Using DriverWorkbench Tools

Version 3.1

COMPUWARE

Technical support is available from our Technical Support Hotline or via our FrontLine Support Web site.

Technical Support Hotline:
1-800-538-7822

FrontLine Support Web Site:
http://frontline.compuware.com

November 22, 2003

Table of Contents

Preface

Chapter 1

Getting Started with DriverWorkbench

Chapter 2

Using BoundsChecker Driver Edition

Chapter 3

Using TrueTime Driver Edition

Chapter 4

Using TrueCoverage Driver Edition

Chapter 5

Using DriverMonitor

Preface

Purpose of This Document

This document is intended for DriverStudio™ users who want to use the DriverWorkbench™ tools: Boundschecker® Driver Edition, TrueTime® Driver Edition, and TrueCoverage™ Driver Edition, to analyze drivers designed for Windows® Millennium Edition and the Windows NT® family (Windows 2000, Windows XP, and Windows Server 2003) platforms.

- **BoundsChecker Driver Edition** – The driver edition of the famous BoundsChecker software has the unique ability to invoke Compuware's SoftICE® debugger when an error is detected, freezing the entire system at the point where the error occurred. In addition, developers can easily configure BoundsChecker to search for specific error conditions such as memory allocations, driver calls, etc. Note that this can be done either on the developer's local machine or on one or more remote machines. This tool is especially powerful and effective for finding those elusive intermittent bugs.
- **TrueTime Driver Edition** – TrueTime Driver Edition provides sophisticated performance analysis that allows device driver developers to easily identify and fix performance bottlenecks. TrueTime gathers comprehensive performance information, statistics and data on function call frequency, I/O request packet (IRP) completion times, data throughput, deferred procedure call (DPC)

timings, and more. This information presented graphically, enabling developers to visually analyze a driver's performance.

- **TrueCoverage Driver Edition** – True Coverage Driver Edition collects code coverage information on device drivers and provides a Windows application for display and analysis of the collected data. It displays the number of lines of code tested, number of times the code was executed, functions tested and the percentage of code and functions tested.

Users of previous versions of DriverStudio should read the Release Notes to see how this version differs from previous versions. This document assumes that you are familiar with the Windows interface and with software driver development concepts.

What This Document Covers

The *Using DriverWorkbench Tools* document is organized as follows:

- Preface – Briefly describes the components and features of the DriverStudio Tools products. The Preface also explains how to contact the Compuware Technical Support Center.
- Chapter 1, "Getting Started with DriverWorkbench" – DriverWorkbench is an integrated Technology Environment that provides easy access to information about device driver errors and performance problems in a distributed environment.
- Chapter 2, "Using BoundsChecker Driver Edition" – Describes the errors detected by BoundsChecker, how to view BoundsChecker data, event catagories and types , and other BoundsChecker operations.
- Chapter 3, "Using TrueTime Driver Edition" – Describes TrueTime performance analysis and the Probe Point API.
- Chapter 4, "Using TrueCoverage Driver Edition" – Explains how to control the TrueCoverage data display, as well as the relationship of coverage data to source code.
- Chapter 5, "Using DriverMonitor" – Explains how to control the display of messages and how to use timestamps.
- Glossary
- Index

Conventions Used In This Document

This document uses the following conventions to present information:

Convention	Description
Enter	Indicates that you should type text, then press RETURN or click OK.
italics	Indicates variable information. For example: *library-name*.
monospaced text	Used within instructions and code examples to indicate characters you type on your keyboard.
small caps	Indicates a user-interface element, such as a button or menu.
UPPERCASE	Indicates directory names, file names, key words, and acronyms.
Bold typeface	Screen commands and menu names appear in **bold typeface**. For example: Choose **Item Browser** from the **Tools** menu.
commands and file names	Computer commands and file names appear in `monospace typeface`. For example: The ***Using SoftICE*** manual (`Using SoftICE.pdf`) describes...
variables	Variables within computer commands and file names (for which you must supply values appropriate for your installation) appear in *`italic monospace type`*. For example: Enter `http://`*`servername`*`/cgi-win/itemview.dlll` in the **Destination** field...

Customer Assistance

For Non-Technical Issues

Customer Service is available to answer any questions you might have regarding upgrades, serial numbers and other order fulfillment needs. Customer Service is available from 8:30am to 5:30pm EST, Monday through Friday. Call:

- In the U.S. and Canada: 1-888-283-9896
- International: +1 603 578-8103

For Technical Issues

Technical Support can assist you with all your technical problems, from installation to troubleshooting. Before contacting Technical Support, please read the relevant sections of the product documentation as well as the Readme files for this product. You can contact Technical Support by:

- **E-Mail:** Include your serial number and send as many details as possible to:

  `mailto:nashua.support@compuware.com`

- **World Wide Web:** Submit issues and access additional support services at:

  `http://frontline.compuware.com/nashua/`

- **Fax:** Include your serial number and send as many details as possible to:

  `1-603-578-8401`

- **Telephone:** Telephone support is available as a paid* Priority Support Service from 8:30am to 5:30pm EST, Monday through Friday. Have product version and serial number ready.

  - In the U.S. and Canada, call: 1-888-686-3427
  - International customers, call: +1-603-578-8100

  * Technical Support handles installation and setup issues free of charge.

When contacting Technical Support, please have the following information available:

- Product/service pack name and version.
- Product serial number.
- Your system configuration: operating system, network configuration, amount of RAM, environment variables, and paths.
- The details of the problem: settings, error messages, stack dumps, and the contents of any diagnostic windows.
- The details of how to reproduce the problem (if the problem is repeatable).
- The name and version of your compiler and linker and the options you used in compiling and linking.

Chapter 1
Getting Started with DriverWorkbench

Introduction to DriverWorkbench

DriverWorkbench is an integrated technology environment that provides easy access to information about device driver errors and performance problems, including remote analysis in a distributed environment. Completely rewritten from the ground up, DriverWorkbench provides seamless integration of Visual SoftICE™ with DriverStudio tools such as BoundsChecker Driver Edition, TrueTime Driver Edition, and TrueCoverage Driver Edition for developers using Visual Studio 6.

Note: BoundsChecker Driver Edition, TrueTime Driver Edition, and TrueCoverage Driver Edition can also be used inside the Visual Studio .NET environment.

DriverWorkbench allows the developer to more easily and quickly build device drivers. By being able to move from one tool to the other without ever leaving the development environment, the driver developer can build a high-quality device driver more efficiently.

Driver Testing and Debugging Made Easy

DriverWorkbench tools allow you to test and debug device drivers on either a local system or a properly-configured remote network-connected system. Depending on the tools you've installed, you can:

- Debug virtually any type of code with Visual SoftICE
- Debug system crashes

- Test drivers on a "live" system
- Launch and View BoundsChecker data
- Launch and View TrueTime performance data
- Launch and View TrueCoverage data
- View debug messages with DriverMonitor

Debugging Code with Visual SoftICE

Visual SoftICE is an advanced, all-purpose debugger that can debug virtually any type of code. This includes, but is not limited to, interrupt routines, processor level changes, and I/O drivers. Visual SoftICE combines the power of a hardware debugger with the simplicity of a symbolic debugger. It provides hardware-like breakpoints and sticky breakpoints that follow the memory as the operating system discards, reloads, and swaps pages.

Visual SoftICE displays your source code as you debug, and lets you access your local and global data through symbolic names. Unlike conventional debuggers, which are restricted to application space, Visual SoftICE has complete system access and can trace difficult problems between the system and application layers.

To learn more about Visual SoftICE, see the *Using Visual SoftICE* document.

Debugging System Crashes

A postmortem debugger is similar to a regular debugger, except that it can debug software that has already crashed by using crash dump files. You can use Compuware's postmortem debugger to view all aspects of a program as if the program were running under a regular debugger, paused at a breakpoint. When you open a crash dump file, you will get information about:

- **Stacks** – The current stack of function calls leading to the present program state. Each layer in the stack is called a frame. The present program state in the debugger is the point at which the system crashed.
- **Local Variables** – These are variables that are in scope at the currently selected frame.
- **Registers** – Data that is in the computer registers at the present program state.
- **Memory** – Contents of virtual memory at the present program state.

- **Disassembly** – A view of virtual memory that is interpreted in the form of machine code.
- **Source** – Code associated with functions and variables of the program.
- **System Data** – Properties of the system at the time of the crash, such as the reason for the crash (the bug check codes), the list of processes, loaded drivers, and active threads.
- **Debugger Extensions** – Kernel debugger extension commands (KD extensions) that allow you to examine crash dump information in different contexts.

If there is not enough information in the crash dump file to detect the cause of your crash, you can increase the type and amount of data that BoundsChecker Driver Edition logs and reproduce the condition. If the crash occurred on a "remote" computer, you can use the remote capabilities of BoundsChecker to debug the file. Assuming that the remote computer has Compuware's driver tools installed on it, you can use the Target Selector dialog to browse to the remote computer. Once connected, you can debug the remote computer as if it were right in front of you.

Testing Drivers on a "Live" System

Compuware provides many useful tools for debugging your driver on a "live" system. "Live" means that the system is in front of you and you are not attempting to debug a system crash file. With these tools, you can:

- **Start/stop non-boot drivers** – You no longer have to switch to the control panel to start your driver.
- **Log calls to DebugOut** – You can intercept calls to DebugOut and displays them in the output window. When used in conjunction with DriverWorks and DriverNetworks, channels can be used to filter out any unwanted strings. You no longer need a separate tool.
- **Take a snapshot of the system** – When you take a snapshot of the system, you get information like the Driver List, device stacks and other information that is useful for you to determine if you driver is behaving as expected. When used in conjunction with BoundsChecker Driver Edition or TrueTime Driver Edition, even more information is at your fingertips.

With the abilities of the Compuware driver tool set, you can test your driver from startup to shutdown.

Launching and Viewing BoundsChecker Data

BoundsChecker Driver Edition automatically detects dozens of programming errors in drivers for Windows Server 2003, Windows XP, Windows 2000, and Windows Millennium Edition by watching all calls into the operating system kernel. The trace log exposes the operation of the operating system and helps developers understand the interface between the operating system and the device driver.

In addition to detecting parameter errors, the tool maintains a record of the operation of selected device drivers. This record is available for display and analysis by both SoftICE and the DriverWorkbench tools. Without requiring any source code or driver modification, BoundsChecker can monitor the behavior of any driver, package a user-selected event into a data file and attach it to an e-mail to a user — in real time. It can also track memory allocations and de-allocations while exposing the source code that manipulates memory.

Developers can easily configure BoundsChecker to collect only the information needed, whether they want a broad collection of all systems events or a specific set of APIs in only one driver. Developers can also solve remote problems by distributing a configured BoundsChecker engine to collect events from deployed drivers. Event data can be stored and retrieved for detailed analysis.

BoundsChecker analyzes crash dump files generated by Windows when the target system being debugged crashes. These files can be analyzed locally or remotely from a DriverStudio host.

You can view data logged by BoundsChecker Driver Edition, including:

- Events logged by BoundsChecker including parameters, return values and certain structures (i.e., IRPs)
- Errors logged by BoundsChecker
- List of blocks of memory currently allocated including size and who allocated it
- List for resources currently allocated including the owner

Using BoundsChecker will give you insight into the way your driver behaves with the system as well as when your driver was called.

Launching and Viewing TrueTime Performance Data

Driver performance is largely defined by latency and data throughput, rather than by algorithmic or raw computational performance. TrueTime Driver Edition is a performance analysis tool allowing Windows Server 2003, Windows XP, and Windows 2000 device driver developers to

identify and fix local or remote performance bottlenecks easily, and all without requiring recompilation.

Without changing source code, TrueTime gathers comprehensive performance information, statistics and data for device drivers, and individual functions and operations within the drivers. The information gathered is presented graphically and permits developers to analyze driver performance visually. Using multiple views and filters, developers can review performance data on input/output packet lifetimes, function call frequency, code latencies, deferred procedure call timings and interrupt timing.

TrueTime is also capable of instrumenting both graphics drivers and NDIS drivers, including the automatic profiling of DirectDraw and Direct3D functions. In addition, TrueTime permits you to merge two or more instrumentation files into a single file. The current version of TrueTime hooks the IRP dispatch functions, ISRs, DPCs and I/O completion routines. Additional options allow the function tables exported by NDIS and video drivers to be hooked as well. Each time a dispatch function is called, TrueTime logs the function prolog and epilog time stamps.

Launching and Viewing TrueCoverage Data

TrueCoverage Driver Edition is an automatic coverage analysis tool that makes it easy to test drivers thoroughly. Newly redesigned from the ground up, TrueCoverage performs coverage analysis without the need for source code or symbols. An untested piece of code can cause a system crash. TrueCoverage helps developers figure out which parts of the code have been tested and which still need to be tested. The ability to measure and track code execution and stability during development saves testing time and improves code reliability.

Without changing source code, TrueCoverage gathers coverage data for drivers and allows developers to view the data in the context of the source code. Developers can also merge data accumulated over multiple sessions to aggregate coverage information.

How TrueCoverage Works

TrueCoverage organizes binary code into blocks interconnected by jump instructions. Then it counts how many times each binary block has been executed and displays a coverage report. In addition, TrueCoverage generates both two- and three-dimensional graphs of the code being analyzed, so that developers can see how the code is being executed.

When you use TrueCoverage Driver Edition during your testing, you know exactly how much of your code has been tested. You know exactly how many lines and how many functions were never tested. Since it is unlikely that you will test everything in one execution of an application, you can also accumulate coverage data from several sessions by merging the session files.

Viewing Debug Messages

DriverMonitor can display debug messages from any of the following sources running at local or remote machines:

- Kernel mode DbgPrint statements on Windows NT family systems
- VxD Out_Debug_String statements on Windows 9x systems
- User Mode OutputDebugString statements on both Windows 9x and Windows NT family systems

The DriverMonitor output is reachable either from within DriverWorkbench or from the MSVC.NET Shell.

About the DriverWorkbench GUI

Various DriverWorkbench tools support a number of ease-of-use features, including:

- The support of standard Cut, Copy and Paste using the clipboard
- The ability to save the contents of the page to a file
- The ability to Print and Print Preview
- The ability to create custom keyboard definitions
- The ability to load a custom workspace
- The ability to create and save a custom workspace
- The ability to create and save script file paths
- The ability of Visual SoftICE to set and follow a main context for all pages, and overriding page-specific contexts for Locals and Stack pages
- Special Visual SoftICE icons that indicate such things as Target State and Breakpoint Types

Preferences

Preferences are set on a global basis for all DriverWorkbench plugins, as well as on a per-workspace basis for specific components such as Visual SoftICE. You can set these global, per-workspace, toolbar, status bar, font, color, and keyboard preferences using the Preferences dialog.

To access the Preferences dialog:

1. From the File menu, select Preferences.
2. Click on one of the following Preferences dialog tabs to make your settings.
   a. Global Settings
   b. Per-Workspace Settings
   c. Keyboard
   d. Toolbars & Status Bar
   e. Fonts & Colors

Global Settings

When you select an element, the settings for that element are listed along with any value they may currently have. If you click on a setting, a description of that setting, any ranges (if applicable), and other rules on data entry (if applicable) are displayed in the Setting Explanation window below the list. The types of behavior controlled by these settings ranges from path definitions, global page properties for specific pages, and workspace saving and loading behavior.

Visual SoftICE Global Settings

Visual SoftICE global settings act on Visual SoftICE alone, and do not change with different workspaces. When you select an element, the settings for that element are listed along with any value they may currently have. If you click on a setting, a description of that setting, any ranges (if applicable), and other rules on data entry (if applicable) are displayed in the Setting Explanation window below the list.

Most of the settings deal with page properties, with the exception of Event Handling and Scripts.

Per-Workspace Settings

When you save your workspace, the values in the workspace take precedence from then on, even if they are automatically concatenated from the Application Settings data.

Workspace Save and Load

In addition to loading and saving custom workspaces manually, you can configure several automatic workspace loading and saving behaviors in DriverWorkbench. To configure workspace behavior, access the Workspace Save/Load element on the Global Settings tab of the Preferences dialog. You can configure the Workspace Properties shown in Table 1-1.

Table 1-1. Workspace Properties

Option	Description
Query to save workspace changes?	This option causes DriverWorkbench to ask you if you want to save the workspace, as a reminder, if you have made changes.
Save workspace on close.	This option automatically saves the workspace when you close it, or when you exit DriverWorkbench.
Startup: Load this workspace ...	This option allows you to designate a workspace to automatically load on startup.
Startup: Reload last workspace.	This option allows you to configure DriverWorkbench to automatically load the last workspace you had open on startup.

To change a True or False value for a workspace property, click in the Value field to toggle it. To specify a workspace for the Startup: Load this workspace property, click in the Value field to open it for edit and enter the workspace path and name. You can also use the browse button to browse your file system and select a saved workspace. If you click on a setting, a description of that setting, any ranges (if applicable), and other rules on data entry (if applicable) are displayed in the Setting Explanation window below the list.

Chapter 2
Using BoundsChecker Driver Edition

Introduction to BoundsChecker Driver Edition

The ability to automatically detect errors and log system calls has been the main focus of the BoundsChecker family of products for years. This functionality has been extended for driver writers with the BoundsChecker Driver Edition driver, BCHKD. An automatic error detection and event logging tool for driver writers, BCHKD monitors your drivers for common programming errors. BCHKD also collects data on a wide and configurable range of events, which can be viewed using the DriverWorkbench environment.

When a driver loads, BoundsChecker can hook over one thousand different API calls, entry points and driver callbacks. Users can configure BoundsChecker to monitor any subset of the system drivers for a detailed look at the interaction between drivers and the system. Since BoundsChecker Driver Edition logs the driver calls, it also does automatic error detection. (See “Errors Detected By BoundsChecker Driver Edition” on page 10 for a complete list.) A user can receive automatic feedback

about an error by configuring BoundsChecker to notify SoftICE or BoundsChecker Monitor that an error occurred.

BoundsChecker Driver Edition is supported on all Windows NT family platforms as well as on Windows ME.

What's New for Version 3.1?

BoundsChecker Driver Edition features for DriverStudio 3.1 include:

- Spinlock Tracking. (See page 14.)
- Automatic Configuration Updating (also known as "Dynamic Reconfiguration"). (See page 21.)
- Driver-Internal Function Logging. (See page 17.)
- BoundsChecker Select Functions page in the DriverStudio Configuration dialog. (See page 15.)
- Updated Summary Pane in the Events Window. (See page 34.)
- New Calling Function column in Resource Allocations View. (See page 39.)

Errors Detected by BoundsChecker Driver Edition

BoundsChecker automatically finds the following types of errors:

- **Memory Overrun:** A write occurred past the end of an allocated block
- **Memory Underrun:** A write occurred prior to the start of an allocated block.
- **Memory Leak:** Memory was allocated but not freed.
- **Resource Leak:** A resource was allocated but not freed.
- **Invalid Parameter:** BoundsChecker validates certain parameters to driver APIs.
- **Invalid Return Values:** BoundsChecker checks the return values for failure codes.
- **IRQL Violations:** BoundsChecker validates that the calls are made at the right IRQL.
- **DMA related errors:** Monitors for the correct handling of DMA related objects and routines.

- **Stack Overflows:** BoundsChecker monitors the current stack pointer to determine if it is within a specified number of bytes of the stack limit.

Enabling or Disabling BoundsChecker Driver Edition

The first step to successfully using BoundsChecker Driver Edition is properly configuring it to collect only events that are pertinent to your particular situation. BoundsChecker-DE can watch any combination of drivers and has the ability to intercept thousands of system level calls. Unfortunately, if not configured properly, BoundsChecker will collect so much data that the calls you are most interested in will be obscured. To configure BoundsChecker, select Settings from the Compuware DriverStudio menu. Once you have successfully set up your configuration, save it so you can quickly restore the settings at a later date.

BoundsChecker is enabled or disabled from the DriverStudio Configuration Dialog 's Control Panel Startup page in the DriverStudio program group.

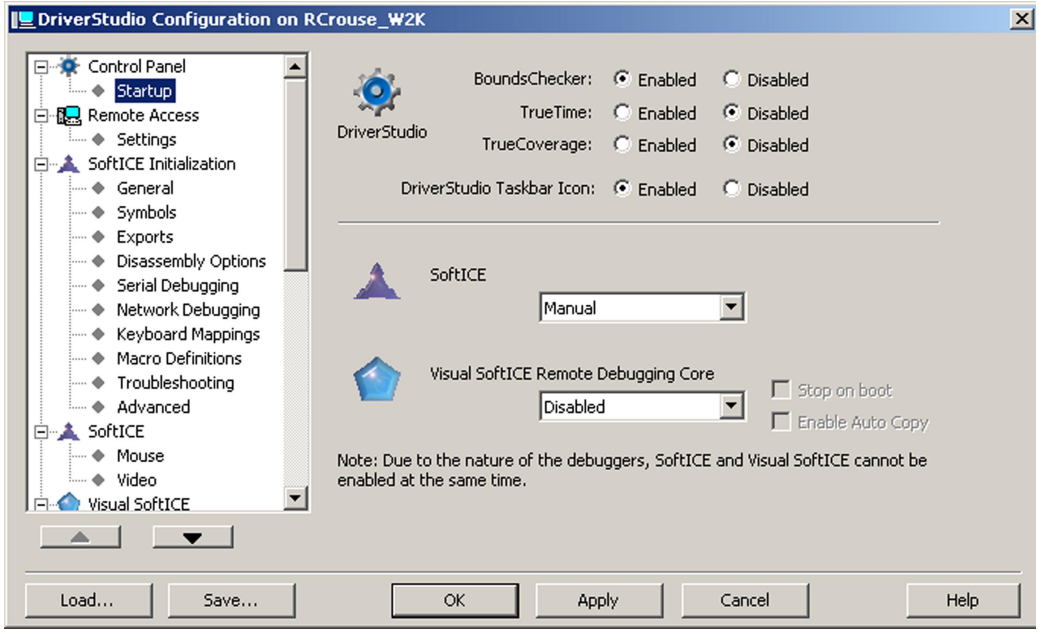


Figure 2-1. DriverStudio Configuration Dialog Startup Page

Configuring BoundsChecker Driver Edition

The first step to successfully using BoundsChecker Driver Edition is properly configuring it to collect only events that are pertinent to your particular situation. BoundsChecker Driver Edition can watch any combination of drivers and has the ability to intercept thousands of system level calls. Unfortunately, if not configured properly, BoundsChecker will collect so much data that the calls you are most interested in will most likely be obscured. To configure BoundsChecker, select **Settings** from the Compuware DriverStudio Start Menu, and go to the BoundsChecker General Page.

General Page

The BoundsChecker General Page is where you can configure BoundsChecker error notification and error trapping. Figure 2-2 shows the General Page for BoundsChecker settings.

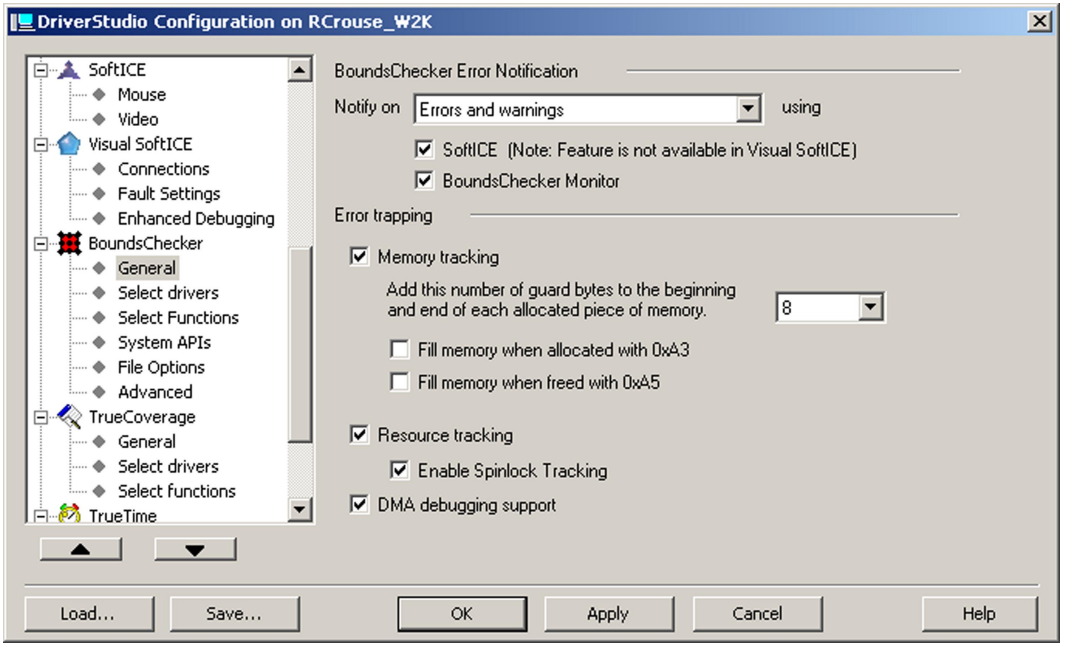


Figure 2-2. BoundsChecker Settings – General Page

BoundsChecker Error Notification

BoundsChecker can be configured to notify SoftICE and/or BoundsChecker Monitor when an error occurs.

When SoftICE is notified of an error, SoftICE will pop-up, stopping the machine and allowing you to see exactly what the last events that occurred were. For more information, see “Using BoundsChecker with SoftICE” on page 27.

When BoundsChecker Monitor is notified of an error, it takes a snapshot of the BoundsChecker event buffer and saves it to a file. The error is displayed in the BoundsChecker Monitor interface and further details can be found using DriverWorkbench. For more information, see “Using BoundsChecker Monitor” on page 46.

Error Trapping

Memory Tracking

When BoundsChecker intercepts a call to an allocation function like ExAllocatePool, it will put guard bytes before and after the allocated block of memory. BoundsChecker will periodically check these guard bytes to see if they have changed. If so, BoundsChecker knows that an overrun or an underrun occurred and will flag an error. The number of guard bytes is configurable (0, 8, 16, or 32) giving you a greater chance of finding wild pointer writes.

BoundsChecker also gives you the option of Memory Filling. Memory filling is the act of filling a piece of memory on allocation with 0xA3 or on de-allocation with 0xA5. By doing memory filling, you can quickly tell if your driver is reading uninitialized memory or accessing a memory block that has been freed.

When your driver unloads, BoundsChecker will log every block of memory that was allocated and not freed as a memory leak.

Following are some of the memory allocation functions that BoundsChecker logs:

- ExAllocatePool
- ExAllocatePoolWithQuota
- ExAllocatePoolWithQuotaTag
- ExAllocatePoolWithTag
- ExAllocatePoolWithTagPriority
- MmAllocateContiguousMemory
- MmAllocateContiguousMemorySpecifyCache
- MmAllocateNonCachedMemory

- NdisAllocateMemory
- NdisAllocateMemoryWithTag
- RtlAnsiStringToUnicodeString
- RtlUnicodeStringToAnsiString
- EngAllocMem

Resource Tracking

BoundsChecker also monitors the number and types of resources that your driver allocates. A resource is an object such as a registry key that your driver might create. At any time, you can get a list of currently allocated resources by using DriverWorkbench or SoftICE. When your driver unloads, any resource allocations that are still attributed to your driver are marked as resource leaks.

Following is a list of the types of resources that BoundsChecker monitors and an example of a call that would allocate such a resource:

- Directory objectsZwCreateDirectoryObject
- File handlesZwCreateFile
- Section objectsZwOpenSection
- EventsIoCreateNotificationEvent
- InterruptsIoConnectInterrupt
- IRPsIoAllocateIrp
- MDLIoAllocateMDL
- ThreadPsCreateSystemThread

Spinlock Tracking

Spinlock tracking is a subset of resource tracking (i.e., if resource tracking is turned off, Spinlock tracking is disabled). When this option is selected, BoundsChecker verifies that your driver is correctly using Spinlock detection. It will look for errors in Spinlock, Mutexes, and Fast Mutexes. It can find errors of the following types.

- Non-Initialized errors – User has not initialized a lock before using it.
- Double Initialized errors – User has initialized a lock more than once.
- Double Acquired errors – User has attempted to acquire the same lock twice.
- Double Released errors – User has attempted to release a lock twice.

- Thread Release errors – User attempts to release from wrong thread.
- Deadlock errors – User attempts to wait on a lock on more than one CPU.

DMA Debugging Support

For drivers that perform DMA operations between the host computer and a hardware device, BoundsChecker can assist in the debugging of DMA-related problems. When this feature has been enabled, BoundsChecker will monitor the selected driver's use of all DMA routines including those invoked through the DMA_OPERATIONS function pointers.

BoundsChecker will detect improper use of DMA buffers, adapters, and map registers. It will also detect improper or invalid function call sequences, such as a MapTransfer call before an associated AllocateAdapterChannel call, or a Master device making Slave operation calls.

Select Drivers

After selecting what types of errors to find, the next step in configuring BoundsChecker is to select the drivers that you want to monitor. This is done in the Select Drivers settings page (Figure 2-3). You may select any combination of drivers in this list. This list is obtained using the Win32 service control manager API. If a driver you are interested in is missing, you can add it using the **Add Driver** button. However, it would be better to keep the number of selected drivers as small as possible in order to keep unwanted information out of the event log.

Sometimes you need to select more than just your driver to get all of the information you are looking for. If your driver interacts with other drivers in the system, you will need to select those as well. For example, if you are writing a miniport driver, you will probably need to select the host controllers in order to get the information you are looking for.

Select Functions

After selecting which drivers to monitor on the Select Drivers Page, you can select which of each driver's internal functions you wish to monitor. The Select Functions Page (Figure 2-4) has three options:

- **Driver** – Select the driver-specific functions to monitor.
- **Symbol File** – Enter the name of the file where symbols will be found. You can browse for a different file rather than the default. Symbols must be located on the local machine.

- **Select functions to monitor** – This list contains all of the function names found in the symbol file. You can select the functions you want BoundsChecker to log in addition to the System APIs.

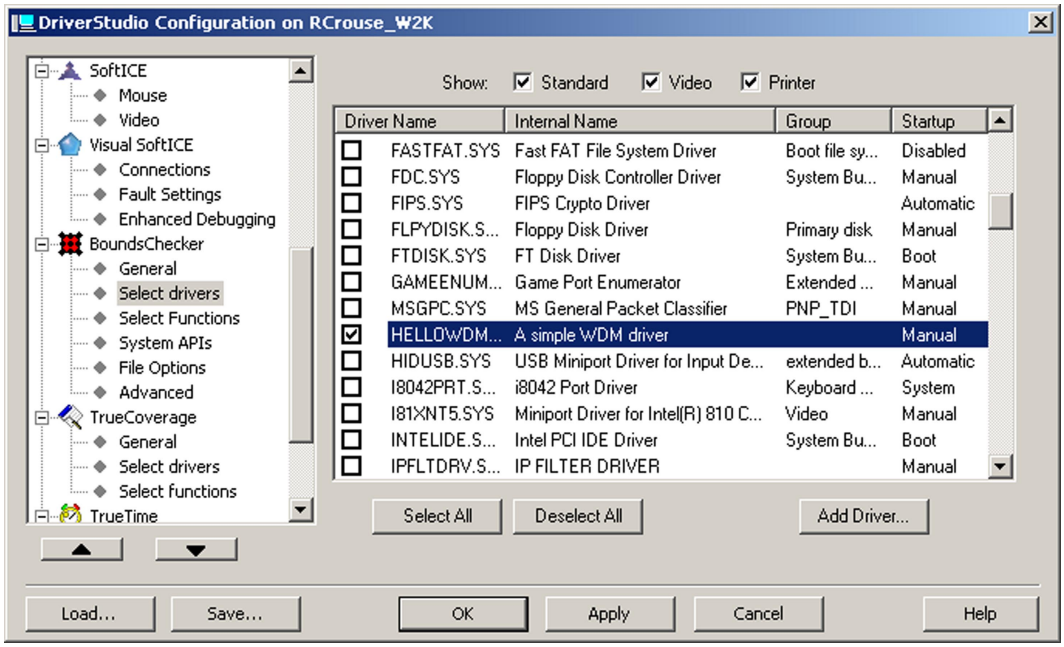


Figure 2-3. BoundsChecker Settings – Select Drivers Page

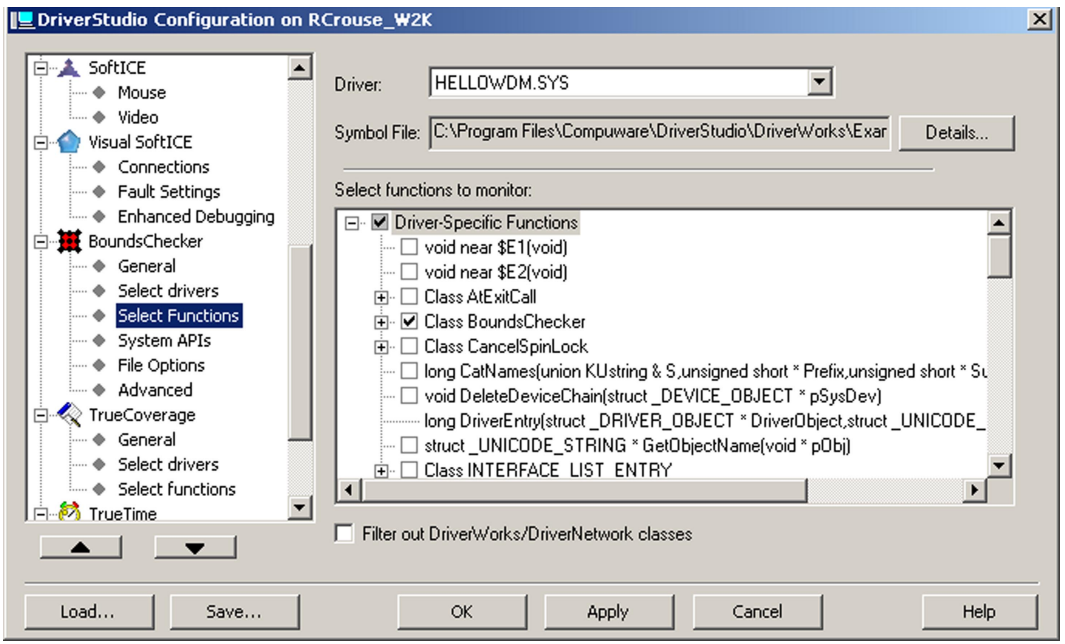


Figure 2-4. BoundsChecker Settings – Select Functions Page

Driver Internal Function Logging

BoundsChecker has always been able to monitor and log your driver's use of the system APIs and calls. For Version 3.1, BoundsChecker Driver Edition has added the ability to monitor and log functions that exist only in a driver under development. You can now see a logged output that contains every function that has executed in your driver. Using the new BoundsChecker Select Functions page, you can select each and every function that you want to monitor. This includes C++ class member functions; even static and overloaded functions. All parameters to the monitored functions are also included in the logged output, so you can examine your driver architecturally, as well as for parameter-passing defects. The logged results is more than often extremely useful in understanding the twists and turns that the driver takes on its way to a bug or defect.

System APIs

Normally, when you have selected only the drivers you are interested in, there is no need to modify which APIs to monitor. After all, if your driver made the call, then you probably want to see it. However, if you do want to selectively choose which APIs to monitor, you can do this on the System APIs page (Figure 2-5). The APIs that BoundsChecker logs are broken up into logical groups and each group has anywhere from 8 to 250 entries.

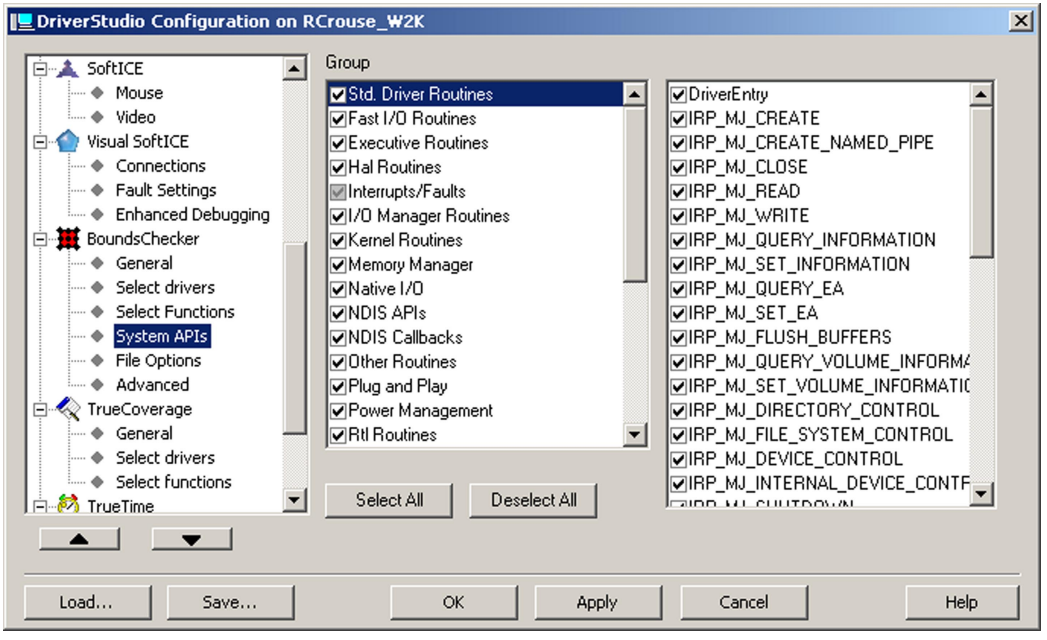


Figure 2-5. BoundsChecker Settings – System APIs Page

The groups of APIs are:

- **Standard Driver Routines** – Call from the operating system into a driver. Standard driver routines include IRP handlers, ISRs, and DPCs.
- **Fast I/O Routines** – A registered entry point in a driver that allows Windows NT to perform I/O without the overhead of creating and passing IRPs through packets. File system or transport drivers typically use these routines, but any driver can use them.
- **Executive Routines** – Support routines provided by the kernel to allocate memory, manage lists, and manipulate various other data structures and kernel objects.
- **HAL Routines** – The Hardware Abstraction Layer (HAL) provides interfaces to the CPU and hardware resources. It is designed to abstract hardware architectures across Windows NT platforms.
- **I/O Manager Routines** – Allows a driver to manage IRPs by communicating with the kernel I/O subsystem, as well as other drivers. These routines can allocate, deallocate, and cancel IRPs, and call lower drivers.
- **Kernel Routines** – Controls the scheduling and synchronization of threads, and manages the operation of kernel-mode applications and drivers. Kernel routines manipulate dispatcher and control objects. Dispatcher objects control thread scheduling and synchronization. They include events, mutexes, queues, semaphores, threads, and timers. Control objects manage all other kernel-mode operations. They include APC, DPC, device queue, interrupt, and process objects. Kernel routines whose names begin with Kf are the fastcall versions of the corresponding Ke routines.
- **Memory Manager** – Maps, frees, queries, or adjusts the attributes of memory address ranges. The Windows NT Memory Manager provides these routines.
- **Native I/O** – Internal file I/O support routine. These undocumented routines are very similar to the corresponding Win32 counterparts.
- **NDIS API/NDIS Callbacks** – NDIS Library function used by a network driver or a driver-registered MiniportXxx or ProtocolXxx routine. Windows NT supports Network Interface Card (NIC) drivers, intermediate protocol drivers, and upper-level protocol drivers. This can be a call from the driver into the kernel, or a callback into the NDIS driver.
- **Other Routines** – Object Manager routines, Process Structure routines, and Security Reference Monitor routines.

- **Plug and Play** (Windows 2000 only) – Kernel support routine used by drivers to interface with the Plug and Play subsystem.
- **Power Management (**Windows 2000 only) – Kernel support routine used by drivers to interface with the Power Management subsystem.
- **RTL Routines** – Call to one of the C Runtime Library Routines provided by the operating system for kernel mode drivers.
- **ZwXxx Event** – Documented service typically used by drivers to access the registry and file system from kernel mode.
- **USB Client Support Routine** (Windows 2000 only) – The USBD_Xxx events are routines exported from USBD.SYS. The USB_IRP_Xxx events represent DEVICE_CONTROL and INTERNAL_DEVICE_CONTROL messages sent to USBD.SYS.
- **Device Driver Interface Routine** – Allows Device Independence. The Display DDI events represent callbacks into the display driver. Display drivers export DrvEnableDriver at their entry point to initialize the driver and register a list of supported functions. GDI uses these registered functions to communicate with the display driver and produce the appropriate output. DDI function names are in the for DrvXxx.
- **GDI Event** – Corresponds to a call to services exported by win32k.sys. The names of general service routines begin with Eng. The names of specialized routines begin with the name of the related object (for example, BRUSHOBJ_pvAllocRbrush). GDI routines are divided into the following categories: surface management, palette services, path services, rendering services, or font and text services.
- **DirectDraw Event** – Functions exported from a display driver allowing the implementation or acceleration of DirectDraw functions. BoundsChecker monitors all callbacks provided by the following structures: DD_CALLBACKS, DD_SURFACECALLBACKS, DD_PALETTECALLBACKS, D3DNTHAL_CALLBACKS2, D3DNTHAL_CALLBACKS3, DD_MISCELLANEOUSCALLBACKS, DD_COLORCONTROLCALLBACKS, DD_VIDEOPROTCALLBACKS, DD_KERNELCALLBACKS.
- **SCSIPORT Routine** – Routine exported by the Microsoft SCSI Port driver.
- **SCSI Miniport Routine** – Driver-supplied callback that a SCSI driver registers with the SCSI Port driver
- **SCSI Driver Routine** – Driver-supplied callback that a SCSI Class driver registers with a SCSI Class system driver.

- **SCSI Class Routine** – Routine exported by the CLASS2.SYS SCSI Class system driver.
- **Class Plug and Play Routine** – Routines exported by the CLASSPNP.SYS SCSI Class system driver.
- **VideoPort Routines** – These routines are exported by the videoprt.sys system driver.
- **DriverWorks Event** – Generated by a DriverWorks device driver. These events are generated automatically by the checked version of the DriverWorks class library. DriverWorks drivers can also generate these events with an explicit function call to BoundsChecker.
- **DebugPrint Event** – Output sent from a driver or application to be displayed in a system debugger. BoundsChecker hooks these events globally; they are hooked for all drivers or none.
- **Interrupts/Faults** – Events such as page faults (Int0E), Int2Es and debug prints.

A driver writer can select only those categories and events that are pertinent to the type of driver they are running. For instance, a programmer writing video drivers probably doesn't need to hook the SCSI routines.

You can turn off entire groups by clicking the check box in the left hand list, or individual APIs by clicking checkboxes in the right hand list. By default, there are only 3 APIs turned off for you. They are:

- Int0E
- Int2E_AppMode
- Int2E_KernelMode

The reason these are turned off is they occur at such a high frequency that all other data in the BoundsChecker log will be lost. Turn these on only if looking for a particular problem that would require them. Otherwise it is highly recommended that you leave these APIs turned off.

Even after selecting only the drivers you are interested in, there is still the possibility that too much information will be logged to the BoundsChecker event buffer. Use the Driver APIs to filter this down further.

File Options

The File Options Page (Figure 2-6) has three options.

- **Output Directory** – Use this option to specify the directory for the BoundsChecker temporary files. These temporary files include event captures, temporary source files, and BoundsChecker event logs.
- **Write BoundsChecker events to file** – If this option is selected, BoundsChecker will write events to a log file called BCHKD.EVT. (This log is a text file and can be found in the Output Directory.)
- **Maximum Log Size** – This selection determines the overall size of the log file BCHKD.EVT. When this limit is reached, writing to this file is stopped.

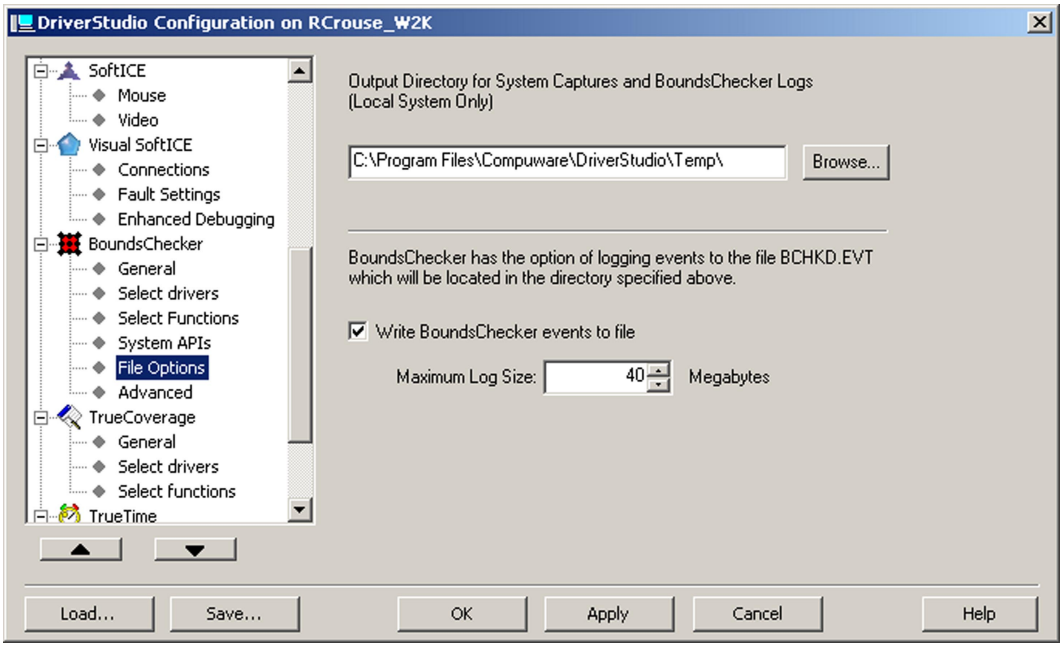


Figure 2-6. BoundsChecker Settings – File Options Page

Advanced

The Advanced Page (Figure 2-7) has three settings:

- **Stack Checking.** BoundsChecker checks the value of the stack pointer to see if it is within a certain number of bytes of the top or bottom of the stack. This way, stack problems can be avoided. You can set the number of bytes by changing this value.

Caution: Setting the number of bytes too high may cause BoundsChecker to incorrectly log errors, slowing down the boot process.

- **Event Buffer Size.** The BoundsChecker event buffer is circular. Thus, events will be overwritten after a certain point. You can specify how large the buffer should be to limit the entries that will be removed.
- **Dynamic Reconfiguration.** This option allows you to apply configuration changes without rebooting.

Note: The automatic configuration update feature will work on any driver that DOES NOT have a start value of zero in the service registry. Drivers that have a start value of zero are loaded by NtLdr rather than the operating system. These drivers can be configured and monitored in the same manner as in previous versions of BoundsChecker. Changes to their configuration settings can only take effect on a subsequent machine boot-up. These "boot" drivers can be identified easily using the SoftICE EVENT –d command.

There are a couple of caveats that you should be aware of before you use this feature:

- When the configuration is immediately updated, all previous driver tracking information will be lost. This affects such features as Memory Tracking, Resource, or Spinlock tracking.
- There are several BoundsChecker features that monitor pairs of functions or separately occurring events and statistics. If the configuration update occurs in the middle of such monitored events, the reporting of these events and features will be skewed.

Tip: *Restarting your monitored driver (either by unloading or by reboot) after the configuration has been modified always insures clean and accurate driver information.*

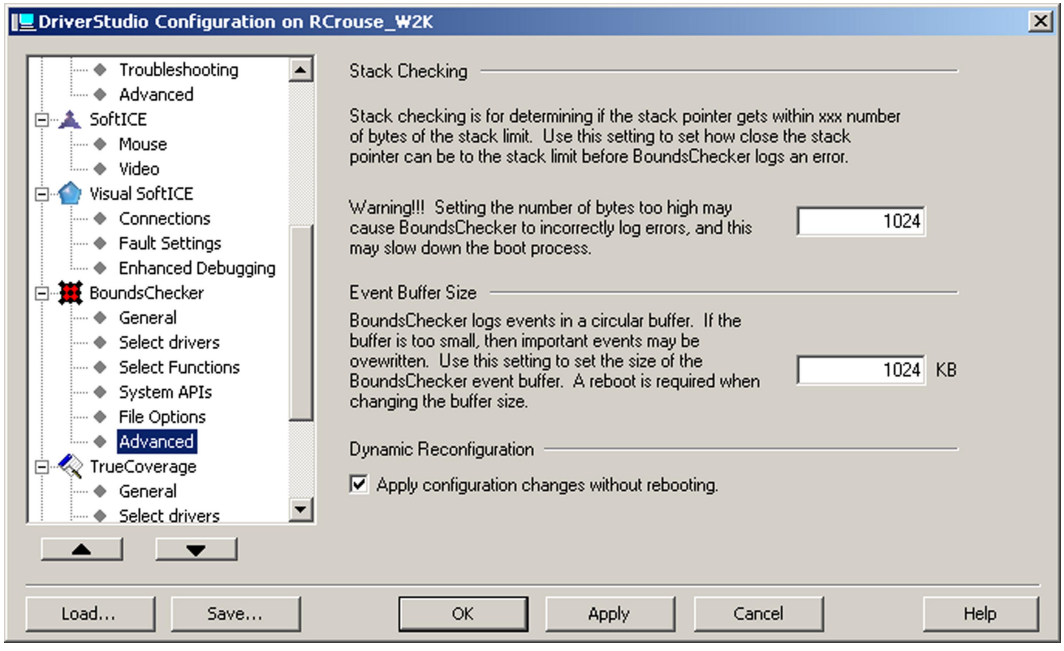


Figure 2-7. BoundsChecker Settings – Advanced Page

Controlling BoundsChecker Event Logging

Overview

One of the major functional points of the BoundsChecker Driver is that it exposes functions for the control of the logging buffer. The controls are can be used by several of the DriverStudio products and can be compiled into a users driver. There are four log controls exposed by the driver. They are Start, Stop, Reset and a special log function for custom messages. The selective use of the logging control can help the driver writer to quickly analyze and fix driver errors.

There are three ways to control event logging:

- Through DriverWorkbench or a development environment plugin
- By adding the logging control to your driver
- Through the SoftICE kernel debugger

Logging Control from DriverWorkbench

Logging controls for BoundsChecker are available through the DriverWorkbench menu bar (Figure 2-8). These logging controls are typically used when the driver to be tested will be run in a situation that closely duplicates its end use.

- **Start Event Logging:** Activates the logging when it is stopped.
- **Stop Event Logging:** Deactivates the logging when it is running.
- **Reset Events:** Clears the log buffer.
  - Events and Leaks
  - Events Only
  - Memory/Resource Leaks

One typical sequence of a test might be that the driver logging is stopped before the driver is loaded. At this point the buffer would be cleared using the reset menu. Logging would be restarted and the user's driver is now loaded. The driver would then be run in a manner that would give the driver a real mode test.

The next step is that the driver would be unloaded and the logging stopped. It is now that the developer or tester would use the "capture system information" menu to bring the log from the driver into the IDE. At this time the events can be analyzed.

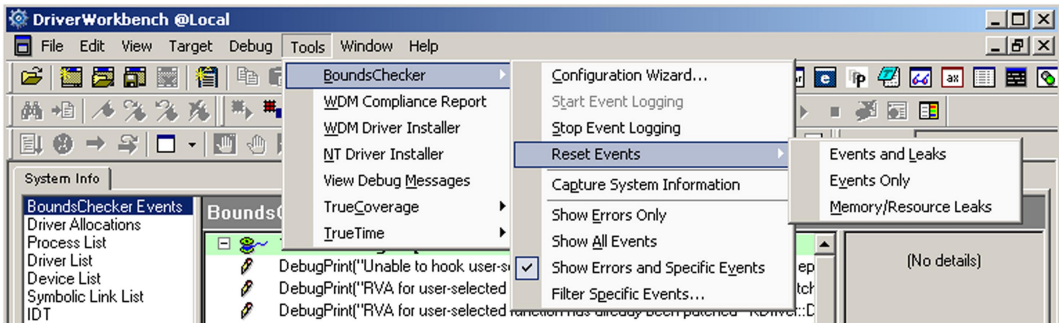


Figure 2-8. DriverWorkbench Tools Menu - BoundsChecker Submenu

Another possible scenario would be to stop the logging after driver load to analyze only that portion. Then moving on to another section of the driver. The idea of using the logging in this manner is to break the analysis of the driver into smaller sections.

Controlling Logging from Your Driver

The BoundsChecker Driver exposes several I/O controls (IOCTLs) to control logging activities. These simple controls are Start, Stop, and Reset. This gives you some flexibility in what the BoundsChecker driver will show for errors. You might use this feature to hide an error that you already know about and do not want displayed. It could also be used to give you a finer control of errors so that several areas of the driver can be refined while other sections are left for future debugging. If you are familiar with driver-to-driver communication, the method of calling the logging controls will be familiar to you, for it is the same as calling an IOCTL in a driver-to-driver situation.

Testing an entire driver can be very overwhelming. Using logging correctly allows you to concentrate on small areas of the driver at a time.

The fundamental steps are discussed in the following paragraphs.

Define the IOCTLs

The control codes for communicating with the BoundsChecker driver are shown below. Typically you would want to have these is a header file that is common to all drivers. This would give the highest degree of reusability. However, these definitions could be placed anywhere the driver write desires.

```
// BCHKDLogger.h
//
```

```
// General BoundsChecker logging definitions.
// These define the BoundsChecker Driver logging interface. //

#ifndef __BCHKDLogger_h__
#define __BCHKDLogger_h__

// Hard-coded number for the BCHKD device
#define FILE_DEVICE_BCHKD  0x00008400

// IOCTL for BoundsChecker logging stop.
#define IOCTL_BCHKD_LOGGINGSTOP(ULONG)CTL_CODE(FILE_DEVICE_BCHKD,0x40,
METHOD_BUFFERED, FILE_ANY_ACCESS )

// IOCTL for BoundsChecker logging start.
#define IOCTL_BCHKD_LOGGINGSTART(ULONG)CTL_CODE(FILE_DEVICE_BCHKD,0x41,
METHOD_BUFFERED, FILE_ANY_ACCESS )

// IOCTL for BoundsChecker logging clear.
#define IOCTL_BCHKD_LOGGINGCLEAR(ULONG)CTL_CODE(FILE_DEVICE_BCHKD,0x42,
METHOD_BUFFERED, FILE_ANY_ACCESS )
#endif// __BCHKDLogger_h__
```

Connecting to the BoundsChecker Driver

This will most likely be done in your driver in the function *DriverEntry*. Then, you can use it whenever it is required throughout the driver.

```
//This would be declared Globally (Device extension maybe?)
PDEVICE_OBJECT bchkdDeviceObject;
PFILE_OBJECT   bchkdDeviceFileObject;

status=IoGetDeviceObjectPointer(&bchkdNameString,STANDARD_RIGHTS_ALL,
&bchkdDeviceFileObject, &bchkdDeviceObject);

//BylookingatIoGetDeviceObjectPointerinMSDNhelponecanfigureout
the aurguments.

Using a device IOCTL to control the buffer.
Theactualloggingcallrequiresthreesteps.Theevent,thecalltothe
BCDriver, and the wait for the event.
```

```
// Initialize the event before IOmanager gets it
KeInitializeEvent(&event, NotificationEvent, FALSE);

//Create an IRP to send to BoundsChecker.
irp = IoBuildDeviceIoControlRequest(
< IOCTL CODE >,  //The IOCTL we want called.
bchkdDeviceObject,  //The BoundsChecker device object.
          <MSG>,                          // user comment label
<MSG> ? strlen(<MSG>)+1 : 0,   // length of string and terminator
           NULL,
           0,
           FALSE, // use IRP_MJ_INTERNAL_DEVICE_CONTROL
          &event, // event to be signaled on completion ( wait
for it below )
           &ioStatus);

//Send the created IRP to BoundsChecker.
status = IoCallDriver(bchkdDeviceObject, irp);
```

The steps described above are straight forward. A little additional detail will allow you to use the logging feature effectively.

The event and the wait on event are described in MSDN. The call to **IoBuildDeviceIoControlRequest** is detailed there also. However, there are two items in this call that deserve special attention: the <IOCTL CODE> and the MSG.

The IOCTL CODE is dependent on which of the three logging controls you wish to execute. The choices are:

- IOCTL_BCHKD_LOGGINGSTOP
- IOCTL_BCHKD_LOGGINGSTART
- IOCTL_BCHKD_LOGGINGCLEAR

The MSG is a char* location that can be used to mark the logging events with a message from the driver developer. This message can be used for almost anything. For example, you might use this message to mark the type of test that was run, or possibly mark the reason for executing the IOCTL.

Using the SoftICE Kernel Debugger to Control Logging

Probably the most versatile and less intrusive way of using BoundsChecker logging control is by working these controls in

conjunction with SoftICE. The distinct advantage of using SoftICE is because areas of the driver can be checked in a tightly-controlled situation.

Some simple commands allow for the logging control. The SoftICE commands are “Event –o” to turn on/off the logging and “Event –r” to reset the log buffer. A typical sequence for controlling the logging of errors with Softice would be to set a breakpoint in the driver at the opening and closing brackets to the function to be checked. When the breakpoint is hit and SoftICE pops up (on the opening bracket), you can issue the Softice command “Event –r” (clearing the log).

Breaking out of Softice places the machine back into run mode. The driver executes the current function and breaks on the closing bracket of the function. From this point, SoftICE event commands can be used to analyze the information. Refer to the *Using SoftICE* manual for more details on event views.

Using BoundsChecker with SoftICE

After you install DriverStudio, you can view BoundsChecker events in SoftICE with the EVENT command. SoftICE can display events in the Command window or in a separate, scrollable Event window.

```
EVENT [-? | -a | -lx | -nd | -o | -pd | -r | -s | -t |
-x]
[start-event-index [Levent-count]]
```

-?	Displays descriptions of the supported command switches.
-a	Turns API return display on or off. The default setting is on. When this setting is off, SoftICE does not display API return events.
-lx	Specifies the stack-checking level (0x40 - 0x4000). The default setting is 0x800.
-nd	Specifies the nesting depth used to display events. Legal values are 0 to 32 (decimal format). The default nesting level is 10. If events nest past the specified nesting depth, SoftICE does not display them as indented.
-o	Turns event logging on or off. The default setting is on.
-pd	Specifies the SoftICE pop-up level for BoundsChecker events. The default setting is 0. 0 - SoftICE does not pop up on BoundsChecker events. 1 - SoftICE pops up on errors only.

	2 - SoftICE pops up on all errors and warnings.
-r	Clears the event buffer.
-s	Displays the current status of event viewing and logging. The number of logged events is the total that have been trapped since the system was started. It is displayed in decimal format.
-t	Turns display of thread switches on or off. The default setting is on. When this option is on and event n-1 is in a different thread than event n, SoftICE displays event n in reverse video indicating a thread switch has occurred. When this option is off, SoftICE does not display thread switches.
-x	Displays all events with their parameters, as well as general summary information for each event, including elapsed time, current thread and current IRQL. If you do not specify this switch, SoftICE displays a single summary line for each event.
start-event-index	Displays events starting at the specified event index.
event-count	Displays the logged events in the Command window, starting from the specified *start-event-index* for a length of *event-count* events. If you do not specify a length, SoftICE displays the events in a scrollable window starting from *start-event-index* (if one is specified).

Viewing Events in the Event Window

Enter the following command at the command prompt to display events in the Event window.

```
EVENT
```

When you do not specify *start-event-index* or *event-count*, SoftICE displays the Event window in place of the Command window. You can use this command with one of the EVENT command switches or with a *start-event-index* to customize the display. Press **Esc** to close the Event window.

You can specify whether SoftICE displays the events in the Event window with summary or detail information. While the Event window is open, you can use F1 to expand or collapse all events. You can place the cursor on a line and double-click or press Enter to expand or collapse a single event. The Event window supports the following keys.

Table 2-1. Event Window Keys

Key	Description
Enter	Toggles the display state of the event at the current cursor position between summary information and detail information.

Table 2-1. Event Window Keys (Continued)

Key	Description
Esc	Closes the Event window. When you re-open the Event window, SoftICE preserves the previous window state (i.e. current event, expansion state, and filters are the same).
PageUp	Scrolls the screen up one page.
PageDown	Scrolls the screen down one page.
Up Arrow	Moves cursor up one line. If on the top line, it scrolls the window up one line.
Down Arrow	Moves cursor down one line. If on bottom line, it scrolls window down one line.
Shift-Left Arrow	Scrolls the window left one column.
Shift-Right Arrow	Scrolls the window right one column.
Home	Moves the cursor to the top row. If the cursor is already on the top row, starts display at the first event.
End	Moves the cursor to the bottom row. If the cursor is already on the bottom, starts display at the last event.
*	Undoes the last Home or End operation.
F1	Toggles the display state of all events between summary information and detail information.
F2	Displays the Event filtering dialog.
F3	Displays the Parameter filtering dialog.
F4	Displays error events only.
F	Closes the Event window and returns focus to the Command window. Use this key if you want to use other SoftICE commands on data that is displayed in the Event window. If you bring up the Event window again, SoftICE preserves the previous window state (i.e. current top event, expansion state, and filters are the same).
R	Toggles the display state of API returns between showing all API returns and showing no API returns.
T	Toggles the highlighting of thread switches. Thread switches are indicated by displaying the summary line of the first event in the new thread in reverse video.
E	Toggles the highlighting of errors on API returns. SoftICE displays the summary line of API return errors in bold.

Table 2-1. Event Window Keys (Continued)

Key	Description
S	Displays the event at the current cursor position at the top of the Event window.
N	Finds the next event that matches the search criteria selected with the right mouse button.
P	Finds the previous event that matches the search criteria selected with the right mouse button.
0 - 7	Filters events by CPU number on SMP machines. Each key acts as a toggle for displaying all events that occurred on a specific CPU. These keys also appear as buttons on the top line of the Event window.

Viewing Events in the Command Window

Enter the following command at the command prompt to display events in the Command window starting at event *start-event-index* for a length of *event-count* events.

```
EVENT start-event-index Levent-count
```

SoftICE can display any number of events in the Command window, starting from any specific event index, and with summary or detail information. The summary display includes only a single line for each event. The detail display includes the summary information plus all event parameters. You can use the EVENT command switches to customize the display output.

It is useful to view events in the Command window when you want to either view a small group of functions, or save the event data to a SoftICE History file. A SoftICE History file contains current contents of the SoftICE history buffer. Use the Command window scroll bars to view the contents of the SoftICE history buffer.

To Save a SoftICE History File

1. Open the Symbol Loader.
2. On the **File** menu, select **Save SoftICE History**.
3. Choose a location and file name.
4. Click **Save**.

Specifying Source and Symbol Paths

To set source and symbol paths from within DriverWorkbench, select **Preferences...** from the File menu. On the **Global Settings** tab, under **Element,** expand the **General** option and select **Paths.** There are several options here that you may choose from, but the entries that pertain to BoundsChecker views are **Source Search Paths** and **User Symbol Paths**.

Note: Since Version 3.0, BoundsChecker views no longer use the symbol server paths. For setting symbol paths, see the description "**Setting OS Symbol Paths**."

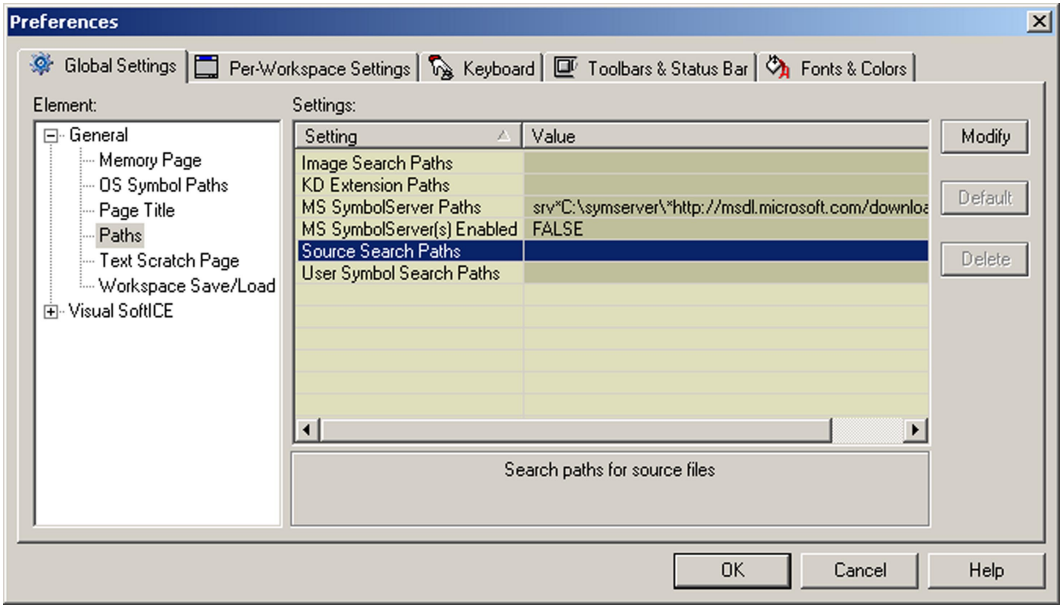


Figure 2-9. DriverWorkbench Preferences Dialog - BoundsChecker

To change either path, click once in the Value column to do in-place editing, or double-click on the entry in the Value column to bring up the Path List Edit dialog (Figure 2-10).

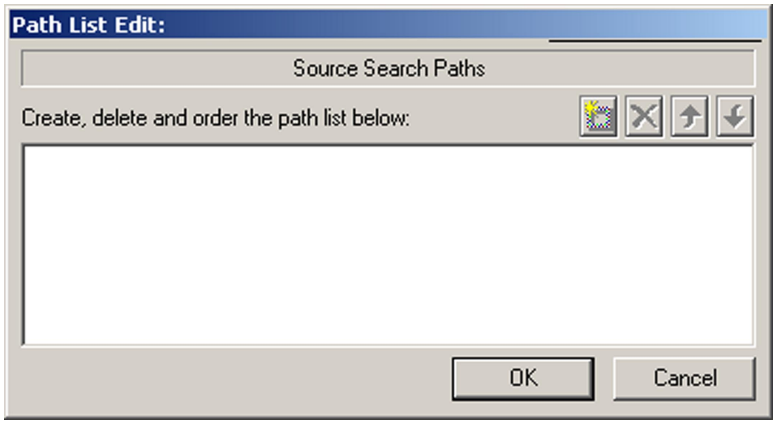


Figure 2-10. Path List Edit Dialog - Source Search Path

The four buttons on the dialog are used as follows:


Click on this button to browse for a path and add it to the list.


Click on this button to remove a path.


Move a path up in the search order. Paths that appear first will be searched first.


Move a path down in the search order.

Note: The "\..." following a path instructs DriverWorkbench to search all sub-directories of the specified path. If you do not want DriverWorkbench to perform this search, remove the "\...".

Once search paths have been specified, DriverWorkbench looks for symbol files using the following steps:

1. DriverWorkbench looks for the name of the file with a .DBG extension. Symbol files for the OS (prior to WinXP) were DBG files.
2. DriverWorkbench looks for the actual file (i.e. testdrv.sys). If this file is found, DriverWorkbench opens it and pulls the location of the symbol file from the PE header. If the symbol file does not exist in the same directory it was created in, it must exist in the same directory as the driver.
3. If the driver is not found, DriverWorkbench looks for the file name with a **.PDB** extension.

4 If none of the above steps work, DriverWorkbench looks for the file name with a **.SYM** extension

Specifying OS Symbol Paths

When viewing BoundsChecker data, DriverWorkbench uses the OS Symbol paths to find and load symbols for the operating system. Users can set up different symbol paths for different OS. When DriverWorkbench loads BoundsChecker data, it retrieves the OS information and automatically uses the correct symbol path.

BoundsChecker Options

Options relating to BoundsChecker can be found under **BoundsChecker** on the **Tools** menu. The options are:

- **Configuration Wizard**. Select this option to start the BoundsChecker settings dialog and quickly configure the BoundsChecker settings. After configuring BoundsChecker, the driver will be enabled and a reboot will be required.
- **Start Event Logging**. Notify BoundsChecker that it should start logging events to the buffer. For examples of why you might do this, refer to the "Controlling BoundsChecker Logging" section of this document on page 23.
- **Stop Event Logging**. Notify BoundsChecker that it should stop logging events to the buffer. For examples of why you might do this, refer to the "Controlling BoundsChecker logging" section of this document on page 23.
- **Reset Events**. Using this selection you can reset the event buffer, which means removing all events, and memory allocations. Memory leaks are separate from the event stream so when trying to track down memory leaks, it is suggested that you clear the leak entries. Otherwise, you will get leaks from a previous load of your driver.
- **Capture System Information**. With this option, you can take a snapshot of the BoundsChecker event buffer. Refer to the "Controlling BoundsChecker Logging" section on page 23 for more details about capturing system information.

System Info View

After selecting ***Capture System Information***, BoundsChecker data is displayed in the System Info view.

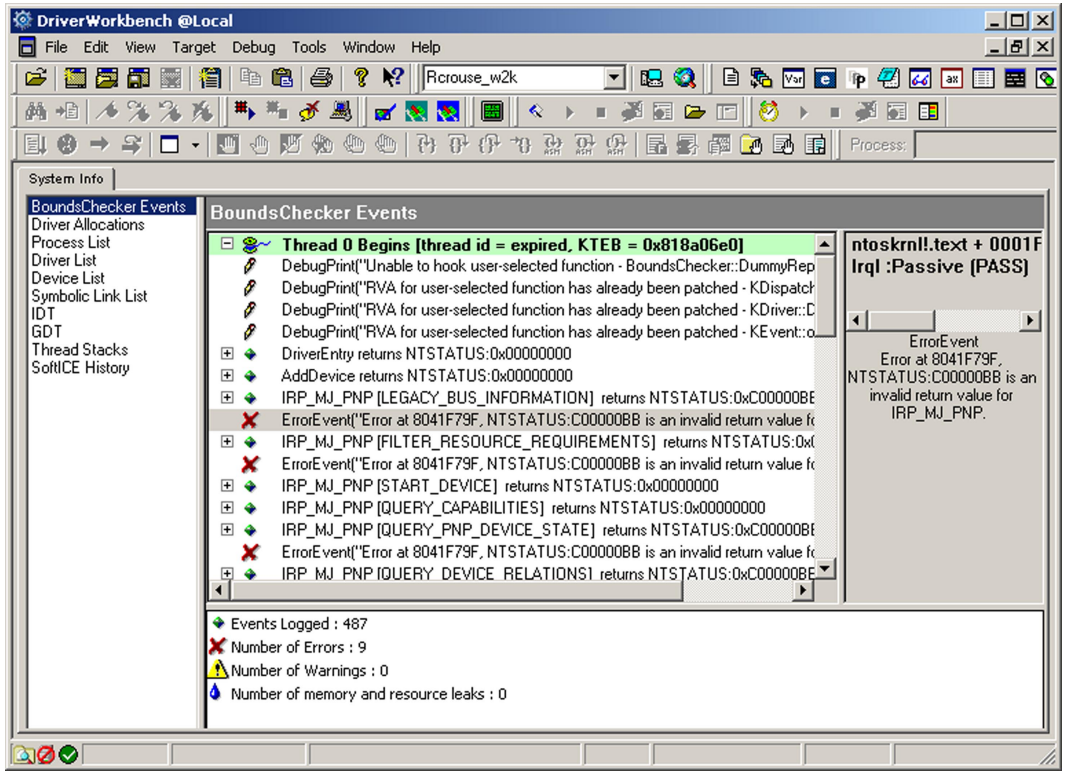


Figure 2-11. System Info View

BoundsChecker Events

The BoundsChecker Events view is a hierarchical display of the events logged by the BoundsChecker driver. The event view comprises three panes: the *Event Pane*, the *Details Pane*, and the *Summary Pane*.

Event Pane

The event pane lists all events and errors that the BoundsChecker driver has logged. The events are displayed as a tree in order to properly show nested calls. Besides the name of the event, there is additional information available for display in the event pane. This information includes:

- **Arguments**. Parameters passed to the function.
- **Elapsed Time**. Number of milliseconds that passed between the execution of this event and the one previous to it.
- **Absolute Time**. Number of milliseconds that have passed since BoundsChecker loaded and the execution of this event.

- **Sequence Numbers**. The index of the selected event in the BoundsChecker event stream.

Each of these options is available from the **View** menu. The Event Pane also displays thread interactions as context switch events (i.e., when one thread gains controls of the CPU). When a machine has multiple CPUs, the Event Pane uses color-coded bands on the left side to identify which CPU executed each event.

Users can also filter the displayed events in the event pane. For more information, see "Filtering BoundsChecker Events" on page 36.

For every event, BoundsChecker logs two levels of the call stack (that called the function). Users can display the source for these levels from the Event Pane context menu (right-click to select).

Select the option **Go To Source of > Caller** and choose the stack entry you would like to see. If source code for this location is available, it will be displayed, otherwise the default view will be an assembly listing. For more information on setting up source and symbol paths for your drivers, see "Specifying Source and Symbol Paths" on page 31.

Details Pane

The Details pane displays information specific to the current event. For example, if an API call/return event is selected, the Details Pane will display:

- The name or address of the calling function.
- If applicable, the name or address of the callee function.
- The IRQL this event was executed at.
- The types and values of each parameter passed to the function.

When displaying the parameters, in all cases you will be shown the type of the parameter (i.e.,, DRIVER_OBJECT), a default parameter name, and the parameter value. For some structures, such as DRIVER_OBJECT, IRP and UNICODE_STRING, BoundsChecker logs the entire structure to the event log. This information is displayed in the Details Pane.

You can display the memory location associated with any parameter in the memory window by right clicking on the parameter value and selecting **Go To Address**.

Note: The event that you are looking at in the event view might have occurred several seconds or minutes before you did the actual capture. In this case, there is a good chance that when these addresses are displayed, the memory window will not show the data you expect. That's why BoundsChecker logs the data as part of the event stream. Display memory for a parameter may only be valid for the last few events in the view.

Summary Pane

The summary pane shows:

- Total number of events logged.
- Total number of errors.
- Total number of warnings.
- Total number of memory and resource leaks.

The summary pane can be turned on and off by selecting Event Summary Pane from the DriverWorkbench View menu or the BoundsChecker Events context menu.

Filtering BoundsChecker Events

After you have captured the events you are interested in, you will find that there is still a lot of data to sort through. You can filter the displayed events by placing the mouse cursor in the Event Pane, launching the BoundsChecker context menu and selecting ***Filter Specific Events***. This launches the Filter Specific Events dialog (Figure 2-12).

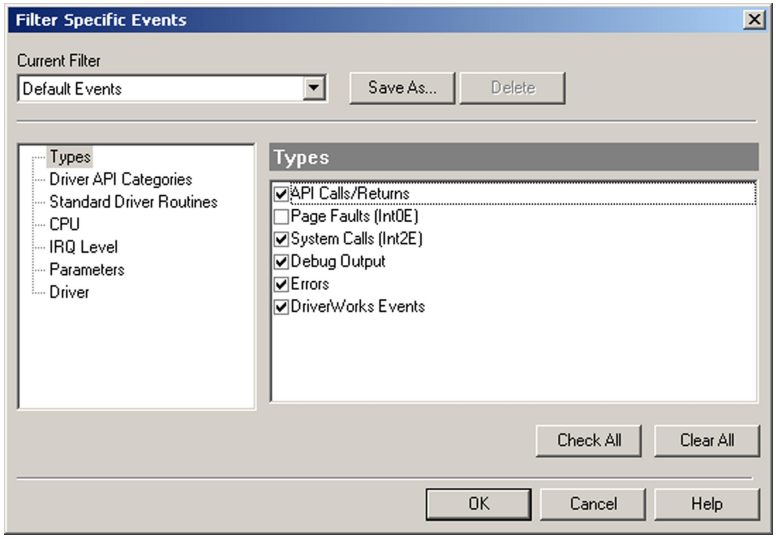


Figure 2-12. Filter Specific Events Dialog

Filtering allows you to perform tasks like following the lifespan of a particular IRP, or seeing which events might have executed at a higher IRQL level.

Filters are available for:

- **Types**. These are common groups of events that you may want to filter on.

- **Driver API Categories**. Filter based on an entire API (Application Programming Interface) group. For more information on the API groups, see “Driver APIs.”
- **CPU**. Filter events based on the CPU that executed them.
- **IRQL**. Filter events based on the IRQL when the event was executed.
- **Parameters**. Filter events based on specific parameters. For instance, if you wanted to see every event that took a particular IRP as a parameter, you would type PIRP (name of the parameter) into the Type field and the value of the IRP into the Value field. Finally, click Insert. This will add the filter for the IRP and selecting OK will apply the filter to the current view.
- **Driver**. When monitoring more than one driver, you may want to only see events where a particular driver was either the caller or callee.

Once you have set up your filters, you can save the filter by clicking the Save As button. All saved filters will be displayed in the Current Filter drop-down list.

Driver Memory Allocations

The Driver Memory Allocations list (Figure 2-13) contains all memory blocks currently allocated to the drivers BoundsChecker is monitoring.

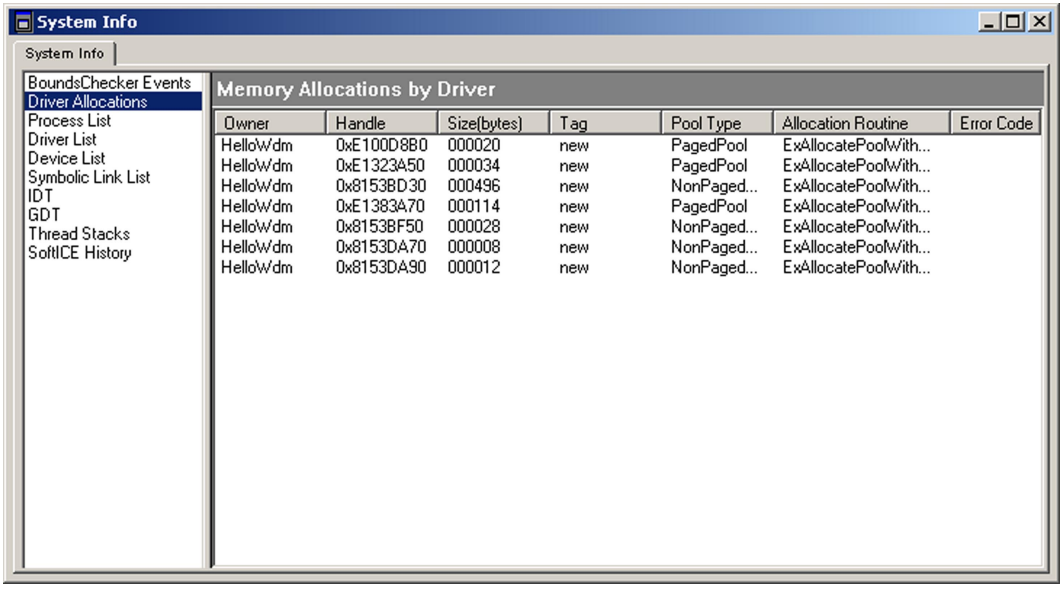


Figure 2-13. Driver Memory Allocations List

The following information is displayed for each allocated block of memory.

- **Owner**. This is the name of the driver that called the allocation function.
- **Handle**. The pointer to the memory block returned by the allocation function.
- **Size**. The size, in bytes, of the allocated piece of memory.
- **Pool Type**. The memory pool from which the memory was allocated.
- **Allocation Routine**. The name of the function that was called to do the allocation.
- **Error Code**. This reflects if there is a problem with the block of memory. The possible memory errors are Memory Overrun, Memory Underrun or Memory Leak. For more information on these errors, see "Errors Detected by BoundsChecker" on page 10.

Displaying Source Code for an Allocated Block of Memory

When BoundsChecker logs a memory allocation call, it also logs who made that call. You can view the source of the caller from within DriverWorkbench by selecting the memory allocation you are interested in, and then from the context menu, select **Call Stack** (Figure 2-14). There are two entries (for two stack levels). Select the stack entry you would like to see. If a source file is available, it will be displayed, otherwise you will be shown a disassembly listing.

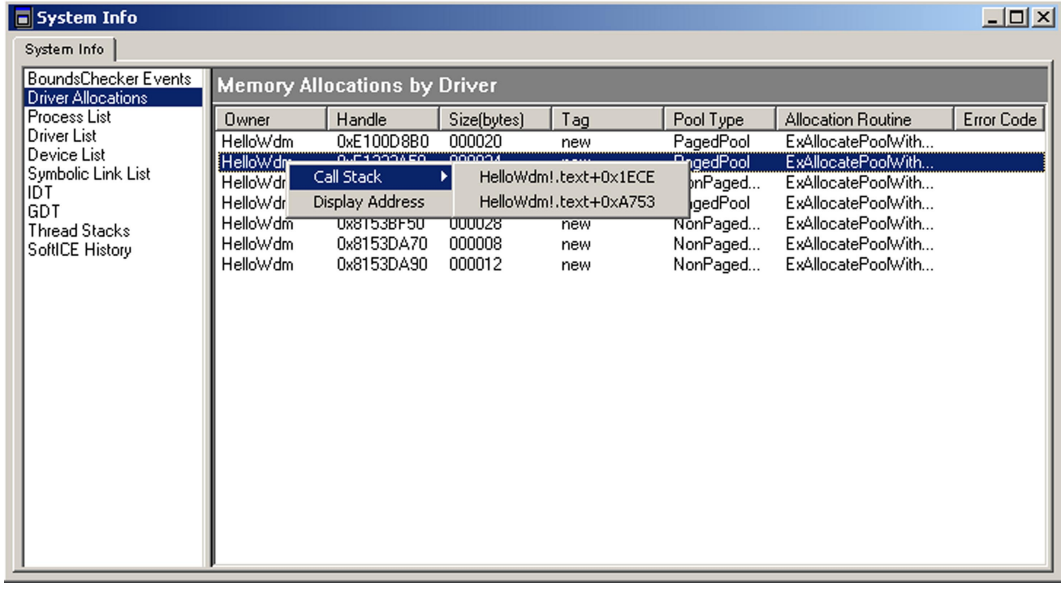


Figure 2-14. Selecting the Call Stack

Process List

The **Process List** window displays a list of the active processes on the system at the time of the crash. Located at the bottom of the window are the Modules tab (Figure 2-15) and the Threads tab (Figure 2-16). The table below lists the possible process and provides a description for each one.

- **Process.** Process Description
- **CR3**. Physical address of the process page table.
- **LDT Base.** Base address of the local descriptor table.
- **LDT Limit**. Size of the local descriptor table (if one is present).
- **KPEB**. Address of the kernel process environment block.
- **PID**. Process ID.
- **Status**. Process status.

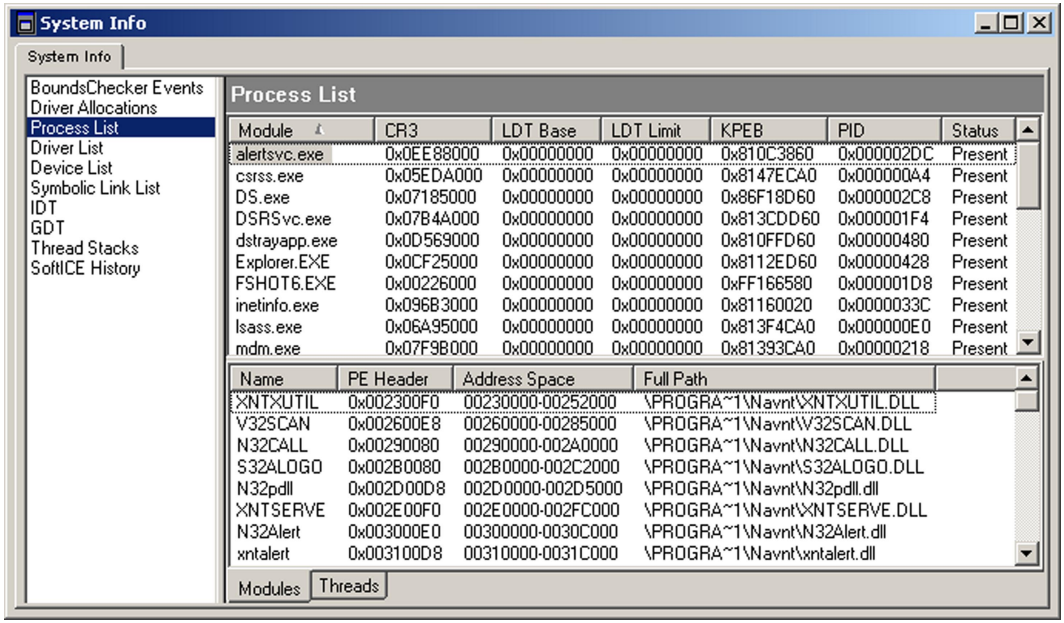


Figure 2-15. Process List (Modules Tab Selected)

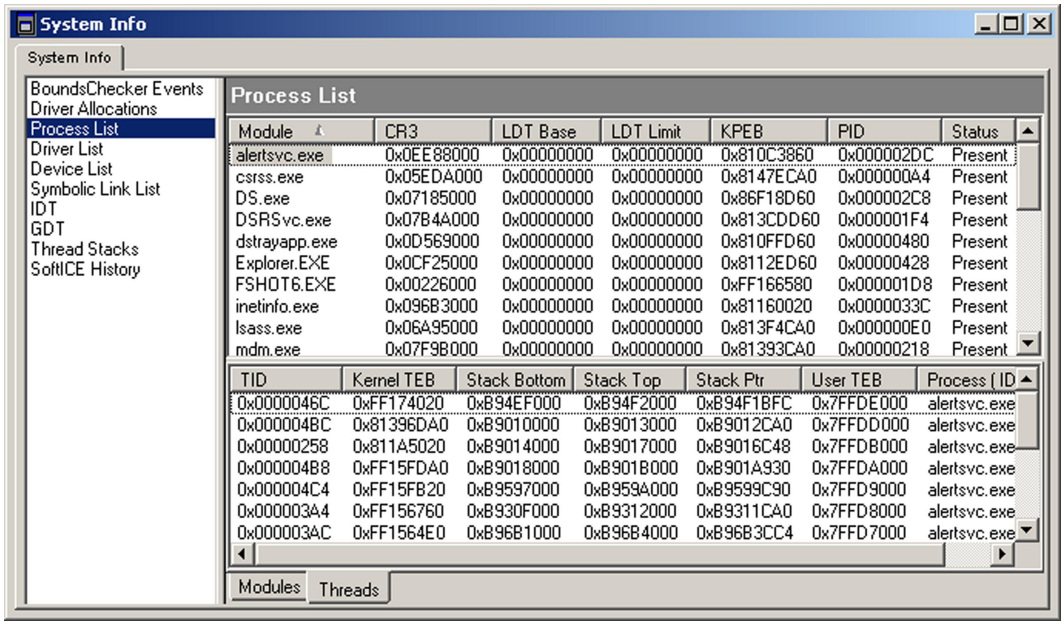


Figure 2-16. Process List (Threads Tab Selected)

Driver List

The **Driver List** (Figure 2-17) contains a list of all the drivers loaded on the system at the time of the crash or creation of the system information file.

- **Name**. Name of the driver.
- **Address Space**. Address space where the driver was loaded.
- **Memory Allocations**. Number of memory allocations made by the driver.
- **Memory Used.** Total bytes allocated by the driver.
- **Memory Errors.** Number of BoundsChecker memory errors generated by the driver.
- **General Information**. Information about the driver from the driver object structure.
- **Entrypoints**. List of the entry point functions for the driver.

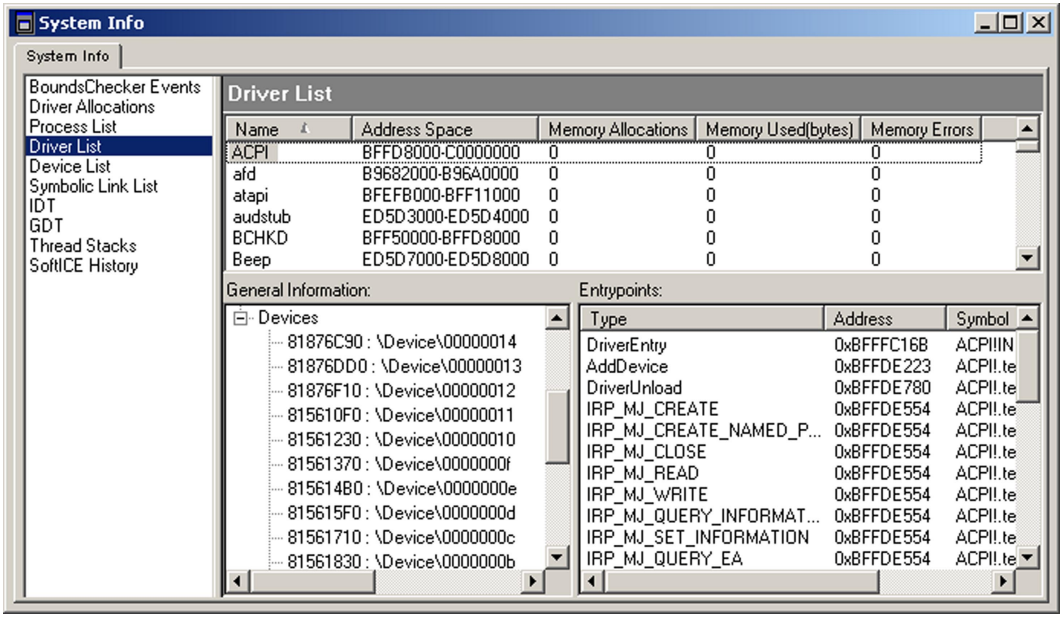


Figure 2-17. Driver List

Device List

The Device List (Figure 2-18) contains a list of all the devices on the system at the time of the crash or creation of the system information file.

- **Name**. The name of the device.
- **Type**. Type of the device object. (Device types are defined in NTDDK.H file.)
- **Owning Driver**.
- **General Information**. Information about the device from the device object structure.
- **Device Stack**.

Symbolic Link List

Symbolic Link List (Figure 2-19) contains a list of the names of every symbolic link object in the system as well as the name of the corresponding object that the link aliases.

- **Name.** Symbolic device name.
- **Linked Object Name**. Device object represented by Name.

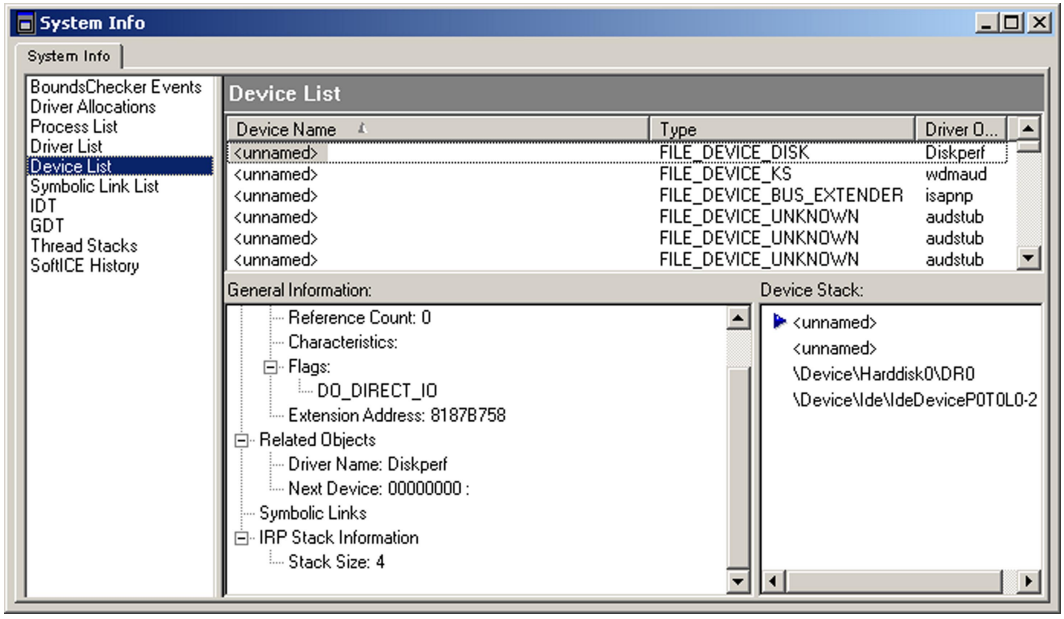


Figure 2-18. Device List

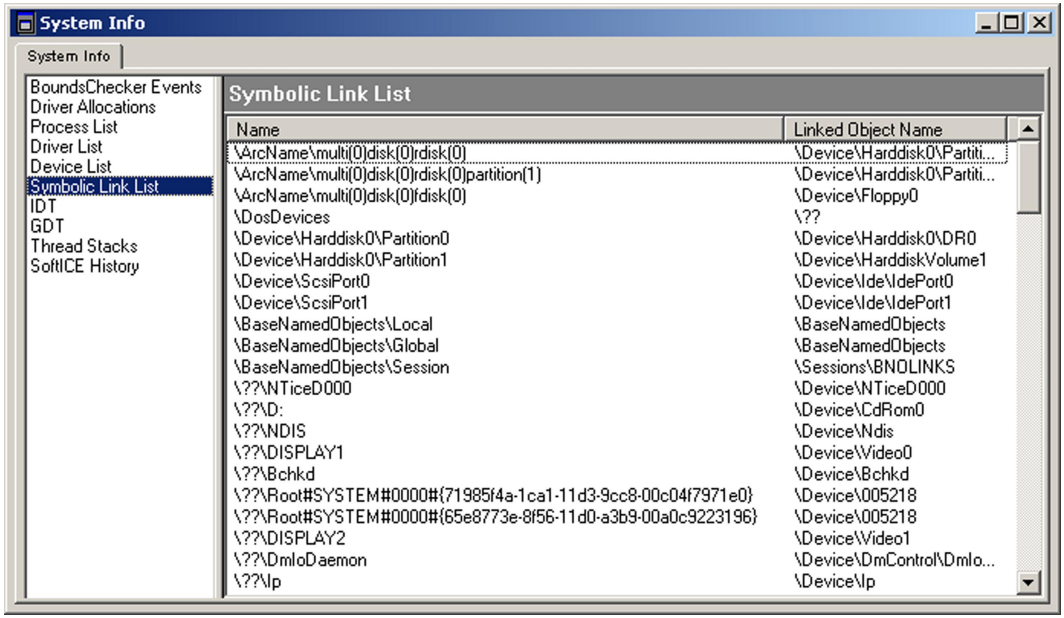


Figure 2-19. Symbolic Link List

IDT

The **IDT** (Figure 2-20), or Interrupt Descriptor Table, gives the following information:

- **(CPU)Vec**. Interrupt vector number (0 through 255).
- **DPL.** Interrupt gate privilege level (0 is kernel level; 3 is user level).
- **Type**. Type of interrupt gate (interrupt, trap, or task gate).
- **Sel:Offset**. Address of the interrupt handler procedure.
- **Owner**. Name of the module that contains the interrupt handler procedure. A symbol name is displayed if one is available.

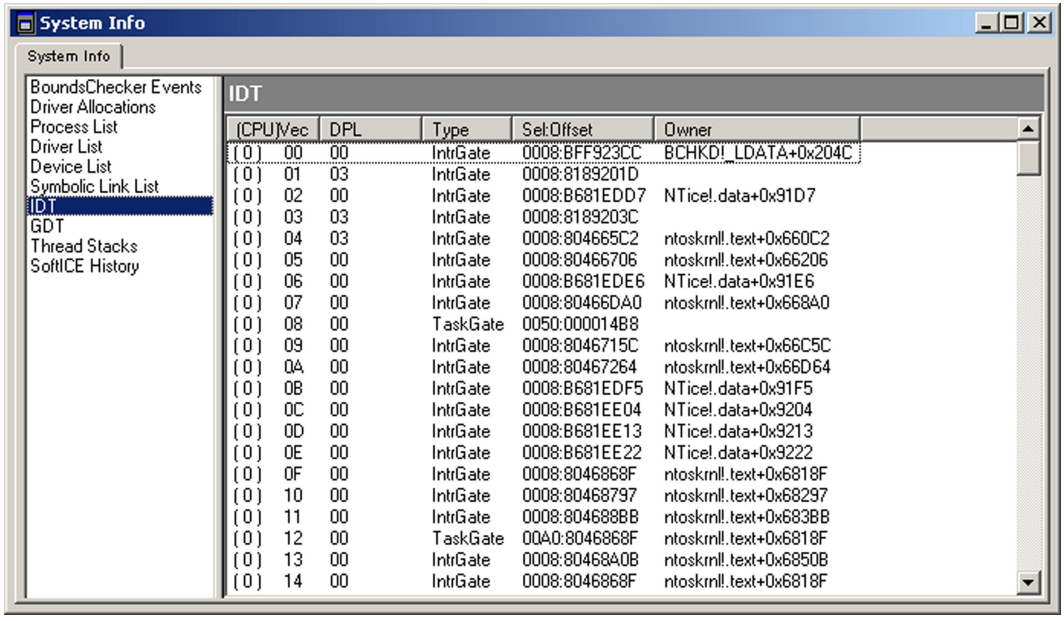


Figure 2-20. Interrupt Descriptor Table

GDT

The Global Descriptor Table (GDT), shown in Figure 2-21, gives the following information:

- **(CPU)Selector**. Selector value loaded in the segment registers.
- **DPL**. Descriptor privilege level (0 – kernel level; 3 – user level).
- **Base**. Base address of the selector.
- **Limit**. Size of the address space covered by the selector.

- **Type**. Descriptor type (i.e., code, data, system, etc.).
- **Flags**. Attributes of the descriptor (i.e., read-only, read-write, execute, etc.).

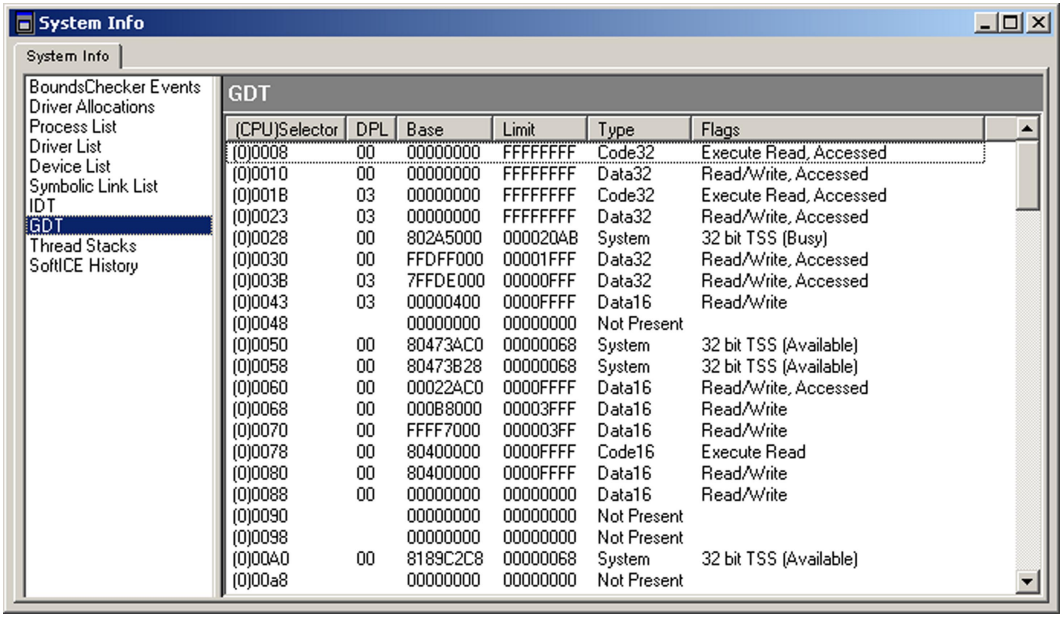


Figure 2-21. Global Descriptor Table

Thread Stacks

Selecting **Thread Stacks** displays the call stack for any process thread running in the system when the dump file was created. The following columns are viewable (by default) when you select Thread Stacks (Figure 2-22):

- Function – The name of the function. Only available if symbols are loaded.
- File – The file where the function exists. If file information is not available, the module name will be displayed.
- Line – The line number in the file.

Right-clicking on an entry in the Thread Stacks pane will take you to the context menu and allow you to display these additional columns:

- EBP – The EBP address of this stack frame.
- EIP – The EIP address of this stack frame.

- Arguments – Three DWORDs following the EIP value on the stack. These are potential arguments to the function.

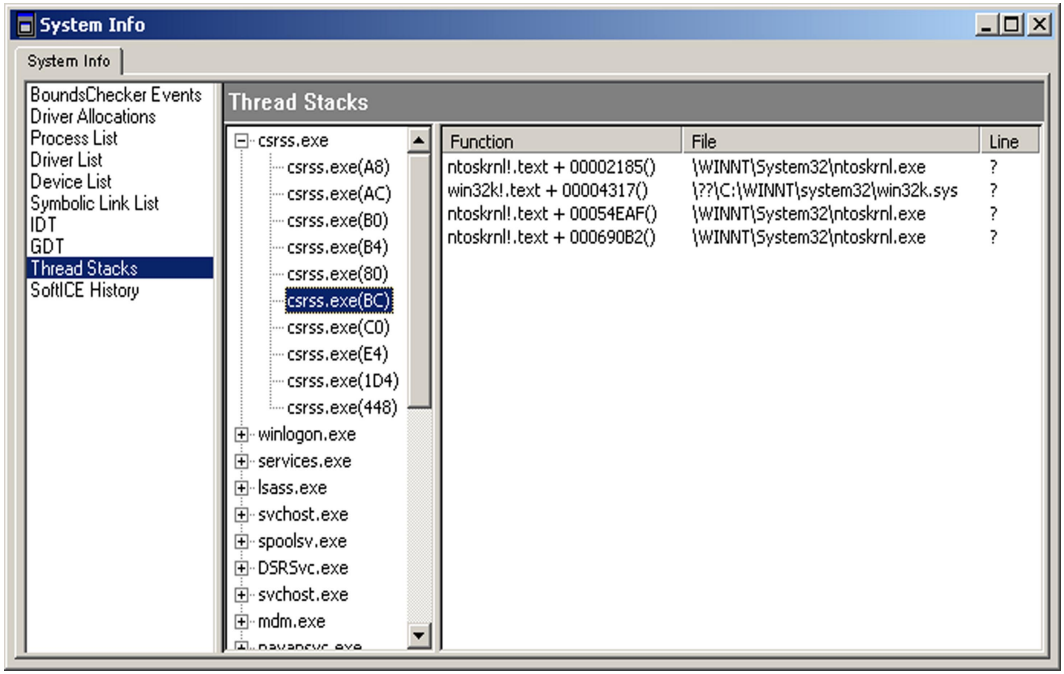


Figure 2-22. Thread Stacks

SoftICE History

If SoftICE is running, you can view the contents of the SoftICE history buffer (Figure 2-23). Selecting **SoftICE History** displays output debug strings, SoftICE commands and their output, and other informational messages that can be useful in piecing together what happened just before the system crashed or the system information file was created.

Accessing BoundsChecker Events in a Memory Dump File

In the event of a crash, a Windows NT family platform can be configured to create a memory dump file, which is a raw dump of memory to a file on your hard drive. When BoundsChecker is loaded, the entire event buffer gets logged to the hard drive as well. These are the last calls made before the system crashed. This fact could be useful when figuring out what happened. There are plenty of memory dump readers available, but only DriverWorkbench allows you to analyze a crash dump file along with viewing BoundsChecker events.

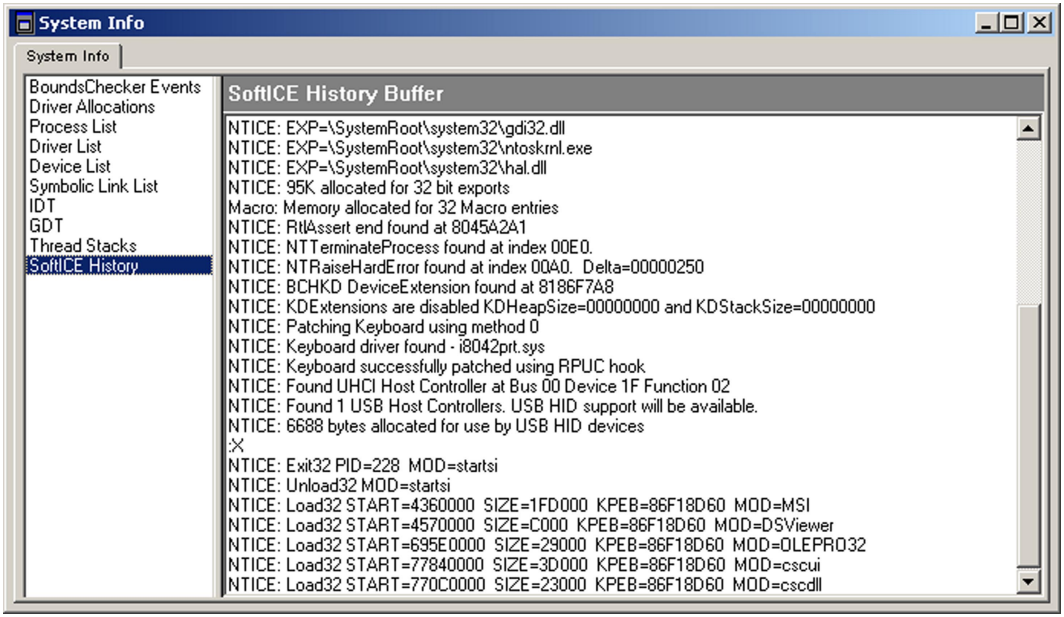


Figure 2-23. SoftICE History Buffer

Using BoundsChecker Monitor

What is BoundsChecker Monitor

BoundsChecker Monitor is an application that receives notification from the BoundsChecker driver when an error occurs. When this notification is received, BoundsChecker Monitor temporarily turns event logging off, captures all of the events that led up to the error in a file and then turns event logging back on. Basic information, including the offending driver and a summary of the error, is displayed in the BoundsChecker Monitor Errors Detected dialog (Figure 2-24).

The control buttons on the right side of the dialog function in the following manner:

Collapse the dialog so as not to show the error list.

Expand the dialog to show the error list.

Remove all errors for a particular driver. Select the driver in the top pane and click this button to remove all error entries for this driver.

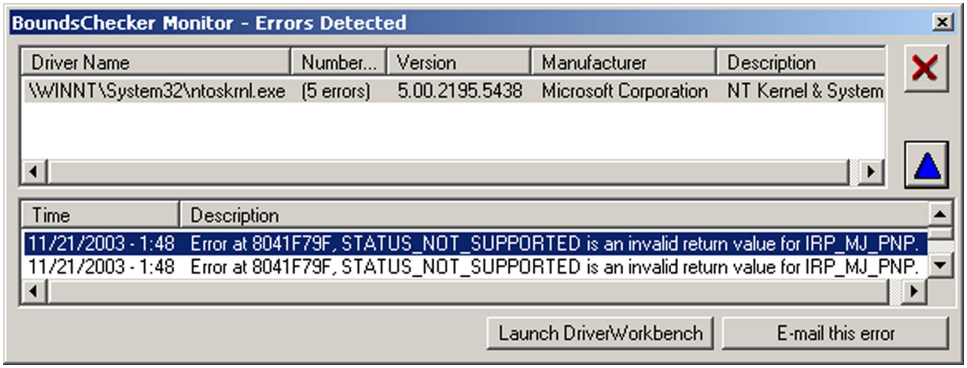


Figure 2-24. BoundsChecker Monitor Errors Detected Dialog

BoundsChecker Monitor Options

After an error entry has been made to the Errors Detected dialog, you have several options.

- **Ignore the error.** In the case of API failure, the driver could be handling the error correctly.
- **View the error in DriverWorkbench.** This can be done by expanding the window to show errors, selecting the error you are interested in, and pushing the **Launch DriverWorkbench** button.
- **E-mail the information about the error back to the driver developer.** To do this, expand the window to show errors, select the error you are interested in, and push the **E-mail this error** button.

Starting BoundsChecker Monitor

BoundsChecker Monitor is part of the DriverStudio Tray Application, which is enabled from the Startup page in the DriverStudio Configuration dialog. When users want BoundsChecker Monitor to provide feedback on errors, they need to tell BoundsChecker this in the BoundsChecker Settings-General Page.

Frequently Asked Questions (FAQs)

General FAQs

What types of errors will BoundsChecker find?

The types of errors that can be found are shown in an earlier section of this document, **“**Errors Detected by BoundsChecker Driver Edition**”** on page 10.

I found the settings for checking memory, Resources and DMA, but what about the other items?

The other things like API validation and IRQL failures are inherently checked and do not need to be configured.

Does BoundsChecker for Drivers require “instrumentation” or changes to my code?

BoundsChecker Driver Edition does not require recompilation of the driver or insertion of code into the users driver. It works strictly on dynamic hooks in the code as it runs

I have run BoundsChecker on my driver and got a lot of information. I really don’t know were to begin looking for errors. Do you have any hints?

The first major step is to be sure that BoundsChecker Driver Edition is configured properly. You may have selected to many functions to examine. Try to refine the functions you are checking. After you can deselect the ones you have checked and select others. See the setup hints in this document in the section entitled “Configuring BoundsChecker Driver Edition” on page 12. If there is still to much information or information that you do not want to see then you can use filtering to show only those types of errors you are interested in.

I would like to have BoundsChecker look at only some sections of my driver and not others. Is there a way to do that?

If you would like to look at all the APIs, but only in one section of a driver, then the best way to do that is by controlling where BoundsChecker logging is active. See the information on logging control entitled “Controlling BoundsChecker Event Logging” on page 23.

What is a memory leak?

A memory leak is a block of memory that was allocated but not freed. With drivers, the question is when does a block of memory become a leak? The answer is when the driver unloads. Until that point, memory blocks that were allocated are deemed valid. Since BoundsChecker hooks memory allocation and de-allocation routines, it is simple to keep track of memory blocks. When a driver unloads, BoundsChecker looks in an internal memory list to see if any blocks belong to the unloading driver. If so, the block is marked as a leak.

How do I find a memory leak in my driver?

By default, BoundsChecker will track memory allocations and report leaks. As a user, you should narrow the scope of what BoundsChecker watches. Try to limit the number of drivers you are watching on the Select Drivers page in the DriverStudio settings. BoundsChecker by default watches *all* APIs so, if selecting sub-groups, make sure to include memory allocation routines.

How do I get a leak report on my driver?

There are two ways to get reports of memory leaks. The first is to use the EVMEM command in SoftICE. The EVMEM command lists all blocks of memory that BoundsChecker has logged. The ERR column tells you if the block was a leak. The second way is to use DriverWorkbench. After your driver has unloaded, select Capture System Information from the BoundsChecker menu. In the System Information view, select Driver Allocations. A list of all allocations will be displayed, including an Error column. Sort the list by driver and you can see all allocations that belonged to your driver. These are leaks (since your driver has unloaded).

Why does BoundsChecker still report leaks in my driver after I've fixed them?

Remember that memory leaks and other errors are persistent. That is, the errors will appear from run to run of your driver. For example, say you had a USB driver that loaded and unloaded and you discovered that you had leaks in it. You fix the leaks and re-run it. BoundsChecker will report the same leaks in the second run, even though you fixed them. Why? Memory tracking does not reset when a driver reloads. What you should do is select Reset from the BoundsChecker menu in

DriverWorkbench to reset memory tracking before you reload your driver.

What do I do if my driver is a boot driver?

What happens if your driver doesn't unload until the OS shuts down? After all, as soon as the machine reboots, BoundsChecker will lose all data regarding leaks and other errors. You don't know if you have problems or not. There are two possible solutions here.

Solution 1 – Just before shutting down, capture System Information in DriverWorkbench. Sort the Driver Allocations list by driver and note all memory blocks that belong to your driver. Manually check each one to see if it was freed before the driver shut down.

Solution 2 – In the settings dialog under BoundsChecker > File options you have the ability to log BoundsChecker events to a file. This file is a text representation of the events and errors that BoundsChecker logs. While it does not contain all of the information you might need, when used in conjunction with the information above, you should be able to easily figure out which blocks have leaks and whether you should fix them or not.

Tracking Down Memory Overruns and Underruns

Configuring BoundsChecker to Detect Memory Overruns and Underruns

The BoundsChecker General page of the DriverStudio Configuration dialog (Figure 2-1) has options which allow you to detect memory overruns and underruns.

- **Notify on** – This will determine what debugger and under what circumstances BoundsChecker active the chosen debugger.
- **Error trapping** – Selecting memory tracking will tell BoundsChecker to detects overruns and underruns by padding allocations with guard bytes. BoundsChecker checks all guard bytes at 1 second intervals and when the block is freed or the driver it belongs to unloads. If the pattern of the guard bytes is no longer 0x5A or 0xA3 (depending on selections) repeated for the number of bytes (also depending on selection), an error is reported.

How BoundsChecker Detects a Memory Overrun or Underrun

BoundsChecker intercepts calls to memory allocation functions such as ExAllocatePool for the purposes of memory tracking. When one of these

calls is made, BoundsChecker inserts the selected number of guard bytes with the selected pattern at the beginning and end of each allocated block. The guard bytes can be seen when dumping a memory block inside a debugger such as SoftICE. BoundsChecker checks all guard bytes at one second intervals, when the block is freed, or if the driver it belongs to unloads. If the pattern of the guard bytes is no longer the correct pattern repeated for the number of bytes, an error is reported.

Determining the Location of the Overrun

As stated above, BoundsChecker checks the guard bytes at one-second intervals. Sometimes you find that a lot has happened since the offending memory write. There is a way to make BoundsChecker check more often. You can add an undocumented registry key in the registry at HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services\Bchkd. The key name is **GuardCheckTime**, it is a DWORD and it is in tenths of a second. By changing this number, you change how often BoundsChecker looks at memory blocks for overruns. Be aware, if the interval is set too often, your machine may become unusable, especially if you are monitoring drivers such as NTFS and FASTFAT. These drivers allocated large number of memory blocks and checking all of those over and over will really slow the machine down.

Another way to track down an overrun would be to use SoftICE. When BoundsChecker reports an overrun, information about where the block was allocated is available. By setting a breakpoint on this line in SoftICE, you can get the address of the memory block returned by the allocator function. Now, by using BPM breakpoints, you can set a breakpoint on write over the guard byte area. When the breakpoint goes off, you have the culprit.

Helpful Hints

Try to limit the drivers you are monitoring. Drivers such as NTFS and FASTFAT allocate thousands of memory blocks. If you are trying to do overrun checking constantly on these blocks, your system will become unusable.

There are no underrun bytes for allocations done with HeapAlloc or string routines such as RtlInitAnsiString.

Chapter 3
Using TrueTime Driver Edition

Introduction to TrueTime Driver Edition

TrueTime Driver Edition is a performance analysis tool that allows a Windows NT family device driver writer to identify and fix a driver's performance bottlenecks. In fact, TrueTime Driver Edition has been engineered to meet the needs of not only the device driver writer, but the kernel-mode code developer as well.

About Driver Performance

Driver performance is largely defined by latency and data throughput, rather than by algorithmic or raw computational performance. TrueTime Driver Edition is a performance analysis tool engineered to meet the needs of device driver and kernel-mode code developers. With TrueTime Driver Edition, device driver writers can identify and fix a driver's performance bottlenecks. In addition, kernel-mode code developers can

isolate latency and bandwidth problems, quickly identify performance hot spots, and accomplish all of this without recompilation.

TrueTime hooks IRP dispatch functions, as well as ISRs, DPCs and I/O completion routines. Additional options allow the function tables exported by NDIS and video drivers to be hooked as well. Each time a dispatch function is called, TrueTime logs the function prolog and epilog time stamps.

Using TrueTime

A user of TrueTime Driver Edition works with two main components: a **Performance Measurement Driver** and a multi-pane **Performance Analysis Window**.

You can enable the **Performance Measurement Driver** at boot time. Once enabled, you can start and stop logging dynamically during execution. (The Driver collects data according to the specifications you have set in the TrueTime | Settings menu). You can instruct TrueTime to collect performance data about any device driver in the Windows system, provided that symbolic information is available for the driver.

The collected data will be written to a **Log File** in the user-specified logging directory. This information is accumulated by the Measurement Driver, and saved to a file with an .ttd extension. You can open this file to display and analyze its gathered performance data.

Once the user is satisfied with the collected data, the **Performance Analysis Window** can be brought in with the performance data from the session's Log Files. Alternatively, the user can open a Log File and show Performance Analysis data on those files.

Frequently Asked Questions (FAQs)

How does Function Hooking Work?

After a driver to be monitored is loaded into memory, TrueTime patches each monitored function with a jump instruction that transfers control to a piece of code in the x9tt.sys driver that logs a timestamp. This jump instruction is placed at the beginning of the function. At runtime, the return address on the stack is changed to point back into x9tt.sys to record the total time spent in the function.

How is Hooking different from Sampling?

Sampling involves taking a measurement at a fixed or randomly selected time interval. Each measurement records where program control is at the

time of measurement. If enough samples are taken of a representative test run this approach will theoretically show you where your driver is spending the most time. It will not give you accurate measurements of best and worst cases function performance over all function calls in a driver. It is likely to completely miss infrequently called or very short functions.

Function hooking records accurate timing data for every monitored function every time it is called. This makes it possible to accumulate accurate call counts and statistics for best, worst, first, last, and average time spent in a function.

Do I have to rebuild my driver in order to work with TrueTime?

No. TrueTime will automatically patch the entry points without requiring you to rebuild your driver. This occurs after the driver is loaded into memory. It does not effect the driver file on disk. This patching occurs only when TrueTime is enabled.

How do I get TrueTime to collect data?

First, make certain that the TrueTime driver is set to **enabled** in the DriverStudio Configuration dialog **Control Panel Startup** page. Next, choose **TrueTime/General** from the list of choices in the left panel of the Configuration dialog. Select the appropriate general configuration items from this page. Then, to monitor your driver, choose **Select drivers** from the list of choices in the left panel. This will display a list of all drivers installed on your machine. Select the check box to the left of your driver to enable it to be monitored by TrueTime. Next, choose **Select functions** to choose the driver functions that you want to monitor. Click **OK** to close the Configuration dialog. Data collection will begin automatically when your computer is rebooted.

After the initial driver selections have been made, you can change your selections by going to the DriverWorkbench menu bar and choose **Tools**/**TrueTime**/**Configuration Wizard**. When the DriverStudio Configuration dialog appears, choose **TrueTime/General** from the list of choices in the left panel of the Configuration dialog. Select the appropriate general configuration items from this page. Then, click **Next** or **Select drivers** from the list of choices in the left panel. This will display a list of all drivers installed on your machine. Select the check box to the left of a driver to enable it to be monitored by TrueTime. Click **Next** or **Select functions** to choose the driver functions that you want to monitor. Click **Finish** to close the Configuration dialog.

TrueTime hooks your dispatch entry points and begins logging data the next time your DriverEntry routine is called. (You will need to stop and restart your driver to force DriverEntry to be called.)

Note: In order for a driver to show up in the Select Drivers list, it must have an entry under [HKEY_LOCAL_MACHINE\System\CurrentControlSet\Services]. You can use the DriverStudio Configuration dialog **TrueTime/Select drive**r menu choice to create a Services entry in the registry for your driver.

If you want to select a driver for monitoring without adding an entry under the Services key, use the **Add Driver** button in the DriverStudio Configuration dialog **TrueTime/Select driver** page.

Where, when, and how does TrueTime store performance information?

TrueTime stores performance information in log files. These log files have a .ttd extension. Log files are stored in a location specified in the Output Directory field in the TrueTime settings dialog. TrueTime creates a log file as soon as the DriverEntry function for a driver to be monitored is called.

Note: All times shown in the Analyze Performance window are relative to the start of DriverEntry.

How is data stored and displayed when more than one driver is selected?

If more than one driver is selected for monitoring a separate ttd file is created for each driver. The filename is composed of the driver name, a unique session id, and the driver restart count. Each time you reboot your machine a new session id is assigned. Each time you restart a monitored driver within a session the restart count is incremented. The following are examples of log file names:

- **`i8042prt_sys_9089eb78_0000.TTD`** – The first time `i8042prt.sys` was started in session 9089eb78.
- **`i8042prt_sys_9089eb78_0001.TTD`** – The second time `i8042prt.sys` was started in session 9089eb78.
- **`serial_sys_9089eb78_0000.TTD`** – The first time `serial.sys` was started in session 9089eb78.
- **`serial_sys_bc5b796c_0000.TTD`** – The first time `serial.sys` was started in session bc5b796c.

The current (or last) session number is stored in the registry. When you choose Analyze Performance TrueTime merges the results from all of the driver files in the latest session. For each driver in the session, the file

with the largest restart count is used and the other files for that driver are ignored. You must choose **File > Save As** in order to write out the merged session file if you want to view the merged session at a later time. When viewing merged session data the timestamp of the earliest DriverEntry over all monitored drivers in that session is used as the 0 timestamp. All times shown in the Analyze Results window are relative to that time.

You can also open TTD files manually using the File > Open menu selection and specifying the file by name.

Does TrueTime support storing performance data in compressed folders?

Yes. To reduce the amount of space that the .ttd files take, set the output directory to point to a compressed folder on an NTFS volume. The compressed files are one-third the size of an uncompressed file.

What are .tti files and where are they stored?

TrueTime stores the list of user-selected functions for monitoring in a file with the same name as the driver but with a `.tti` suffix instead of the `.sys` suffix. The `.tti` file is always written to the directory containing the driver file.

How does TrueTime maintain consistency between the .sys, .pdb, and .tti files?

When the settings dialog is opened, TrueTime compares the internal timestamps stored in the `.sys`, `.pdb`, and `.tti` files. If the `.sys` and `.pdb` timestamps don't match, TrueTime will not display symbol information for the driver. Also, you will not be able to use the settings dialog to select additional functions for monitoring. If the tti file timestamp does not match the driver timestamp TrueTime will display a warning message about this. TrueTime stores only the function offsets when logging data. Modifying and rebuilding your driver can cause these offsets to change. TrueTime will try to resynchronize the user-selected function offsets after a rebuild. However, the list of user-selected functions should be verified to ensure they are still correct. When the data is collected and saved, symbolic function information is saved with it so no external symbol files are required.

Configuring the Settings Control Panel

TrueTime Driver Edition is enabled on the Control Panel Startup page. If the enabled radio button is selected, the TrueTime Performance Measurement Driver is loaded at boot time.

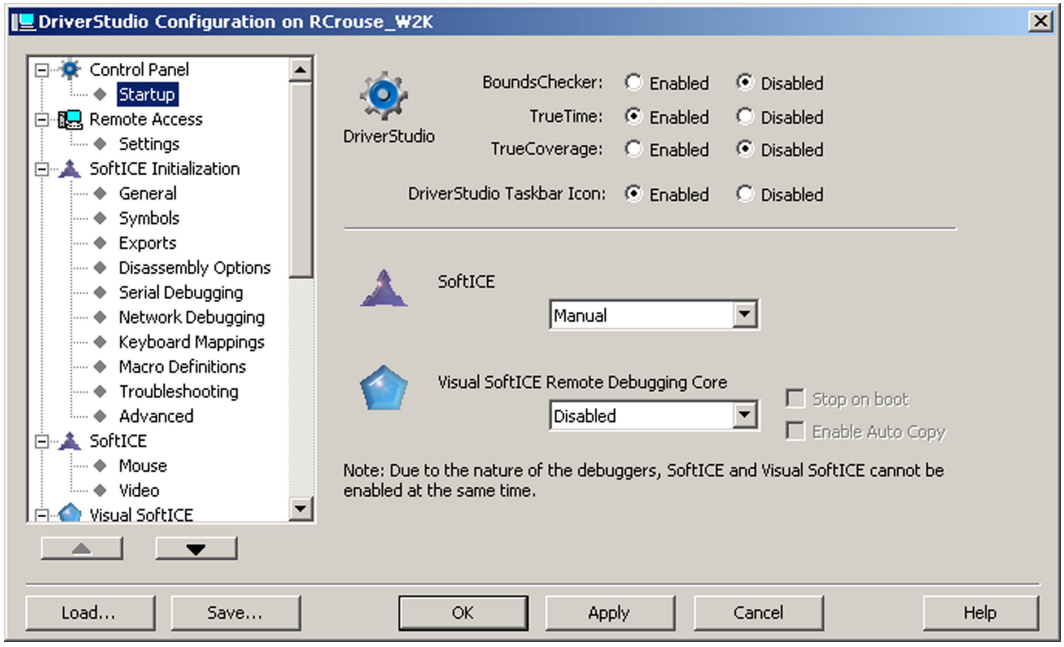


Figure 3-1. DriverStudio Configuration Dialog –Control Panel Startup Page

Note: Due to the overhead when using BoundsChecker, it is not possible to collect meaningful performance data with TrueTime while BoundsChecker is active. TrueCoverage will also interfere with performance data collection. As a result, the DriverStudio control panel enforces exclusions between the TrueTime driver and the BoundsChecker and TrueCoverage drivers. Enabling TrueTime will automatically disable BoundsChecker and TrueCoverage. Enabling either BoundsChecker or TrueCoverage will automatically disable TrueTime.

TrueTime General Page

TrueTime can be configured from the **Settings** option of the DriverStudio program group beginning on the General page (Figure 3-2).

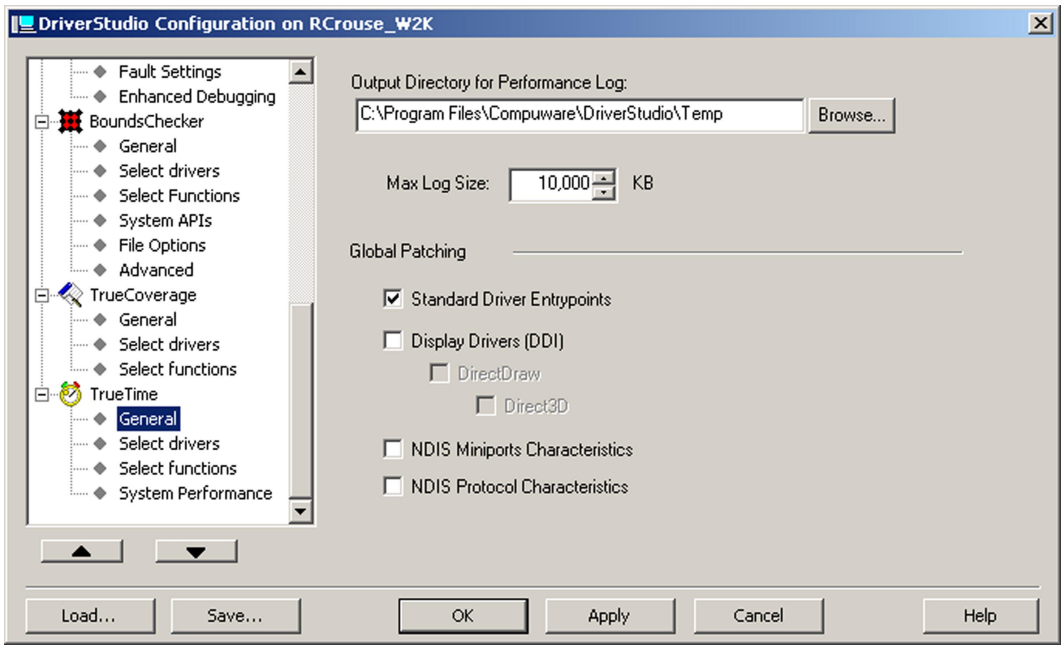


Figure 3-2. DriverStudio Configuration Dialog –TrueTime General Page

This page allows you to determine how TrueTime stores the data it gathers when it interacts with the system under test.

The following selections are available on this page:

- **Output Directory for Performance Log.** The path listed in the data entry box sets the output directory for performance log files. The TrueTime Measurement Driver always creates log files in the output directory. Use the Browse button to select a path.
- **Max Log Size**. This field sets the maximum size of the Performance Log. Note that the Performance Log is a linear buffer. This means that if the Performance Measurement Driver is recording data into the Log and the Max Log size is reached, recording simply ceases.
- **Global Patching.** The TrueTime configuration page has global patching options to select Standard Driver Entrypoints as well as the Display Drivers (DDI), DirectDraw, and Direct3D functions.
  - **Standard Driver Entrypoints** – This is enabled by default. Disabling this option will override the standard patching of StartIO, AddDevice, DriverUnload, and the IRP Dispatch table.
  - **Display Drivers (DDI)** – Enabling this option patches the DrvEnableDriver entry point as well as the Graphics DDI function table.

- **DirectDraw** – DirectDraw function profiling requires patching the DDI. Selecting this option enables the *Direct3D* option. Patching DirectDraw functions will occur at runtime provided the video driver being profiled supports DirectDraw. The DDI callbacks DrvEnableDirectDraw and DrvGetDirectDrawInfo are used to determine the driver supplied DirectDraw callbacks.
- **Direct3D** – Direct3D function profiling requires patching DirectDraw. D3D driver supplied callbacks are located at runtime from the DD_HALINFO struct and DdGetDriverInfo queries.
- **NDIS Miniport Characteristics** – Enabling this option allows TrueTime to patch the NDIS MiniportCharacteristics function table. The tables are determined when your driver calls NdisMRegisterMiniport, NdisMRegisterDevice, and NdisIMRegisterLayeredMiniport
- **NDIS Protocol Characteristics** – Enabling this option allows TrueTime to patch the NDIS ProtocolCharacteristics function table. The table is determined when your driver calls NdisRegisterProtocol.

Select Drivers for Data Collection

The TrueTime Select Drivers page (Figure 3-3) lets you choose those drivers for which the TrueTime Performance Measurement driver will gather data. Select the driver categories that you want to analyze. The driver categories are:

- **Standard** – This option is selected by default.
- **Video** – This option is selected by default.
- **Printer** – This option is selected by default.
- **VxD** – This Win9x option is grayed-out when TrueTime is loaded on a WinNT family platform..

In the work pane, TrueTime displays a list of drivers in your system for each of the categories selected. Information shown for each driver includes:

- **Driver Name –** Displays the driver's name.
- **Internal Name –** Displays the driver's internal name.
- **Group –** Displays the driver's Load Order Group.
- **Startup –** Displays the driver's Start Group (Manual, Automatic, Boot, Disabled, or System).

Tip: *Some columns may not appear in the list. Use the horizontal scroll bar below the pane to reveal columns that are "hidden" to the right of the window.*

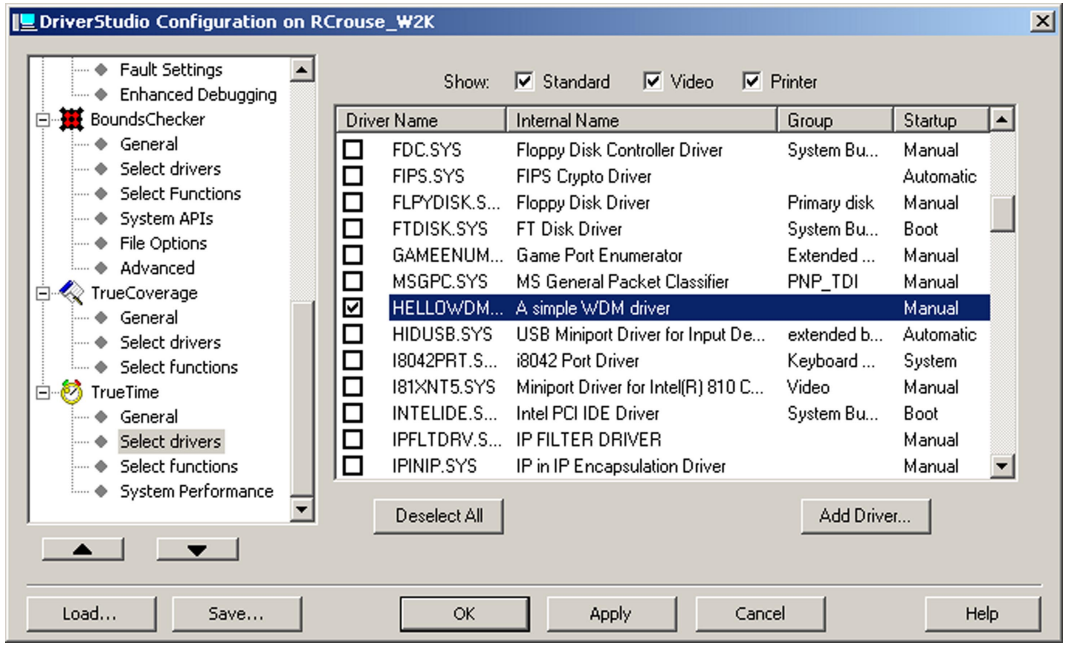


Figure 3-3. DriverStudio Configuration Dialog – TrueTime Select Drivers Page

Select the driver or drivers to be monitored using the checkbox that appears next to the driver name in the list. The **Deselect All** button will reset the list. The **Add Driver** button lets you add a driver to the list.

Note: TrueTime Driver Edition's Performance Analysis driver will only begin gathering information on a driver when that driver is loaded (not merely when the driver has been selected). If the driver is already loaded, you must unload and reload the driver (or reboot) so that the Performance Analysis driver will begin gathering data on that driver.

Select Functions to Monitor

The TrueTime Select functions page (Figure 3-4) allows you choose a driver's specific functions to monitor.

Note: Standard entry points are always hooked for profiling. This is DriverEntry for most drivers, and DrvEnableDriver for video and printer drivers. These are not deselectable.

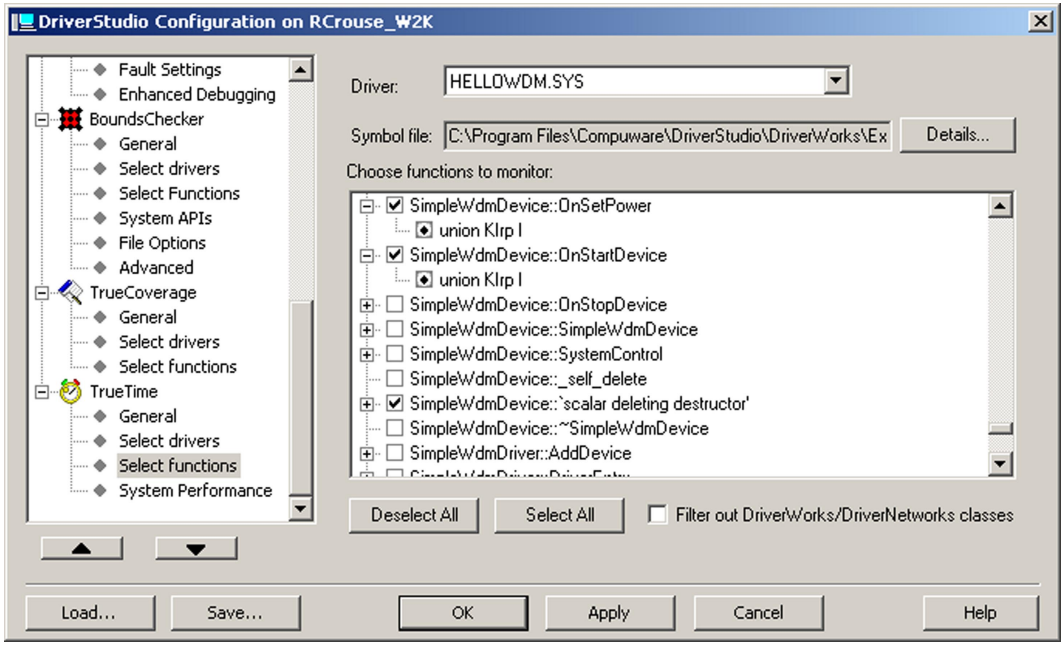


Figure 3-4. DriverStudio Configuration Dialog – TrueTime Select Functions Page

Select the driver in the drop-down list box. (Note that only those drivers that you chose to monitor in the Select Drivers window will appear in the drop-down list box.) When you have done this, TrueTime Driver Edition will populate the pane with a hierarchical list showing the driver's functions.

Select which functions you want TrueTime Driver Edition to monitor by clicking on the checkbox next to the function name.

Selecting Function Parameters

Parameter-based function timing allows a finer granularity of results than presented at function level alone. Different code paths can be measured within functions when based on a passed argument. If you click the + sign next to a function name, the function parameters will be displayed. The function parameters can be expanded in the tree. Selecting one of these parameter nodes will flag the kernel mode component of TrueTime Driver Edition to collect the timing data of this function within the context of this parameter. Parameters flagged in this manner will show up in the IRP List detail display pane.

If the function list displays:

- **No drivers are currently selected for performance analysis** – You didn't select any drivers in the Select Drivers page. Go back to the Select Drivers page and select the driver that you want to analyze.
- **No function information was found in the symbol file** – You will not be able to select additional functions to collect timing for. You will however see function names in your results for the standard patched entry points. See "How To Monitor More Than the Default Number of Functions" for information on building your driver with the necessary symbol information.
- **Could not locate symbols for driver 'drivername'** – First, check the Symbols column on the Select drivers page for the correct path and filename of your symbol file. If this is blank, the symbol file should be located in its original build location. Alternatively we will locate the symbol file if placed in the same directory as the driver. Finally, symbol files can be placed in the user-defined symbol paths.

Note: Symbols are now stored within performance result files. There is no longer any dependency on additional binaries or symbol files once results have been saved.

Click the **Deselect All** button to clear all selections.

Filter out DW/DNW classes – This option will filter out DriverWorks™ and DriverNetworks™ class libraries from the function selection list. Checking this makes locating your functions easier when building with those libraries. The list is unfiltered by default.

Analyzing Driver Performance

Determining Where a Driver is Spending Its Time

How do you determine where a driver is spending its time? In two words: *Drill Down!*

TrueTime for Drivers is designed specifically with device drivers in mind. By default, TrueTime monitors the total time spent in DriverEntry, all dispatch functions, ISRs, DPCs, and I/O Completion routines. For many drivers there is no other way for control to enter your driver. After monitoring your driver with the default settings, TrueTime will tell you which of these functions is taking up the most time. All other functions in your driver are called as children of one of these top level functions. After a single run with the default settings you will know at a high level where your driver is spending the most time.

Data Analysis

A monitoring session must be active in order to choose **Analyze Performance**. You can also open a saved session with the **File > Open** menu selection.

Your primary tool for locating a driver's bottleneck is the Function Stats window. Select **TrueTime > Analyze Performance**, and in the left pane (the index pane) of the Performance Analysis Window select **Function Stats**. This opens the **Function Stats** window in the right pane.

Each row of this window shows information gathered for a particular function. The *% in function* column is the most helpful in locating bottlenecks, since it shows the percentage that each function contributes to the overall time consumed by the driver.

Note: The symbol file should always be in its original location or in the same directory as the driver. If you are running a driver from an alternate location, you should point the symbol search path to the directory containing the driver. If the driver has an ImagePath entry in the registry, this should not be necessary.

Right-click on the window, and select **Go To Source** to see the corresponding source code on a separate window.

Locating Driver Latency Problems

Once you run a measurement session on your driver, there are several graphs in the Analyzer window that might help you. You can bring up the Performance Analysis window by either clicking on TrueTime/ Analyze Results or by loading a TrueTime file by clicking on File/Load and selecting the appropriate results file.

In either case, the Performance Analysis window will show, with two frames in it. The left frame is an index of different reports and graphs available in TrueTime Driver Edition. The reports that may help you are the IRP List, the IRP Completion Histogram, the IRP Completion Times and the DPC Latency Histogram.

Click on IRP List to see an individual list of every I/O Request Packet issued to your driver during the measurement session. You will see minimum, maximum and average latencies for each type of packet, and a count of how many packets of each type were issued. For each I/O Request Packet type, the bottom frame will show a list of all packets, with a timestamp, returning status and number of bytes transferred.

Click on IRP Completion Histogram to see a histogram that groups packets by completion time. Select a subset of the graph to zoom. Right

click and choose Zoom Prev to return to the previous zoom level, or Zoom All to return to the original 1:1 view.

Click on IRP Completion Times to get a scatter chart clustering IRPs by arrival time and time to complete. Select a subset of the graph to zoom the chart. Right click and select Zoom Prev to return to the previous the zoom level, or Zoom All to return to the original 1:1 view.

Click on DPC Latency Histogram to see a histogram of Deferred Procedure Call latency times. Right click on each histogram bar and select View Bin Contents to get a more detailed diagram of that DPC's timing. This diagram has a bar for each DPC, showing how long it takes before the DPC is scheduled, as well as how long the DPC took to execute.

Monitoring More Than the Default Number of Functions

To monitor more than the default number of functions, you will need to build your driver with symbols. You need a free build with symbols, even if no functions are patched; the UI depends on it.

When building with the WinNT 4 or Win2K DDK

Create symbols for a free build by setting the following environment variables before doing a rebuild of your driver:

```
set NTDEBUG=ntsd
set NTDEBUGTYPE=windbg
set USE_PDB=1
```

When building with the Windows XP DDK using build.exe

Add **`set MSC_OPTIMIZATION=/Zi`** in XP DDK free build environment.

When building with the Windows XP DDK using VisualStudio via DriverWorks or DriverNetworks

Add the Compiler option: `/Zi`.

Add the Linker option: `/debug:FULL`.

This will create a `.pdb` symbol file containing symbols for your driver. Generation of symbols in a free build will not effect driver performance or significantly increase the size of the `.sys` file. The `.pdb` file should either remain in the directory where it was built, or should be moved to the directory containing the `.sys` file.

After you have built a symbol file for your driver, bring up the **TrueTime Settings dialog**. Make sure your driver is still checked under **Select**

Drivers. Next, make sure that the correct symbol file and location are listed under the **Symbols** column in the **Select Drivers** list box. Now choose **Select functions to monitor**. TrueTime will display a list of all functions in your driver. Check the functions that are relevant to the entry point that TrueTime has reported as taking up most of your time. Click **OK** to save the settings and close the dialog. You will need to restart your driver in order for these changes to take effect.

Analyzing I/O Request Packet Performance

You must have a Log File open to perform this operation, and the Performance Analysis window should be showing.

1. Click on IRP List on the index pane. The IRP List pane will show.
2. The top subpane shows a list of IRPs by type, with latency statistics. The bottom subpane shows a detail timestamp of the packets. Additional IRP detail timestamps can be captured by indicating which function parameters are IRP pointers within the TrueTime **Select Functions Page** of the DriverStudio Configuration.
3. Right click and use the expand/collapse options, or click on the plus/minus boxes to expand and collapse the timestamp tree details.
4. Right click and click on **Go To Source** to see the source code for the corresponding function.
5. Click on **IRP Completion Histogram** on the index pane. The **IRP Completion Histogram** pane will show.

   The histogram shows the distribution of packets according to type and latency.
6. Rest the mouse pointer on top of a strip to see the type and number of packets in that strip.
7. Select an area of the display to zoom in and use smaller time unit on the horizontal axis.
8. Right click on the window, then click on the **Recalc bins on zoom** toggle to resize the horizontal time scale when zooming.
9. Right click and select the Zoom options to move around in the zoom range: click **Zoom Prev** to return to the previous Zoom view, or **Zoom All** to revert to the original scale.
10. Click on **IRP Completion Times** in the index pane. The **IRP Completion Times** pane will show.

a. Two scatter graphs will show in the window. The top graph always shows an overview of the entire session, as well as the current zoom context for the bottom window. The current area visible in the bottom windows is highlighted in green in the top window. The graphs show time to completion against arrival time.

b. Select an area of the display to zoom in. Note that only the lower graph zooms. Right click and choose one of the zoom options to move about in the zoom scale hierarchy: click **Zoom Prev** to return to the previous Zoom view, or **Zoom All** to revert to the original scale.

c. Rest the mouse pointer on a point in the graph, or on a cluster of points, to see a list of the IRPs the make up that cluster.

Using the TrueTime Probe Point API

TrueTime Probe Points

A TrueTime probe point is a function call inserted in the user's code to generate a timestamp at a specific location. When a probe point is executed, the TrueTime driver logs the timestamp and information that uniquely identifies the probe point.

This information is composed of:

- **The group ID.** This is a user-supplied value that distinguishes one group of timestamps from another.
- **The probe ID**. Also a user-supplied value. Probe IDs serve to define a sequence of execution through the driver. The probe ID of 0 is reserved, and cannot be used. The probe ID of 1 marks the start of a path of execution. Probe IDs > 1 are appended (by the logging system) to the current path of execution in the order they are executed. In addition, you can define a table of labels that assigns a meaningful name to Probe IDs, thereby making the TrueTime output easier to read. (The example program on page 69 will clarify this.)
- **The address of the call.** This is simply the location of the call in the program.

Labels can be associated with each group and with each probe index within a group. Assigning unique probe indices and corresponding labels can make the results display more readable.

Probe Point API

The TrueTime public Probe Point Interface consists of the following two functions:

```
BOOLEAN TTRegisterProbeGroup(
  unsigned short GroupId,
  char *pStringTable[]
);
```

The **TTRegisterProbeGroup**() function assigns a list of labels to the group and probe IDs. *GroupID* must be unique (as compared to other *GroupID* numbers you may assign to other groups). *pStringTable*[] is an array of pointers to null-terminated strings. The string given by *pStringTable*[0] is the group name. The strings given by *pStringTable*[1] . . . *pStringTable*[n] are the probe index labels. Note that *pStringTable*[n+1] must be NULL (to mark the end of the table).

Suppose you are using the TrueTime Probe Points to monitor two drivers, DriverA and DriverB, simultaneously. The probe groups in DriverA should have different group IDs than those in DriverB if you want their generated paths to appear as different groups in the TrueTime Probe List display (shown below). However, the probe groups can share the same group ID if you want to follow a path of execution that spans both drivers. This feature (shared group IDs across drivers) can be useful if you are monitoring two or more drivers that interact.

```
void TTProbe(
  unsigned short GroupId,
  unsigned short ProbeId,
  void *pIrpContext
);
```

Placing a call to **TTProbe**() in your code inserts a probe-point at that location. *GroupID* is the probe's group ID and *ProbeId* is the probe's index in the group. The first useable probe index is 1, which the TrueTime data analysis module recognizes as the start of a sequence of probe points. After a call to **TTProbe**() with *ProbeId* == 1, TrueTime will analyze all subsequent probes with the same group ID as part of the same execution path. When the next call to **TTProbe**() is executed with *ProbeId* == 1 (for that group ID), TrueTime will begin logging a new path of execution.

The *pIrpContext* pointer identifies an IRP to be associated with the probe. Probes with an associated IRP are also displayed as transactions in the DriverWorkbench IRP display.

Note: It is important that, within a given group, the first call to **TTProbe**() executed have an *ProbeId* equal to 1. Otherwise, TrueTime may not

recognize the start of a new probe path and may continue appending probe events to a single path.

The declarations for these functions are in TTProbes.h. The use of these functions is described below.

Example Program

The Probes sample driver and test applications are installed by default in **C:\Program Files\Compuware\DriverStudio\TrueTime\Examples**. (See the readme.htm in that location.) This example has been updated to build from the latest XP DDKs.

A code segment from **rwdrv.cpp** is shown below:

```
#define TIMEFUNC_GROUP 200
#define PROBEPOINT_GROUP 300
#define TIMEFUNC1( _funcname, _arg1) \
{  \
  TTProbe(TIMEFUNC_GROUP, 1, NULL);  \
  _funcname(_arg1);  \
  TTProbe(TIMEFUNC_GROUP, 2, NULL);  \
}

#define PROBESTART() \
TTProbe(PROBEPOINT_GROUP, 1, NULL);

#define PROBELOOP() \
TTProbe(PROBEPOINT_GROUP, 2, NULL);

#define PROBELOOPEXIT() \
TTProbe(PROBEPOINT_GROUP, 3, NULL);

#define PROBEEND() \
TTProbe(PROBEPOINT_GROUP, 4, NULL);

#define PROBEERRORRETURN() \
TTProbe(PROBEPOINT_GROUP, 5, NULL);
```

The above #defines specify two probe groups: TIMEFUNC_GROUP and PROBEPOINT_GROUP. The TIMEFUNC_GROUP probe group provides an easy way - via the TIMEFUNC1() definition - to time the execution of any function that takes a single argument as input.

PROBEPOINT_GROUP is designed to be used in a more complex function. It provides calls to TTProbe() for function entry and exit, entry and exit points for a loop within the function, and an error return exit point.

The Labels are associated with the GroupID, Probe Index pairs inside of DriverEntry using the following code:

```
char *TimeFuncStringNames[] =
{
  "TimeFunc",
  "TimeFuncStart",
  "TimeFuncEnd",
  NULL
};

char *ProbePointStringNames[] =
{
  "Probe",
  "Test Probe Group Start",
  "Enter Dump Loop",
  "Exit Dump Loop",
  "Test Probe Group End",
  "Error Return in WDump",
  NULL
};

TTRegisterProbeGroup(TIMEFUNC_GROUP, TimeFunc-
StringNames);
TTRegisterProbeGroup(PROBEPOINT_GROUP, Probe-
PointStringNames);
```

Building a Driver Containing Probe Points

To build a driver using TrueTime Probe Points, follow these steps:

1. Include `TTDProbes.h` located in the `DriverStudio\Common\Include\` subdirectory.
2. Insert the array definitions for the string tables that define the labels.
3. Add calls to the code to register the Probe Point groups (TTRegisterProbeGroup()).
4. Add calls to TTProbe() in the appropriate places in your code to add the probe points.
5. Add a line to your SOURCES file similar to the next line:

```
LINKER_FLAGS=c:\DriverStudio\Common\Lib\x9tt.lib
```

Note: The above step assumes that you have installed DriverStudio on the C: drive. If you have installed DriverStudio on some other drive, modify the LINKER_FLAGS assignment accordingly. If you are using a

.dsp project file with Microsoft Visual C++, you will need to add the **x9tt.lib** library to the list of libraries to link against. This setting is found under the Project/Settings/Link page.

Probe Points without Assigned Labels

The TrueTime output generated from running 30 iterations of the sample driver is shown in Figure 3-5. Notice that this version of the driver does not include labels for the Probe Points. Consequently, all Probe Points are displayed by their indices within a probe group.

Probe List

Group	Avg	Called	Min	Max	First	Last
Group 300	88.7074	124	1.2726	278.5886	116.2157	110.6087
Group 200	141.7659	31	111.2274	278.6371	116.2324	111.5217

Path	Avg ET	Called	Min	Max	First	Last
Path 0	**1.4799**	**31**	**1.2726**	**2.5920**	**2.5920**	**1.4331**
Index 1 in rwdrvIWDump	0.0000	31				
Index 5 in rwdrvIWDump	1.4799	31				
Path 1	**90.8395**	**31**	**68.7692**	**234.5619**	**75.9013**	**234.5619**
Index 1 in rwdrvIWDump	0.0000	31				
Index 2 in rwdrvIWDump	1.0301	31				
Average loop interval	1.8127	1519				
Index 3 in rwdrvIWDump	90.8395	31				
Path 2	**131.2542**	**62**	**109.8829**	**278.5886**	**116.2157**	**110.6087**
Index 1 in rwdrvIDump	0.0000	62				
Index 2 in rwdrvIDump	1.5251	62				
Average loop interval	1.2542	6138				
Index 3 in rwdrvIDump	126.7375	62				
Index 4 in rwdrvIDump	127.6722	62				
Index 4 in rwdrvIWDump	141.7742	62				

Figure 3-5. Probe List Showing Points without Assigned Lables

Probe Points with Associated Group and Index Labels

The same code is shown in Figure 3-6. However, these Probe Points are displayed with group and index labels added.

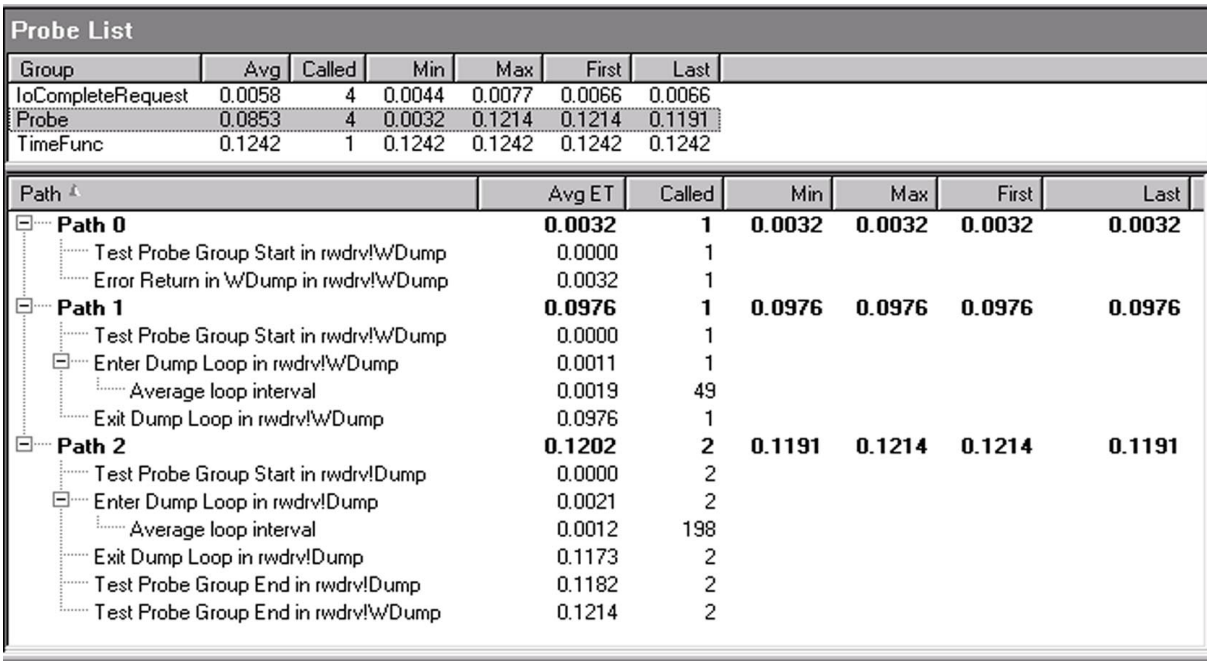


Figure 3-6. Probe List Showing Points with Assigned Lables

Probe Point Path Analysis

TrueTime generates aggregate statistics for each unique path through the user's code that is encountered. For example, if you have a probe group GROUP_1 with probes G1_PROBE_BEGIN, G1_PROBE_ERROR, and G1_PROBE_END defined as follows:

```
#define GROUP_1 100
#define G1_PROBE_BEGIN() TTProbe(GROUP_1, 1,
NULL)
#define G1_PROBE_ERROR()TTProbe(GROUP_1, 2, NULL)
#define G1_PROBE_END() TTProbe(GROUP_1, 3, NULL)

and a routine Dump(PVOID pData) defined as

int Dump(PVOID pData)
{
  G1_PROBE_BEGIN();
  if (NULL == pData)
  {
    G1_PROBE_ERROR();
    return 0;
  }
```

```
    G1_PROBE_END();
    return 1;
}
```

If all paths are followed the DriverWorkbench Performance Analysis/ Probe Points window will generate the following data:

```
Stats for Group_1
 Stats for Group_1/Path 0
  G1_PROBE_BEGIN
  G1_PROBE_END

 Stats for Group_1/Path 1
  G1_PROBE_BEGIN
  G1_PROBE_ERROR
```

This provides a finer level of analysis than would otherwise be possible, allowing you to compare the execution times that different paths may take through the same function. In a more elaborate case of the above example, it is likely that the error return will execute much more quickly than a "normal" return from the function.

Loops

When analyzing probe point information, the DriverWorkbench will assume that any repeating probe point in a sequence is part of a loop and will aggregate all of the probe points into a single point. This ensures that the probe point window will generate the same path for a single probe point used in a loop no matter how many iterations of the loop are encountered. The data display will show the elapsed time for the first iteration of the loop (that is, the time from the beginning of the path — the probe point at Index 1 — until the first execution of the probe point in the loop) and will count all loop iterations and calculate the average time for a loop iteration. Figure 3-6 illustrates this behavior.

The entries for **Path 0**, **Path 1** and **Path 2** show the elapsed time from the beginning of the session, as well as the number of times the Dump Loop was entered during the data logging session. Meanwhile, the associated *Average loop interval* entry in Path 1 and Path 2 shows the average loop interval and the number of loop iterations. (The number of times the loop was entered plus the number of iterations equals the total number of times the probe in the loop was hit.)

Displaying NDIS Packet Transactions

Select **TrueTime** > **Analyze Performance**, and in the left pane (the index pane) of the Performance Analysis Window, select **NDIS Packet List**. This opens the NDIS Packet List window in the right pane.

When monitoring drivers within the NDIS subsystem, TrueTime will log each packet allocation.

In the top pane, you select which driver you want to display packet details from.

In the lower details pane, each packet is listed in order of allocation. The total time until the packet is freed is shown in order of allocation timestamp. Additional timing details are indicated when a packet is expanded for certain NDIS calls.

Tracked NDIS calls:

Begin packet:	NisAllocatePacket
	NdisDprAllocatePacket
	NdisDprAllocatePacketNonInterlocked
End packet:	NisFreePacket
	NdisDprFreePacket
	NdisDprFreePacketNonInterlocked
Packet transactions:	NdisSend
	NdisGetReceivedPacket
	NdisCopyFromPacketToPacket
	NdisIMCopySendCompletePerPacketInfo
	NdisIMCopySendPerPacketInfo
	NdisIMGetCurrentPacketStack

Creating Measurements and Log Files

You can enable or disable the TrueTime Driver Edition Measurement Driver. If you choose to enable the Driver, it is actually enabled at start time. You can then start and stop logging as described below. If you choose to disable the Driver, then starting or stopping logging has no effect.

Turning on the Measurement Driver from startup is useful in those situations where data is needed about the behavior of a driver in early stages of the Windows boot and initialization process.

Measurements are collected into Log Files, created by the Measurement Driver into the user's Output Directory. Log Files have a ***.ttd*** extension.

1. To turn measurements **on**, from the DriverWorkbench menu select **Tools > TrueTime > Start Performance Logging**.
2. To turn measurements **off**, from the DriverWorkbench menu select **Tools > TrueTime > Stop Performance Logging**.
3. To **discard** accumulated measurements, from the DriverWorkbench menu select **Tools > TrueTime > Clear Performance Log**.
4. After you perform an **Analyze Performance** on the current session, you can click on **File > Save As** to save this session's Log File.

Loading a Previously-Recorded TrueTime Log File

To load a previously-recorded TrueTime Log (***.ttd***) File:

1. Select **File > Open**.
2. Choose ***.ttd*** as your filter at the **Files of Type** entry.
3. Select the desired Log File and click on **Open**.

The session defined by the Log File is now open, and its data can be analyzed. The Performance Analysis window should be open on the DriverWorkbench desktop.

Exporting Collected TrueTime Data

Data can be exported from any of the TrueTime displays for use in other programs such as spreadsheets using the following procedure.

1. Select the desired TrueTime display
2. Set the view to the scale and precision desired for the exported file.
3. Select **File > Export to text...**
4. For the **Files of Type** entry, choose **.txt** or **.csv** as your filter. Files saved as **.csv** output the function timing in a comma-separated value format. Files saved as **.txt** will be tab-delimited.
5. Name the desired Export File or leave the default name.
6. Click on **OK**.

Using Parameter-Based Timing

Parameter-based timing allows you to get a better idea of the amount of time a function takes based on a particular argument that is passed into the function. This will enable you to view the past performance of your driver on control paths dependant on the input argument.

Setting Up the Parameter to be Watched

After you have decided which driver you are profiling and which functions in this driver you want to profile, you can now choose the specific argument on which to generate parameter-based data. In the DriverStudio Configuration dialog under **TrueTime > Select Functions** you can expand any functions that take arguments upon which to base the profiling. Simply expand the function and select the argument that you want to watch.

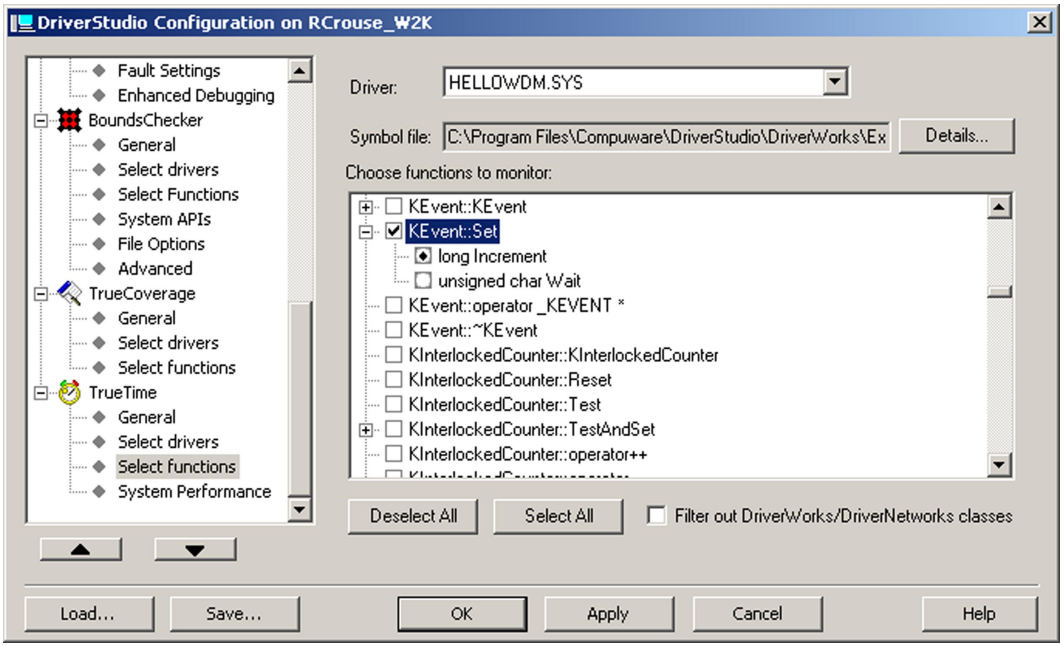


Figure 3-7. Selecting the Argument KEvent::Set, long Increment

Restart the driver (whether by rebooting the machine or by stopping and starting the driver).

After restarting the driver, go into DriverWorkbench and select **Tools > TrueTime > Analyze Performance** to get TrueTime information. If you look at the bottom pane of the Function Stats window (Figure 3-8), you will see the argument selected for the function highlighted in the

window above is listed at the top of the first column. All of the different values that were passed into the function in this argument are listed

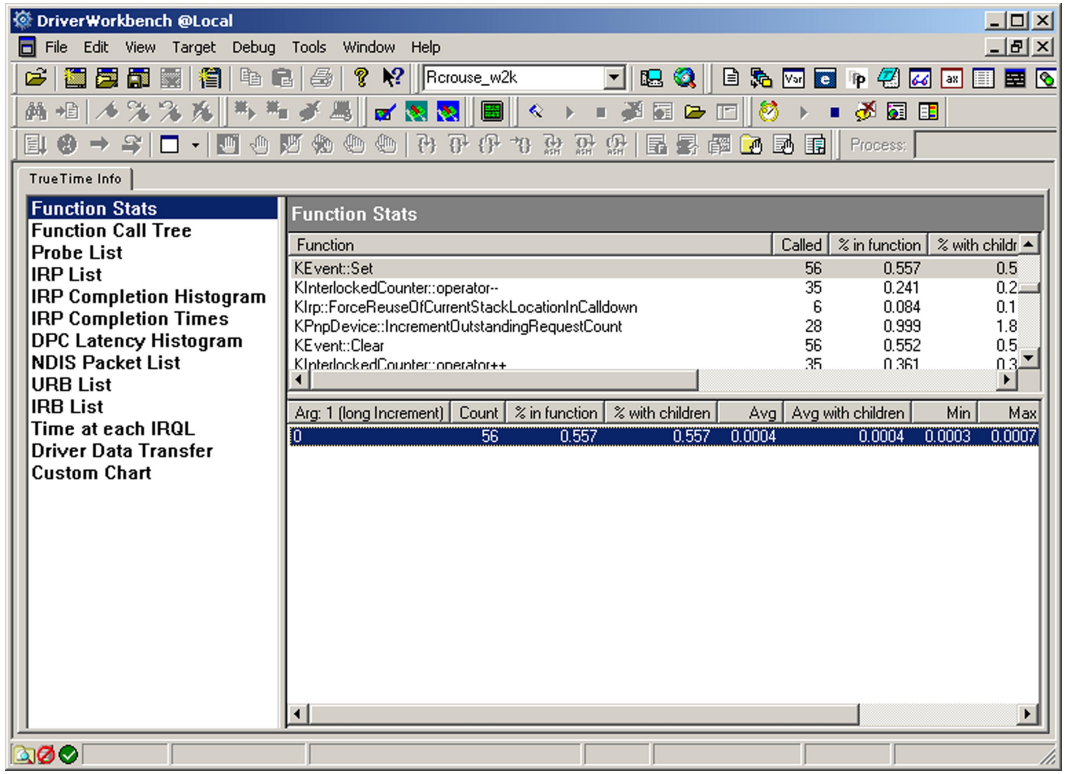


Figure 3-8. Function Stats for KEvent::Set, long Increment

Parameter-based timing is most effective when the number of possible values for the argument is finite (such as a Bool, a flag, or some sort of switch statement value). In this way, there will only be several lines under the Argument Expansion pane with the timing data being extremely relevant to the argument value which was passed in. Other useful cases would be situations involving a length or size parameter where the timing would vary over different amounts of data to be read/written/processed, like the ROP of a BitBlt call. Basically, any arguments that determine specific code paths within the timed function would be good arguments to use.

On the other hand, selecting an argument like an address that is being passed in will most likely lead to a very large number of different addresses, each with a single time executing. This, in turn, would generate a new line under the argument expansion and make the data no more useful than the original Function Stats.

Making a TrueTime Comparison

Compare performance data against previous profiling sessions to validate code optimizations. TrueTime allows you to take two .TTD files from two different runs on the same driver and get a side by side view of the performance improvements resulting from the changes made.

To do the compare, open DriverWorkbench or the .Net environment and go to Tools > TrueTime > Compare TTD Files. This will bring up bring up the TTD File Comparison dialog (Figure 3-9).

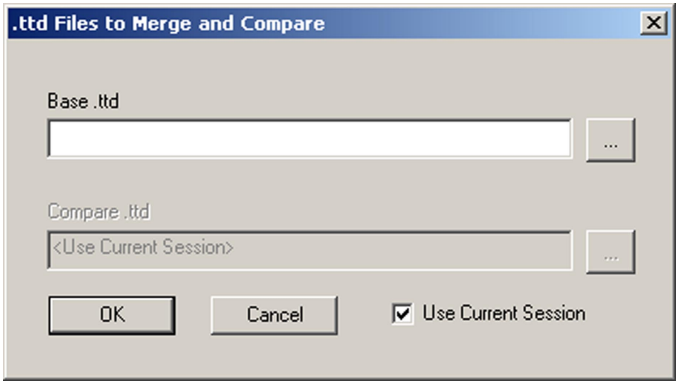


Figure 3-9. TTD File Comparison Dialog

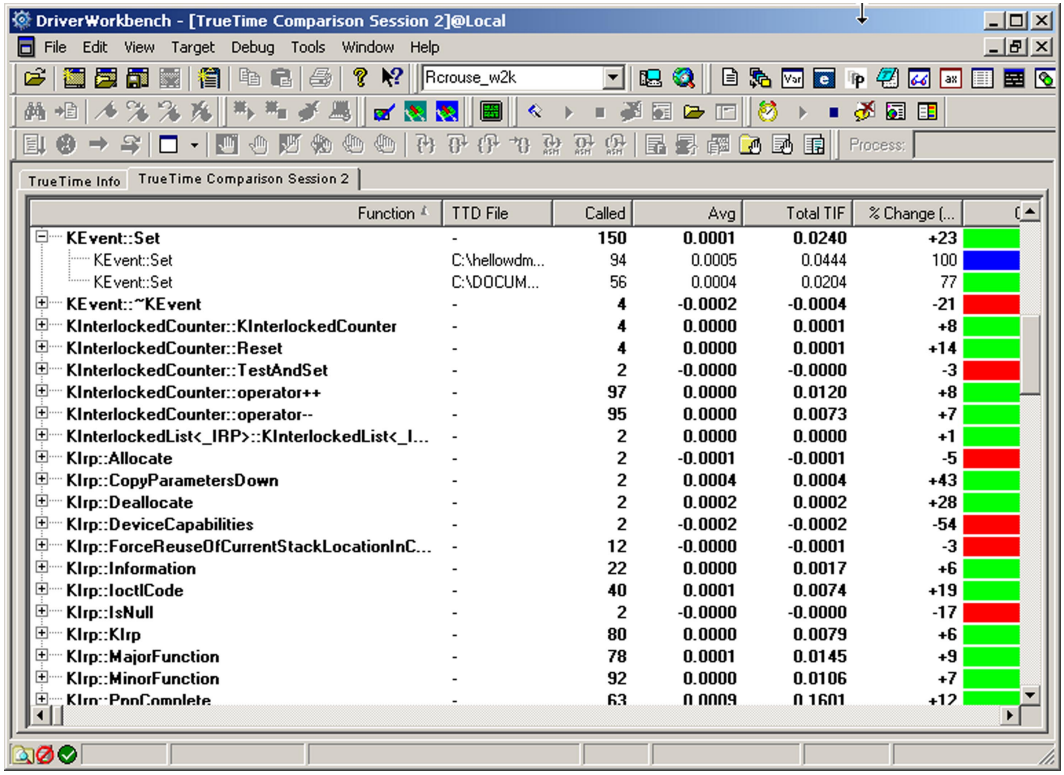
Select a Base file to compare against. This is usually a previously saved data file from an earlier build of the driver you want to compare. Here you are allowed to select any valid .TTD file on your system as the baseline data in the “Base .ttd” field.

Next, select a Compare file to compare to the base file. Use either the currently running TrueTime session to compare against the Base or to select another .TTD file from your system.

Note: If TrueTime is not currently running on your system, the option to “Use Current Session” will not be available and you will have to choose two previously saved .TTD files to compare.

After selecting the two files and clicking OK, you will be presented with the TrueTime Comparison Session window (Figure 3-10). Timing is compared when matching symbol names are found in both data files. The actual times can be seen for each file by expanding nodes in the tree. When a function exists in only one TTD, the timing is displayed without comparison.

The closer to repeating exactly the same tests and duration of test for the two analysis the more useful the Comparison will be, since it will give you a more accurate representation of the difference any code changes may have had on your final timing results.



Function	TTD File	Called	Avg	Total TIF	% Change (...
KEvent::Set	-	**150**	**0.0001**	**0.0240**	**+23**
KEvent::Set	C:\hellowdm...	94	0.0005	0.0444	100
KEvent::Set	C:\DOCUM...	56	0.0004	0.0204	77
KEvent::~KEvent	-	**4**	**-0.0002**	**-0.0004**	**-21**
KInterlockedCounter::KInterlockedCounter	-	**4**	**0.0000**	**0.0001**	**+8**
KInterlockedCounter::Reset	-	**4**	**0.0000**	**0.0001**	**+14**
KInterlockedCounter::TestAndSet	-	**2**	**-0.0000**	**-0.0000**	**-3**
KInterlockedCounter::operator++	-	**97**	**0.0000**	**0.0120**	**+8**
KInterlockedCounter::operator--	-	**95**	**0.0000**	**0.0073**	**+7**
KInterlockedList<_IRP>::KInterlockedList<_I...	-	**2**	**0.0000**	**0.0000**	**+1**
KIrp::Allocate	-	**2**	**-0.0001**	**-0.0001**	**-5**
KIrp::CopyParametersDown	-	**2**	**0.0004**	**0.0004**	**+43**
KIrp::Deallocate	-	**2**	**0.0002**	**0.0002**	**+28**
KIrp::DeviceCapabilities	-	**2**	**-0.0002**	**-0.0002**	**-54**
KIrp::ForceReuseOfCurrentStackLocationInC...	-	**12**	**-0.0000**	**-0.0001**	**-3**
KIrp::Information	-	**22**	**0.0000**	**0.0017**	**+6**
KIrp::IoctlCode	-	**40**	**0.0001**	**0.0074**	**+19**
KIrp::IsNull	-	**2**	**-0.0000**	**-0.0000**	**-17**
KIrp::KIrp	-	**80**	**0.0000**	**0.0079**	**+6**
KIrp::MajorFunction	-	**78**	**0.0001**	**0.0145**	**+9**
KIrp::MinorFunction	-	**92**	**0.0000**	**0.0106**	**+7**
KIrp::PnpComplete	-	**63**	**0.0009**	**0.1601**	**+12**

Figure 3-10. TrueTime Comparison Session Window

Note: A column for standard deviation is available by adding the DWORD `HKEY_LOCAL_MACHINE\Software\NuMega\DriverWorkbench\StdDev` and setting it to 1. This column is not enabled by default because in order to calculate the standard deviation correctly it takes a significant amount of time. If the amount of time it takes is not a factor than the column will give you an extremely good idea of how precise the average is in relation to the spread of the different times called. The smaller the standard deviation is the better the average represents what will happen each time.

Comparison sessions can be saved from the **File menu** > **Save As**...These have a .TTM extension and can be opened without the original .TTD files present.

Explanation of the Comparison Session Window

The window presents the function name in the “Function” column. If there is a matching function that was profiled in both .TTD files there will be a “+” arrow allowing you to see the statistics from both runs listed underneath the main function, the base file is always on top.

The “TTD File” column gives the name of the .TTD file that the function came from.

The “Called” function gives the aggregate number of calls in the root of the tree and the number of calls for each .TTD file in the branches.

The “Avg.” column gives the difference between the average times it took for the two functions to run (Base time – Compare time).

Additional columns can be displayed by right clicking in the chart or from the Tools menu, TrueTime, “Show/Hide Columns”. % in function, % with children, Average with children, Minimum time, Maximum time, First, Last, and Total Time can be added to the default columns displayed.

The “% Change” column gives the percentage change based upon the graph key. By default, the average time used. The graph key can be changed either by right-clicking in the chart or from the Tools menu: Use **TrueTime > Graph Key** to key off any of the columns, visible or hidden. This column is also the basis for the graph on the right.

The bar graph is based on a 0% to 100% scale with a black line denoting the 100% mark. You can drag the 100% bar left or right and the graph will resize accordingly allowing you to view the data in any relative scale you would like.

The color-coding is as follows:

Color	% (Percentage)
Blue	100
Red	> 100
Green	< 100

Performance Analysis Windows

About the Performance Analysis Windows

TrueTime has the following Performance Analysis Windows:

- **Function Stats**. The Function Statistics window, as its name implies, displays statistical data collected for each function that had been selected for monitoring in the current session.
- **Function Call Tree**. This window provides a hierarchic view of a driver's functions.
- **Probe List**. From this window, you can view probe event data produced by probe points in the driver. (See the topic “Using the

TrueTime Probe Point API” on page 67 for more information on probe points.)

- **IRP List**. This window provides detailed information about I/O Request Packet handling performance.
- **IRP Completion Histogram**. This window provides user-adjustable histogram views that illustrate how long a driver's IRPs are taking to complete.
- **IRP Completion Times**. The IRP Completion Times window provides yet another view of IRP completion times. In this case, the graphs plots how long it takes an IRP to complete with respect to its arrival rate.
- **DPC Latency Histogram**. The DPC Latency Histogram displays the delay between DPCs' being queued and actual execution.
- **NDIS Packet List**. This window describes NDIS packet events.
- **URB List**. This window gives you URB information such as the name and path of the driver that sent URBs and the total number of URBs send by this driver.
- **IRB List**. This window gives you IRB information such as the name and path of the driver that sent IRBs and the total number of IRBs send by this driver.
- **Time At Each IRQL**. This window displays a “pie” chart that shows the time that the driver spends at various interrupt request levels, shown as a percentage of overall time.
- **Driver Data Transfer**. The Driver Data Transfer window's *scatter plot* shows the quantity of data transferred by IRPs, plotted against the time since the driver started.
- **Custom Chart.** The Custom Chart provides an integrated, customizable display of all function, IRP and system performance data for a TrueTime session. The time spent in each individual call to a function or IRP can be plotted versus system performance counters across any specific time span of execution. Sophisticated zoom and pan functionality provides fast and easy browsing of large data sets.

The functional operation of these windows is described in detail in the TrueTime Driver Edition Help file, TTDE.chm.

Chapter 4
Using TrueCoverage Driver Edition

Introduction to TrueCoverage Driver Edition

When you test code changes, you want to make sure you are testing all of your changes. Untested changes can produce unexpected results and introduce bugs. The best way to make sure you test everything is to use a coverage tool like TrueCoverage.

TrueCoverage helps you detect which parts of your code have been tested, and which still need to be tested. You can save testing time and improve code reliability by measuring and tracking code execution and stability during development.

Benefits of TrueCoverage

TrueCoverage provides the following essential coverage analysis features:

- Comprehensive data collecting and reporting
- Session data merging

Comprehensive Data Collecting and Reporting

TrueCoverage gathers comprehensive coverage data for drivers, including functions and individual lines of code.

Session Data Merging

You can collect coverage data every time you run your driver, and accumulate the data in a merge file. If several members of your development team use TrueCoverage, you can merge the files generated by all the team members to determine project-wide coverage statistics.

Understanding TrueCoverage Data

TrueCoverage internally gathers data at the assembly instruction level, It scans all the possible code path in a driver image and record how many times each path was executed during a driver run (between the times the driver was loaded and unloaded).

TrueCoverage can present the gathered data in two different ways depending on whether the source code for the driver under test is available or not. For both representations, a graphical view of the possible code paths for each function is shown.

Binary Mode

In binary mode, the basic unit used to calculate the coverage statistics (percent of driver executed, percent of function executed…) is the "block." A block is the smallest sequence of continuous assembly instructions. Some examples of basic blocks are shown in Figure 4-1.

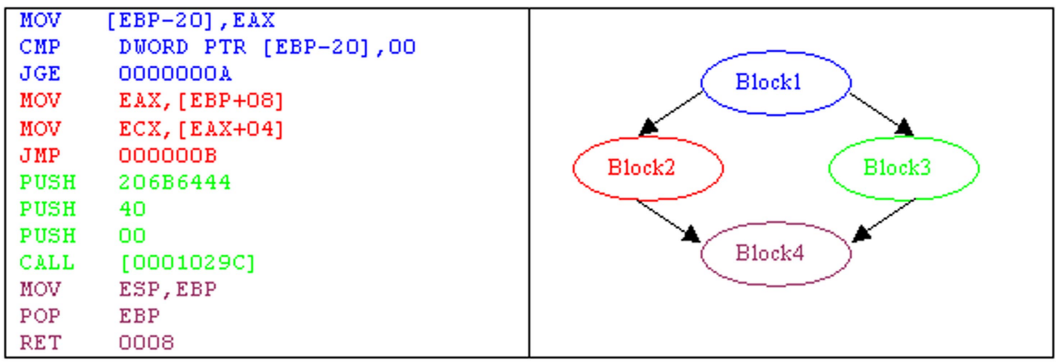


Figure 4-1. Basic Unit of Coverage

When the code path reaches an instruction that modifies the code flow (like a jump), it marks the end of a block as well as the beginning of one or more new blocks.

TrueCoverage records how many times each of these blocks was executed and uses those values to calculate the coverage statistics for each function and for the whole driver.

Source Mode

In source mode, the line of source code is used as the basic unit to calculate the different coverage statistics (percent of driver executed, percent of function executed…). Blocks are still used internally, but the coverage statistics are calculated on a per-line basis. This means that coverage statistics values will differ between binary mode and source mode, because several blocks may represent one line of source code or several lines of source code may be represented by only one block. Figure 4-2 shows an example of how blocks of assembly code map to source code. The red block represents two lines of code: the call to f() and the "else" keyword.

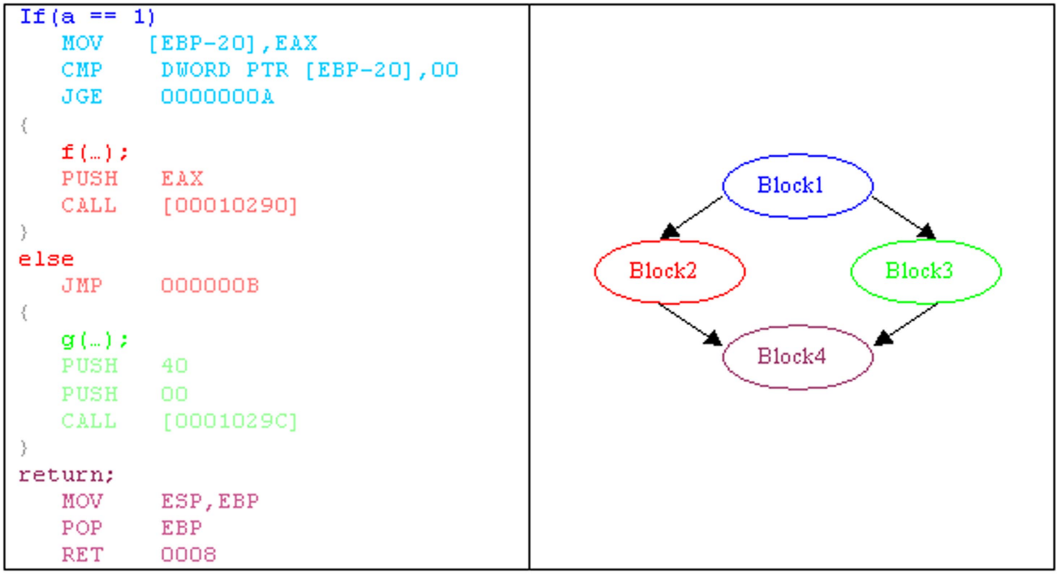


Figure 4-2. Blocks of Assembly Code Mapping to Source Code

Configuring TrueCoverage Driver Edition

TrueCoverage can be configured locally on the computer that will run the monitored driver or remotely from any computer connected by network to the computer that will run the monitored driver. In both cases the configuration procedure is identical.

1 Launch **Compuware > DriverStudio > Settings** and enable TrueCoverage on the **Startup** page. TrueCoverage can be enabled

simultaneously with BoundsChecker and one of the debuggers (SoftICE or Visual SoftICE).

2. On the TrueCoverage **General** page, enter the Output Directory in the appropriate box. The Measurement Driver always creates Log Files in the Output Directory.

3. If you built your driver with DriverWorks or DriverNetworks, you can check the “Filter out DW/DNW classes” box to filter out the classes that are embedded in your driver by the framework. This way, you will only get coverage information on your code.

4. On the TrueCoverage **Select drivers** page, choose those drivers for which the TrueCoverage Measurement driver will gather data. You can monitor up to three drivers simultaneously.

5. On the TrueCoverage **Select functions** page you can select which functions will be monitored, they are all selected by default. Note that the real function names are only shown if a symbol file for your driver is available. If no symbols are available the Settings application creates names concatenating the word “Function” with the hexadecimal value of the relative virtual address of the function. The same names are used in the coverage report.

6. Click **OK** to save your settings. The Settings application will ask to reboot your system if TrueCoverage was not previously enabled or if you selected a boot driver or a system driver.

   Note that the real function names are only shown if a symbol file for your driver is available. If no symbols are available, the Settings application creates names concatenating the word “Function” with the hexadecimal value of the relative virtual address of the function. The same names are used in the coverage report.

Getting TrueCoverage Data

The Coverage data can be viewed in two ways: using DriverWorkbench or using Visual Studio .Net:

- In DriverWorkbench, TrueCoverage options are located under the **Tools** menu and on the toolbar.
- In Visual Studio.Net, the TrueCoverage options are located under the **Tools > DriverStudio** menu and on the toolbar.

TrueCoverage starts logging data as soon as a monitored driver loads. The coverage data is gathered as the driver is being exercised and written to disk when the driver unloads.

While the driver being monitored is loaded, you can take a “snapshot” of the coverage data by selecting the **Analyze Current Session** option in the TrueCoverage menu or on the toolbar.

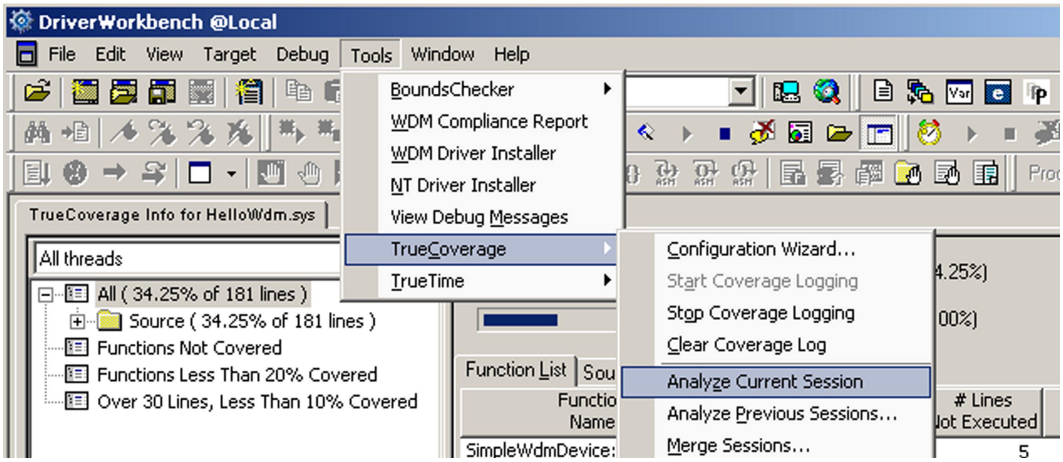


Figure 4-3. Selecting the **Analyze Current Session** Option

A window containing the coverage report appears (Figure 4-4).

After the driver being monitored unloads, you can view the coverage data by selecting the **Analyze Previous Sessions...** option. This opens a dialog with a list of the coverage sessions that were saved to disk when the monitored driver unloaded but were never opened. (See Figure 4-5.)

You can select one or more sessions and click **Open** to display those coverage sessions. Once a session has been opened it disappears from the **Unreviewed Sessions** dialog.

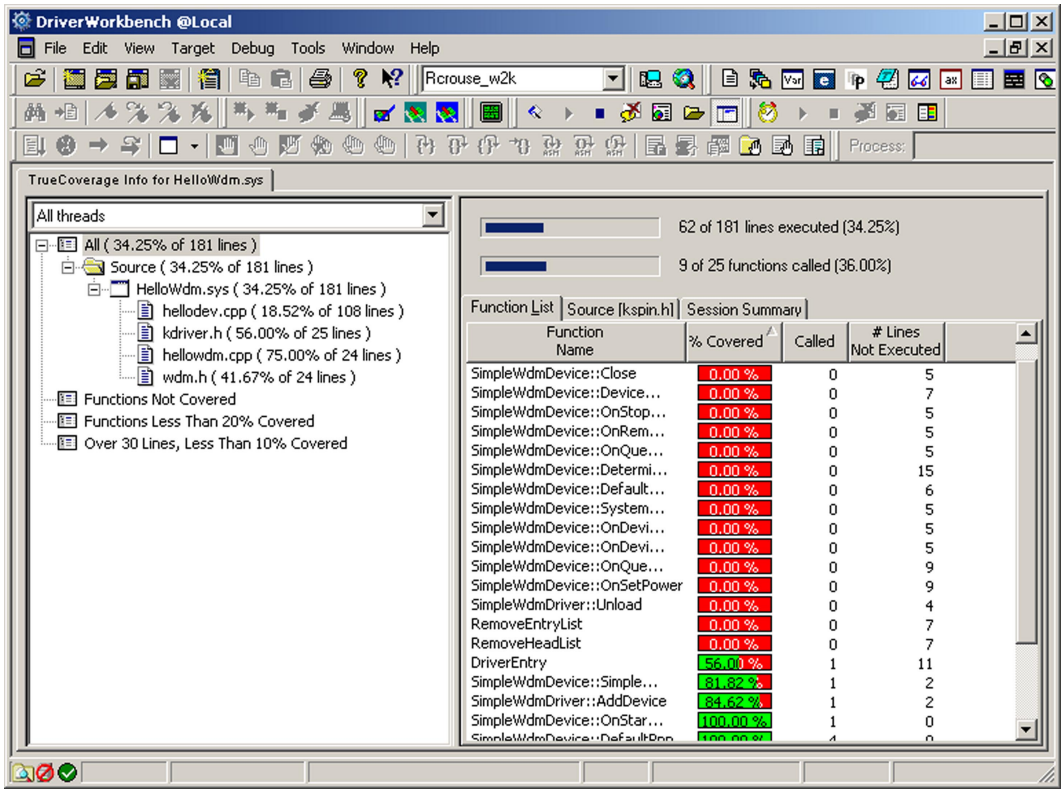


Figure 4-4. TrueCoverage Info from Current Session

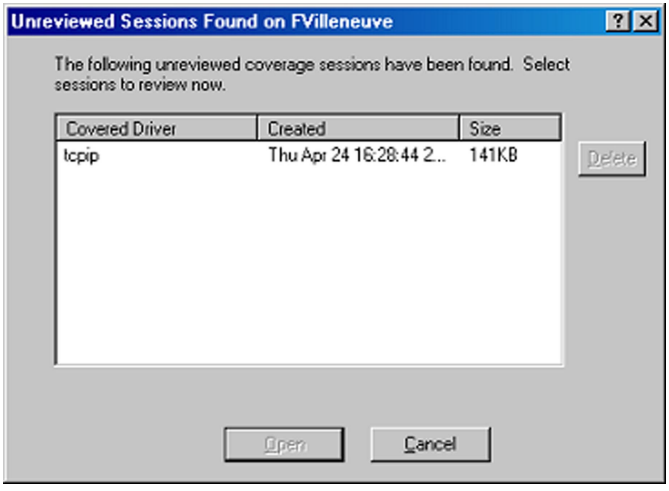


Figure 4-5. Unreviewed Sessions Dialog

Opening/Saving Session Files

Using one of the two methods listed above creates a temporary session files and displays it. You can use the option **File > Save** to save the session file and be able to reopen it later with **File > Open**. TrueCoverage session files are saved with the file extension **tcd**. If you close the report window before saving the session, a dialog will display proposing to save the file. Remember, if you don't save the data, it will be lost.

- Use **File > Open** to open a previously saved TrueCoverage report file.
- Use **File > SaveAs** to save the active TrueCoverage report file with a different name.

Analyzing TrueCoverage Data

The left pane of the TrueCoverage Info session (Figure 4-4) is the *Filter Pane*. Filters are very useful in isolating some functions depending on how they were covered. The **All** filter displays coverage data for every function. In Source mode there is a filter for each source file that displays coverage data for the functions located in that source file. There are three predefined filters that display data for the functions that were not covered or were covered less than twenty percent. It is also possible to add custom filters. (See "Filtering Data" on page 91.)

The right pane of the report contains the Function List tab, Source/Assembly tab, Session Summary tab, and Coverage Meters.

The **Coverage Meters**, located at the top of the pane, summarize the line and function coverage for the item selected in the Filter Pane.

Function List Tab

The data shown in Table 4-1 is available on the Function list tab. To show or hide columns on the Function list tab, see "Showing and Hiding Data Columns" on page 95.

Table 4-1. Function List Tab Data

Column	Definition
Function name	Name of the function
% Covered	Percentage of code that was executed in the function
Called	Number of times the function was called

Table 4-1. Function List Tab Data (Continued)

Column	Definition
#Lines/Blocks not executed	Number of lines or blocks that were not executed
#Lines/Blocks executed	Number of lines or blocks that were executed
Total #Lines/Blocks	Total number of lines or blocks
Image	Binary image containing the function
Source	Source file containing the function
Address	Relative Virtual Address (RVA) of the function

Source/Assembly Tab

The data shown in Table 4-2 is available on the Source or Assembly tab.

Table 4-2. Source/Assembly Tab Data

Column	Definition
Count	Number of times the line was executed
Line#	Line number
RVA	Relative virtual address of the line
Source	Source or Assembly code
Condition	Indicate whether one, all, or none of the cases of a conditional statement were covered

The Condition column (defined in Table 4-2) uses the symbols shown in Table 4-3..

Table 4-3. Condition Column Symbols

Column	Definition
->	Only the right case of a conditonal statement was covered
<-	Only the left case of a conditonal statement was covered
<->	Both cases of a conditional statement were covered

To show or hide columns on the Source or Assembly tab, see “Showing and Hiding Data Columns” on page 95.

TrueCoverage uses color on the Source tab to differentiate among lines of code that were not executed, lines that were partially executed, lines that were executed, and non-executable lines.

By default, TrueCoverage uses the color settings shown in Table 4-4. To make changes these settings, select **Tools > TrueCoverage > Color Settings** .

Table 4-4. TrueCoverage Color Settings

Event	Color
Lines executed	Green
Lines partially executed	Yellow
Lines not executed	Maroon
Non-executable lines	Black

Summary Tab

The Session Summary tab provides summary data about the session, such as creation date, operating system and processor information. It also contains a summary of the coverage statistics for the driver and each source file.

Controlling the Display of Data

You can control your view of coverage data by:

- Filtering data to show information for only the selected source files.
- Sorting the source files in the Filter pane.
- Showing or hiding data columns in the Function List and Source/ Assembly tabs.
- Sorting data using any of the data columns in the Function List tab.
- Selecting the precision for the displayed data.
- Displaying coverage data for a selected thread.

Filtering Data

The Filter Pane lists the source files that your driver used during the session and specially defined filters. You can use the Filter Pane to filter data in the Function List tab and to navigate among source.

In the Filter Pane, select:

- **All** to display data for all source files.
- **Source** to display data for all functions.
- **A driver** to display data for all functions in that driver. Double-click a driver to display a tree view of the source files used in that driver.
- **A source file** to display data for all functions in that file. Double-click a source file to display its code in the Source tab.
- **A filter** to display data for a selected group of functions. For example, *Functions Not Covered* displays data for all the functions that you have not tested, no matter which source file or image contains them.

Double-click any source file in the Filter pane to view its contents in the Source tab.

Creating a New Filter

The coverage report for a large driver contains a lot of data. To filter the report to only view the functions that match a predefined criteria:

1 Right-click on the Filter pane and select **Create Filter** (Figure 4-6)**.**

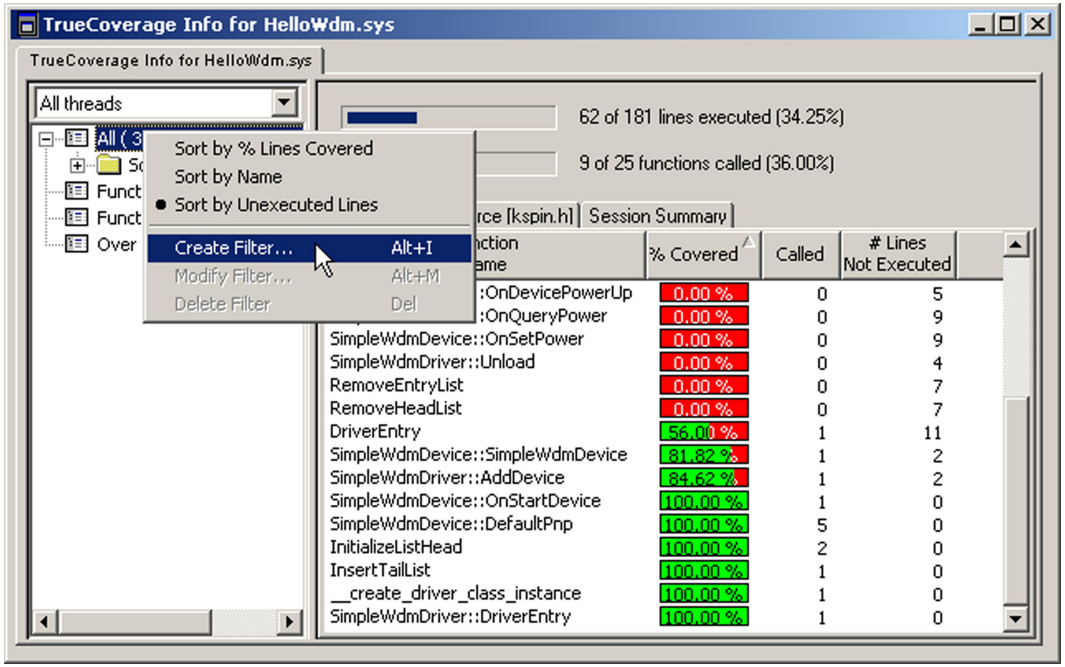


Figure 4-6. Creating a Filter

The **Create Filter** dialog appears (Figure 4-7).

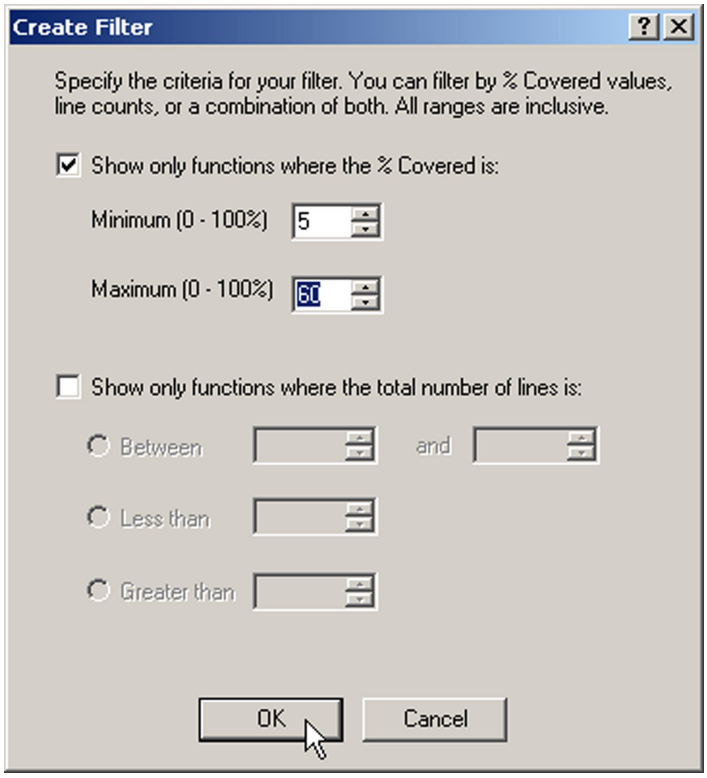


Figure 4-7. Create Filter Dialog

2 Choose the type of filter. You can filter by **% Covered**, **total number of lines**, or a combination of both.

   If you choose % Covered, specify **Minimum** and **Maximum** values.

   If you choose total number of lines, click **Between**, **Less than**, or **Greater than**, and specify the appropriate values.

3 Click **OK** to close the Create Filter dialog.

4 Select your new filter on the Filter Pane to view the list of function(s) that match the filter criteria.

Modifying a User-defined Filter

Complete the following steps to modify an existing filter.

1 Right-click on the Filter Pane and select Modify Filter from the context menu.

The Modify Filter dialog appears.

2 Make changes to the settings.

3 Click **OK** to close the Modify Filter dialog and update the Filter pane.

Deleting a User-defined Filter

Right-click on the user-defined filter to delete and select **Delete Filter.**

Sorting the Filter Pane

You can sort the source file nodes in the Filter pane by percent lines covered or name. Complete the following steps to change the sorting in the Filter pane.

1 On the View Menu, select **Sort > Filter Pane**.
The **Sort Filter Pane** dialog appears.

2 Select one of the following options from the **Sort by** combo box:

- ◇ **Percentage of Lines Covered** to sort by the percent of lines executed in the source file.
- ◇ **Name** to sort alphabetically by the name of the source file.
- ◇ **Number of Unexecuted Lines** to sort by the percent of lines unexecuted in the source file

3 Click **OK** to close the dialog (Figure 4-8) and re-sort the Filter Pane.

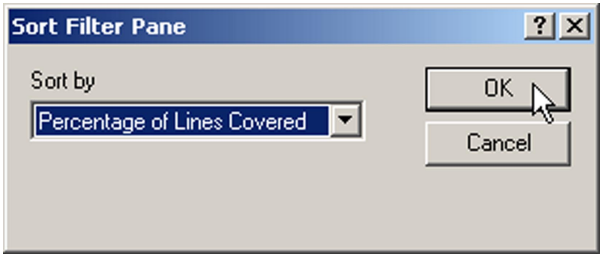


Figure 4-8. Sort Filter Pane Dialog

Sorting Data

In the Function List tab, you can click a column header to sort the data by that column. Click the column header again to reverse the sort order.

Showing and Hiding Data Columns

Right-click on the column headers. When the drop-down menu appears, check the column that you want show and uncheck the column that you want to hide.

Changing Precision

All TrueCoverage data columns can be displayed with one to four digits to the right of the decimal point. Complete the following steps to change the decimal precision.

1. On the View menu, point to Precision.
2. Select the number of digits you want to the right of the decimal point.

Relating the Coverage Graph to Source Code

The Coverage data graph offers a different perspective to understanding the code path that was executed, particularly in binary mode.

Relating Coverage Data to Source Code

TrueCoverage associates coverage data with specific lines of source code. Use the Source tab to review this data.

Displaying a Selected Source File

In the Filter Pane, double-click the source file.

Displaying Source Code for a Selected Function

Complete the following steps to display the source code for a function in the Function List.

1. Click the Function List tab to make it active.
2. Select a function name.
3. Right-click to display the context menu and select **Go to Source**.

To display a specific function on the Source tab.

1. Click the Source tab to make it active.
2. Select the name of the function you want to view from the combo box above the graph.

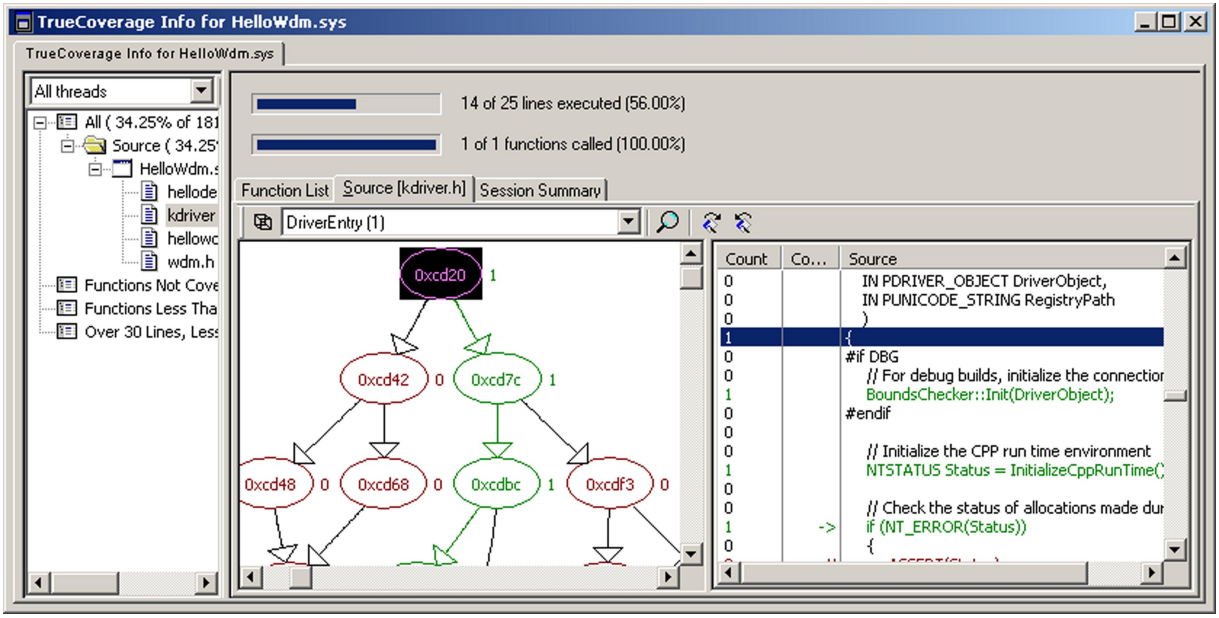


Figure 4-9. Displaying Source Code

Displaying Coverage Data for a Selected Thread

If you have checked the **Log Thread Information** box on the TrueCoverage General Page of the DriverStudio Configuration dialog, you can view the coverage data for a specified thread.

Just go to TrueCoverage Info and click on the combo box at the top of the filter pane. Then, select the thread for which you would like to see coverage data (Figure 4-10), or select **All Threads** to view the global coverage report.

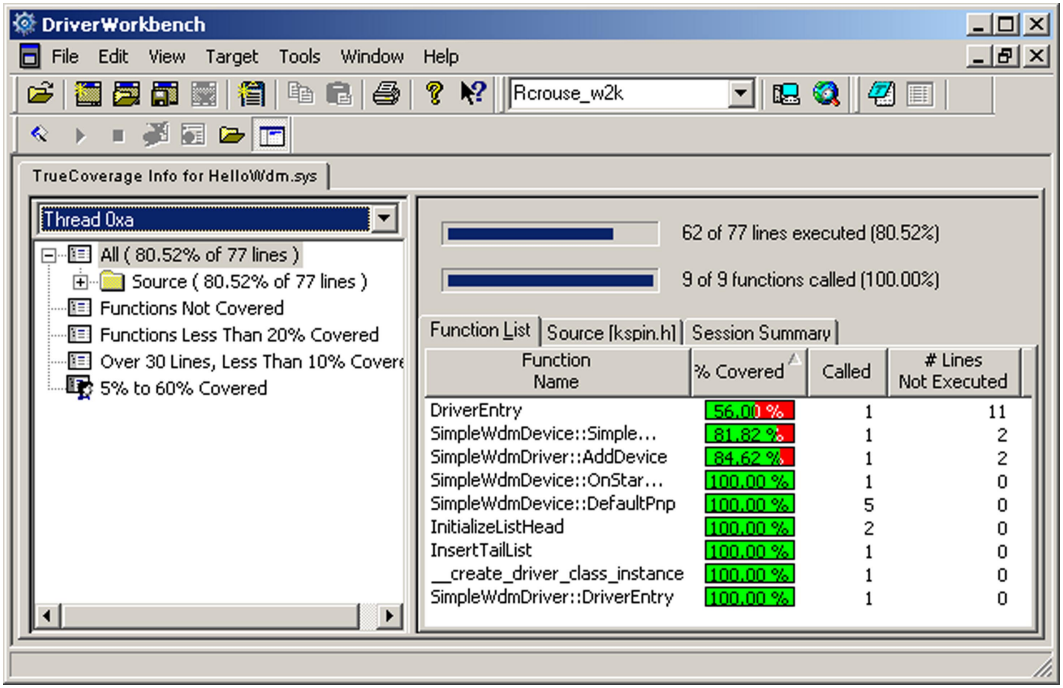


Figure 4-10. Single Thread Coverage Data

Merging TrueCoverage Data

When you use TrueCoverage during your testing, you know exactly how much of your code has been tested. You also know exactly how many lines and how many functions were never tested. Since it is unlikely that you will test everything in one execution of a driver, you want to know how much of your code has been tested overall. You can accumulate coverage data from several executions (**sessions**) of your driver by merging the session files. **Merging** combines the coverage data from multiple sessions in a single file.

About the Merging Process

Merging is the process of accumulating coverage data from multiple sessions. It is important because it is not likely that you execute all, or even most of your code in a single session.

You can merge coverage data by selecting the **Tools > TrueCoverage** menu, and clicking on the **Merge Sessions...** option. When the Merge dialog appears (Figure 4-11), click on the **Add...** button and choose the

TrueCoverage file(s) to merge. You can either merge the selected files into a new file or append them to an existing file.

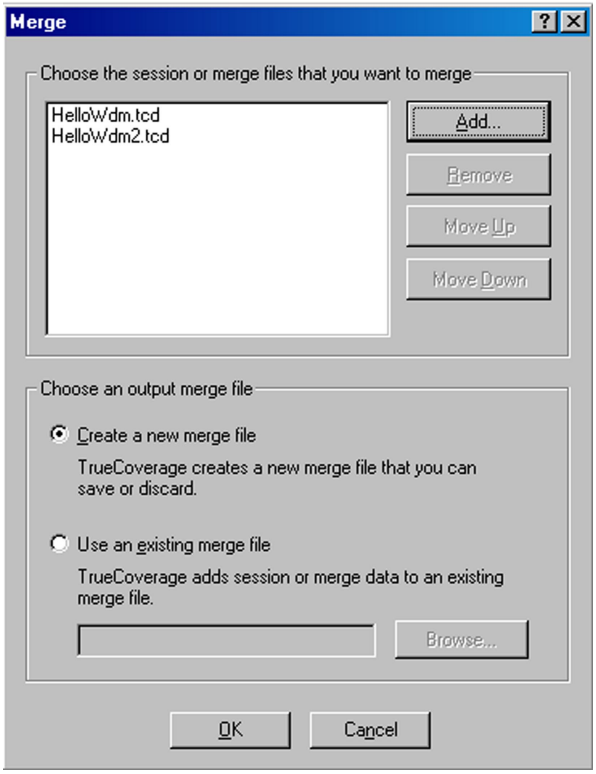


Figure 4-11. TrueCoverage Merge Dialog

Performing a Merge

When you merge session data, TrueCoverage:

- Compares the percentage of covered values and returns the superset of the data. The percentages are not just added together, they are accumulated. For example, if you merge a session with thirty percent of the functions covered and a session with twenty percent of the functions covered, you probably have not reached fifty percent coverage. There are likely parts of the code that were executed in both sessions.
- Calculates "% Volatility" values for each source file and image. These values represent the percentage of functions that changed in your code between sessions, and demonstrates the stability of your code.

- Uses data from the session or merge file that ran the newest image to determine the states of your functions and images. TrueCoverage uses the time stamps of the images to determine which is the most recent image.
- Creates the **Merge History** tab.
- Creates the **Merge Summary** tab.
- Merge files maintain a record of all of the images and functions that were loaded in any of the contributing session files.

Reading Session Data

After you convert an intermediate session file, TrueCoverage displays the data for that session in the Session window. The Session window contains the Filter Pane and Session Data Pane.

The **Filter Pane** contains a tree view of your application images and source files, and a list of filters. TrueCoverage displays the percent of lines covered and total number of lines in parentheses to the right of each source file or image. Select a file or image to display only the functions from that component on the Function List tab. Select a filter at the bottom of the tree to display only the functions in that subset on the Function List tab.

The **Session Data Pane** contains the Function List tab, Source tab, Session Summary tab, and Coverage Meters. The Function List tab displays a list of the functions called during the session with data about each function. The Source tab displays the source code for the selected source file and data collected for each line of code that was executed. The Session Summary tab provides summary data about the session, such as operating system and processor information.

The **Coverage Meters** summarize the line and function coverage for the item selected in the Filter Pane.

You can use various methods to access session data from the Function List and Source tabs.

Merged States for Functions and Images

When you merge sessions, TrueCoverage tracks the state of your functions. For example, TrueCoverage knows when you have changed, added, or removed a function. TrueCoverage displays information about these states in the State column on the Function List and using filters in the Filter Pane.

TrueCoverage uses the following states for functions:

When you...	TrueCoverage marks the function as...
Create a new function	New
Make a change to a function	Changed
Delete a function	Removed
Add an image	New
Remove an image	Removed

Functions that have not changed are displayed with blank entries in the State column. Removed functions are displayed in the **Removed Functions** filter. They are not used to calculate coverage statistics. All other function states are represented in the State column on the Function List tab.

Note: When you merge sessions that used executables with different optimization options, TrueCoverage marks some functions that you did not change as "Changed." When you make any change to a function that changes the number of lines in the function (e.g., add or remove a comment), TrueCoverage also marks the function as "Changed." TrueCoverage cannot distinguish between major and minor code changes.

Chapter 5
Using DriverMonitor

- Introduction to DriverMonitor
- Controlling the Display of Messages
- DriverMonitor Channels

Introduction to DriverMonitor

DriverMonitor displays debug messages issued by software running at local or remote machines. DriverMonitor output is reachable either from within DriverWorkbench or from the MSVC.NET Shell.

Displaying Debug Messages

DriverMonitor can display messages from any of the following sources:

- Kernel mode *DbgPrint* statements on Windows NT family systems
- User Mode *OutputDebugString* statements on both Windows 9x and Windows NT family systems

DriverMonitor can display debug messages issued by both Kernel and User Mode code. These are messages issued by calling the DbgPrint( ) Windows API. The tight integration of DriverMonitor as a plugin for both the DriverStudio DriverWorkbench and .NET shells allows it to display messages issued both locally or by remote Target machines.

DriverMonitor is actually comprised of two parts: the DriverMonitor Plugin and the DbgMsg data collection driver. As long as the driver is installed on a Target DriverStudio machine, local or remote, the DriverWorkbench Plugin can collect debug messages from that Target and display them in a Host window.

Collecting Outstanding Messages

DriverMonitor can be used to collect outstanding messages and write them to a file, or it can be turned on and monitor debug messages dynamically. The availability of DriverMonitor means that debug messages can be collected even while a driver is being debugged from a C++ shell. You can use DriverMonitor either with the DriverStudio DriverWorkbench or inside Microsoft Visual Studio.NET. For example, it is possible to compile and load a DriverWorks driver within Microsoft Visual Studio.NET, while running DriverMonitor inside the .NET shell in a totally-integrated development and debugging environment for kernel mode code.

Running the DbgMsg Collection Driver

Before you can collect messages with your DriverMonitor plugin at your Host machine, you must have the DbgMsg data collection driver installed and running at each Target Machine. Installation of the driver is performed by the DriverStudio installation wizard. The DbgMsg driver starts at "automatic" time, and hence will possibly be actively running by the time the Plugin starts.

The driver is organized in channels, which are logical partitions of data to be sent up to the Plugin. There are two local channels, the *Default* channel and the *Win32* channel. There are also as many remote channels as there are machines within the DriverStudio intranet that have the DbgMsg driver installed. All those channels appear in the local and remote channel choice menu dialogs, and the user can select which ones to turn on. By default the two local channels are started in the running state, therefore debug messages will be pouring in from whatever channels that are selected open. Each channel has a memory queue assigned to it, from where messages are sent up to the application for displaying on the user's window.

The queues are organized as circular queues. Therefore, if the application doesn't pick up debug information from the driver on a prompt basis, some queues may overflow, meaning that some messages will be lost. Of course local channels will fill much faster than remote channels due to the much slower speed of the network interconnect.

About the Plugins

DriverStudio 3.1 Plugins are dynamic libraries structured with the Microsoft Foundation Classes. They adhere to the Microsoft Document/View architecture, and implement one additional class to communicate between the Workbench or .NET shell and the user program itself.

DriverWorkbench knows which plugins are present in the system by looking them up in the appropriate run-time directory and loading them; the .NET shell uses a .CTC file to declare which plugins and which menus are present.

Functionality

Much of the original DriverStudio Version 2.7 functionality has been retained by DriverMonitor. Some functionality migrated from original DriverWorkbench itself into the Workbench or .NET environment (for example, standard file handling procedures such as open, save and close). Other functionality was moved from DriverWorkbench to other plugins (for example, the starting and stopping of drivers).

Menu and Toolbar Structure

The menu and toolbar structure of the DriverMonitor Plugin is somewhat different from that of the DriverStudio 2.7 application. For DriverStudio 3.1, DriverMonitor menus have been split into Permanent and Document/View menus. Permanent menus are visible at shell level even if DriverMonitor is not visibly displaying information (i.e., they control functionality that is independent of the view). Document/View menu entries are considered as tied to the document and to the view, and consequently are not treated as permanent menus. Such entries were moved up to the File or View menus as appropriate. However, these menus are context sensitive in both DriverWorkbench and .NET. This means that whenever DriverWorkbench is running and has “focus,” the Workbench or .NET shell will display the DriverMonitor File or View menu choices among those available in its native File and View menus.

Running DriverMonitor under DriverWorkbench

DriverMonitor runs on the Host machine and allows information accumulated by the DbgMsg driver to be displayed. If DriverStudio has been installed in the proper way, DriverWorkbench will be able to see DriverMonitor as one of its Plugins. This will cause DriverWorkbench to add one top-level menu entry for DriverMonitor. Clicking on that Monitor entry will bring up a single *Monitor Debug Messages* command item.

Clicking on the *Monitor Debug Messages* item will start DriverMonitor as a new page inside the current DriverWorkbench Pad. If you desire to use a separate pad for DriverWorkbench, you can invoke the "View/Pad/Create" sequence, and then click on **Monitor** and select **Monitor Debug Messages**.

You can also start DriverMonitor by clicking on its icon visible among the DriverWorkbench's toolbars.

Note that only one copy of DriverMonitor is active at any time, so, if DriverMonitor is already running, both the *Monitor Debug Messages* menu item and the DriverMonitor icon will be disabled. Once the DriverMonitor Plugin is started, the first debug messages will begin to appear.

Messages come Time-stamped from the Driver to the Application. The timestamp is relative to the time when the plugin starts. When the DriverMonitor Plugin is first started, the messages retained in the DbgMsg driver's queues will be sent to the Host DriverMonitor Plugin, and they will hence be displayed with negative timestamps. Thus, a debug message with a negative timestamp means that the message occured before the DriverStudio applicaiton started.

You should also note that DriverMonitor can run in its own pad, or it can run as a page in a pad shared with some other DriverStudio Plugin. Under both DriverWorkbench and Visual Studio .NET, Plugin menus are split into two groups: Persistent Menus, and View Menus. A Persistent Menu is active and can provide user choices even if the DriverWorkbench Plugin is not running on the Host machine; but a View Menu will only be available if the Plugin is open and running.

In DriverWorkbench, Persistent Menu entries are extensions of the specific Top-level Plugin Menu. For example, the *Monitor Debug Messages* menu is a Persistent entry, and it extends from the Persistent Monitor Top-level menu. View menus, however, are context dependent: if TrueCoverage has the focus, the Top-level View menu will include TrueCoverage View menu entries, but if DriverMonitor is running, the Top-level View menu will rather show the DriverMonitor View entries merged with the rest of the DriverWorkbench's own View menu. Note that Plugins can function on the same pad, as two separate pages.

Running DriverMonitor under Microsoft Visual Studio.NET

You can also run DriverMonitor within Microsoft's Visual Studio.NET. As pointed out before in this document, DriverStudio plugin menus are separated into two classes: Permanent menus and View menus. Permanent menus are visible and can be acted upon even if the plugin is not open on the desktop, while View menus require the plugin to be running before the menu is displayed and made available for use.

In Visual Studio .NET, Permanent menus extend from the DriverStudio submenu, which is itself an item of the .NET Shell *Tools* menu. The menus for all plugins that are visible to DriverStudio are displayed as an

extension of the Tools > DriverStudio menu, grouped by plugin and with separators keeping apart the different groups. DriverMonitor has a permanent menu entry in the Tools > DriverStudio menu.

Just like the DriverWorkbench environment, the .NET Visual Shell will merge view-related menus into its own menus, and selectively present these menus on a context-dependent basis. For example, if TrueCoverage is currently running in the focus window, clicking on the View menu will bring up TrueCoverage View items merged with other .NET menu entries; however, if the focus window is running DriverMonitor, the View-menu items for DriverMonitor appear if you click on *View*.

DriverMonitor Channels

DriverStudio has the concept of a Message Channel through which debug messages are made available to DriverMonitor. Each DriverStudio target has at least two channels: the Win32 channel and the Default channel. In addition, DriverWorks drivers can specify additional custom channels to allow for increased selectivity when displaying debug messages. DriverMonitor displays kernel mode messages from either the Win32 channel or the Default channel, as specified by you. These messages can either be from the local machine or from remote Target machines that are visible through the DriverStudio remote capability.

Available Options

DriverMonitor has additional options that allow better user control over the debug message transmission and displaying process. The user can filter the messages on a per-channel basis, using Regular Expression pattern matching techniques to select which messages get displayed. Other options include auto saving the DriverMonitor message log at regular intervals, setting the timing period for checking messages from remote channels, changing the number of debug messages that are kept in the DriverMonitor buffer, and changing display options such as the font and the format of the time display.

Controlling the Display of Messages

Setting the Maximum Number of Messages

DriverMonitor displays messages from the various channels in its main window. After a certain number of lines have accumulated, the oldest messages become unavailable. The program enables you control the threshold at which old messages are discarded.

To set the threshold at which messages are discarded, select View > Options > Set Message Count. A dialog box appears allowing you to specify a number. The number you specify determines the maximum number of messages that DriverMonitor will maintain in its buffer at any given time. The buffer is refreshed in a circular manner. This means that the oldest messages will be discarded when necessary.

Saving Messages to a File

Saving the Messages Currently Available

To save all the messages currently in the buffer to a text file, select File > Save As from the menu. DriverMonitor then copies all the currently available messages to the file that you specify.

Exporting Messages to a Text File

To export the messages currently in the buffer to a text file, select File > Export Messages to Text, then navigate to the desired directory and enter the destination filename for your export.

Automatically Saving After Every N Messages

DriverMonitor can automatically save the available messages to a text file after every N messages are received, where N is a value that you specify. To set this value, select View > Options > Set Autosave Count and specify the desired value of N in the dialog box. To disable this feature, set the Autosave count to zero.

Controlling Message Scrolling

DriverMonitor has two modes of scrolling messages:

- Keep the most recent message in view
- Keep the messages in the viewing window in sight. The window will not be scrolled if less than the maximum number of allowed messages are in the buffer. Otherwise, the window will be scrolled to keep the current messages in sight even if those messages change position in the window as older messages are discarded. (See “Setting the Maximum Number of Messages” on page 105.)

By default, DriverMonitor automatically scrolls the display to keep the most recent message in view. You can toggle this feature by selecting **View > Options > Keep last message in view**.

Filtering Messages

DriverMonitor gives you the ability to filter messages according to your own criteria (i.e., messages that don't satisfy your filter will not be displayed). You have the following options:

- **Display only messages matching a particular expression.** Any new messages that do not match the expression will be discarded.
- **Do NOT display any messages matching a particular expression.** Any new messages that match the expression will be discarded.
- **Highlight any messages matching a particular expression.** Any messages that match the expression will be highlighted.

In order to edit the filters, select View > Filter Messages. This will bring up the Filter Messages dialog (Figure 5-1) with text boxes for each of the three types of filters.

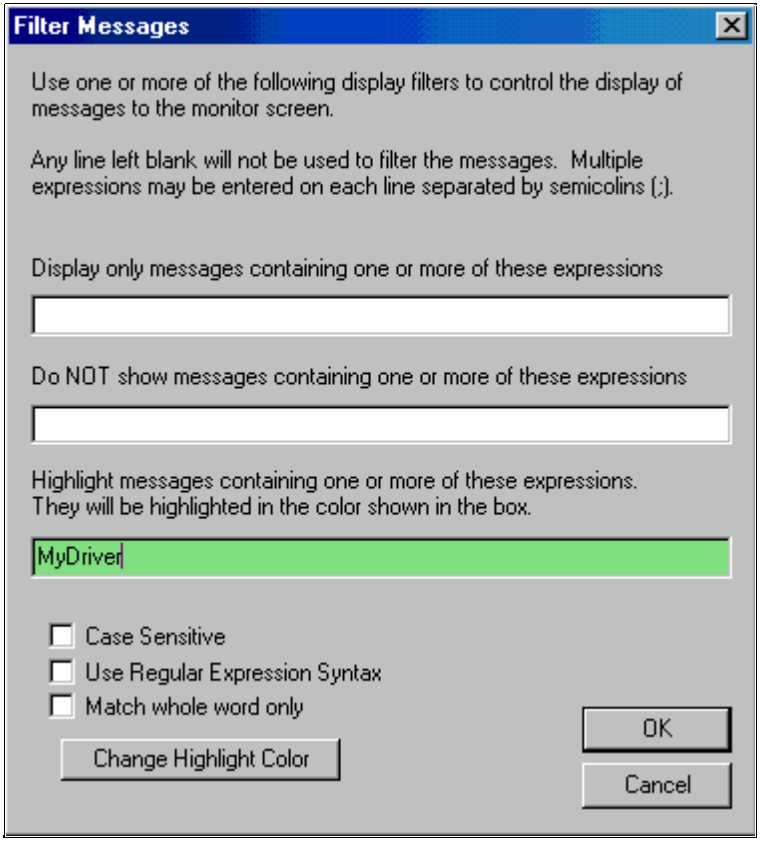


Figure 5-1. Filter Messages Dialog

Note: A filter that prevents a particular message from being displayed only applies to new messages. It will not delete existing messages. Once a message has been excluded, it cannot be recovered.

It is possible to use any combination of the above filtering methods at once. In addition, the dialog gives you choices of turning case sensitivity on or off, to match partial or whole words in the debug message, and to use either a simple wildcard syntax or a more complete regular expression syntax. An additional button lets you select the color of your highlight.

For any one filtering method, it is possible to enter multiple expressions separated by semicolons (;) on the line. Any filter lines left blank will be ignored, and will not filter any messages.

Options for Filter Expressions

The following options simultaneously apply to all three types of filters:

- **Search for whole words only**
- **Use regular expression syntax**. Toggles between wildcard syntax and regular expression syntax.
- **Case Sensitive**. Uppercase and lowercase letters are treated as different characters.

These are picked by clicking within the corresponding checkboxes in the dialog. Additionally, you can choose the highlight color by clicking on the **Change Highlight Color** button.

Expression Matching

Simple Expression Matching

If **Use Regular Expression Syntax** is not checked, the DriverMonitor Plugin uses a simple wildcard syntax. According to that syntax,

- '*' – Matches zero or more characters that are not spaces, tabs, or new lines
- '?' – Matches a single character that is not a space tab, or a newline

Example

My* matches 'My', 'MyDriver', and 'MyVariable'.

M?y matches 'may', but not 'my'.

Complex Expression Matching

If you use Regular Expression Syntax, you can perform much more complex expression matching. Table 5-1 shows some of the more common identifiers used in regular expressions.

Table 5-1. Common Identifiers Used in Regular Expressions

Identifier	Description	Example
*	Zero or more of the character appearing just before the '*'.	BA* matches 'B', 'BA', and 'BAA.
+	One or more of the character just before the '+'.	BA+ matches 'BA', 'BAA', and 'BAAA.'
.	Any single character.	
[x]	One of the characters within the brackets.	'[13579]' matches any odd number.
[x-x]	Matches any character in the range.	'[a-z]' matches any letter.
[^x]	Matches any character not in the brackets.	'[^0-9] matches any non-number.
\s	Matches a space, tab, or new-line.	
\S	Matches anything but a space, tab or new line.	
\w	Any word character. That is, any letter, number, or underscore (_).	
\W	Any character not matching \w.	
^	The beginning of the string.	'^MyDriver:' matches all messages beginning with 'MyDriver:'
$	The end of the string.	

Note: The DriverMonitor Plugin accepts all regular expression identifiers, and is not limited to the ones listed above.You can consult a standard text on Computer Science to find out more about what kind of regular expressions you can use in your pattern matching.

About Timestamps

DriverMonitor displays a timestamp with each message. This timestamp can be displayed as a relative value or as an absolute value. Timestamps displayed in relative format are expressed in milliseconds, while timestamps displayed in absolute format are shown as hh:mm:ss time of

day. The default mode is to display the timestamp in relative format, and the **View > Options > Clock Time** menu pick allows toggling between these two formats.

For local messages, relative timestamp values are computing relative to the time that DriverMonitor started. When you first start DriverMonitor, you may see that some timestamps may be negative. This happens because those messages were emitted by drivers prior to the start of DriverMonitor.

For remote messages, relative timestamp values are computed relative to the time the DriverMonitor Target driver DbgNet.sys started on the remote machine.

A buffer overflow condition may corrupt a timestamp. DriverMonitor can usually detect this condition, but occasionally a spurious timestamp value may appear.

DriverMonitor Channels

Understanding Trace Channels

A trace channel is a stream of information, usually coming from a driver. A driver may write to several different channels, or, it may send all of its information to the default channel. A DriverWorks driver may open custom channels by specifying a channel name in the constructor of a KTrace object. All trace information sent to that object will be sent to the custom channel.

DriverMonitor may display one or more trace channels in its main window, originating either from the local machine, or from a remote machine running DriverStudio.

It is possible to have multiple DriverMonitor windows open, each receiving a different set of channels.

Special Channels

DriverMonitor has two special channels:

- Default Channel
- Win32 Channel

Default Channel

The Default channel receives all system debug output. It receives messages sent to the following services:

- ***DbgPrint*** – Available on WinNT, Win2K, and WinXP.
- ***DbgPrintEx*** – Available on WinXP.
- ***Out_Debug_String*** – Available on Win95, Win98, and WinMe.

The default channel also receives the output from any DriverWorks KTrace object that did not specify a custom channel in its constructor. The default channel is present on all systems running DriverMonitor.

Win32 Channel

The Win32 channel displays messages sent to *OutputDebugString* by user mode applications that are not running under a debugger. The Win32 Channel is available on Win98, WinMe, WinNT, Win2K, and WinXP machines.

Configuring Local Channels

A DriverMonitor host can receive debug messages from local or remote DriverMonitor targets. These messages are received through monitor Channels. If only local channels are to be used, you can configure them by selecting View > Local Channels. You will see a dialog box similar to the one shown in Figure 5-2.

Figure 5-2. Local Channel Status Dialog

The dialog displays a checklist containing all the channels on the local machine. Check the box next to any channel to select it. Once you click the OK button, DriverMonitor will display the debug messages sent through that channel, subject to the filtering you specify with the View > Filter Messages menu pick and dialog.

Typically you will have the Default and Win32 channels available, and maybe more if you're running a DriverWorks driver. Note that if a local channel is open on your machine, its messages will not be transmitted to any remote DriverMonitor hosts connected to that target computer. (For more information on DriverMonitor channels, see the "Understanding Trace Channels" topic on page 110.)

Configuring Remote Channels

A DriverMonitor Host can receive debug messages from local or remote DriverMonitor targets. These messages are transmitted through Monitor Channels. The channels allow you to select groups of messages depending on where they are issued. A target machine will typically have the Default and the Win32 channels, and maybe more if a DriverWorks driver is being monitored.

Remote Channels are channels fed from remote target machines. You can have a host receive input from one or more remote channels by selecting them in the Select Channels dialog. This dialog is displayed when you select View > All Channels (Figure 5-3).

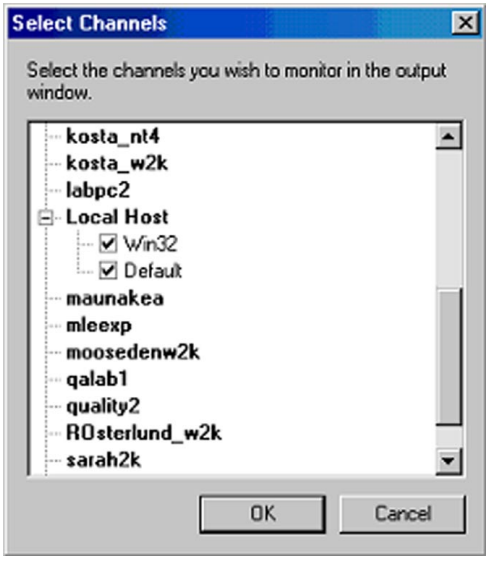


Figure 5-3. Select Channels Dialog

The first time this dialog is selected, the network will be queried for available machines, so there may be a slight delay depending on your network load and on the number of available remote machines. This dialog contains a list of remote target machines whose DriverMonitor drivers can be reached over the network. Clicking on any remote machine name will connect you to the selected machine, and display the machine's available channels. You can select the channels to display at your local host's DriverMonitor window by checking the box next to the channel name. You will be able to select multiple remote channels in addition to your local channels.

Channels may only be received by one machine at a time. This means that any channels already being displayed on another host machine will show up with their checkboxes marked. These channels are unavailable to your host until the machine receiving the channels closes them.

Local channels may also be configured in this dialog. They appear under the machine name *Local host*. Check the box next to each channel to display it.

Note: The local machine has priority over any remote machine asking for a channel. If a channel is received by a remote machine, opening the channel on the local machine will stop the remote machine from receiving DriverMonitor output through that channel. The remote machine's channel feed will resume as soon as that channel is closed on the local machine.

Controlling the Remote Channel Update Frequency

The DriverMonitor host periodically accesses the network to find all eligible targets. The access rate can be changed with the View > Options > Set Auto Channel Check Period menu selection. Frequent updates will improve the DriverMonitor's response time at the cost of increased network traffic; longer periods between updates will decrease network traffic but increase the time it takes for DriverMonitor to find targets over the network.

Glossary

Application Programming Interface (API)	A language and message format used by an applications program to communicate with the operating sysem or some other control program such as a communications protocol. An API is implemented by writing a function call in your program which links to a required subroutine for execution.
# Lines Executed Column	Total number of executed lines in the function.
# Lines Not Executed Column	Total number of unexecuted lines in the function.
% Covered	Percentage of lines in the function that were executed.
% Volatility	Percentage of functions that have changed since the last merge was performed in the current merge file. This value represents the stability of the function. If it remains high, it will be hard to significantly increase your % Covered values.
Called Column	Number of times the function was called.
Filters	Subsets of functions listed at the bottom of the tree view in the Filter pane. When you select a filter, the Function List tab is populated with the functions in that subset.
Functions	Refers to the methods, procedures and statements used by your application.
Image	Images are compiled driver files. .SYS files are images.
Image Column	Name of the executable, DLL, or OCX that contains the function.
Instrumentation	The process of adding instructions to an image that the DriverStudio tool uses to collect data at run time.

Intermediate Coverage Session File (.ics)

Intermediate coverage session files are generated by TrueCoverage when you unload your driver. They are intermediate files that contain coverage data. TrueCoverage places an intermediate session file in the DriverStudio\Temp directory.

To convert intermediate session files to session files, you can open them in TrueCoverage.

Merge File (.dcm)

Merge files store data from two or more sessions. To create a new merge file, you must have saved session files.

Merging

The process of accumulating coverage data from multiple sessions.

Project Merge File

A project merge file is a type of merge file that TrueCoverage creates for each project the first time you select Merge Current File. By default, when you close a new session file that you did not merge into the project merge file, TrueCoverage prompts to ask if you want to merge.

You can restart the data accumulation in a project merge file by deleting the project merge file. If you want to save the data that is already in the project merge file before restarting, you can use Save As.

RVA

Relative Virtual Address.

Session

A session is a single execution of a driver.

Source Column

Name of the file that contains the source code for the function.

State Column

Merged stated of the function. Functions that have not changed are displayed with blank entries in the State column.

- **New** – Function has been added since the last merge was performed in the current merge file. TrueCoverage also marks a function as *New* when the image that contains it is added or re-loaded.
- **Changed** – Source code for the function has changed since the last merge was performed in the current merge file.
- **Removed** – Function or image that contains the function was removed. TrueCoverage also marks a function as *Removed* when the image that contains it is removed.

Total # Lines Column

Total number of lines in the function.

TrueCoverage Driver Session File (.dcs)

Session files store session data. Each session file contains data for one session. You can open a session file without opening the associated project.

User-defined Filters

User-defined filters are filters for which you specify the criteria. With your cursor on the Filter Pane, right-click and select **Create Filter** from the context menu to define a new filter. Choose **Modify Filter** to alter an existing filter.

Index

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X, Y, Z