Debugging Blue Screens

A Technical Paper from

Compuware

September 1999

Debugging Blue Screens

Introduction

The "Blue Screen of Death," often just called the Blue Screen, or abbreviated BSOD, is one of the most feared displays on Windows NT and Windows 2000. For end users, it means that the operating system has crashed, losing work and bringing downtime to what is often a mission-critical operation. For developers, it means that they have failed their end users. Under most circumstances, only errors in kernel mode programs can bring down Windows 2000 or Windows NT. And most by far of the third-party kernel mode programs are device drivers.

The BSOD is also one of the most difficult debugging challenges faced by developers. This is for two reasons. First, getting visibility into the behavior of a kernel mode program is difficult; few programming tools provide a mechanism for watching kernel mode execution. Second, the information and technical resources on kernel mode programming are limited, disorganized, and difficult to understand. Kernel mode programmers are often on their own, or forced to rely on advice and direction provided their colleagues on news groups.

How do you debug something that you can't watch?

What is a Blue Screen?

The Blue Screen is Windows 2000's and Windows NT's way of handling situations where the next processing step either isn't clear or could result in corrupt data. In the example later in this paper, we introduce a divide-by-zero error into a driver, an error that illustrates both of these types of situations.

The Blue Screen is called with a STOP code (also called a BugCheck parameter) that describes the reason for a system halt. There are about 150 different STOP codes (available in one place in bugcodes.h found in the DDK), but only about half a dozen are common. The STOP code can have up to four parameters that are related to some of the STOP codes, and which are also printed out on the Blue Screen.

The next section of the Blue Screen includes the CPU ID, the IRQL at the time of the crash, and the operating system build number. Below this is a list of each driver loaded in the system, its base address, and a date stamp (actually the number of seconds from December 31, 1969 to when the driver was built). Below the loaded drivers is a stack trace, starting at the point directly above the call that created the Blue Screen.

Some of this information is useful in debugging the system crash, if you understand what it means and how it's generated. But to many kernel mode developers, it just signifies that something went wrong, somewhere. And Windows 2000 provides much less information in the Blue Screen on the cause of system crashes than Windows NT 4.0. Is it possible that better information can be provided for debugging kernel mode drivers that crash the system?

There's got to be a better way.

The Traditional Approach

There are several possible solutions to the challenge of debugging system crashes. Microsoft offers a set of tools and techniques that, when used together, provide a rudimentary solution. However, the approach, outlined below, offers just barely enough information to get started with the crash debugging process. These debugging tools aren't packaged in a single place, and include little or no documentation on using them.

- You can log an event message with bugcheck information – a STOP code and parameters. To do this, on the Startup/Shutdown tab in System | Properties, select the "Write an event to the system log" box. When you reboot, an event is written into the event log and can be viewed by the event monitor. The description of the event includes the following information:

  ```
  Event ID:  1001
     Source:  Save Dump
     Description:
     The computer has rebooted from a bugcheck.  The bugcheck
     was: 0x0000001e (0xc0000094, 0xf73606b9, 0xc0000094,
     0xffffffff). Microsoft Windows NT [v15.1381]. A dump was
     saved in:  C:\WINNT\MEMORY.DMP
  ```

- Dumpchk.exe, first available on the Windows NT 4.0 Service Pack 3 CD, provides the address (in hexidecimal) of the driver that generated the STOP message. The address can be used to identify the driver using the output from Pstat.exe, described below. Dumpchk.exe is run from the command line.
- Pstat.exe, a utility found in the Resource Kit, provides size, address, and date information on processes and drivers running on the system. It provides a starting address for the driver, which can be compared to the address provided with Dumpchk.exe to pinpoint the driver.

In other words, this debugging process provides you with the identity of the driver that caused the Blue Screen, and nothing more. As a developer, you probably already know that the driver you have under development is the cause of the crash. You need more detailed information about what driver event caused the crash, and how the event did so.

Microsoft's solution is to use a kernel-mode driver in the system being debugged, along with a second PC connected via a serial cable. A Windows debugging tool called WinDbg runs on the second PC, gathering data from the test PC across the serial cable and displaying it. WinDbg has a host of limitations, including the requirement to use two PCs for debugging, uneven and inconsistent documentation (WinDbg is included on the MSDN distribution rather than sold as a product), and not easy to use WinDbg is the right solution if no other tools are available.

There is a better way.

What You Really Need

Let's start with the device driver itself. Every driver is different, certainly. But thanks to the WDM model implemented for the more recent Windows operating systems, the architecture and much of the standard operations are the same from driver to driver. If you can make use of both a standard architecture and proven, reusable code for much of the driver, you accomplish two things. First, you accelerate your entire driver development process; and second, you make it much easier to isolate and identify coding errors.

Then there's the Blue Screen itself. At the point of the Blue Screen, Windows gathers a limited amount of information about the state of the system at that moment in time, and throws it on the Blue Screen. Some of this information is in a human-readable form, while the rest looks like something only a machine could love. The reason for this is that the system does the minimal amount of processing in order to best preserve the state of the machine in the dump. You get worthwhile information, but the system doesn't spend any time organizing and formatting it.

The obvious solution is to freeze the state of the system at the point the Blue Screen routines are called. Rather than letting it crash and lose information leading up to Blue Screen, an NT debugging tool should prevent the system from rebooting until the developer can get all possible information from the state of the system at the time of the crash. You should be able to query the system on what failed and what events led up to the failure.

Once the system has crashed and rebooted you can, of course, collect a crash dump file. But the crash dump file contains all of the contents of memory, and is both huge and difficult to read. You do have a head start, if you use Dumpchk.exe to find the starting address of the offending driver, and Pstat.exe to associate the name of the driver with this address. Once there, you can search the crash dump file for further references to your driver. But the crash dump file still takes a great deal of time and effort to read and understand.

Instead, you would like to be able to filter the crash dump file to give you just the information you need on your driver, without having to search through a mountain of irrelevant and difficult to read data. It should only include information on the driver or drivers you specify, and expressed and organized in a way that lets you readily identify your driver code.

Once you've filtered the crash dump file to the point where it's useful for tracing the execution of a specific driver, it's really a short step to being able to tie the specific driver instructions back to its source code. If this were possible, you could find the error in memory or in the kernel call, then automatically trace it back to event generating the error, and from their directly to the exact line of the driver source code. By relating the fatal error directly to the driver code, you can debug a driver in a matter of minutes rather than hours or days.

NuMega DriverStudio from CompuWare – A Better Way to Build, Debug, Test, and Deploy Device Drivers

DriverStudio components address development, debugging, tuning, testing, and deployment of Windows device drivers. The NuMega DriverWorks component is a class library that enables developers to apply object-oriented practices to device driver development. The library includes a framework for building kernel mode and WDM drivers, in addition to a number of other non-framework classes. The product also provides a code generator, numerous samples, and other utility programs. Using DriverWorks for driver development assists in debugging Blue Screens by easily identifying driver events and relating them to source code during the debugging process. It also abstracts the Windows DDK into C++ classes, enabling the programmer to work at a higher level of abstraction.

DriverWorks includes DriverWizard, a Microsoft Visual Studio plug-in that automates the creation of the driver framework. The DriverWizard provides a step by step process for setting up the details of a driver, then uses an extensive collection of C++ library code to assemble the driver source files. DriverWizard then inserts comments explaining the function of the source and where the driver developer has to insert device-specific code to complete the driver.

DriverStudio also delivers strong debugging tools, no matter what development environment used for building drivers. The DriverStudio solution to debugging Blue Screens combines the use of the award-winning NuMega SoftICE along with NuMega BoundsChecker Driver Edition and NuMega DriverWorkbench to debug interactively at the time of the crash, and by examining the events leading up to the crash after rebooting. We call the combination “on-demand” and “post-mortem” debugging, and it provides unprecedented scope and depth of kernel mode driver debugging.

To illustrate a simple debugging problem using DriverStudio tools, I’ve built a driver using DriverWorks and intentionally introduced an error that will crash a Windows NT or Windows 2000 system. I’ll show how DriverStudio tools can halt the system at the instant of the crash to examine the state of the system, then save a crash dump file for analysis after the system is rebooted. DriverStudio debugging tools will pinpoint the exact line of source code responsible for the Blue Screen.

Using the defaults in the DriverWorks DriverWizard, you can create a driver skeleton in just a minute. While this driver won’t actually control hardware, it compiles into a .SYS and can be loaded into the kernel with DriverWorkbench. When compiling, we select the checked build to make sure we have debugging information. Once we’ve debugged the driver, it can be recompiled into the free build. Note that none of the DriverStudio debugging tools require instrumenting the code or linking special libraries into the build.

In this particular example, I’ve introduced a simple error into the Unload() function – a divide by zero right before the DriverWorks base class KDriver::Unload() is called (Figure 1). This driver will load without difficulty, and will cause no problems while loaded. However, when you try it unload it, the divide by zero error in the kernel will crash the operating system.

Figure 1. Introducing a simple error into the Unload() function of a driver built with DriverWorks.

SoftICE Provides Interactive System-Wide Debugging

SoftICE is the only interactive Windows debugger available that provides complete information on the processor, operating system, drivers, and programs on a single computer. To be most effective, SoftICE requires that debugging symbols be generated during the driver build process. Take, for example, a sample driver built using the DriverWorks DriverWizard in Visual Studio. You should use the checked versions of the driver libraries needed for your platform, and choose the checked target for your driver. The build process in Visual Studio will note that debugging symbols have been created, and whether or not SoftICE is active.

Before attempting to load the symbol file for the Demo driver, make sure SoftICE is active on the system. You can configure SoftICE to load at boot time, when Windows starts, or manually while the system and Windows are running.

To load the symbol file into SoftICE, select the Symbol Loader utility (Figure 2). Open the .NMS file, and select Load from the Module menu. The Symbol Loader will display a message indicating that the process completed successfully. SoftICE is now ready for debugging the driver.

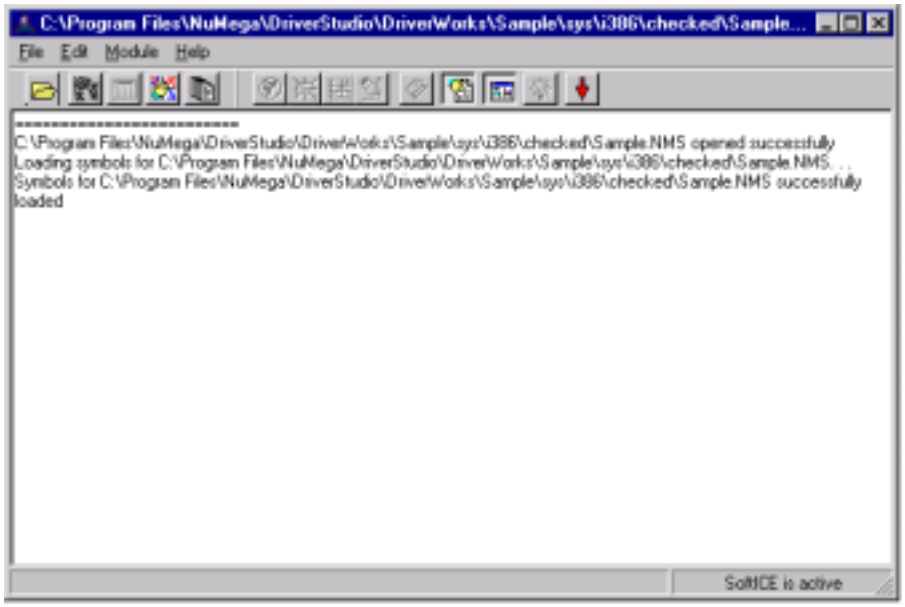

Figure 2. The SoftICE Symbol Loader lets you load the driver's symbol file for SoftICE to work with.

BoundsChecker Driver Edition Collects System Events

BoundsChecker Driver Edition is a kernel mode driver that loads before all other system drivers, and monitors kernel events for drivers on that system. It collects kernel events and stores them in a ring buffer whose size can be adjusted by the user.

BoundsChecker Driver Edition can monitor any combination of drivers and kernel events. To tell it which drivers to monitor, open DriverWorkbench and choose BoundsChecker | Configure (Figure 3). You can select from the list of drivers those you want to monitor.

If your driver isn't listed, you can add it to the list by selecting the Add Driver button at the bottom of the dialog box and choosing a driver on the system. For our Demo driver, we select it from its directory, then check its box. This will be the only driver we monitor on the system, since it's the one we're debugging.

The next dialog lets you choose the event categories and individual events you would like BoundsChecker to monitor. There are literally thousands of events that you can choose to follow using BoundsChecker. The final dialog in the configuration process lets you adjust the size of the buffer for storing events, and determine when SoftICE becomes active. For our Demo driver, we'll monitor all system events, keep the BoundsChecker buffer size at its default value of 1024 KB, and set SoftICE to pop up on errors only. If you require more than a 1024 KB buffer to track events, you can increase the value, at the expense of setting aside that amount of memory for storing events.

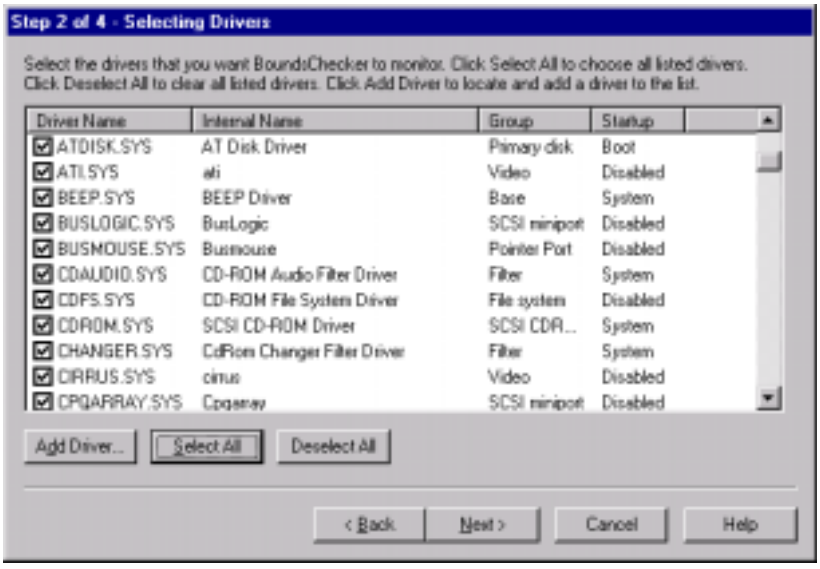

Figure 3. DriverWorkbench lets you select the drivers and events you want BoundsChecker Driver Edition to monitor. If a driver isn't listed, you can add it to the list by selecting Add Driver.

Getting Driver Information Through DriverWorkbench

Within DriverWorkbench, you can manually load your driver by selecting Drivers | Select, then Drivers | Start. This loads your test driver and calls its driver entry point. In our sample case, the driver does nothing, but you can take a BoundsChecker event capture and view its Load event by selecting BoundsChecker | Capture System Information. This may take a few seconds, depending on how large you set the buffer size, the speed of your system, and how many other drivers and events you selected to monitor.

The results of the Capture System Information are displayed in two windows. The first window, called the System Information window, displays information about the active processes, running drivers and their address space, the interrupt descriptor table (IDT), and the global descriptor table (GDT).

The second window, called the Driver Events window, displays all of the kernel events for those drivers you chose to monitor, in the order that they occurred on the system. For those events generated by your driver or other driver for which you have source code, you can link DriverWorkbench to your driver source code by selecting View | Options, and providing the search paths to the source and symbol files. To get the most value out of BoundsChecker events in general, and in debugging crash dump files in particular, you should set up the search paths for the source and symbol files, as this provides a way of directly linking driver events to the code that generated them.

Crashing the System

To crash the system running the Demo driver, we simply select Drivers | Unload in DriverWorkbench. This initiates the call to the Unload() function on the Demo driver, causing the divide by zero code to execute in the kernel, and crashes the entire system. You won't see the usual Windows NT/2000 Blue Screen right away, however, because the SoftICE window will pop up to enable you to perform immediate debugging activities on the system crash.

At this point, the system has crashed, and is not recoverable without a reboot. SoftICE freezes the system state at the instant of the crash, and lets you view a great deal of information at this point to determine what driver and what code crashed the system. If you're a driver developer testing a new driver, you probably already know that your driver caused the crash, so you can start looking at kernel events, stack data, register values, and source code.

If the bug code is the result of a page fault, GP fault, stack fault, or invalid opcode, SoftICE attempts to restart the faulting instruction. Control stops on the actual faulting instruction with all the registers in their original state. In this case, the divide by zero generated a page fault, so we have all of the register information.

Individual windows in the SoftICE user interface provide information that may be useful in debugging this or similar system crash. These windows include displays of unassembled instructions and/or source code, the current stack frame, the current state of the registers and flags, memory contents, call stack values, and the current state of the FPU (Floating Point Unit) stack registers.

You can also view BoundsChecker events in SoftICE using the EVENT command. This is the obvious point to begin debugging this particular Blue Screen. You'll also see BoundsChecker events from the crash dump file in DriverWorkbench after you reboot, but looking at them now could help you pinpoint which events to examine in greater detail later.

You can also normally use SoftICE to set breakpoints and step through code in both kernel mode and user mode of the system. SoftICE lets you follow execution across any system boundaries. However, with your system crashed, it's not possible to step through your driver code until you reboot and bring SoftICE back up.

If individual variable values are in the call stack or the registers, one obvious place to seek answers to your Blue Screen is in these views. If you know the address of your driver entry point, you can examine the memory locations around that address to determine if there is an invalid value.

The exact process you use to perform initial debugging activities within SoftICE as your system has crashed are dependent on your type of driver and where it caused the crash. In all likelihood, you'll try to collect some general information on the state of the system, then examine the BoundsChecker events once you reboot. You'll also want to see if the error is reproducible under the same circumstances as the initial crash.

Tracing the Bug After the Fact

After you've rebooted the crashed system, the crash dump file (named MEMORY.DMP by default and written to %SYSTEMROOT%, but it can be given any name) can be loaded into DriverWorkbench. Note that the crash dump file is the size of physical memory, so the system swap file must be defined to be at least the size of physical memory in order to successfully capture this information. The crash dump file is configured to provide BoundsChecker events in addition to memory values normally captured. This information is similar to that displayed when you perform a Capture System Information from within DriverWorkbench, but includes those events leading up to the crash. This means you are likely to have the exact driver event that caused the crash.

Examining in the System Information window shows that DriverWorkbench includes an additional panel titled Fault Information. This panel presents similar information as would be found in the event log if you wrote the STOP information to the log, although it also includes the bugcheck parameters and exception address (Figure 4).

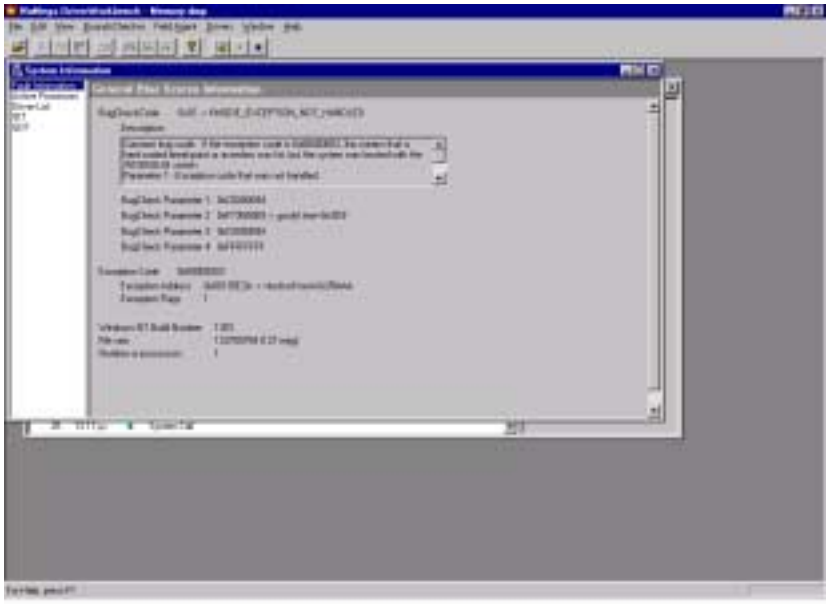

Figure 4. DriverWorkbench displays system information gathered from the crash dump file, including event information generated by the STOP command, bugcheck parameters, and exception address.

The Driver Events Window in DriverWorkbench displays all of the events generated by your driver while it was running in the kernel. You can select BoundsChecker | Show Errors Only to narrow in to the event that generated the system crash. Once you have that particular event, you can expand once again to show all events, to examine the sequence of events that led up to the error.

One very quick and easy was of identifying the offending code is to select the event that generated the system crash, and right-click to bring up its menu. You can then go to the

source of the caller or the callee, depending on which applies in that particular situation (Figure 5). If you don't have the source code available for a particular call, DriverWorkbench displays dissassembled object code.

In the case of our divide by zero error, the page fault is displayed as an error event in the Event Window, close to the bottom of the event listing. Selecting the event immediately above the page fault and choosing *Go To Source of Caller* takes you to the beginning of the Unload() function in your source code, which was the last system event before the error.

You're now at the beginning of the short sequence of code that caused the Blue Screen. Much of this code was the framework generated by DriverWorks, so you can open your original driver framework to determine the modifications and additions you made. Knowing that the error in your source code was between this point and the call to the DriverWorks base class KDriver::Unload(), you can identify the offending logic in just a few seconds.

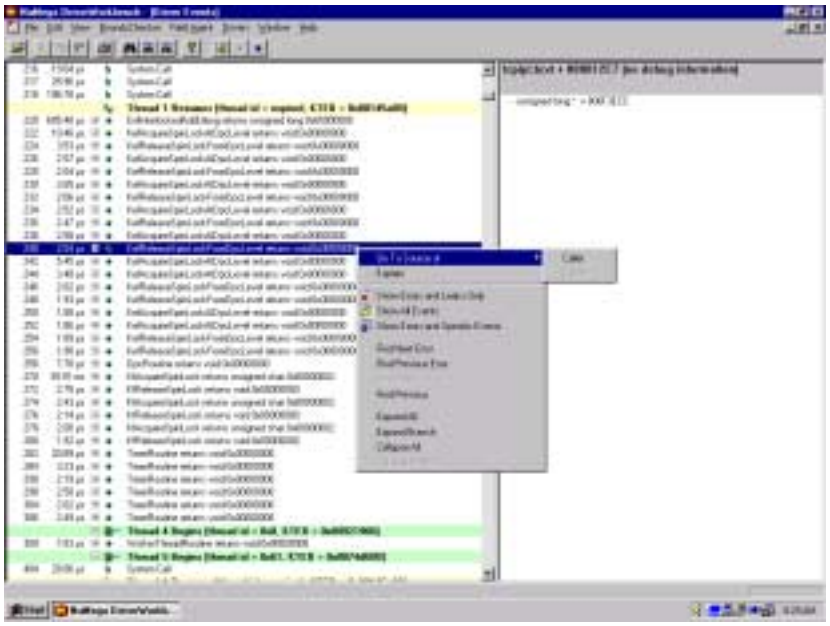

Figure 5. After loading the crash dump file into DriverWorkbench, you can select an individual BoundsChecker event and go to the source of either the caller or callee of that event.

Debugging on Remote Systems

DriverStudio provides two alternatives for remotely debugging drivers. First, SoftICE can be controlled over a network. You can install the full SoftICE on the system to be debugged, use the NET command to activate its integrated TCP/IP stack. On the viewing workstation, install SINET.EXE, a version of the SoftICE user interface that enables user interaction with the remote system. You'll have the same SoftICE view of the remote networked debugging system as you do on a local machine. If a network isn't available,

you can also debug remotely using a serial connection, either by connecting with a serial cable or by using a modem.

Second, you can package a preconfigured copy of BoundsChecker Driver Edition into an application called NuMega FieldAgent, which can be sent to users on the same floppy disk as the driver itself. Once installed on the remote system, users can periodically capture BoundsChecker events, or collect BoundsChecker events in the crash dump file if the system crashes, just as they would with a full BoundsChecker installation. This is useful for deploying to beta testers, who frequently encounter hardware and software conflicts that aren't readily identified on development workstations.

The crash dump file can be sent back to the developer, who can use DriverWorkbench for post-mortem debugging just as if the crash had occurred on the development system. FieldAgent is licensed for unlimited distribution. While there's not the same opportunity to use SoftICE to freeze the state of the crashed system and examine the system at the point of failure, FieldAgent provides you with far more information than you currently have available for working with beta users.

Getting Past Your Blue Screen

Of course, not all Blue Screens are this easy to debug. If they were, you wouldn't have a need for a powerful debugging combination like SoftICE, BoundsChecker Driver Edition, DriverWorkbench, and FieldAgent.

But debugging Blue Screens caused by driver errors and inconsistencies is one of the toughest tasks a systems programmer faces. The error could be a simple coding mistake, as illustrated here, or it could be a misuse of the kernel programming interfaces or an inadvertent pointer error. The possible causes can number in the thousands, and may take a detailed knowledge of Windows NT or Windows 2000 architecture and operation to diagnose and fix.

Pinpointing the offending driver and examining a crash dump file is only the beginning of the process. The rest can either be long, tedious, and error-prone, or quick and streamlined, with lots of information on the behavior of your driver on the system.

It's your choice.