kernel debugger, the trap handler for "int 2d" constructs an EXCEPTION_RECORD structure with an exception code of STATUS_BREAKPOINT and specific information about the service in the exception parameters, including the service class and any other particular information, and then raises an exception. This exception is ultimately handed to the kernel debugger through the call-outs to KiDebugRoutine embedded in the exception dispatcher.

The trap handler for "int 2d" expects a debugging service class in the Eax register, with Ecx and Edx containing additional information specific to each service. So the fields of the EXCEPTION_RECORD structure are initialised with the following values:

```
ExceptionCode = STATUS_BREAKPOINT;
ExceptionFlags = 0;
ExceptionRecord = 0;
ExceptionAddress = Eip;
NumberOfParameters = 3;
ExceptionParameters[0] = Eax;
ExceptionParameters[1] = Ecx;
ExceptionParameters[2] = Edx;
```

The native debugging services defined by Win2k are:

```
#define DbgKdPrintService       0x01
#define DbgKdPromptService      0x02
#define DbgKdLoadImageSymbolsService    0x03
#define DbgKdUnLoadImageSymbolsService  0x04
```

Of all the debugging services, only DbgKdPromptService is an interactive service, the rest are all informative services. System components that wish to invoke any of the above debugging services typically call the routine DebugService, which prototype is as follows:

```
NTSTATUS DebugService(UCHAR
        ServiceClass, PVOID Argument1,
        PVOID Argument2);
```

Internally, this routine sets Eax to ServiceClass, Ecx to Argument1, Edx to Argument2 and then issues an "int 2d" instruction. The contents of Argument1 and Argument2 are specific to each particular debugging service. The Win2k kernel further defines four routines that provide direct access to each of the native debugging services. DbgPrint outputs diagnostic information to the remote debugger; DbgPrompt requests input from the remote debugger's user; DbgLoadImageSymbols informs the kernel debugger that a system-image has been loaded into system space; and its counterpart, DbgUnLoadImageSymbols, informs the kernel debugger that a system-image has been unloaded from system space. All four ultimately call into DebugService to convey the service request to the kernel debugger.

The Memory Manager's routines MmLoadSystemImage and MmUnLoadSystemImage rely on DbgLoadImageSymbols and DbgUnLoadImageSymbols respectively to inform the kernel debugger that a system-image is being loaded and unloaded from system space.

The debug breakpoint exception generated by an "int 3" instruction also finds its way into the kernel debugger using the same mechanism that the trap handler for "int 2d" uses to convey a debugging service request to the kernel debugger. The only difference is that in this case ExceptionParameters[0] is set to zero. The debugging routines DbgBreakPoint and DbgBreakPointWithStatus cause a debug breakpoint exception by issuing "int 3". This last routine sets the status code in the Eax register before executing "int 3".

Another way of breaking into the kernel debugger is when the target detects a break-in request from the remote debugger. The routine KeUpdateSystemTime constantly polls the serial line by calling KdPollBreakIn and, when it detects a break-in command, it generates a debug breakpoint exception by calling DbgBreakPointWithStatus. Internally, KdPollBreakIn also checks the value of KdpControlCPending and if it is TRUE, returns TRUE indicating that a break-in has been detected.

So far, you have seen that all the debugging events of interest to the kernel debugger are conveyed to it in the form of exceptions, and that the kernel debugger differentiates among them by examining the ExceptionCode and ExceptionParameters[0] fields in the EXCEPTION_RECORD structure. I will now explain what the kernel debugger does when it receives an exception from KiDispatchException.

### Kernel debugger internals

The kernel debugger functionality is fully implemented in KdpTrap. Every time this routine is called by KiDispatchException to deal with an exception, the kernel debugger is up and running. KdpTrap is responsible for notifying the exception or debugging service to the remote debugger and, interacting with the remote debugger to accept and process debugging commands to control the system.
KdpTrap begins its execution by determining which debugging event caused its invocation. To do so, KdpTrap examines the ExceptionCode field in the EXCEPTION_RECORD structure. If this is STATUS_BREAKPOINT, the exception was either caused by an "int 3" instruction or by a debugging service request (DebugService), depending on the value of ExceptionParameters[0]. If this field is zero, the exception was caused by a debug breakpoint; otherwise it should contain one of the native debugging service classes. If ExceptionCode is not STATUS_BREAKPOINT, then it should be any other exception of interest to the kernel debugger, such as illegal instruction (STATUS_ILLEGAL_INSTRUCTION) or integer overflow (STATUS_INTEGER_OVERFLOW).

Recall from the description about the exception-handling architecture that the kernel debugger may be given two chances to deal with an exception if it originated in kernel mode. On a first-chance notification, KdpTrap immediately deals with STATUS_BREAKPOINT (debug breakpoints or native debugging services) and STATUS_SINGLE_STEP exceptions. If the FLG_STOP_ON_EXCEPTION bit in NtGlobalFlag is set, KdpTrap also deals with any other exception on the first-chance notification. This flag prevents frame-based exception handlers to be given a chance to handle the exception. If the flag is not set, KdpTrap deals with any other exception only on the second-chance notification. Moreover, KdpTrap only handles a STATUS_PORT_DISCONNECTED exception in the second chance notification to give frame-based exceptions handlers a chance to handle this particular exception.

KdpTrap proceeds to notify the remote debugger about the debugging event that occurred in the system. Before continuing, KdpTrap first freezes the operation of the system by calling KdEnterDebugger, which in turn calls KeFreezeExecution to disable interrupts in the system.

KdpTrap next performs appropriate actions depending on the debugging event. If the event is due to an exception or a debug breakpoint, KdpTrap calls KiSaveProcessorControlState to dump the processor state (execution context plus special registers) to the processor state frame for the kernel debugger to examine or modify if necessary. Next, KdpTrap calls KdpReportExceptionStateChange to report the exception or debug breakpoint.

Internally, KdpReportExceptionStateChange first calls KdpCheckTracePoint to perform any necessary instruction counting and trace data collection for any existing internal breakpoints and special calls. Next, it calls KdpSetStateChange to construct a DBGKD_WAIT_STATE_CHANGE32 structure with a description of the exception and the processor state when the exception occurred:

```
typedef struct _DBGKD_WAIT_STATE_CHANGE32 {
        ULONG NewState;
        USHORT ProcessorLevel;
        USHORT Processor;
        ULONG NumberProcessors;
        ULONG Thread;
        ULONG ProgramCounter;
        union {
                DBGKM_EXCEPTION32 Exception;
                DBGKD_LOAD_SYMBOLS32 LoadSymbols;
        } u;
        DBGKD_CONTROL_REPORT ControlReport;
        CONTEXT Context;
} DBGKD_WAIT_STATE_CHANGE32,
                *PDBGKD_WAIT_STATE_CHANGE32;
```

The NewState field is set to DbgKdExceptionStateChange to indicate that the structure contains a description of an exception in the Exception union member. Afterwards, KdpReportExceptionStateChange calls DumpTraceData to dump collected trace data to the packet to be sent to the remote debugger. Finally, KdpReportExceptionStateChange calls KdpSendWaitContinue to send the packet to the remote debugger and initiate the command accept/process loop.
KdpSendWaitContinue is passed the packet to be sent to the remote debugger, this contains a context record and a packet type so that the remote debugger can easily recognize the content and layout of the packet. In the case of an exception report, the packet type is PACKET_TYPE_KD_STATE_CHANGE64.
KdpSendWaitContinue immediately calls KdpSendPacket, which is passed the packet data and a packet type. KdpSendPacket constructs a kernel debugger packet with the packet data and the specified packet type and

transmits the packet by the serial line using the proprietary debugging protocol. If KdpSendWaitContinue detects that contact with the remote debugger cannot be established (by examining the variable KdDebuggerNotPresent), it returns TRUE to the caller indicating success anyway. When KdpTrap eventually receives this return value, this routine in turn returns this same value to KiDispatchException indicating that the exception has been successfully handled.

At this point, KdpSendWaitContinue has already informed the debugging event to the remote debugger. Now, KdpSendWaitContinue starts a loop to accept and process commands from the remote debugger. This routine calls KdpReceivePacket to read a packet from the serial line and can be specified the packet type it is expecting to receive. Command packets from the remote debugger have a packet type of PACKET_TYPE_KD_STATE_MANIPULATE, so when KdpSendWaitContinue calls KdpReceivePacket, this last routine is indicated that is should only accept command packets. Command packets contain a structure called DBGKD_MANIPULATE_STATE32, which defines the basic operations that the remote debugger is allowed to perform upon the target:

```
typedef struct _DBGKD_MANIPULATE_STATE32 {
        ULONG ApiNumber;
        USHORT ProcessorLevel;
        USHORT Processor;
        NTSTATUS ReturnStatus;
        union {
                DBGKD_READ_MEMORY32 ReadMemory;
                DBGKD_WRITE_MEMORY32 WriteMemory;
                DBGKD_READ_MEMORY64 ReadMemory64;
                DBGKD_WRITE_MEMORY64 WriteMemory64;
                DBGKD_GET_CONTEXT GetContext;
                DBGKD_SET_CONTEXT SetContext;
                DBGKD_WRITE_BREAKPOINT32 WriteBreakPoint;
                DBGKD_RESTORE_BREAKPOINT
                        RestoreBreakPoint;
                DBGKD_CONTINUE Continue;
                DBGKD_CONTINUE2 Continue2;
                DBGKD_READ_WRITE_IO32 ReadWriteIo;
                DBGKD_READ_WRITE_IO_EXTENDED32
                        ReadWriteIoExtended;
                DBGKD_QUERY_SPECIAL_CALLS
                        QuerySpecialCalls;
                DBGKD_SET_SPECIAL_CALL32 SetSpecialCall;
                DBGKD_SET_INTERNAL_BREAKPOINT32
                        SetInternalBreakpoint;
                DBGKD_GET_INTERNAL_BREAKPOINT32
                        GetInternalBreakpoint;
                DBGKD_GET_VERSION32 GetVersion32;
                DBGKD_BREAKPOINTEX BreakPointEx;
                DBGKD_READ_WRITE_MSR ReadWriteMsr;
                DBGKD_SEARCH_MEMORY SearchMemory;
        } u;
} DBGKD_MANIPULATE_STATE32,
                *PDBGKD_MANIPULATE_STATE32;
```

By examining the ApiNumber field of this structure, KdpSendWaitContinue determines which operation to perform on behalf of the remote debugger (DbgKdReadVirtualMemoryApi to read from virtual memory, for example) and responds accordingly by sending a packet with the requested data back to the remote debugger if necessary. The bulk of the debugging commands provided by the remote debugger are implemented based on these primitives. This loop is repeated until the remote debugger orders the system to resume execution or to perform a reboot.
Finally, KdpTrap calls KiRestoreProcessorControlState to restore the processor state previously saved in the processor state frame. Note that the processor state may have been modified by a set context request from the remote debugger.

The above explanation describes the steps KdpTrap goes through to report an exception or debug breakpoint to the remote debugger. A similar set of steps is followed by this routine to report a system-image load or unload notification service. The only difference is that instead of KdpReportExceptionStateChange, KdpReportLoadSymbolsStateChange is now called to report these informative events. Internally, this routine constructs DBGKD_WAIT_STATE_CHANGE32 with NewState set to DbgKdLoadSymbolsStateChange and particular information about the system-image in the LoadSymbols union member. The UnloadSymbols field of DBGKD_LOAD_SYMBOLS32 indicates whether the report relates to a system-image unload (TRUE) or load (FALSE). The pathname of the system-image immediately follows the DBGKD_WAIT_STATE_CHANGE32 structure in the packet to be sent to the remote debugger. KdpReportLoadSymbolsStateChange then calls KdpSendWaitContinue to send the packet and start the command accept/process loop.

To service a string output request (DbgPrint), KdpTrap invokes KdpPrintString.

Before invoking this routine, KdpTrap first calls KdLogDbgPrint to log the string to the print log buffer. Internally, KdpPrintString constructs a DBGKD_DEBUG_IO structure with ApiNumber set to DbgKdPrintStringApi to indicate the structure contains appropriate information about the string to print in the PrintString union member:

```
typedef struct _DBGKD_DEBUG_IO {
        ULONG ApiNumber;
        USHORT ProcessorLevel;
        USHORT Processor;
        union {
                DBGKD_PRINT_STRING
                        PrintString;
                DBGKD_GET_STRING GetString;
        } u;
} DBGKD_DEBUG_IO, *PDBGKD_DEBUG_IO;
```

The null terminated string immediately follows the structure. Later, KdpPrintString calls KdpSendPacket to transmit the packet to the remote debugger, specifying PACKET_TYPE_KD_DEBUG_IO as the packet type. Finally, KdpPrintString calls KdpPollBreakInWithPortLock to check if the break-in command has been issued

by the remote debugger during the string output. Because the system operation is frozen when KdpPrintString is invoked, KeUpdateSystemTime cannot poll the serial line to check for the break-in command and therefore, the system cannot break into the kernel debugger if such a command is detected immediately after outputting a string to the remote debugger. That's why KdpPollBreakInWithPortLock is called at this time by KdpPrintString and if the break-in command is detected, this routine returns a TRUE result. The return value of KdpPollBreakInWithPortLock is in turn returned by KdpPrintString to KdpTrap. If KdpTrap detects that a TRUE result has been returned by KdpPrintString, KdpTrap sets the Eax register in the context record from its parameters to STATUS_BREAKPOINT; otherwise it sets Eax to zero. The value of Eax in the context record is the result eventually returned to the routine which invoked the string output debugging service. Internally, DbgPrint checks the return value of DebugPrint (remember this routine directly invokes the string output debugging service through DebugService) and if it is STATUS_BREAKPOINT, DbgPrint calls DbgBreakPointWithStatus to generate a debug breakpoint and break into the kernel debugger.
To service a string input request (DbgPrompt), KdpTrap calls KdpPromptString, but before this, KdpTrap calls KdLogDbgPrint to log the string to be displayed in the remote debugger to the print log buffer.
KdpPromptString constructs a DBGKD_DEBUG_IO structure with ApiNumber equal to DbgKdGetStringApi to indicate the structure contains appropriate information about the prompt string and the maximum number of characters to read in the GetString union member. The null terminated prompt string immediately follows the structure. KdpPromptString next sends the packet to the remote debugger by calling KdpSendPacket with a packet type of PACKET_TYPE_KD_DEBUG_IO. Now KdpPromptString calls KdpReceivePacket to receive a PACKET_TYPE_KD_DEBUG_IO packet with the string read. The received DBGKD_DEBUG_IO structure contains the number of characters actually read in the LengthOfStringRead field of the GetString union member.

Note that neither KdpPrintString nor KdpPromptString call KdpSendWaitContinue to start the command accept/process loop. This means that after a device driver or system component calls DbgPrint or DbgPrompt to request the particular debugging service, the system continues its normal operation as if nothing had happened.

The last step of KdpTrap is to unfreeze the system operation by calling KdExitDebugger. Internally, this routine calls KeThawExecution to re-enable interrupts in the system and allow other system components to continue their normal execution.

As I mentioned earlier, KiDebugRoutine points to KdpStub if kernel debugging is not enabled in the system. When the exception dispatcher calls KdpStub to handle an exception, this routine checks if the debugging event is one that can be ignored and, if so, returns TRUE to KiDispatchException indicating that the exception has been handled. Specifically, all native debugging services except DbgKdPromptService can be ignored. When KdpStub receives any other exception, it returns FALSE to the exception dispatcher to give frame-based exception handlers a chance to deal with the exception.

### Serial line debug protocol

As you have will have noted, the remote debugger and the kernel debugger communicate by interchanging debugging messages. These messages include debugging commands to control and get information about the target, descriptions of exceptions or debugging events that occur in the target or the special break-in command issued by the remote debugger to cause the target to break into the kernel debugger.
The low level data structure used to pass debugging data between the remote debugger and the kernel debugger is a kernel debugger packet. All packets begin with a packet header defined in KD_PACKET as shown below, follows the packet data and terminate with a trailing byte:

```
typedef struct _KD_PACKET {
        ULONG PacketLeader;
        USHORT PacketType;
        USHORT ByteCount;
        ULONG PacketId;
        ULONG Checksum;
} KD_PACKET, *PKD_PACKET;
```

The PacketLeader field in KD_PACKET classifies the packet in one of three possible packet categories: normal packet, control packet and break-in packet.
The PacketType field in the packet header indicates the packet type. This field allows both the remote debugger and the kernel debugger to recognize the content and layout of the packet data. Control packets can be any of: PACKET_TYPE_KD_ACKNOWLEDGE, PACKET_TYPE_KD_RESEND or PACKET_TYPE_KD_RESET. Control packets are used to control the flow of packets between the host and the target computer. Normal packets can be typed as: PACKET_TYPE_KD_STATE_CHANGE32, PACKET_TYPE_KD_STATE_MANIPULATE, PACKET_TYPE_KD_DEBUG_IO, or PACKET_TYPE_KD_STATE_CHANGE64. Recall from the previous section that these packets are used to send debugging commands to the target and to report debugging events to the remote debugger. The special break-in packet is actually a control packet with a PacketLeader of BREAKIN_PACKET.

The PacketId field is just a sequence number for the packet being transmitted. The other two fields in KD_PACKET are self-explanatory. ByteCount contains the number of bytes in the packet data and CheckSum contains a very simple checksum computed by adding all the bytes in the packet data.

The content and layout of the packet data is directly related to the packet type. The table below illustrates the structures to be filled in the packet data for each normal packet type:

**Packet data structures**

| Packet Type | Structure in Packet Data |
|---|---|
| PACKET_TYPE_KD_STATE_CHANGE32 | DBGKD_WAIT_STATE_CHANGE32 |
| PACKET_TYPE_KD_STATE_MANIPULATE | DBGKD_MANIPULATE_STATE32 |
| PACKET_TYPE_KD_DEBUG_IO | DBGKD_DEBUG_IO |

PACKET_TYPE_KD_STATE_CHANGE64 DBGKD_MANIPULATE_STATE64

Control packets consist only of a KD_PACKET header with PacketLeader set to CONTROL_PACKET_LEADER and the particular control packet type in the PacketType field.

The sequence of steps for reading a packet from the serial line is as follows:

1. Read four bytes to get the packet leader. If the read times out or the packet leader is incorrect, then retry the read.
2. Read two bytes to get the packet type. If read times out or if the packet type is incorrect, go to step 1.
3. Read two bytes to get the byte count. If read times out or if the byte count is greater than PACKET_MAX_SIZE, go to step 1.
4. Read four bytes to get the packet id. If read times out, go to step 1. If the packet id is not the expected value, then send a PACKET_TYPE_KD_RESEND control packet to request the sender to resend the complete packet and restart from step 1.
5. Read four bytes to get the checksum. If checksum is invalid, then send a PACKET_TYPE_KD_RESEND control packet and restart from step 1. The packet data immediately follows the packet header. There should be ByteCount bytes following the packet header.
6. Read the packet data. If read times out, go to step 1. The debug protocol requires that every time a normal packet is received, the receiving party must reply with the control packet of type PACKET_TYPE_KD_ACKNOWLEDGE. On the kernel debugger side, this algorithm is implemented in KdpReceivePacket.

### Conclusion

By now you should have a solid understanding of how the Win2k kernel debugger works. Armed with this knowledge, you can have a clearer picture of what is going on behind the scenes when performing kernel-mode debugging on an NT-based operating system employing Microsoft's debugging tools. For a complete definition of the structures and constants mentioned in this article, refer to the header file Windbgkd.h in the Windows 2000 DDK.

Albert Almeida is an industrial engineering student, specialising in the design and development of distributed enterprise applications. He also researches Windows NT/2000/XP/.NET internals and security. He can be reached at magito@bigfoot.com.