Using Visual SoftICE™

Release 1.3.0

Windows NT®
Windows® 2000
Windows® XP

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

Doc. 11581
May 19, 2003

Table of Contents

Chapter 6
Using Breakpoints

Preface

Purpose of This Manual

Note: Unless stated otherwise, this document will use “Windows NT/2000/XP” to refer to the Windows NT, Windows 2000, and Windows XP operating systems, and characteristics of Windows NT described in this manual also apply to all supported operating systems.

Visual SoftICE is an advanced, all-purpose debugger that can debug virtually any type of code including applications, device drivers, EXEs, DLLs, and OCXs.

Audience

This manual is intended for programmers who want to use Visual SoftICE to debug code for Windows NT/2000/XP platforms running on either 32- or 64-bit processors.

Organization of This Manual

The *Using Visual SoftICE* manual is organized as follows:

- Chapter 1, "Choosing Your Visual SoftICE Version"
  Helps you decide which version of Visual SoftICE, Visual SoftICE or Classic Visual SoftICE, is best suited for a given debugging scenerio. It also gives an overview of the Visual SoftICE product, and highlights the many new and useful features.
- Chapter 2, "Visual SoftICE Target Transport Configuration"
  Provides an overview of the many available transports, and compares the benefits of each.
- Chapter 3, "Overview of the Visual SoftICE User Interface"
  Explains the GUI for Visual SoftICE, including the meaning of various icons and available menu options. This chapter also discusses how to use the Visual SoftICE configuration settings to customize your environment, pre-load symbols and exports, connect to a target, modify keyboard mappings, create macro-definitions, and set troubleshooting options.
- Chapter 4, "The Visual SoftICE User Interface Pages"
  Explains the GUI pages available for Visual SoftICE, including the functionality and related commands for each.
- Chapter 5, "Visual SoftICE Symbol Management"
  Explains how to manage symbols in Visual SoftICE, including dynamically and persistently loaded symbols, MS Symbol Server setup, and path configuration.
- Chapter 6, "Using Breakpoints"
  Explains how to set breakpoints on program execution, on memory location reads and writes, on interrupts, and on reads and writes to the I/O ports.
- Chapter 7, "Using Expressions"
  Explains how to form expressions to evaluate breakpoints. This concept differs between Visual SoftICE versions, so pay particular attention to this chapter if you are not familiar with Visual SoftICE.
- Chapter 8, "Exploring Windows NT and XP"
  Provides a quick overview of the Windows NT/2000/XP operating systems.
- Appendix A, "Troubleshooting Visual SoftICE"
  Explains how to solve typical problems you might encounter.

- Appendix B, “Kernel Debugger Extensions”
  Explains the available Kernel Debugger Extensions for use with Visual SoftICE, and how to work with them in the Visual SoftICE environment.
- Glossary
- Index

Typographical Conventions

The following conventions are used consistently throughout this manual to identify certain types of 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. Also indicates directory names, and file names.
small caps	Indicates a user-interface element, such as a button or menu.
UPPERCASE	Indicates key words and acronyms.

How to Use This Manual

The following table suggests the best starting point for using this manual based on your level of experience debugging applications.

Experience	Suggested Starting Point
No experience using debuggers.	Read Chapter 1, Chapter 8, and then Chapters 2 through 7.
Experience with other debuggers or another release of SoftICE.	Read Chapter 1 through Chapter 5.

Other Useful Documentation

In addition to this manual, Compuware provides the following documentation for Visual SoftICE:

- Visual SoftICE Command Reference
  Describes all the Visual SoftICE commands in alphabetical order. Each description provides the appropriate syntax and output for the command as well as examples that highlight how to use it.
- Visual SoftICE on-line Help
  Visual SoftICE provides context-sensitive and topic-oriented HTML help as well as command line help for Visual SoftICE commands in the Command page.
- On-line documentation
  Both the *Using Visual SoftICE* manual and the *Visual SoftICE Command Reference* are available as PDF. To access the PDF version of these books, start Acrobat Reader and open the *Using Visual SoftICE* or the *Visual SoftICE Command Reference* PDF files.

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 and the release notes.

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

Note: 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
Choosing Your SoftICE Version

SoftICE or Visual SoftICE?

DriverStudio 3.0 and SoftICE Driver Suite 3.0 include two unique debuggers: SoftICE, the single-machine debugger, and Visual SoftICE, a new GUI-based dual-machine debugger. Depending on the debugging task you are facing, it may or may not be obvious which debugger you should use. This section will help you decide which tool best fits your needs.

In some situations, your choice will be simple: some processor architectures and operating systems are only supported by one of the two debuggers. Table 1-1 shows the platforms supported by SoftICE and Visual SoftICE.

Table 1-1. Supported Platforms

Processor	Operating System	SoftICE	Visual SoftICE
Intel x86 and compatibles	MS-DOS, Windows 3.0/ 3.1/3.11, Windows 9x	Yes	No
Intel x86 and compatibles	Windows NT 3.x, Windows NT 4.0	Yes	No

Table 1-1. Supported Platforms (Continued)

Processor	Operating System	SoftICE	Visual SoftICE
Intel x86 and compatibles	Windows 2000, Windows XP, Advanced Server, .Net Server	Yes	Yes
Intel Itanium1 and Itanium2 (IA64)	Windows XP 64bit Ed., .Net Server 64bit Ed.	No	Yes
AMD Opteron, Hammer (x86-64 / K8)	Windows XP 64bit Ed., .Net Server 64bit Ed.	No	Yes

If you are debugging on DOS or the Windows 9x family, SoftICE is your only choice. If you are working on a 64-bit architecture, only Visual SoftICE will do. If your target is Windows NT/2K/XP and the x86 or compatible architecture, either debugger will work. In that case, read on for an overview of the differences between these two tools.

Single Machine Debugging: SoftICE

SoftICE is a single-machine debugger, meaning simply that all of its code runs on the same machine as the code being debugged. When running, SoftICE has two basic states: popped up, where the SoftICE window is displayed, and popped down, where SoftICE is invisible and the machine runs as normal. When SoftICE is popped up, all processes on the machine are stopped, the operating system does not run, and SoftICE commands are available to the user. SoftICE can pop up in response to user input (the CTRL-D hotkey), breakpoints, exceptions, or system crashes. SoftICE is popped down by issuing one of the go or exit commands, at which point the SoftICE screen is erased and all processes in the system resume operation.

The fact that SoftICE halts the operating system when it is popped up means that it must operate without making use of any of the OS services. This has a number of consequences. For one, the SoftICE user interface does not resemble that of a normal Windows application. Although SoftICE supports keyboard and mouse input, it does not use Windows fonts, nor does its interface contain the enhancements common to Windows applications. In addition, SoftICE cannot assume that it is safe to perform disk access whenever it is popped up, so loading or saving symbol information and SoftICE data is done through companion applications, such as Symbol Loader (Loader32.exe).

Another consequence of the SoftICE single machine architecture is that the interface is extremely fast. All the data in the machine is directly accessible to the debugger, so even tasks involving large amounts of memory access are completed with no noticeable delay.

Because symbols and source code must be loaded ahead of time, SoftICE uses a packaged format for symbols called NMS files. Symbols, translated from the DBG or PDB files output by the linker, can be combined with all or some of the source files used to build the module, and loaded into SoftICE all at once using Symbol Loader or its command-line equivalent, NMSYM. In addition, the new Microsoft Symbol Servers can be accessed using Symbol Retriever utility, which is also capable of translating symbols into NMS files and loading them into SoftICE. These tools make the necessary management of symbols for SoftICE as simple as possible.

SoftICE supports a subset of the available KD Extensions defined by Microsoft. Because the operating system is stopped when the debugger is popped up, SoftICE does not support all the available KD Extensions, since it is not able to make system calls.

There are certain situations where debugging on a single machine is impractical. For instance, if your project is a display driver that is not yet working properly, SoftICE may not be able to display its output. SoftICE does include support for remote debugging, which can be used in many of these situations to redirect the SoftICE input and output over a serial or IP networking link. The remote application in this case is SIRemote, which simply acts as a dumb terminal for SoftICE. The operation of the debugger is not otherwise changed by running remotely.

Dual Machine Debugging: Visual SoftICE

Visual SoftICE, on the other hand, is a dual-machine debugger. The user interface and nearly all of the interpretive code runs on the “master” machine; the code to be debugged runs alongside a small core of debugging functions on the “target” machine. Master and target machines are connected via a transport, which can be a serial cable, IP network interface device, or IEEE 1394 connection.

Because the master machine is never stopped by the debugger, the Visual SoftICE user interface is free to take advantage of all of the usual Windows UI devices. The Visual SoftICE user interface will be instantly familiar to anyone who has used sophisticated Windows programs before; in addition, the command set has been duplicated (with a few exceptions) from the original SoftICE, so Visual SoftICE users should find much that is familiar about Visual SoftICE as well.

Visual SoftICE is also able to load symbol information on-the-fly at any time – including retrieving symbols from a Symbol Server site – so this task is generally handled automatically by the debugger. This frees the user from the necessity of manually specifying symbol files to be loaded by the debugger, although that option is still available in Visual SoftICE.

Visual SoftICE supports loading and examining crashdump and minidump files directly, a feature not found in SoftICE (the DriverStudio DriverWorkbench Application also supports this).

Visual SoftICE also provides complete support for the Microsoft KD Extensions, including those that will not run on SoftICE for architectural reasons.

Which One Should I Use?

If your project falls into the wide overlap between SoftICE and Visual SoftICE, and you have never used SoftICE before, you are probably still wondering which debugger is best for you. Obviously, there is not always a single right answer to this question, but in the remainder of this section we will try to cover some of the scenarios where one debugger might be favored over the other. We are down to guidelines here, though; devotees of either debugger will be quick to point out that their favorite still has advantages, even in cases where the other might appear to be the better choice. We encourage you to try them both, and consider them two similar but distinct tools in your debugging toolbox.

- If you prefer a full-featured Windows GUI, you will probably want to use Visual SoftICE. The SoftICE interface is fast and powerful, but it has no GUI, and it takes some getting used to.
- If you are debugging a crashdump file, try Visual SoftICE. You will be able to use many of the debugging commands you are already familiar with, and Visual SoftICE can reveal more information than the crashdump functionality within DriverWorkbench.
- If you need complete KD Extensions support, use Visual SoftICE. SoftICE provides a limited subset of KD Extensions, but not the whole set.
- If you are debugging a network driver, and you are concerned that the Visual SoftICE IP transport layer might affect the results, use SoftICE. Conversely, if you are debugging a video driver's mode initialization, or a Direct3D or streaming app or driver, try Visual SoftICE or run SoftICE remotely.

- If you want direct access to BoundsChecker events from within the debugger, use SoftICE. SoftICE can stop the machine when an event occurs and allow you to diagnose problems as they occur, even after a system crash.
- If you do not have access to a second machine, or you are traveling and debugging code on a laptop, use SoftICE.
- If you need the ability to package source code together with symbolic debugging information in NMS files, use SoftICE. Both debuggers are capable of loading source code separately from symbol files, of course.

And if you are still confused about which debugger to use, skim through the documentation for both of them. Chances are that something you see there will point you in the right direction.

Visual SoftICE Overview

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.

Visual SoftICE has a 32-bit Master that connects to either 32-bit or 64-bit targets over a variety of available transports.

About Visual SoftICE

Some of the major benefits Visual SoftICE provides include the following:

- Source level debugging of 32-bit (Win32) and 64-bit applications, and Windows NT/2000/XP device drivers (both kernel and user mode).
- Support for x86 Itanium and x86-64 (AMD) platforms.
- Debugging virtually any code, including interrupt routines and the Windows NT/2000/XP kernels.
- Setting real-time breakpoints on memory reads/writes, port reads/ writes, and interrupts.
- Setting breakpoints on Windows messages.
- Setting conditional breakpoints and breakpoint actions.
- Displaying elapsed time to the breakpoint trigger using the processor clock counters.
- Displaying internal Windows NT/2000/XP information, such as:
  - Complete thread and process information
  - Virtual memory map of a process
  - Kernel-mode entry points
  - Windows NT object directory
  - Complete driver object and device object information
  - Win32 heaps
  - Structured Exception Handling (SEH) frames
  - DLL exports
- Using the WHAT command to identify a name or an expression, if it evaluates to a known type.
- Supporting the MMX, SSE, SSE2, x86 instruction set extensions, plus other data decoding such as 3DNow
- Creating user-defined macros.
- On-demand symbol handling.
- Full KD extension support.
- Visual SoftICE provides a robust and customizeable GUI with dockable pages for debugging applications across all platforms. The Visual SoftICE user interface is designed to be functional and flexible.

Chapter 2

Visual SoftICE Target Transport Configuration

Visual SoftICE Target Transport Overview

Visual SoftICE offers several debug transport choices, all of which have their own advantages and disadvantages. Some transports might not be available to you depending on your hardware configuration, or a particular problem you are trying to debug. The only time the master does not use a distinct transport is when it opens a crash dump file.

Visual SoftICE supports the following transport connections between the master and a live target:

- Serial
- Dedicated PCI Ethernet Network Interface Card (NIC)
- Universal PCI Ethernet NIC
- OHCI/UHCI USB Host Controller and USB NIC
- Virtual NIC Driver (optional)
- 1394 (Firewire)

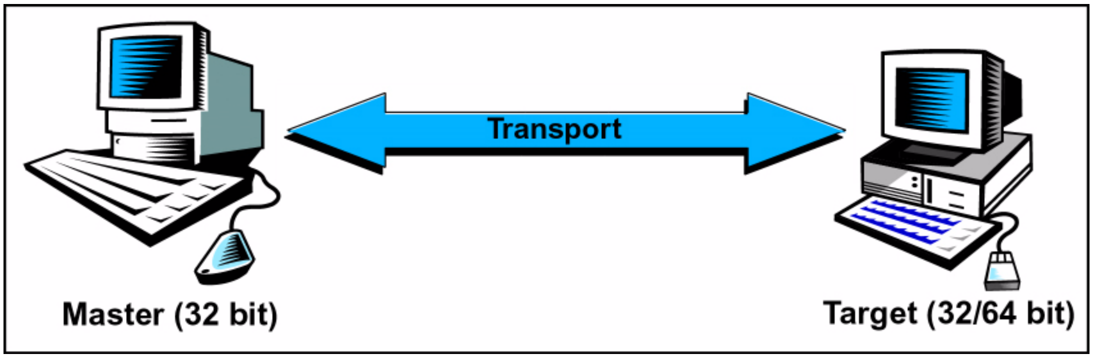


Figure 2-1. Transport Overview

For more information and tips on configuring your transports, refer to the Transport Configuration help.

Serial Target Transport

Standard serial cable into dedicated PCI card serial port. Debugging using the serial interface is quite slow, but probably the easiest to install and configure. Legacy free PC's, however, may not have serial hardware and may require a different transport or an add-in PCI card. If serial is your required transport and you do not have serial hardware built in, you can install the SIIG CyberSerial PCI card.

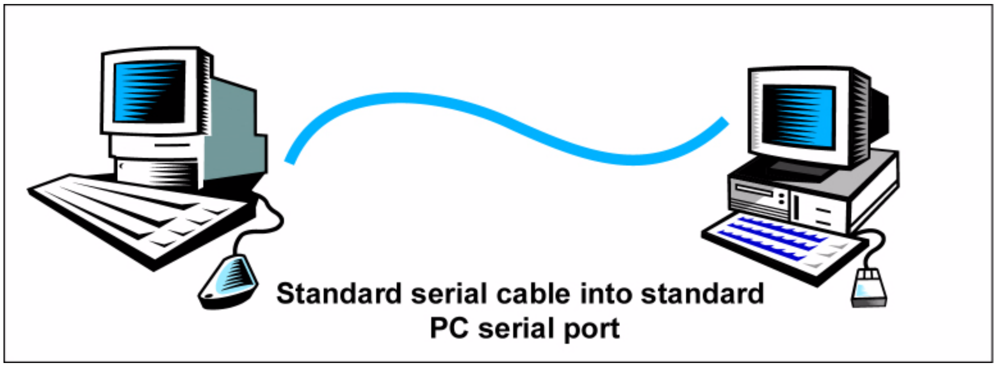


Figure 2-2. Serial Transport

Requirements and Characteristics

Target: UART 8250/16450/16550 or SIIG CyberSerial PCI card

Master: Any serial adapter

Connection Cable: Null modem cable

Speed: ~ 10 KB/sec

Driver: `SISERIAL.SYS`

Dedicated PCI Ethernet Network Interface Card (NIC)

This transport is probably the most flexible choice you have. You can connect the target NIC into your LAN and have access to your machine from anywhere on your network. In addition, you can use a crossover cable to connect the master and target NICs together, thus completely restricting access your target machine. The primary drawback of this debug transport affects you only under the circumstances where you have only one NIC installed in your target machine. After installing our driver for the single NIC, you will not be able to use it for normal windows networking.

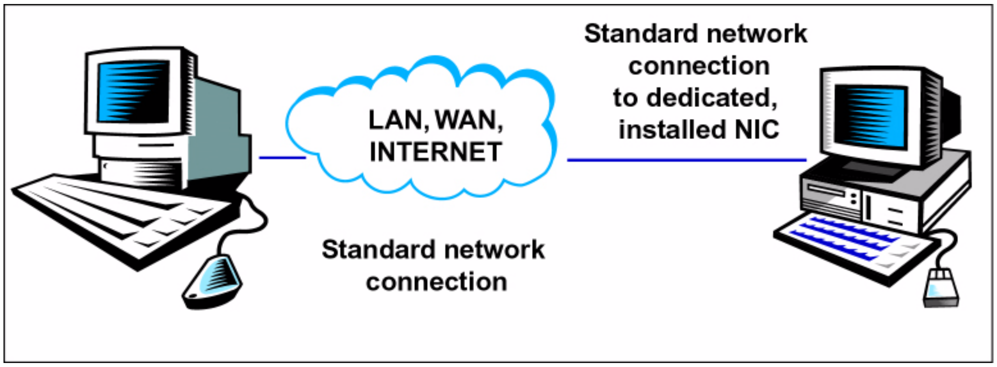

Standard network connection to dedicated, installed NIC
LAN, WAN, INTERNET
Standard network connection

Figure 2-3. Dedicated PCI Ethernet NIC

Requirements and Characteristics

Target: One of the following supported PCI NIC cards:

- ◇ Based on RTL8029 chip
- ◇ Based on RTL8139 chip
- ◇ Intel EtherExpress E100
- ◇ 3Com 3C90X series
- ◇ Based on AMD PCNET chip
- ◇ Based on Lite-On chip

Master: Any Ethernet NIC

Connection Cable: Standard Ethernet cable or Crossover Ethernet cable

Speed: ~ 1-2 MB/sec

Driver: One of the following supported drivers:

- ◇ `SI8029.SYS`
- ◇ `SI8139.SYS`
- ◇ `SIE100.SYS`
- ◇ `SI3C90X.SYS`
- ◇ `SIPCNET.SYS`
- ◇ `SILITE.SYS`

Universal PCI Ethernet NIC

This transport may be right for you if you have a PCI NIC that is not directly supported by our dedicated drivers (refer to the “Dedicated PCI Ethernet Network Interface Card (NIC)” on page 9). The major disadvantage to using this transport configuration is a larger memory footprint on the target than you get with the dedicated NIC drivers.

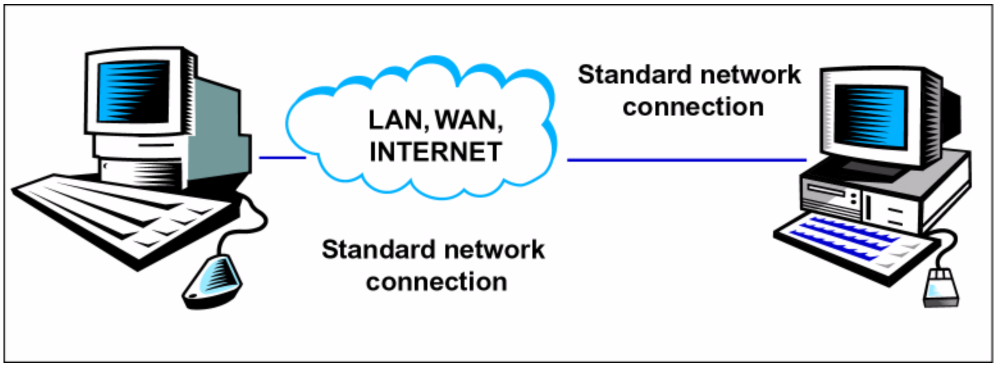


Figure 2-4. Universal PCI Ethernet NIC and Optional Virtual NIC

Requirements and Characteristics

Target: Any PCI Ethernet NIC

Master: Any Ethernet NIC

Connection Cable: Standard Ethernet cable or Crossover Ethernet cable

Speed: ~ 1-2 MB/sec

Drivers: One of the following supported drivers:

- ◇ `SIDN.SYS`
- ◇ `SINIC.SYS`
- ◇ `SIVNIC.SYS` (optional)

OHCI/UHCI USB Host Controller and USB NIC

This transport may be the only one available to you in some situations; for example, if your target is a legacy free laptop. In this case you do not have a serial port or any PCI NICs, thus using an external USB NIC is your only option. The disadvantage of this debug transport solution is that we need to install our own driver, not only for the USB NIC, but also for the USB controller. This means that you cannot plug in any other USB devices under this host controller, even if it has more than one port. A solution to this limitation would be to install an additional USB host controller, for devices like a USB keyboard or mouse, if needed.

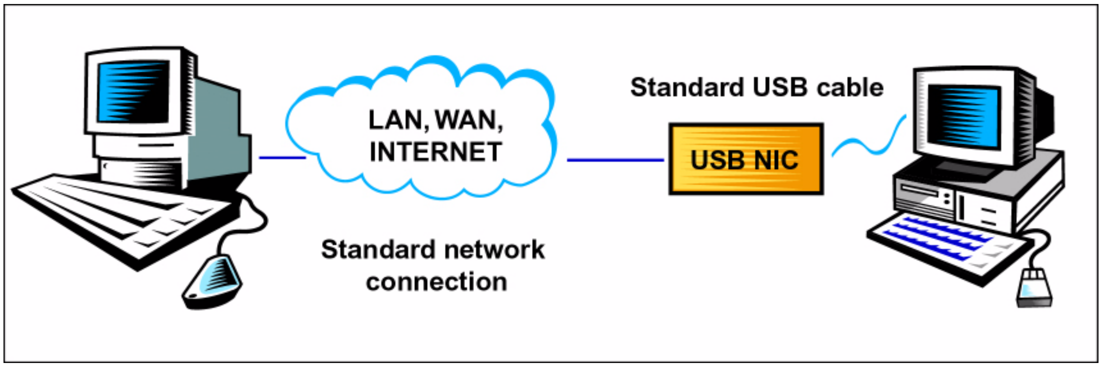


Figure 2-5. OHCI/UHCI USB Host Controller and USB NIC

Requirements and Characteristics

Target: USB 1.1 OHCI\UHCI host controller and Admtek based USB Ethernet NIC

Master: Any Ethernet NIC

Connection Cable: Standard Ethernet cable or Crossover Ethernet cable

Speed: ~ 100-200 KB/sec

Drivers: `SIUSB.SYS`

Virtual NIC Driver (optional)

With the optional Virtual NIC driver, we provide a way to share the target NIC between Visual SoftICE and normal windows networking.

Requirements and Characteristics

Target: Any Supported USB or Universal/Dedicated PCI NIC

Master: Any Ethernet NIC

Connection Cable: See target NIC specifications

Speed: See target NIC specifications

Drivers: `SIVNIC.SYS`

1394 (Firewire)

We use IP over 1394 as a protocol between the master and a target, thus the target machine appears to be on the IP network. This debug transport solution is not shared, and once you install our driver for the 1394 host controller you cannot use it for additional 1394 devices.

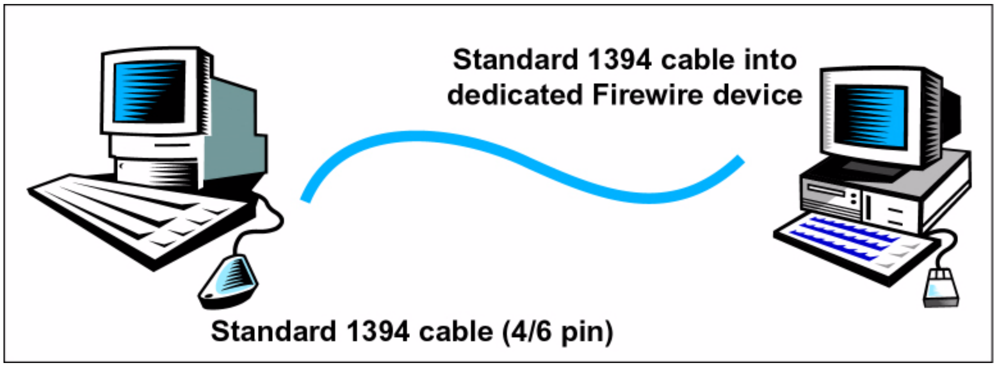

Standard 1394 cable into dedicated Firewire device
Standard 1394 cable (4/6 pin)

Figure 2-6. 1394 (Firewire)

Requirements and Characteristics

Target: OHCI 1394 host controller

Master: Any 1394 host controller and Windows XP or later for OS.

Connection Cable: standard 1394 cable

Speed: ~ 2 MB/sec

Driver: `SI1394.SYS`

Chapter 3

Overview of the Visual SoftICE User Interface

The Visual SoftICE User Interface Overview

Various user interface pages support a number of ease of use features, including:

- The ability to set and follow a main context for all pages, and overriding page-specific contexts for Locals and Stack pages
- The support of standard Cut, Copy and Paste using the clipboard
- The ability to Drag and Drop pages and items (Single element, and element collections between pages)
- The ability to save the contents of the page to a file
- The ability to Print and Print Preview the contents of the page
- The ability to change page modes (snapshot, live, and manual)
- Special Command page features, such as:
  - The ability to use the C++ line comment symbol to append comment text to input (or provide comment lines in scripts)
  - The ability to redirect command output to other pages for display handling (including creation of new pages, on demand)
  - The automatic redirection of appropriate commands from the command page to other existing pages
- 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
- Special icons indicate such things as Target State and Breakpoint Types

Visual SoftICE Icons

Visual SoftICE makes use of many different icons within the GUI to identify such things as breakpoints (type and state), target state, and current context (target IP and UI context). The following sections will help you identify the various icons and their meanings.

Breakpoints

The following icons are used within various pages to indicate the different breakpoint types and states.

Table 3-1. Breakpoint Icons

Icon	Description
🟡 F	This icon represents a Fixed Address breakpoint that has been hit.
🟡 F	This icon represents a Fixed Address breakpoint that has not been hit.
🟢	This icon represents an Image Relative breakpoint that has been hit.
🟢	This icon represents an Image Relative breakpoint that has not been hit.
🔵	This icon represents an I/O breakpoint that has been hit.
🔵	This icon represents an I/O breakpoint that has not been hit.
⚪	This icon represents a disabled breakpoint.

Target State

The following icons are used to indicate the state of the target.

Table 3-2. Target State Icons

Icon	Description
Zzz	This icon indicates the target is in a sleep (power management) mode.
🟩	This icon indicates the target is running.
🟥	This icon indicates the target is stopped.
●	This icon indicates the target is stopped due to hitting a breakpoint.
F	This icon indicates the target is stopped due to a fault.
🟨	This icon indicates the target is instruction or source stepping.
X	This icon indicates the target is stopped due to a BugCheck.
?	This icon indicates the target is in an unknown state.

Current Context

The following icons are used within various pages to identify such things as the target IP and UI context.

Table 3-3. Current Context Icons

Icon	Description
▶	This icon indicates the line you are on when you do a Goto in the Source or Disassembly page.
⏩	This icon indicates the Current UI Context (Process, Thread, or Stack Frame depending on the display it appears in).
⏩	This icon indicates the Current UI Context when a Snapshot of the information was taken.
⇨	This icon indicates the Current Target Instruction Pointer (IP).
⇨	This icon indicates the Current Target IP when a Snapshot of the information was taken.
⇩	This icon indicates the Current Target IP location, and that the next step will move it downward (forward or increasing) into memory.
⇩	This icon indicates the Current Target IP location, and that the next step will move it downward into memory, at the time when a Snapshot of the information was taken.
⇧	This icon indicates the Current Target IP location, and that the next step will move it upward (backward or decreasing) into memory.
⇧	This icon indicates the Current Target IP location, and that the next step will move it upward into memory, at the time when a Snapshot of the information was taken.
☞	This icon indicates the line where the Current IP will be upon executing the next instruction. Displayed when the Current IP and the next IP locations are available in the same screen (Source or Disassembly pages).

Symbol Table State

The following icons are used to indicate the state of the symbol table.

Table 3-4. Symbol State Icons

Icon	Description
🔍	This icon indicates that on-demand symbol loading is enabled (See SET SYMTABLEAUTOLOAD in the Visual SoftICE Command Reference).
🚫	This icon indicates that on-demand symbol loading is disabled (See SET SYMTABLEAUTOLOAD in the Visual SoftICE Command Reference).
🗄	This icon indicates that searching for symbols via symbol servers is enabled (See SET SYMSRVSEARCH in the Visual SoftICE Command Reference).
🚫	This icon indicates that searching for symbols via symbol servers is disabled (See SET SYMSRVSEARCH in the Visual SoftICE Command Reference).
✅	This icon indicates that all tables loaded without warnings (See TABLE in the Visual SoftICE Command Reference).
⚠	This icon indicates that at least one table loaded with a warning condition (See TABLE in the Visual SoftICE Command Reference).
❗	This icon indicates that at least one table was not loaded (See TABLE in the Visual SoftICE Command Reference).

Source Page Symbol State Icons

The following icons are used to indicate the state of the symbols for a source file loaded into a source page. They only have the following meanings when in the status bar of a source page.

Table 3-5. Source Page Symbol State Icons

Icon	Description
✅	This icon indicates that symbols are loaded for the indicated image, and that the source file is part of that image and has line information available.
⚠	This icon indicates that symbols are loaded for the indicated image, and that the source file is part of that image, but there is no line information available.
❗	This icon indicates that the source file is not part of any image in the current/active process.

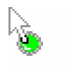
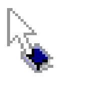
Cursor Types

The following icons are used to indicate various states of the cursor.

Table 3-6. Cursor Icons

Icon	Description
	This icon indicates you have selected a single item and are dragging it from a page.
	This icon indicates you have selected multiple items and are dragging them from a page.
⊘	This icon indicates the cursor is over a window that does not allow dropping of the data you are dragging.
	This icon indicates the cursor is over a window that allows dropping the single item being dragged into it.
	This icon indicates the cursor is over a window that allows dropping the multiple items being dragged into it.

Table 3-6. Cursor Icons (Continued)

Icon	Description
	This icon indicates the cursor is over the breakpoint column, on the Source or Disassembly page.
	This icon indicates the cursor is over the bookmark column.

About the Status Bar

Visual SoftICE provides main status bar fields, an optional one of which displays the status of the symbol engine. The status bar provides the following information:

- Whether on-demand loading of symbols is enabled or disabled.
- Whether using the symbol server is enabled or disabled.
- Whether any symbol table failed to load, or loaded with a warning, or all symbol tables loaded successfully.
- The name of the current default symbol table

The Source page also has a page-specific symbol status bar.

Status Bar Examples

The following examples show various states of the symbol engine, as indicated by the status bar:

Table 3-7. Visual SoftICE Status Bar Examples

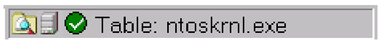
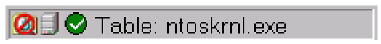
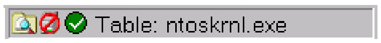
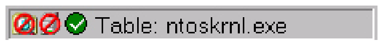
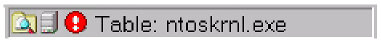
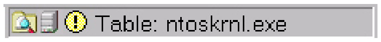
Status Bar	Meaning
 Table: ntoskrnl.exe	Auto-load is enabled, the symbol server is enabled, and all tables loaded successfully.
 Table: ntoskrnl.exe	Auto-load is disabled, the symbol server is enabled, and all tables loaded successfully.
 Table: ntoskrnl.exe	Auto-load is enabled, the symbol server is disabled, and all tables loaded successfully.
 Table: ntoskrnl.exe	Auto-load is disabled, the symbol server is disabled, and all tables loaded successfully.
 Table: ntoskrnl.exe	Auto-load is enabled, the symbol server is enabled, and at least one table failed to load.

Table 3-7. Visual SoftICE Status Bar Examples (Continued)

Status Bar	Meaning
Table: ntoskrnl.exe	Auto-load is enabled, the symbol server is enabled, and at least one table loaded with a condition.

Source Page Status Bar

The source page has its own status bar that uses icons to display the state of symbols for the source file loaded into that page. Icons indicating successful loading of symbols are followed by the image name.

About the Context Bar

Visual SoftICE provides a main context bar which displays the current context the debugger is following, and allows you to change the current context using its drop-down list. The Locals page and Stack page also each have page-specific context bars available. The page-specific context overrides the main context. If you activate a page-specific context bar, you change the context that page is following. Both the main context bar and any page-specific context bars are comprised of three controls and a button.

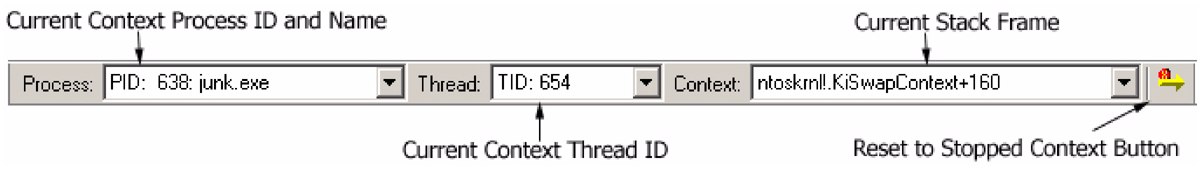

Current Context Process ID and Name
Current Stack Frame
Process: PID: 638: junk.exe
Thread: TID: 654
Context: ntoskrnl!.KiSwapContext+160
Current Context Thread ID
Reset to Stopped Context Button

Figure 3-1. Visual SoftICE Context Bar

Context Bar Controls

The main context bar is comprised of controls allowing you to track the current context right down to the stack frame level. The page-specific context bars are comprised of a sub-set of two of the controls, depending on which page the context bar belongs to. The controls and their functions are as follows:

Table 3-8. Context Bar Controls

Control	Description
Process	Displays the **Process ID** and **Name**. You can use this control to select a new process from the drop-down list of available processes. This is equivalent to changing the current process from the Process List page or via the ADDR command.
Thread	Displays the **Thread ID**. You can use this control to select a new thread from the drop-down list of available threads, if the current process is multi-threaded. This control is the most convenient way to change threads within a multi-threaded process.
Context	Displays the current **Stack Frame**, including any symbolic representation of the frame instruction pointer. You can use this control to select a new stack frame from the drop-down list of available stack frames. This control is the most convenient way to change the current stack frame.
Reset to Stopped Context Button	Switches you back to the context that was current when you stopped the target. This button is located on the main context bar, and becomes active when you stop the target, and switch contexts. Clicking the button does not restart the target.

Page Modes

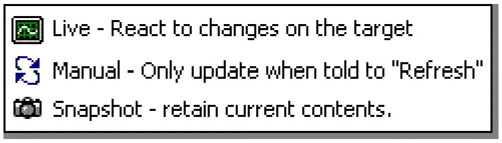

Live - React to changes on the target
Manual - Only update when told to "Refresh"
Snapshot - retain current contents.

Figure 3-2. Page Mode Icons

Each page of the user interface may support one or more of the following 3 modes:

Table 3-9. Page Mode Descriptions

Mode	Description
Live	Events from the target cause the page to refresh automatically.
Manual	You decide when the page should be refreshed (menu(s) > Refresh).
SnapShot	The page is protecting its current contents from over-write. You must elect to destroy the data. Changes to the data, and actions against the target, are not allowed in this mode.

Some pages do not support all the modes, as is appropriate for their function, (for example the Command Page only supports Live mode).

Automatic Changes in Mode State

Each page that supports SnapShot mode will automatically switch to this mode when a target disconnection is detected. This is done to indicate that the connection to a target is gone, and that changes to the data contained in a page are no longer valid, or supported.

Live Mode On Connection

Many pages provide a feature called “Live Mode On Connection” (in the preferences > settings dialog). This feature controls how a pre-existing page behaves when a connection to a target is established. If this feature is active, the page will switch to Live Mode when the connection is detected, and any contents it previously held will be abandoned.This feature is on by default, for pages that support it.

Note: If you wish the interface to always preserve the contents of a page, turn this feature off. Doing so will ensure that when the page changes mode to SnapShot, no automatic event will cause that data to be lost. You can always manually change the state back to Manual or Live, and you will get a confirmation dialog box about abandoning the page’s data, if it has any.

User Interface Preferences

You can set global, per-workspace, toolbar, status bar, font, color, and keyboard preferences using the Preferences dialog. Preferences are set on a global basis for all of DriverStudio and its components, as well as on a per-workspace basis for specific components. This dialog provides the following groups of preferences you can set.

- Global Settings
- Per-Workspace Settings
- Keyboard
- Toolbars & Status Bar
- Fonts & Colors

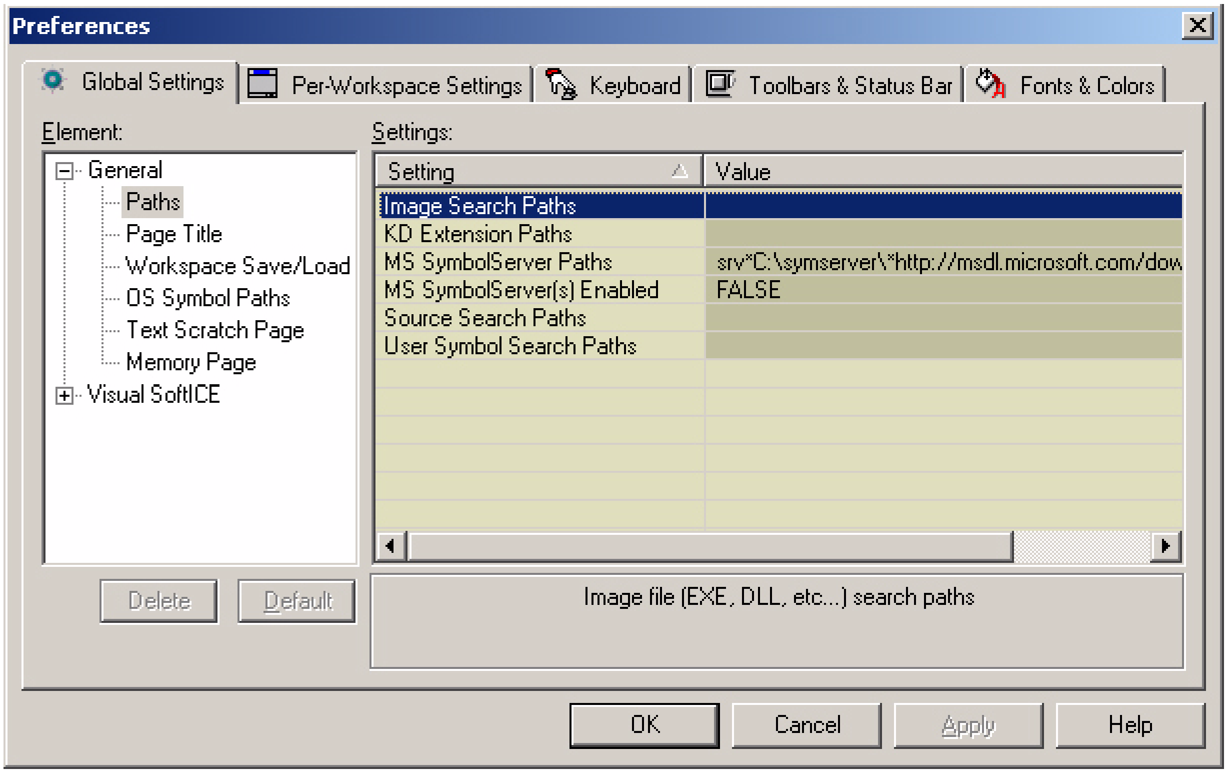
The Per-Workspace settings are provided for users who wish to use different workspaces for different debugging conditions, targets, or target types. Available per-workspace settings fall into two categories: Paths and Scripts. The Per-Workspace paths values prepend or replace the global path values of the same name.

Global Settings

Global Settings are global to the executable, and do not change with different workspaces.

General 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.



Setting	Value
Image Search Paths	
KD Extension Paths	
MS SymbolServer Paths	srv*C:\symserver\*http://msdl.microsoft.com/dow
MS SymbolServer(s) Enabled	FALSE
Source Search Paths	
User Symbol Search Paths	

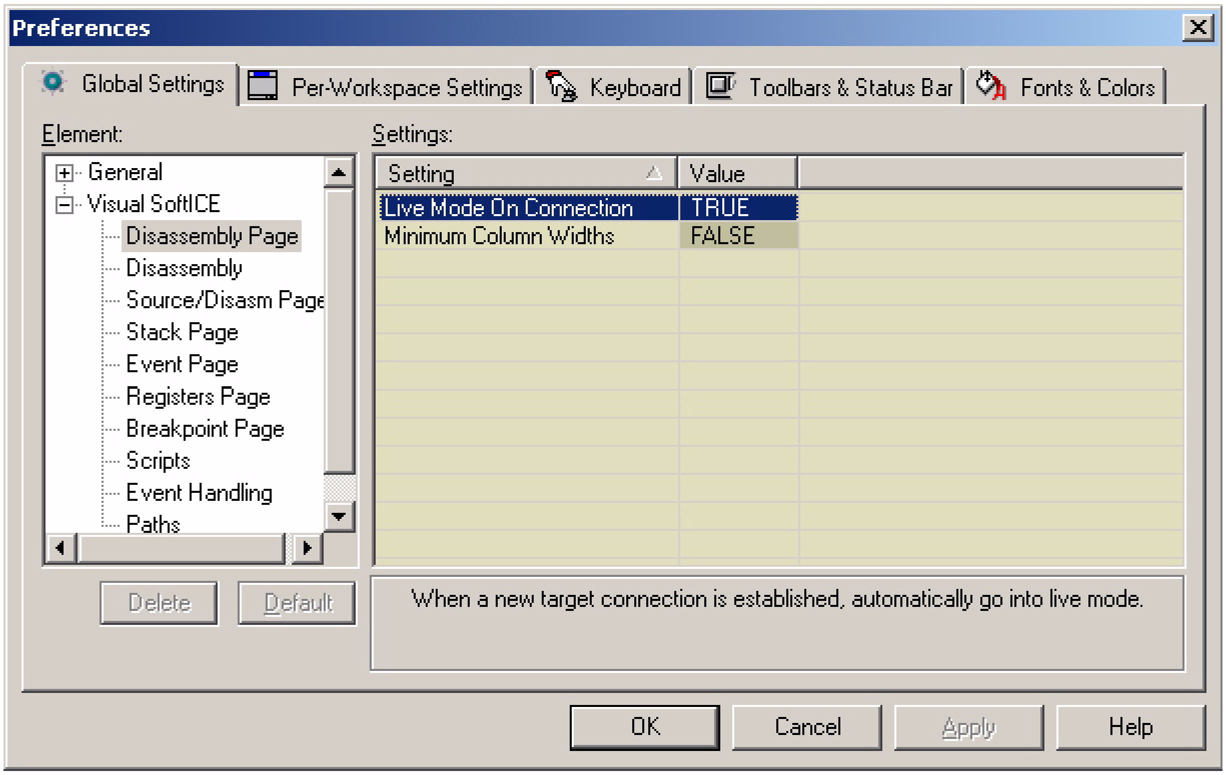
Image file (EXE, DLL, etc...) search paths

Figure 3-3. General Global Settings

Visual SoftICE Global Settings

Visual SoftICE global settings act on the Visual SoftICE component 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.



Setting	Value
Live Mode On Connection	TRUE
Minimum Column Widths	FALSE

When a new target connection is established, automatically go into live mode.

Figure 3-4. Visual SoftICE Global Settings

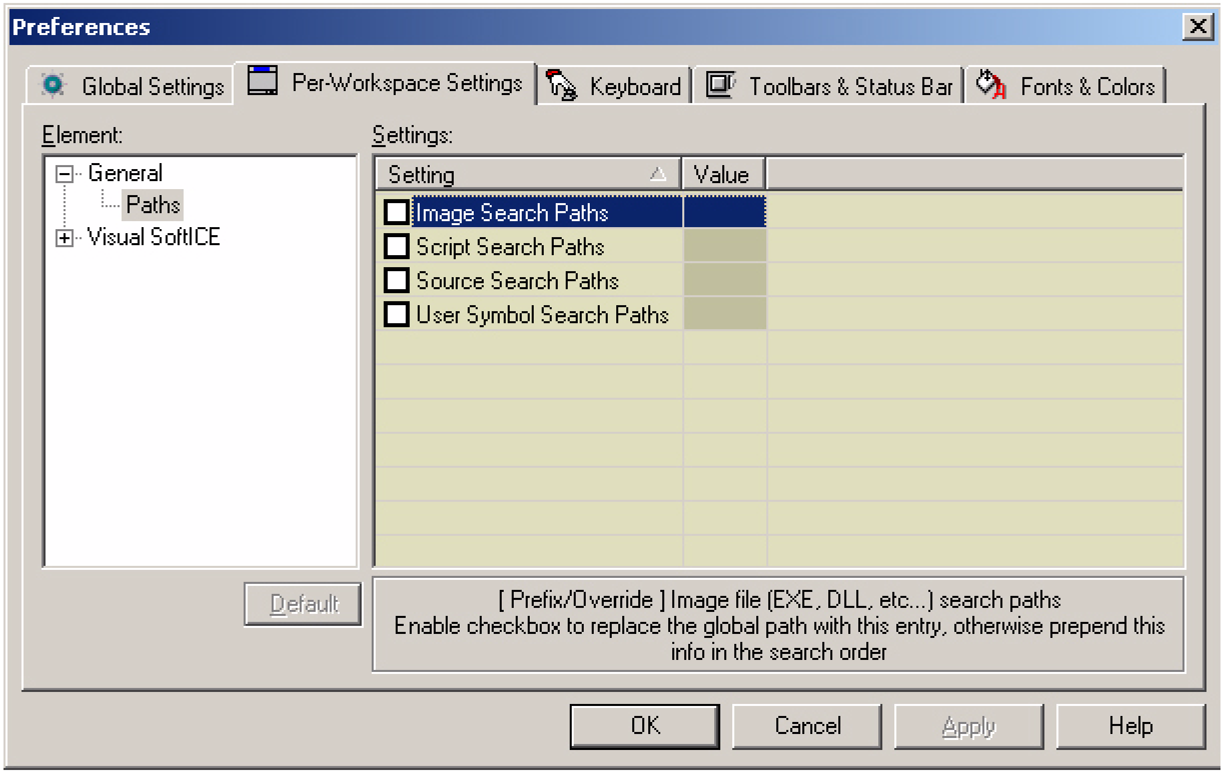
Per-Workspace Settings

Per-Workspace settings are provided for users who wish to use different workspaces for different debugging conditions, targets, or target types. The properties you set here are specific to the current workspace. Available per-workspace settings fall into two categories: Paths and Scripts.

Paths

The Paths element in the Per-Workspace Settings tab allows you to define specific paths to prepend or override the global path settings. These paths, along with the global settings they prepend or replace, define the Visual SoftICE configuration. The current configuration can be seen by issuing the SET command.

If you click on a path type, a description of that path, any ranges (if applicable), and other rules on data entry (if applicable), are displayed in the Setting Explanation window below the list.



Setting	Value
☐ Image Search Paths	
☐ Script Search Paths	
☐ Source Search Paths	
☐ User Symbol Search Paths	

[Prefix/Override] Image file (EXE, DLL, etc...) search paths
Enable checkbox to replace the global path with this entry, otherwise prepend this info in the search order

Figure 3-5. General Per-Workspace Path Preferences

To edit the value of a path, click on its **Value** field. Visual SoftICE opens the **Path List Edit** utility.

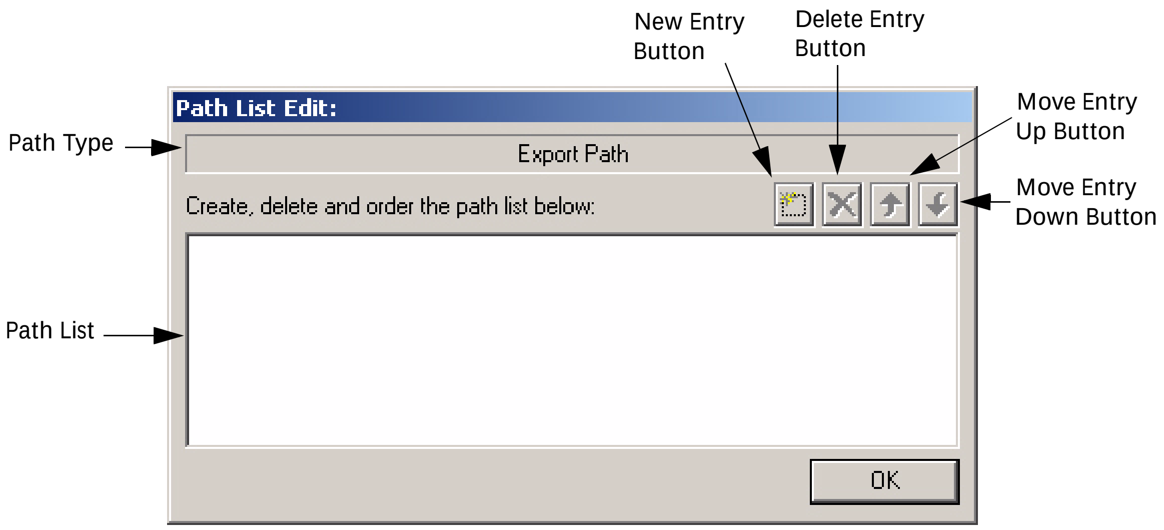
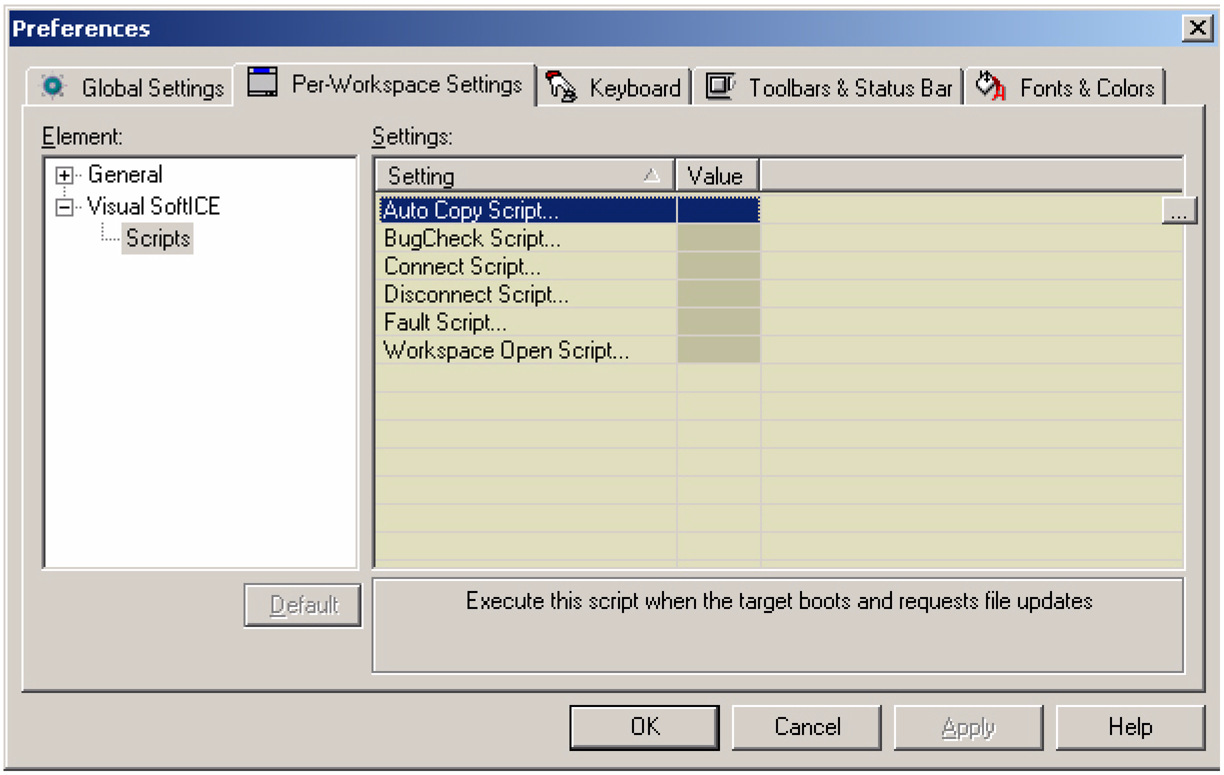


Figure 3-6. Path List Edit Utility

Using the Path List Edit utility, you can add new path definitions, delete obsolete path definitions, and move list items up and down to change the order in which they are searched.

Scripts

The Scripts element in the Per-Workspace Settings tab allows you to associate scripts with certain events in Visual SoftICE. If you click on a script type, a description of that event association and any other rules on data entry (if applicable) are displayed in the Setting Explanation window below the list.



Setting	Value
Auto Copy Script...	
BugCheck Script...	
Connect Script...	
Disconnect Script...	
Fault Script...	
Workspace Open Script...	

Execute this script when the target boots and requests file updates

Figure 3-7. Visual SoftICE Per-Workspace Script Preferences

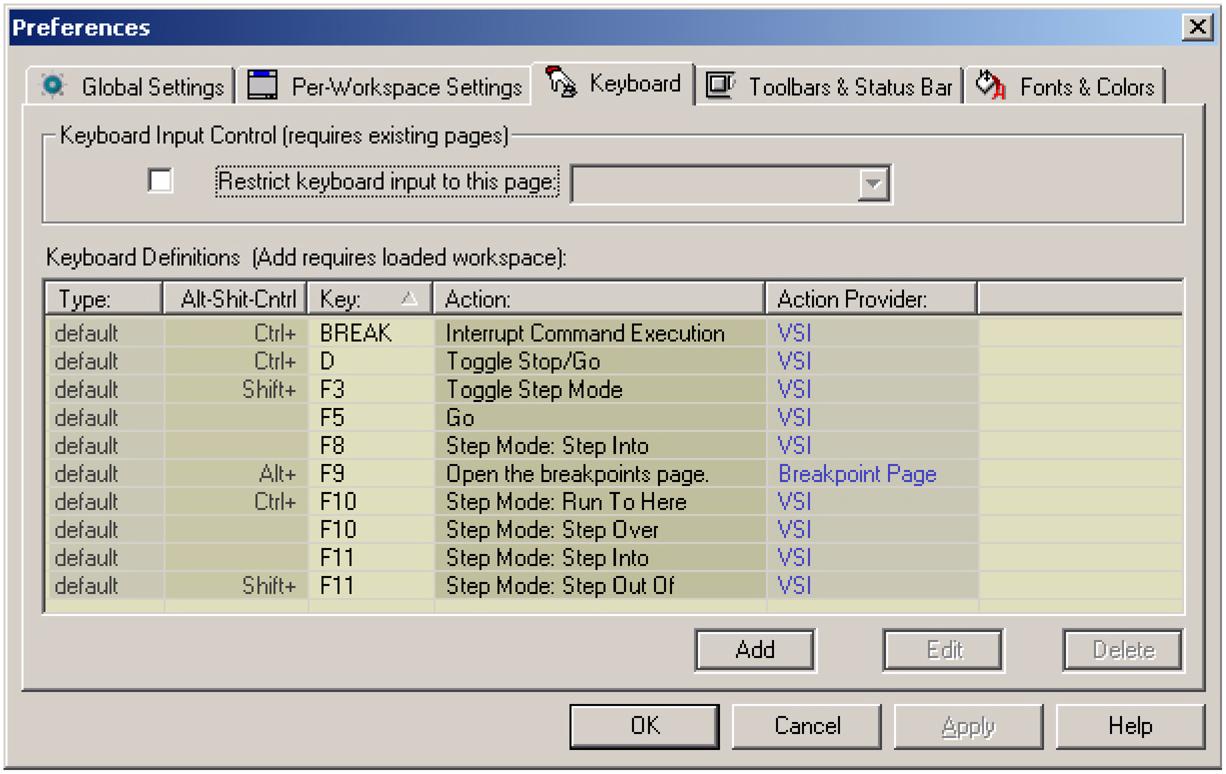
To edit the value of a path, click on its **Value** field. Visual SoftICE opens the value field for editing, allowing you to enter the script file location and name. It also provides a browse button in the right margin, which you can use to browse your file system and select the script file. For more information on Script File Execution, refer to “Script Execution” on page 43.

Keyboard Settings

DriverStudio allows you to create custom keyboard definitions, or edit existing keyboard definitions, and save them in your workspace file. DriverStudio components that work within the DriverWorkbench environment provide default keyboard definitions on a workspace level and sometimes on a page component level. You cannot edit default definitions; however, you can override them by adding your own custom keyboard definitions for the same keystroke. Table 3-10 on page 31 provides the definitions that are global to the DriverWorkbench environment, which you cannot override.

Table 3-10. Global Keyboard Definitions

Keystroke	
CTRL-TAB	Next page on a pad.
SHIFT-CTRL-TAB	Previous page on a pad.
CTRL-F4	Close the current pad.



Keyboard Input Control (requires existing pages)

☐ Restrict keyboard input to this page:

Keyboard Definitions (Add requires loaded workspace):

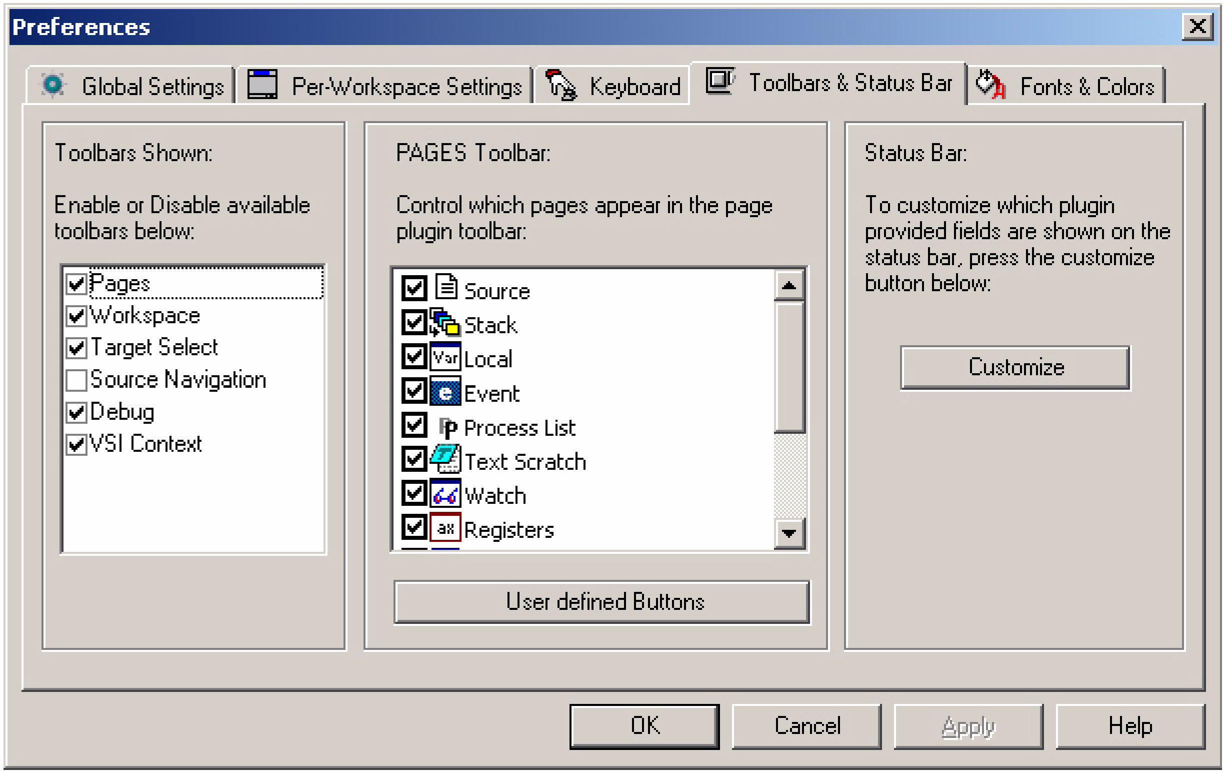
Type:	Alt-Shit-Cntrl	Key:	Action:	Action Provider:
default	Ctrl+	BREAK	Interrupt Command Execution	VSI
default	Ctrl+	D	Toggle Stop/Go	VSI
default	Shift+	F3	Toggle Step Mode	VSI
default		F5	Go	VSI
default		F8	Step Mode: Step Into	VSI
default	Alt+	F9	Open the breakpoints page.	Breakpoint Page
default	Ctrl+	F10	Step Mode: Run To Here	VSI
default		F10	Step Mode: Step Over	VSI
default		F11	Step Mode: Step Into	VSI
default	Shift+	F11	Step Mode: Step Out Of	VSI

Figure 3-8. Keyboard Preferences Tab

Keyboard definitions are global, and will function regardless of what page or control has focus.

Toolbars & Status Bar Settings

DriverStudio has specific toolbars and page-access icons that appear in the master toolbar, and a Status Bar that is displayed in the master frame. You can customize the toolbars and the status bar using this tab.



Toolbars Shown:

Enable or Disable available toolbars below:

- [✓] Pages
- [✓] Workspace
- [✓] Target Select
- [ ] Source Navigation
- [✓] Debug
- [✓] VSI Context

PAGES Toolbar:

Control which pages appear in the page plugin toolbar:

- [✓] Source
- [✓] Stack
- [✓] Local
- [✓] Event
- [✓] Process List
- [✓] Text Scratch
- [✓] Watch
- [✓] Registers

User defined Buttons

Status Bar:

To customize which plugin provided fields are shown on the status bar, press the customize button below:

Customize

OK Cancel Apply Help

Figure 3-9. Toolbars & Status Bar Preferences Tab

Customizing Toolbars

You can configure which of the toolbars or page-access icons you want to display, and you can add further options to the user-defined area of the Pages toolbar. The Toolbars & Status Bar tab is composed of a left-side Toolbars Shown column displaying a list of available toolbar types, a central Pages Toolbar column displaying a list of available page icons, and a right-side column for customizing the status bar. At the bottom of the central column is a User Defined Buttons button that opens the Customize Page Plugin Toolbar utility.

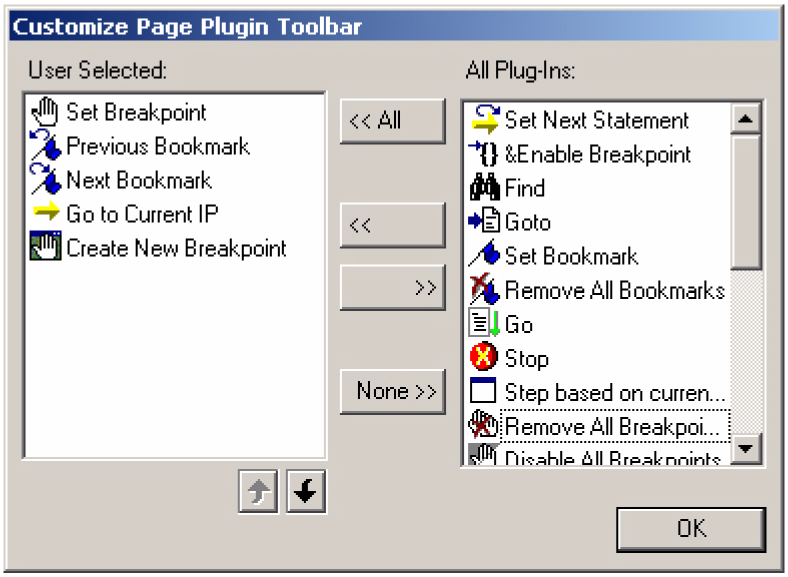
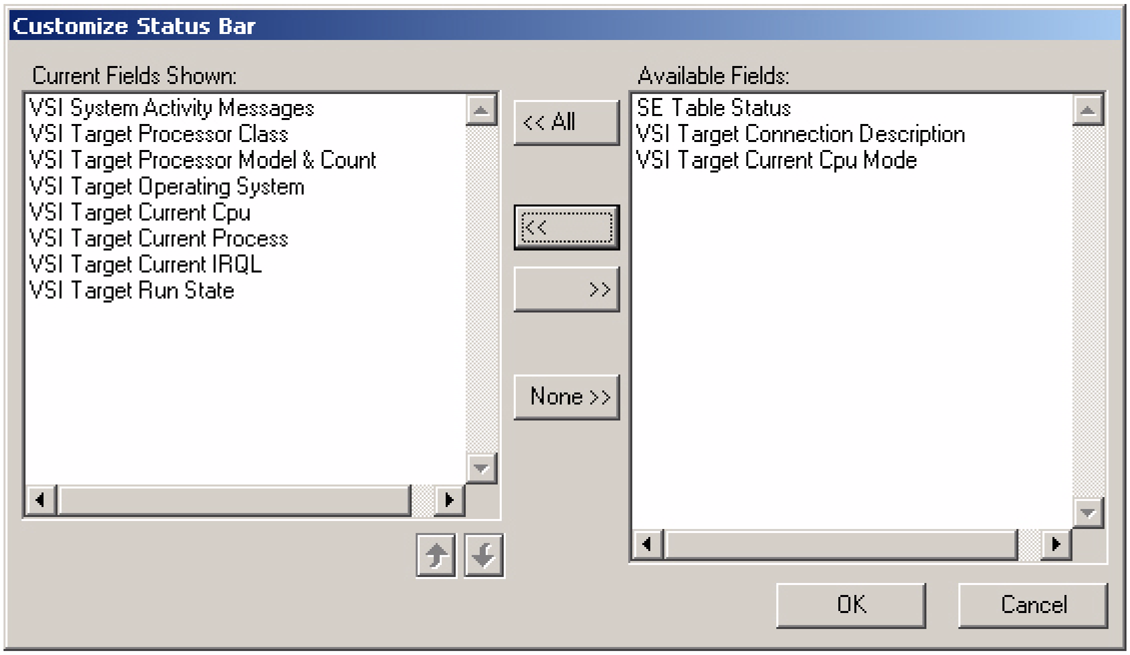


Figure 3-10. Customize Page Plugin Toolbar Utility

Customizing the Status Bar

DriverStudio allows you to customize the information displayed on the status bar. To access the Customize Status Bar utility, click the **Customize** button in the right-hand Status Bar area. To customize the information displayed by the status bar, select or deselect field types by moving them into the appropriate column. You can arrange the fields in the order you would like them to appear in the status bar by using the **Up** and **Down** buttons for the Current Field Shown column.



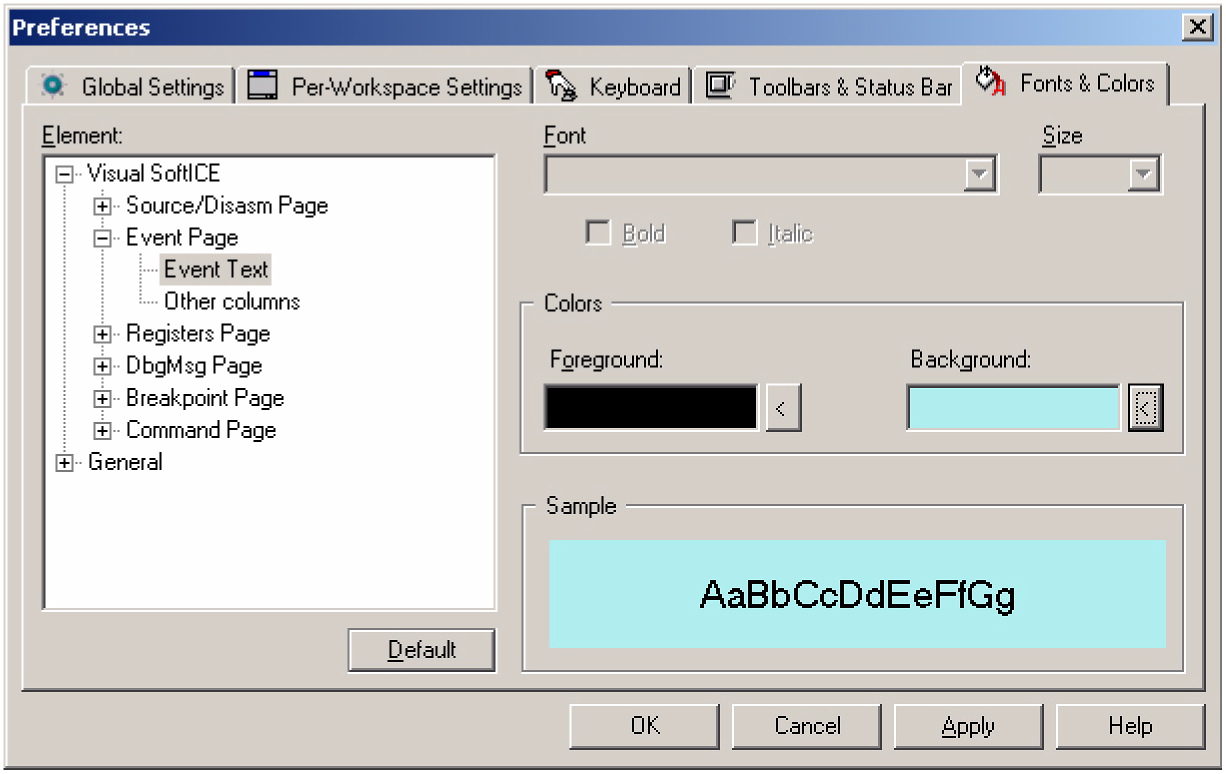
Current Fields Shown:
VSI System Activity Messages
VSI Target Processor Class
VSI Target Processor Model & Count
VSI Target Operating System
VSI Target Current Cpu
VSI Target Current Process
VSI Target Current IRQL
VSI Target Run State

Available Fields:
SE Table Status
VSI Target Connection Description
VSI Target Current Cpu Mode

Figure 3-11. Customize Status Bar utility

Fonts & Colors Settings

Many pages allows you to configure the properties of text that appears in them. DriverStudio provides a Fonts & Colors tab in the Preferences dialog where you can configure various aspects of different pages. The Fonts & Colors tab is composed of a left-side Element column displaying a list of available pages and their settings, and a series of controls (font, size, color, and emphasis). Each page determines the settings available for it, and governs what you can do with each setting.



Element:

- Visual SoftICE
  - Source/Disasm Page
  - Event Page
    - Event Text
    - Other columns
  - Registers Page
  - DbgMsg Page
  - Breakpoint Page
  - Command Page
- General

Font

Size

Bold

Italic

Colors

Foreground:

Background:

Sample

AaBbCcDdEeFfGg

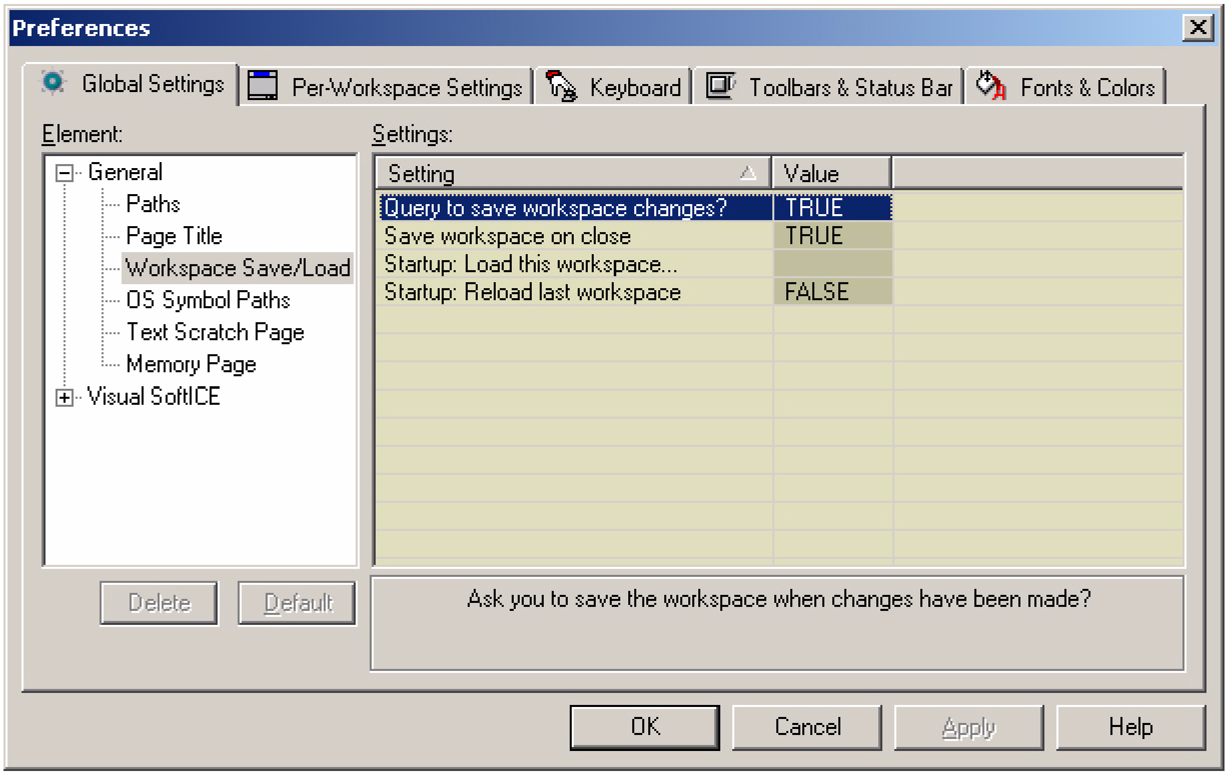
Figure 3-12. Fonts & Colors Preferences Tab

Other User Interface Attributes and Features

The following sections describe other attributes and features of the Visual SoftICE user interface, and how to use them to customize your workspace.

Workspace Save and Load

In addition to loading and saving custom workspaces manually, you can configure several automatic workspace loading and saving behaviors in DriverStudio. To configure workspace behavior, access the **Workspace Save/Load** element on the Global Settings tab of the Preferences dialog.



Setting	Value
Query to save workspace changes?	TRUE
Save workspace on close	TRUE
Startup: Load this workspace...	
Startup: Reload last workspace	FALSE

Ask you to save the workspace when changes have been made?

Figure 3-13. Workspace Save/Load Preferences

You can configure the following workspace properties:

Table 3-11. Workspace Save/Load Properties

Setting	Description
Query to save workspace changes?	This option causes DriverStudio 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 DriverStudio.
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 DriverStudio 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.

Special Command Page Features

Command Comments

The input dialects support a line comment style (as opposed to block style comments). Each dialect can have its own syntax; but, to date, all supported command dialects use the C++ form:

```
// This entire line is a comment – useful in scripts
proc // Command displays all currently running processes
```

Command Syntax Output Redirection

You can redirect the output of a given command to another page by using the dialect's redirection syntax. To date, all the supported command dialects use the following form:

```
Cmd-text /> page-type
```

- The Cmd-text is any normal command that is recognized in the command dialect (e.g. “proc”).
- The redirection syntax “/>” indicates that the remainder of the input stream is a target page type. This form has been chosen so as not to conflict with operands that expression evaluator acts on.
- The page-type can be either of the following:
    - **The name of an existing page**. If the page does not exist, the redirection will fail. If the page exists, it is passed the output for display within its domain. If it cannot handle the data, that page will ignore it, and nothing is displayed.
    - **The “type name” of a known page plugin**. This is a short mnemonic of the page type, such as CMD for the command page, or DBG for the debug output page, (refer to Table 3-12). In this case, a page of “type-name” will be created, and the output of the command passed to it. This page will be created in the current PAD, and will get the text of the command in the Cmd-text as its title.

Table 3-12. Page Plugin Type Names

Page	Name
Command Page	CMD
Debug Message Page	DBG
Event Page	EVT
Text Scratch Page	TXT
Locals Page	LOC
Memory (Data) Page	MEM
Process List Page	PROC
Register Page	REG
Disassembly Page	DISASM
Source Page	SRC
Stack Page	STK
Watch Page	WAT

Examples

The following example sends the output of the proc command to a text page named text1:

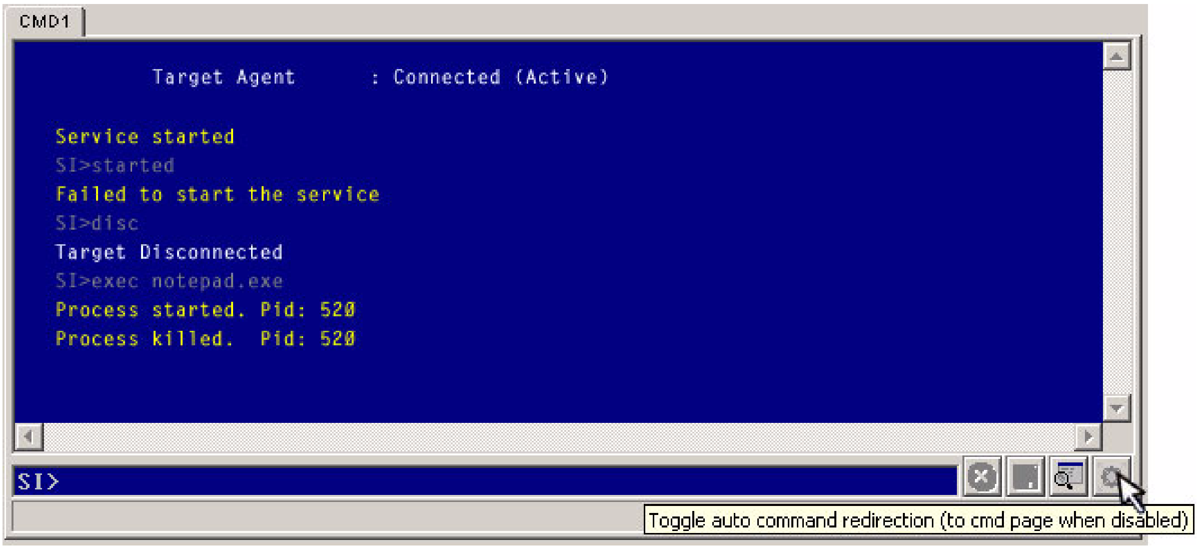
```
SI>proc /> text1
```

The following example sends the output of the proc command to a new text page:

```
SI>proc /> TXT
```

Automatic Output Redirection

The command page supports a mode whereby commands can have output results automatically handled by an appropriate page other than the command page itself. Toggling the “Auto Command Redirection” button on the command page itself enables this.



```
          Target Agent      : Connected (Active)

  Service started
  SI>started
  Failed to start the service
  SI>disc
  Target Disconnected
  SI>exec notepad.exe
  Process started. Pid: 520
  Process killed.  Pid: 520
```

Figure 3-14. Auto Command Redirection Button

When active, after a command is successfully processed, the system looks for registered page handlers for the command. If found, the command output is automatically passed to the first instance of a page found that handles the command.

For example, if your workspace contains a command page and a registers page, you have auto command redirection enabled, and you type the following:

```
r general
```

The register page will change to reflect the current contents of the general register group. If the auto command redirection switch had not been enabled, the output of the command would have appeared within the command page.

Cut, Copy, and Paste

All the controls in all the pages that allow highlight selection support the standard copy operation.

Many input controls that support highlight selection support the standard cut operation.

Most, if not all, input controls that take text support the standard paste operation.

All the user interface controls use the operating systems clipboard buffer.

Drag and Drop

There are 2 types of drag and drop supported in the user interface:

- Pages between and within pads
- Data items from one page to another

Pad Level Page Drag and Drop

All Pads allow any number of pages to be placed within them.

You may rearrange the order of the pages within a pad, by simply dragging and dropping the page to a new place in the horizontal order. You may also drag a page out of one pad and into another pad.

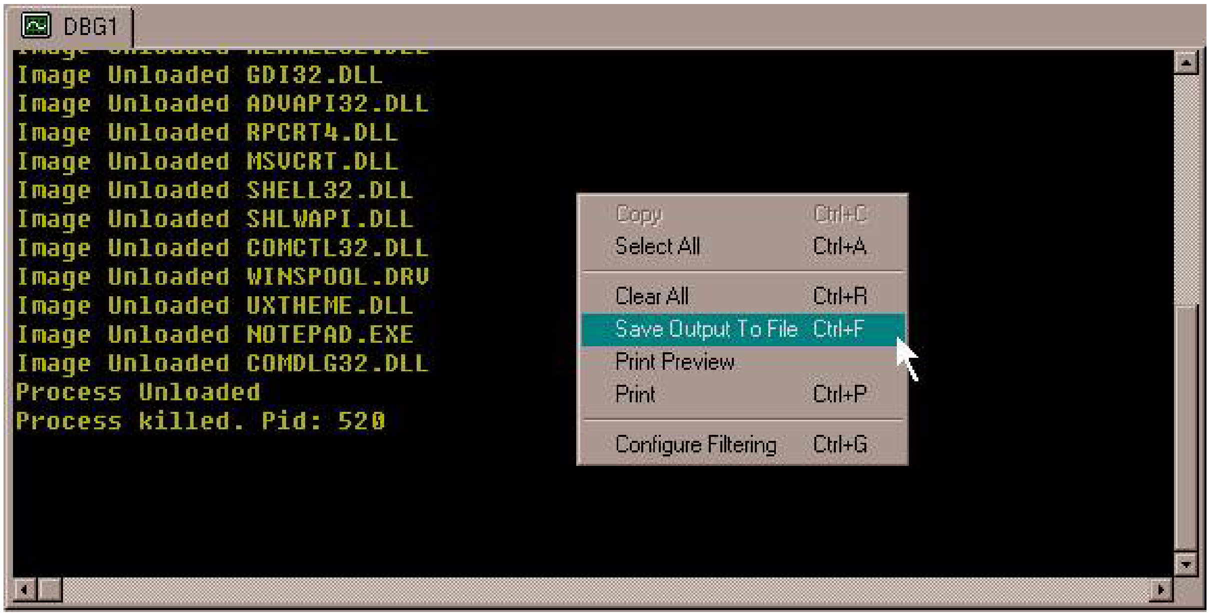
If you press the Esc key while dragging a page, the action is cancelled immediately. If you press and hold the Ctrl key while dragging a page, the action is temporarily cancelled: Releasing the mouse button cancels the drag and drop, whereas releasing the Ctrl key continues the drag and drop.

Data Item Drag and Drop

Many pages of the user interface support dragging a selection from that page to another page's input area. There are 2 different cases: dragging a single element from one page to another, and dragging multiple elements (a collection) between pages. For a table that correlates the graphical representation of each cursor type with an explanation of its meaning, refer to “Visual SoftICE Icons” on page 16.

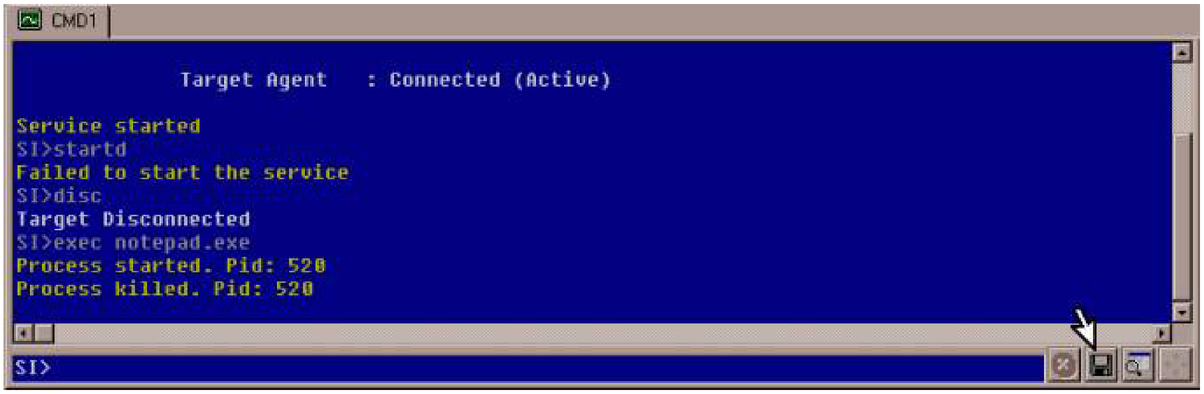
Saving Contents to a File

Many pages of the user interface support saving their contents to file. Saving is normally available via right-clicking on a page and selecting **Save Output To File** from the pop-up menu, or by clicking a button within the page itself (as shown below):



```
Image Unloaded GDI32.DLL
Image Unloaded ADVAPI32.DLL
Image Unloaded RPCRT4.DLL
Image Unloaded MSVCRT.DLL
Image Unloaded SHELL32.DLL
Image Unloaded SHLWAPI.DLL
Image Unloaded COMCTL32.DLL
Image Unloaded WINSPOOL.DRV
Image Unloaded UXTHEME.DLL
Image Unloaded NOTEPAD.EXE
Image Unloaded COMDLG32.DLL
Process Unloaded
Process killed. Pid: 520
```

Figure 3-15. Saving Output Via the Pop-up Menu



```
CMD1

              Target Agent    : Connected (Active)

Service started
SI>startd
Failed to start the service
SI>disc
Target Disconnected
SI>exec notepad.exe
Process started. Pid: 520
Process killed. Pid: 520

SI>
```

Figure 3-16. Saving Output Via the Save Button

Print and Print Preview

Most of the pages of the user interface support printing their contents. This is normally available from the main **File** menu.

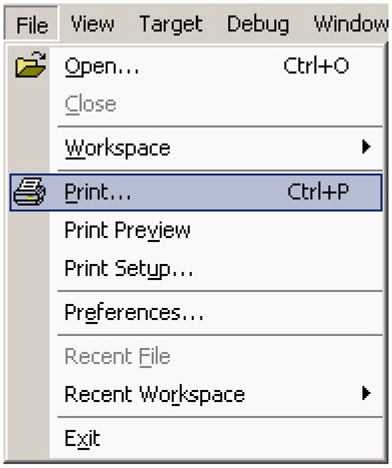


Figure 3-17. Printing from the File Menu

Printing may also be available via right-clicking on a page and selecting **Print** from the pop-up menu.

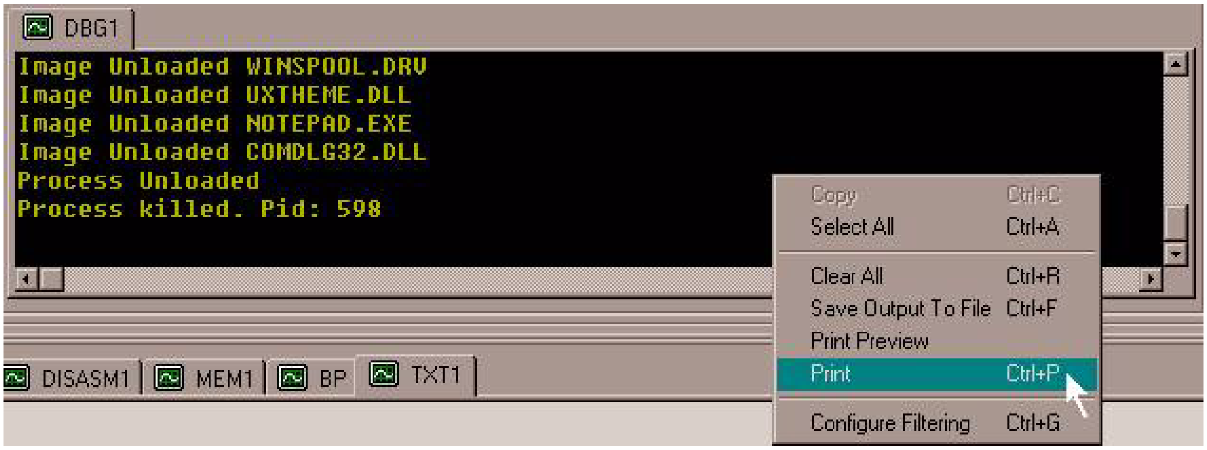

```
Image Unloaded WINSPOOL.DRV
Image Unloaded UXTHEME.DLL
Image Unloaded NOTEPAD.EXE
Image Unloaded COMDLG32.DLL
Process Unloaded
Process killed. Pid: 598
```

Figure 3-18. Printing from the Pop-up Menu

Script Execution

You can configure Visual SoftICE to use scripts you have written upon the triggering of certain events, such as opening a workspace or connecting to a target. You configure scripts via the **Scripts** item in the **Per-Workspace Settings** tab under **Preferences**.

To configure Visual SoftICE to use a script, you need to define the path and script file name in the field corresponding to the trigger event for the script, and activate the event by checking its check-box. You can define scripts for the trigger events listed in Table 3-13.

Table 3-13. Script Trigger Events

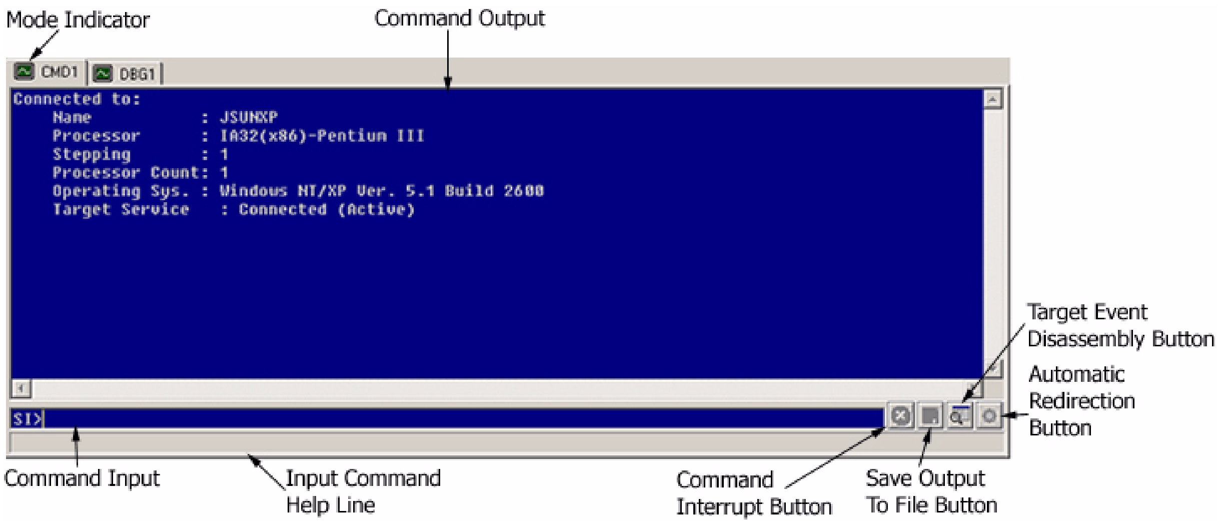
Script	Trigger Event
Auto Copy Script	Execute a script when the target boots and requests file updates.
BugCheck Script	Execute a script upon encountering a BugCheck event (blue screen).
Connect Script	Execute a script upon connecting to a target.
Disconnect Script	Execute a script upon disconnecting from a target.
Fault Script	Execute a script upon encountering a Fault.
Workspace Open Script	Execute a script upon opening a workspace.

Note: Visual SoftICE does not save actual scripts, but saves the paths to the scripts, so you must make sure the scripts exist in the defined location for them to succeed. If you move your workspace to another machine, be certain you also move the script files referenced by the workspace.

Chapter 4
The Visual SoftICE User Interface Pages

The Command Page

The Command page is used to execute Visual SoftICE and KD commands. It is composed of an input command line, an output window, and several control buttons. There is no limit to the number of lines that can be retained in the output pane; the color attributes of all lines, even those no longer visible on screen, are preserved. The Command page also retains its own copy of the status bar when undocked.



```
Connected to:
    Name          : JSUNXP
    Processor     : IA32(x86)-Pentium III
    Stepping      : 1
    Processor Count: 1
    Operating Sys. : Windows NT/XP Ver. 5.1 Build 2600
    Target Service   : Connected (Active)
```

Figure 4-1. Visual SoftICE Command Page

Concepts and Associated Commands

Redirecting Output

The command page supports command redirection of its output either automatically or explicitly to a particular page.

- **Automatic Redirection -** allows you to toggle the automatic redirection of the command output to other pages, whether they are Command pages or other page types.
- **Explicit Redirection -** allows you to redirect the output of a particular command to the specified page or page type.

Message Output

The Command page supports displaying event messages from the target. If the Message Level setting is set to **on** or **verbose**, then messages from the target will be displayed in the output window. For more information on controlling the display of messages in Visual SoftICE, refer to the SET MSGLEVEL command.

Logging

The Command page supports the echoing of the input and/or output of the page to a file. The output window of the page where you execute the command will echo to the specified file in the manner you specify. For more information on controlling the logging of the input and/or output of a Command page, refer to the LOG command.

Scripts and Macros

The Command page supports execution of scripts, and macro definition, execution, and deletion.

- You can echo script commands to the current console by using the SET SCRIPTECHO command. You can set this command on a per-page basis.
- You can set file system search path for scripts by using the SET SCRIPTPATH command. You can set this command on a per-workspace basis.
- You can control whether scripts automatically stop execution when an error occurs by using the SET SCRIPTSTOPONERROR command. You can set this command on a per-page basis.

Refer to the MACRO command in the Visual SoftICE Command Reference for more details.

Output Formatting

You can configure the Command page output format by modifying the following attributes:

- Modify the radix of data for input and output by using the SET RADIX command.
- Modify the uppercase hex disassembly output by using the SET UPPERCASE command.
- Modify the way Visual SoftICE displays 64-bit addresses by using the SET ADDRESSFORMAT command.
- Modify the mnemonics used for register names by using the SET REGNAME command.
- Enable or disable the formatting of FP registers by using the SET FLOATREGFORMAT command.
- Modify the format of the PACKET command by using the SET PACKETFORMAT command.

Customization Settings

You can use the Command page to configure global system settings. To view the current settings use the SET command without any parameters. You can configure the following global system settings:

- Automatically stop the target when a command is issued by using the SET STOPONCMD command.
- Configure the target to stop on embedded INT 1 instructions by using the I1HERE command.
- Configure the target to stop on embedded INT 3 instructions by using the I3HERE command.
- Configure the cache size by using the SET CACHE command.
- Configure the input command dialect by using the SET DIALECT command.
- Configure all fault trapping by using the FAULTS command.
- Configure the way the symbol engine and target attempt to match symbolic data by using the SET IMAGEMATCH command.
- Control thread-specific stepping by using the SET THREADP command.
- Configure the warning and confirmation level by using the SET WARNLEVEL command.
- Configure paths used by Visual SoftICE by using the SET EXEPATH, SET EXPORTPATH, SET KDEXTPATH, SET SYMPATH, SET SRCPATH commands.

Page Features

Open Any Number of Command Pages

There are no restrictions on the number of command pages that you can open; however, Visual SoftICE executes commands from all the pages in serial fashion. For more information on Visual SoftICE commands, refer to the Visual SoftICE Command Reference.

Command Interruption

The command page supports interruption of commands that have been entered on the command line. Some Visual SoftICE commands may take a long time to execute, and are designed to be interruptible. Not all commands support interruption. If execution can be interrupted, then the Command Interrupt button becomes active. Complete one of the following procedures to interrupt execution of a command.

- Click the **Command Interrupt** button
- Press **<Ctrl> <Break>**

Event Disassembly Output

The Command page supports automatic output of disassembly when certain events are received from the target. The Target Event Disassembly button toggles this disassembly off and on. If you enable target event disassembly, then events on the target that should show disassembly will display it in the output window. If you disable target event disassembly, then no disassembly will appear in the Command page.

Saving and Clearing Output

You can save all of the output on the Command page to a specific text file by clicking on the **Save Output** button, or right-clicking on the page and selecting **Save Output To File** from the pop-up menu.

Since Visual SoftICE saves the entire contents of the Output window, you may wish to clear the Output window before executing the command whose results you wish to save. You can clear all of the output on this page by issuing the CLS command, or right-clicking on the page and selecting **Clear All** from the pop-up menu.

Note: If you select an existing file, Visual SoftICE replaces the contents of the file with contents of the Output window. It does not append to the selected file.

Finding Text

The Command page provides a way for you to search for a text string value within the page.

- You can search by trying to match the whole word only, or by finding the best match for a partial string.
- You can search by trying to match the case of the specified search string.

Cut, Copy, Paste, Drag, and Drop

The output window only supports Copy to retrieve its data. The input field supports Cut, Copy, Paste, Drag, and Drop.

- You can select any text within the output window and copy it to the clipboard.
- You can drag a selection from the output window and drop it on the input field. This is useful when entering values for commands on the input field.
- You can use Select All from the pop-up menu to highlight all the lines on the output window before copying them to the clipboard.

Customize the User Interface

There are multiple attributes of the page that you can customize. They are divided into two categories: per-page and application wide settings. Per-page attributes are always remembered by the workspace when you save it.

Per-Page Settings

- You can set the state of automatic command redirection.
- You can set event disassembly options.

Application Wide Settings

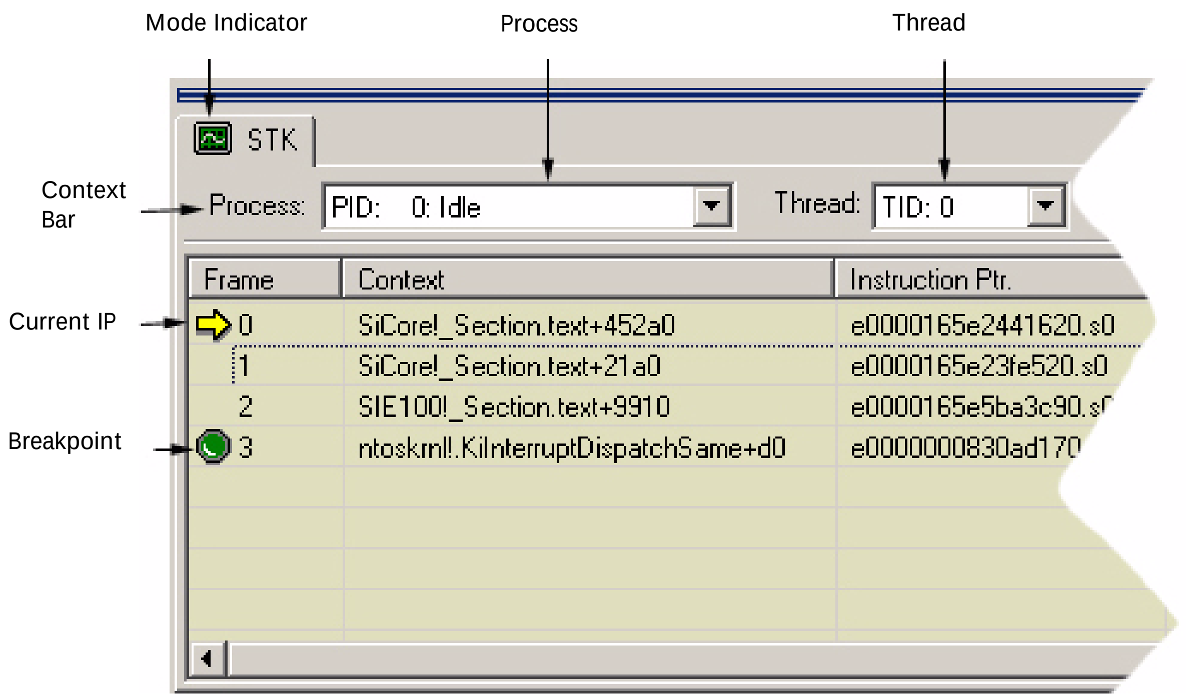
- You can select the font used for text, as well as the background and foreground colors, for the following items: Normal text, Highlighted text, Title text, Dim text, Target Notifications, and Internal Notifications.

Print

The page supports printing, and print-previewing of its contents.

The Stack Page

The Stack page is a read-only page that displays the data on the stack for a selected process and thread context. The Stack page also has a page-specific context bar, allowing you to change the process or thread, which overrides the main context bar.



PID: 0: Idle

TID: 0

Frame	Context	Instruction Ptr.
0	SiCore!_Section.text+452a0	e0000165e2441620.s0
1	SiCore!_Section.text+21a0	e0000165e23fe520.s0
2	SIE100!_Section.text+9910	e0000165e5ba3c90.s
3	ntoskrnl!.KiInterruptDispatchSame+d0	e0000000830ad170

Figure 4-2. Visual SoftICE Stack Page

Concepts and Associated Commands

Displaying Stack Information

There are several commands that can display stack information.

- You can display the stack information from the command line by issuing the STACK command.
- You can display the stack information for a particular thread from the command line by issuing the STACK command with thread-id.
- You can display the stack information for a fiber from the command line by issuing the FIBER command with –s switch.
- You can display the stack page from the command line by issuing the WS command.

Changing Current Context

The stack page displays information based upon the context (thread-id) that is set by either the main context bar, or the page-specific context bar. There are several ways to change the process and context that the Stack page is following.

Frames

The stack page displays a list of items called frames. The frames are listed from the current frame to starting frame of the thread when data is available. The stack page will display the following information for each frame: frame number, current context, instruction address, stack address, and frame address.

- Frame numbers are listed from the current frame to top-most frame of the thread. Each time a new function is called, Visual SoftICE renumbers the frame accordingly.
- Current context displays the function name plus the byte offset from the start of the function of to the instruction address for that frame. If no symbols are available for the frame's function address, then the field will be blank. Refer to the SET SYMPATH command for more information on finding symbols.

Page Features

Context Bar

The Stack page has a page-specific context bar allowing you to select the context (process and thread) for which the Stack page displays stack frames. This page-specific context bar overrides the main context bar and allows the Stack page to track an independent context.

You can switch to a different context to display its stack frames. Use the Process drop-down list to select the process, and the Thread drop-down list to select a specific thread for that process (if the process is multi-threaded).

Source / Disassemble At

The stack page supports displaying a Source or Disassembly page at the instruction address for the selected stack frame. To open either a Source or Disassembly page, right-click on a stack frame and select **Source / Disassemble At**. Visual SoftICE displays a Source page if the symbols and source files can be located for the instruction address. Otherwise, Visual SoftICE displays a Disassembly page starting at the instruction pointer.

- You can change the path that Visual SoftICE uses to search for symbols by issuing the SET SYMPATH command at the command line.
- You can change the path that Visual SoftICE uses to search for source files by issuing the SET SRCPATH command at the command line.

View Memory At

The stack page supports displaying a Memory page for a selected instruction, stack, or frame address. To display a Memory page, right-click on an address and select **View Memory At**.

Breakpoints

The stack page supports the setting, enabling, disabling, or clearing of breakpoints at the instruction address for the selected stack frame. To complete any of these breakpoint operations from the Stack page, right-click on a frame, select **Breakpoint** and then the desired operation (**Set**, **Enable**, **Disable**, or **Clear**).

- You can set a breakpoint for an instruction address of a frame by issuing the BPX command and the address at the command line.
- You can enable, disable, or clear a breakpoint for an instruction address of a frame from the command line by issuing a BL command to get the breakpoint index, and using the BE, BD, or BC command along with that index.

View Locals and Registers for a Frame

The stack page supports displaying either a Register or a Locals page for the selected stack frame. To display one of these pages, right-click on a frame and select **Frame** and then the desired page type (**Registers** or **Locals**).

- You can display the registers for individual frames of a thread from the command line by issuing the `THREAD -r [-f frame] TID` command.
- You can display the locals for the current stack frame from the command line by issuing the LOCALS command.

Stack Status Column

The stack status column provides extra information about the stack walking process. It notifies you about potential problems Visual SoftICE encountered while walking the stack. If Visual SoftICE encountered no problems, and the stack was walked fully, no status messages are displayed. If the stack walk terminates prematurely there will be a status message describing the terminating condition. The most important message is the "Unwind info unavailable" message on IA64.

"Unwind info unavailable" usually means that Visual SoftICE could not read the unwind information from the target. This occurs because the part of the image on the target containing the unwind info is paged out and Visual SoftICE could not access it. A solution for this problem is to have a local copy of all executables (including files in the system32 directory). Add these directories to the exepath so the local copies can be found instead of having to be retrieved from the target.

Open Only One Stack Page

You can only have one Stack page open.

Copy

The page only supports the Copy function to retrieve its data. You can select any cell (intersection of a row and column) and copy its text to the clipboard.

Note: Select a cell by by placing the mouse cursor on the cell and then right-clicking on the element you wish to copy.

Customize the User Interface

There are multiple attributes of the page that you can customize. They are divided into two categories: per-page and application wide settings. Per-page attributes are always remembered by the workspace when you save it.

Per-Page Settings

- You can hide or display the page-specific context bar.
- You can sort the stack frames by any of the displayed columns.
- You can reorganize and resize the columns.

Application Wide Settings

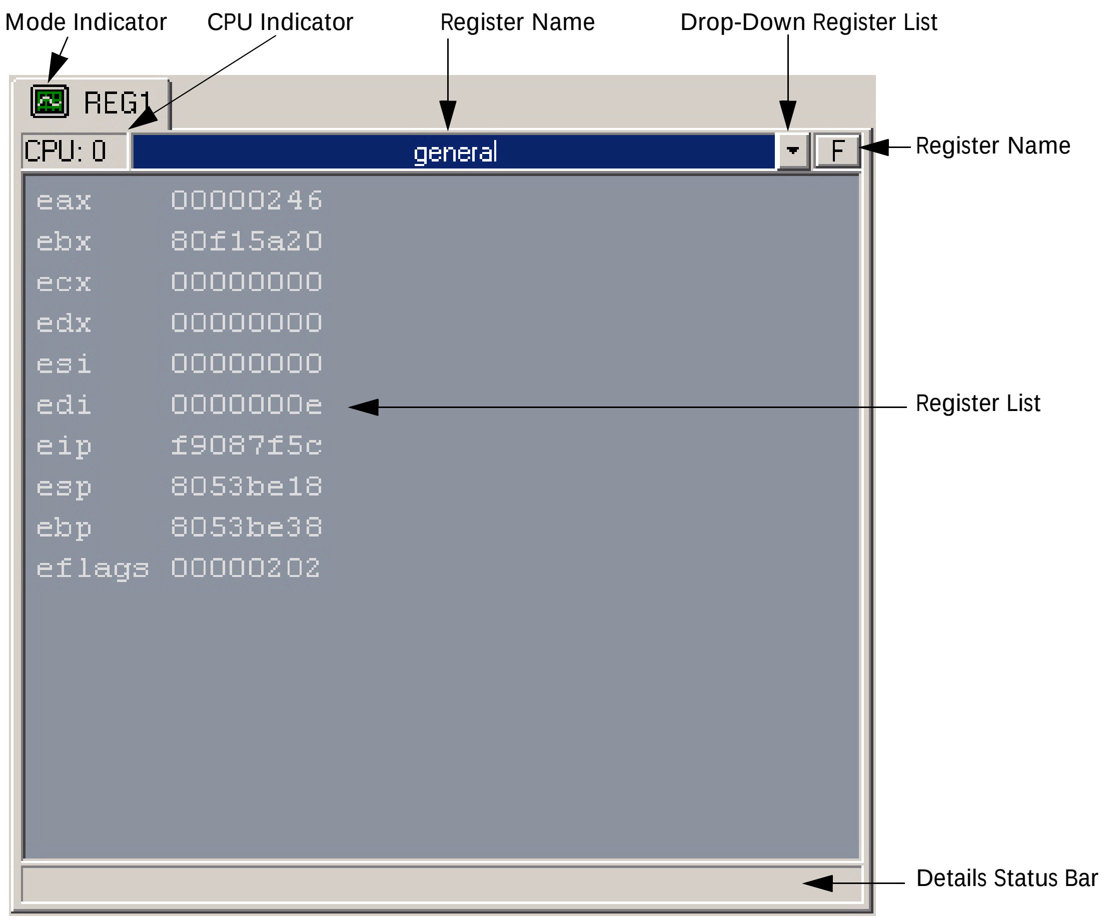
- You can control whether or not the page switches automatically to Live mode when a connection to a new target is established.

Print

The page supports printing, and print-previewing of its contents.

The Registers Page

The Registers page is a container for displaying and editing the names, fields, contents, descriptions, and symbols of the target processor registers.



```
eax     00000246
ebx     80f15a20
ecx     00000000
edx     00000000
esi     00000000
edi     0000000e
eip     f9087f5c
esp     8053be18
ebp     8053be38
eflags  00000202
```

Figure 4-3. Visual SoftICE Registers Page

Concepts and Associated Commands

Types

Different processors contain different registers and register types. Registers are always bit containers, but can be interpreted as bytes, integers, a collection of flags, or floating point values meeting various standard representations and precisions.

Fields

Registers of a given size (for example 32bits wide) very often contain a mixture of information using various subsets of their bits, called fields.

- ◆ You can display fields for any register (or group of registers) in the page by clicking the **Fields** button on the upper right hand corner.
- ◆ You can display register fields from the command line by issuing the R command with the -f switch.

Descriptions

Registers all have additional description information, primarily drawn from details provided by the processor manufacturer, and in some cases as commonly used by operating systems.

- ◆ You can view descriptions for any register in the details status bar at the bottom of the page (if the register does not currently map to a symbol).
- ◆ You can display register descriptions from the command line by issuing the R command with the -d switch.

Symbols

Registers can contain data that currently maps to a known symbol in the current context.

- ◆ If a symbol is available for the current register contents, the symbol is displayed in the details status bar at the bottom of the page.
- ◆ You can display symbols for registers from the command line by issuing the R command with the -s switch.

Allowed Operations

Some registers are read-only, except by the processor itself. Some registers are not available, based on the state of the processor. Modifying some registers can be a dangerous action. As a level of protection, registers are marked in Visual SoftICE as being read-write, or read-only, and can have associated warnings on write operations.

For more information on controlling the display of warnings in Visual SoftICE, refer to the SET WARNLEVEL command.

Groups

Registers that are associated with one another, or are commonly useful to see together, have been gathered into collections called Register Groups.

- All processors support the GENERAL and ALL groups.
- You can display a pop-up menu listing the available register groups by right-clicking on the page (when connected to a stopped target) and selecting Groups.
- You can display register groups from the command line by issuing the RG command.
- You can read from groups, or write to groups. However, if any member is read-only, or has warnings associated with it, the write will fail for any read-only registers, and warnings will be displayed for any warning-associated registers.
- You can read from, or write to, groups from the command line by issuing the R command.

Page Features

View the Registers You Want

The page supports a title area, which tells you the group, register, or set of registers being displayed within it.

- The group, register, or set of registers you have in any page is remembered on a per-page basis in the workspace when you save it.
- You can click on this area to type in a register name, register group name, or list of registers (space delineated) you want displayed.
- You can right-click on this area to select from the available register groups.
- You can set the displayed registers from the command line by issuing the R command and redirecting its output:

  ```
  R eip, esp, eflags />reg
  ```
- You can set the displayed registers from the command line automatically when you issue the R command if you have enabled auto-command redirection.

Open Any Number of Register Pages

There are no restrictions on the number of Register pages, so you can build a workspace with multiple views containing just the registers you are interested in, displayed the way you want, where you want.

View the Registers from a Specific Processor

The page supports a CPU display and selector field. On multi-processor machines you can select the specific processor for which you want to display the registers. Depending on the state of the target, some processors on a multi-processor machine may be inaccessible.

- You can select the CPU via the CPU indicator on the Registers page.
- You can view the registers for a specific processor from the command line by issuing the R command with the -c switch.
- You can control whether the page automatically switches to the stopped CPU on a multi-processor target from the preferences utility.

View Register Data in the Format You Want

Register data can be interpreted for display in a number of ways. The default for display in the page is as integer information in the datalength appropriate for the length of the register itself (or length of the field when displaying fields).

- All formatting preferences are remembered on a per-page basis in the workspace when you save it.
- You can force the display of data to always be BYTE, WORD, DWORD, or QWORD using the pop-up menu.
- You can enable registers that contain floating point data to display that data in a floating point representation using the pop-up menu.
- You can select which floating point representation Visual SoftICE applies to floating point data from the following representations: Double (64bit real), IEEE 80bit, 82bit, and AMD 3DNow.

Edit a Register or Set of Registers

This page provides several ways of editing register values, both for individual registers and groups of registers.

- You can double-click on a single register to open the Modify Register utility.
- You can right-click on a selected register, or group of registers, and select Modify from the pop-up menu.
- You can select the Zero option from pop-up menu to set the selected register, or group of registers, to zero.
- You can write to registers, register groups, and sets of registers from the command line by issuing the R command.

Copy, Paste, Drag, and Drop

The page supports many ways of retrieving its data.

- You can copy register names and values to the clipboard for single and multiple registers.
- You can drag a single register, in which case just the value of the register is placed in the clipboard.
- You can drag multiple registers, in which case the names and values of all the registers are collected in the clipboard, and can then be used by other pages in raw or formatted form. For example, the Text Scratch page prints the collection of names and values in table form.

Customize the User Interface

There are multiple attributes of the page that you can customize. They are divided into two categories: per-page and application wide settings. Per-page attributes are always remembered by the workspace when you save it.

Per-Page Settings

- You can hide or display the details status bar.
- You can display the registers and their values in a vertical list only (default) or in a wrapping horizontal collection.

Application Wide Settings

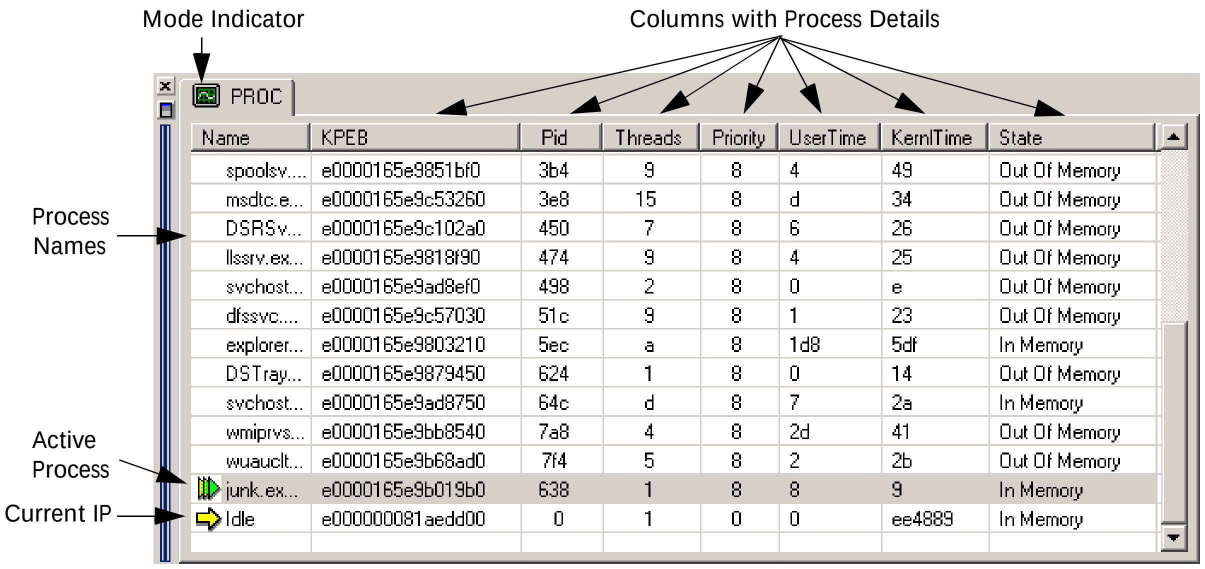
- You can select the font used, background and foreground color, and the color used to indicate a change in the value of a register between updates of the page.
- You can control whether or not the page switches automatically to Live mode when a connection to a new target is established.

Print

The page supports printing, and print-previewing of its contents.

The Process List Page

The Process List page displays a list of the processes running on the target machine. It also uses a triple green arrow to indicate which process is the current process context, and a yellow arrow to indicate the process containing the current IP.



Name	KPEB	Pid	Threads	Priority	UserTime	KernlTime	State
spoolsv....	e0000165e9851bf0	3b4	9	8	4	49	Out Of Memory
msdtc.e...	e0000165e9c53260	3e8	15	8	d	34	Out Of Memory
DSRSv...	e0000165e9c102a0	450	7	8	6	26	Out Of Memory
llssrv.ex...	e0000165e9818f90	474	9	8	4	25	Out Of Memory
svchost...	e0000165e9ad8ef0	498	2	8	0	e	Out Of Memory
dfssvc....	e0000165e9c57030	51c	9	8	1	23	Out Of Memory
explorer...	e0000165e9803210	5ec	a	8	1d8	5df	In Memory
DSTray...	e0000165e9879450	624	1	8	0	14	Out Of Memory
svchost...	e0000165e9ad8750	64c	d	8	7	2a	In Memory
wmiprvs...	e0000165e9bb8540	7a8	4	8	2d	41	Out Of Memory
wuauclt...	e0000165e9b68ad0	7f4	5	8	2	2b	Out Of Memory
junk.ex...	e0000165e9b019b0	638	1	8	8	9	In Memory
Idle	e000000081aedd00	0	1	0	0	ee4889	In Memory

Figure 4-4. Visual SoftICE Process List Page

Concepts and Associated Commands

Current Context

You can use the Process List page to make a process the “current” one. You can ALWAYS make a process current, and that becomes the current context for Visual SoftICE. Refer to “About the Context Bar” on page 22 for more information on the main context bar, and page-specific contexts.

Stopped vs. Running State

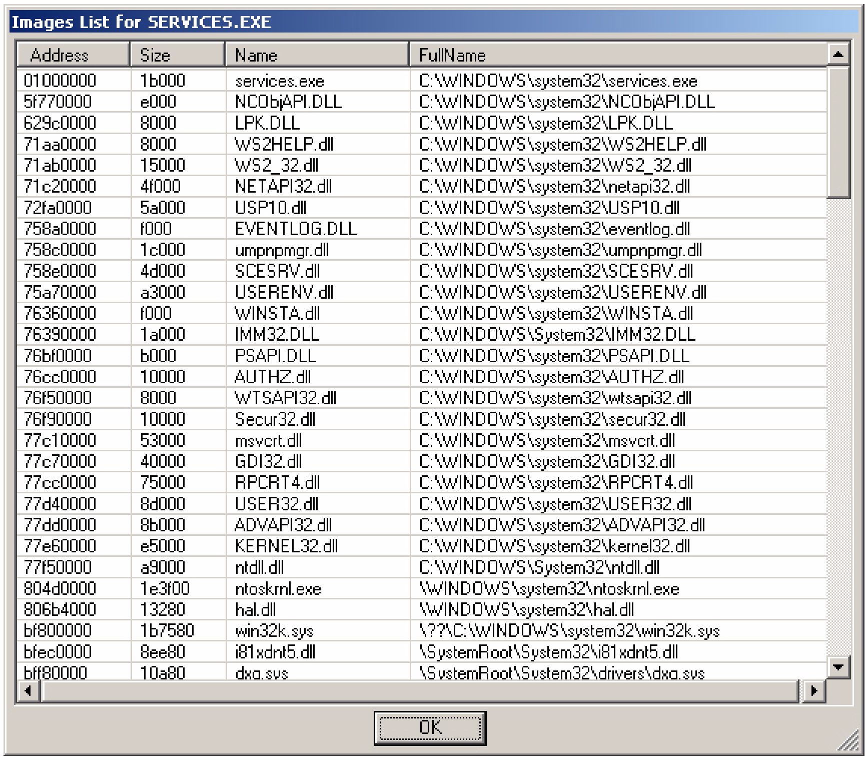
The page supports displaying only two levels of information for a process: basic and extended. The levels are tied to the target state. When the target is running, Visual SoftICE displays only the basic information for the processes. When the target is stopped, Visual SoftICE displays the extended information for each process.

The following basic information is displayed: **Name**, **KPEB address**, and **PID**.

The following extended information is displayed: **Name**, **KPEB address**, **PID**, **Thread Count**, **Priority Level**, **User Mode Time**, **Kernel Mode Time**, and **Process State**.

Images

You can use the Process List page to view the images for a selected process. This works the same as the IMAGE command. To view the images for a process, right-click on the process, and select **Images** from the pop-up menu.



Images List for SERVICES.EXE

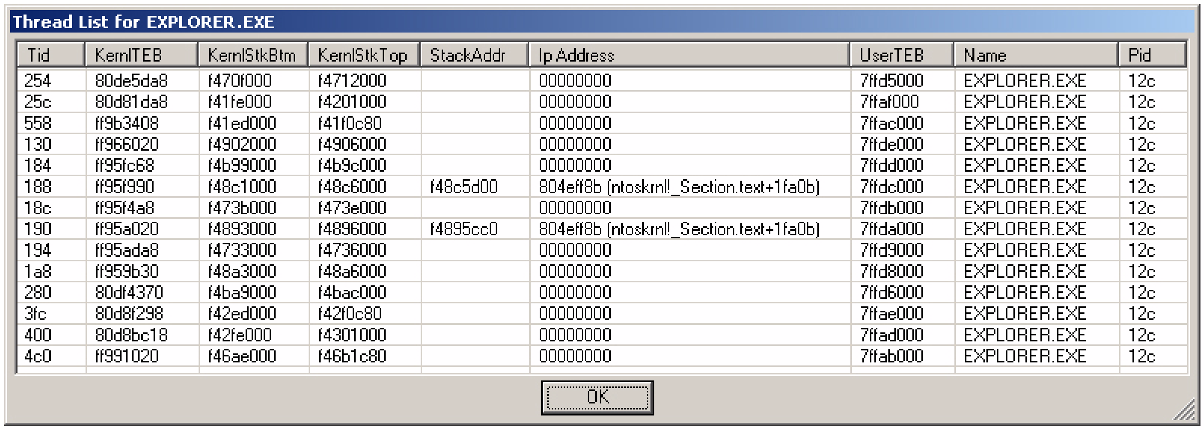
Address	Size	Name	FullName
01000000	1b000	services.exe	C:\WINDOWS\system32\services.exe
5f770000	e000	NCObjAPI.DLL	C:\WINDOWS\system32\NCObjAPI.DLL
629c0000	8000	LPK.DLL	C:\WINDOWS\system32\LPK.DLL
71aa0000	8000	WS2HELP.dll	C:\WINDOWS\system32\WS2HELP.dll
71ab0000	15000	WS2_32.dll	C:\WINDOWS\system32\WS2_32.dll
71c20000	4f000	NETAPI32.dll	C:\WINDOWS\system32\netapi32.dll
72fa0000	5a000	USP10.dll	C:\WINDOWS\system32\USP10.dll
758a0000	f000	EVENTLOG.DLL	C:\WINDOWS\system32\eventlog.dll
758c0000	1c000	umpnpmgr.dll	C:\WINDOWS\system32\umpnpmgr.dll
758e0000	4d000	SCESRV.dll	C:\WINDOWS\system32\SCESRV.dll
75a70000	a3000	USERENV.dll	C:\WINDOWS\system32\USERENV.dll
76360000	f000	WINSTA.dll	C:\WINDOWS\system32\WINSTA.dll
76390000	1a000	IMM32.DLL	C:\WINDOWS\System32\IMM32.DLL
76bf0000	b000	PSAPI.DLL	C:\WINDOWS\system32\PSAPI.DLL
76cc0000	10000	AUTHZ.dll	C:\WINDOWS\system32\AUTHZ.dll
76f50000	8000	WTSAPI32.dll	C:\WINDOWS\system32\wtsapi32.dll
76f90000	10000	Secur32.dll	C:\WINDOWS\system32\secur32.dll
77c10000	53000	msvcrt.dll	C:\WINDOWS\system32\msvcrt.dll
77c70000	40000	GDI32.dll	C:\WINDOWS\system32\GDI32.dll
77cc0000	75000	RPCRT4.dll	C:\WINDOWS\system32\RPCRT4.dll
77d40000	8d000	USER32.dll	C:\WINDOWS\system32\USER32.dll
77dd0000	8b000	ADVAPI32.dll	C:\WINDOWS\system32\ADVAPI32.dll
77e60000	e5000	KERNEL32.dll	C:\WINDOWS\system32\kernel32.dll
77f50000	a9000	ntdll.dll	C:\WINDOWS\System32\ntdll.dll
804d0000	1e3f00	ntoskrnl.exe	\WINDOWS\system32\ntoskrnl.exe
806b4000	13280	hal.dll	\WINDOWS\system32\hal.dll
bf800000	1b7580	win32k.sys	\??\C:\WINDOWS\system32\win32k.sys
bfec0000	8ee80	i81xdnt5.dll	\SystemRoot\System32\i81xdnt5.dll
bff80000	10a80	dxg.sys	\SystemRoot\System32\drivers\dxg.sys

Figure 4-5. Image List

Threads

You can use the Process List page to view the threads for a selected process. This works the same as the THREAD command. To view the threads for a process, right-click on the process and select **Threads** from the pop-up menu.

Note: The target must be stopped to view threads for a process.



Thread List for EXPLORER.EXE

Tid	KernlTEB	KernlStkBtm	KernlStkTop	StackAddr	Ip Address	UserTEB	Name	Pid
254	80de5da8	f470f000	f4712000		00000000	7ffd5000	EXPLORER.EXE	12c
25c	80d81da8	f41fe000	f4201000		00000000	7ffaf000	EXPLORER.EXE	12c
558	ff9b3408	f41ed000	f41f0c80		00000000	7ffac000	EXPLORER.EXE	12c
130	ff966020	f4902000	f4906000		00000000	7ffde000	EXPLORER.EXE	12c
184	ff95fc68	f4b99000	f4b9c000		00000000	7ffdd000	EXPLORER.EXE	12c
188	ff95f990	f48c1000	f48c6000	f48c5d00	804eff8b (ntoskrnl!_Section.text+1fa0b)	7ffdc000	EXPLORER.EXE	12c
18c	ff95f4a8	f473b000	f473e000		00000000	7ffdb000	EXPLORER.EXE	12c
190	ff95a020	f4893000	f4896000	f4895cc0	804eff8b (ntoskrnl!_Section.text+1fa0b)	7ffda000	EXPLORER.EXE	12c
194	ff95ada8	f4733000	f4736000		00000000	7ffd9000	EXPLORER.EXE	12c
1a8	ff959b30	f48a3000	f48a6000		00000000	7ffd8000	EXPLORER.EXE	12c
280	80df4370	f4ba9000	f4bac000		00000000	7ffd6000	EXPLORER.EXE	12c
3fc	80d8f298	f42ed000	f42f0c80		00000000	7ffae000	EXPLORER.EXE	12c
400	80d8bc18	f42fe000	f4301000		00000000	7ffad000	EXPLORER.EXE	12c
4c0	ff991020	f46ae000	f46b1c80		00000000	7ffab000	EXPLORER.EXE	12c

Figure 4-6. Thread List

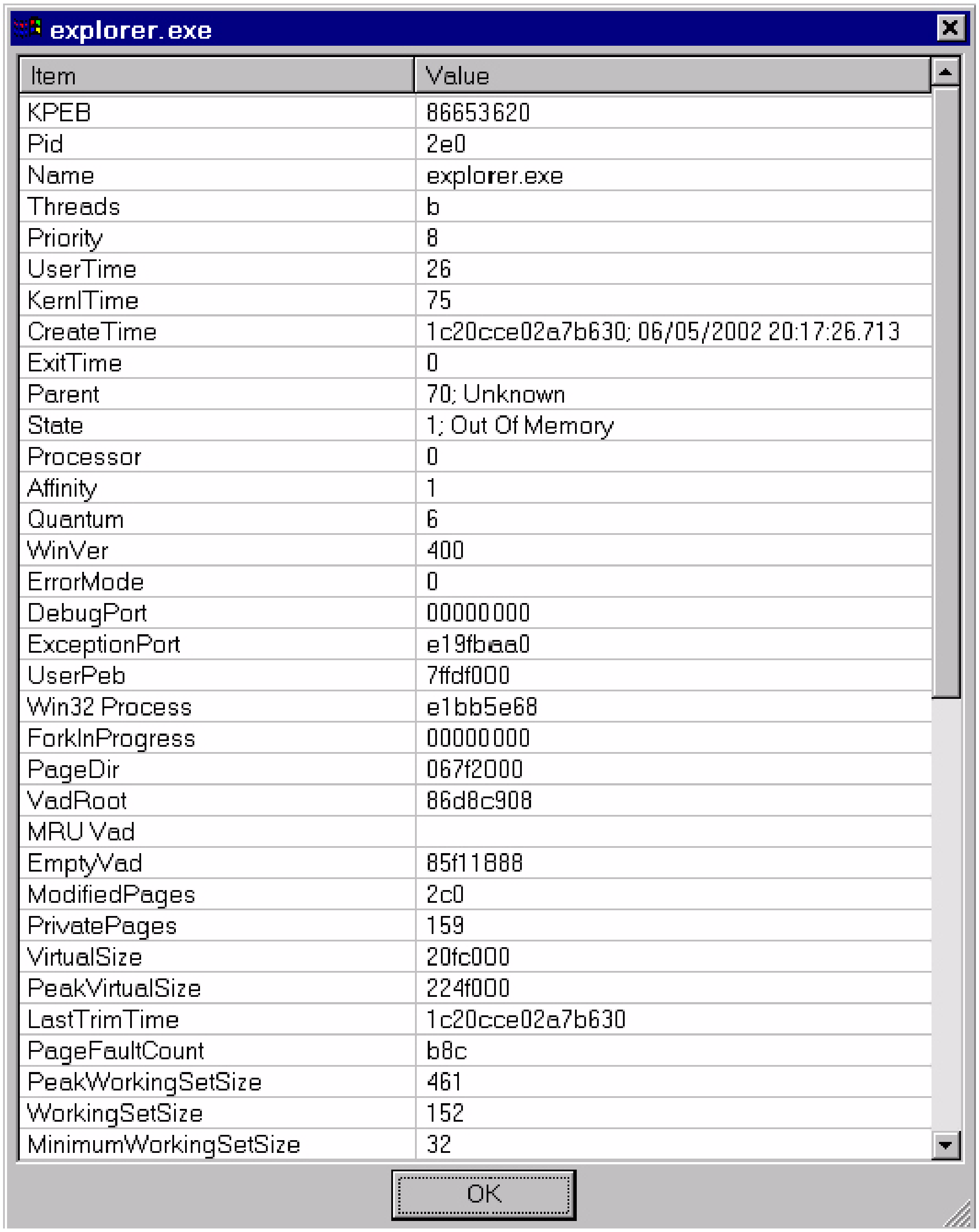
Page Features

Open Only One Process List Page

You can only have one Process List page open.

View Detailed Process Information

You can use the Process List page to view the known details of a process. The Details view for a given process lists all the known information about that process. You can sort the list of details by Item or by Value by clicking on the appropriate column heading.



explorer.exe

Item	Value
KPEB	86653620
Pid	2e0
Name	explorer.exe
Threads	b
Priority	8
UserTime	26
KernlTime	75
CreateTime	1c20cce02a7b630; 06/05/2002 20:17:26.713
ExitTime	0
Parent	70; Unknown
State	1; Out Of Memory
Processor	0
Affinity	1
Quantum	6
WinVer	400
ErrorMode	0
DebugPort	00000000
ExceptionPort	e19fbaa0
UserPeb	7ffdf000
Win32 Process	e1bb5e68
ForkInProgress	00000000
PageDir	067f2000
VadRoot	86d8c908
MRU Vad	
EmptyVad	85f11888
ModifiedPages	2c0
PrivatePages	159
VirtualSize	20fc000
PeakVirtualSize	224f000
LastTrimTime	1c20cce02a7b630
PageFaultCount	b8c
PeakWorkingSetSize	461
WorkingSetSize	152
MinimumWorkingSetSize	32

Figure 4-7. Detailed Process Information

Kill a Process

This page provides a way of terminating a process. You can only kill a process if the processor is running. The killing of a process may fail if the OS refuses to kill the selected process; however, Visual SoftICE does not limit what processes you can attempt to kill.

Copy and Drag

The page supports copying and dragging to retrieve its data.

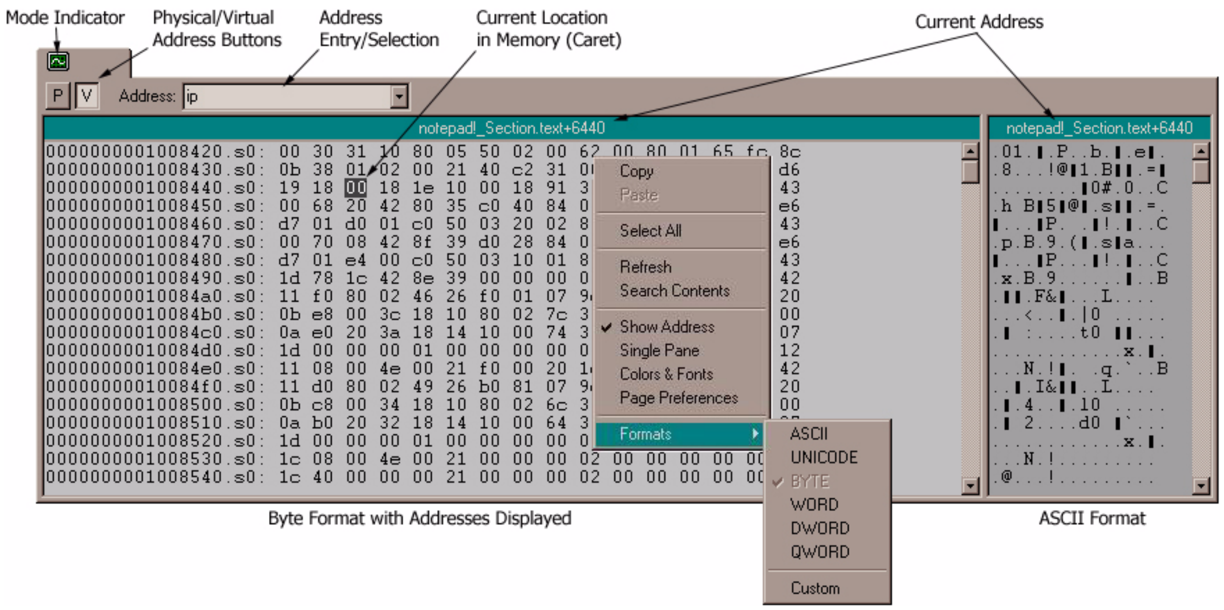
- You can copy data from any column to the clipboard.
- You can drag data from any column to another page.

Print

The page supports printing, and print-previewing of its contents.

The Memory Page

The memory page is a container for displaying and editing blocks of memory on a target machine, or from file sources. This is a shared resource for many DriverStudio components, including BoundsChecker for Drivers, and Visual SoftICE. Memory loaded into this page can be viewed in scalar modes (BYTE, WORD, DWORD...), string, and as structures and classes. Access to physical and virtual memory, display of symbols, formatting to non-scalar types, editing, and data drag and drop to other pages is all driven by the source of the memory loaded into the page (for example, BoundsChecker for Drivers does not support formatting memory into types known to the symbol engine, while Visual SoftICE does). You can retrieve memory by physical or virtual address.



Address: ip

```
0000000001008420.s0:  00 30 31 10 80 05 50 02 00 62 00 80 01 65 fc 8c
0000000001008430.s0:  0b 38 01 02 00 21 40 c2 31 0                  d6
0000000001008440.s0:  19 18 00 18 1e 10 00 18 91 3                  43
0000000001008450.s0:  00 68 20 42 80 35 c0 40 84 0                  e6
0000000001008460.s0:  d7 01 d0 01 c0 50 03 20 02 8                  43
0000000001008470.s0:  00 70 08 42 8f 39 d0 28 84 0                  e6
0000000001008480.s0:  d7 01 e4 00 c0 50 03 10 01 8                  43
0000000001008490.s0:  1d 78 1c 42 8e 39 00 00 00 0                  42
00000000010084a0.s0:  11 f0 80 02 46 26 f0 01 07 9                  20
00000000010084b0.s0:  0b e8 00 3c 18 10 80 02 7c 3                  00
00000000010084c0.s0:  0a e0 20 3a 18 14 10 00 74 3                  07
00000000010084d0.s0:  1d 00 00 00 01 00 00 00 00 0                  12
00000000010084e0.s0:  11 08 00 4e 00 21 f0 00 20 1                  42
00000000010084f0.s0:  11 d0 80 02 49 26 b0 81 07 9                  20
0000000001008500.s0:  0b c8 00 34 18 10 80 02 6c 3                  00
0000000001008510.s0:  0a b0 20 32 18 14 10 00 64 3
0000000001008520.s0:  1d 00 00 00 01 00 00 00 00 0
0000000001008530.s0:  1c 08 00 4e 00 21 00 00 00 02 00 00 00 00 00
0000000001008540.s0:  1c 40 00 00 00 21 00 00 00 02 00 00 00 00 00
```

```
.01.▌.P..b.▌.e▌.
.8...!@▌1.B▌▌.=▌
.........▌0#.0..C
.h B▌5▌@▌.s▌▌.=.
▌...▌P. .▌!.▌..C
.p.B.9.(▌.s▌a...
▌...▌P...▌!.▌..C
.x.B.9......▌..B
.▌▌.F&▌...L....
...<..▌.|0 .....
.▌ :....t0 ▌▌...
.............x.▌.
...N.!▌. .q.`..B
..▌.I&▌▌..L....
.▌.4..▌.10 .....
.▌ 2....d0 ▌`...
.............x.▌.
...N.!..........
.@...!..........
```

Figure 4-8. Visual SoftICE Memory Page

Concepts and Associated Commands

Address Types

You can access memory on the target machine via its physical location (the location on the memory bus where the memory unit of interest is located), or via virtual address per process (as implemented by the OS). Access to memory on the target can fail if one of the following situations is true:

- The virtual address requested is paged out
- The virtual address is unmapped (invalid) in the current process
- The memory is inaccessible based on processor state

Virtual memory access is the most common form of memory access, and is connected with static and dynamic allocations for a given process or the kernel itself.

Physical memory access is useful for looking at I/O registers, video memory, or other regions physically mapped (memory mapped) by the processor. This means the memory of these devices appears to be part of the standard address space of the processor.

You cannot access unmapped memory on the target, or memory held by a specific device on some other bus, in the same manner. Refer to the PCI command for exploring device-managed resources off that bus.

Visual SoftICE can read memory while the target is running or stopped. Reading memory while the target is running retrieves a snapshot, and is not guaranteed to be accurate to current values.

Virtual Memory

You can display virtual memory by:

- Selecting the virtual address type (**V**) button and then entering an address in the **Address** field and pressing **Enter**.
- Issuing the D command in Visual SoftICE.

Physical Memory

You can display physical memory by:

- Selecting the physical address type (**P**) button and then entering an address in the **Address** field and pressing **Enter**.
- Issuing the PHYS command in Visual SoftICE (to display the virtual memory addresses that map to a physical address).
- Issuing the D command with the -p switch in Visual SoftICE.
- Issuing the PEEK command in Visual SoftICE.

Symbols

All addresses may have a symbolic representation. Visual SoftICE automatically updates and displays the symbolic representation on the current address title bar as you move the caret through various memory locations. You can disable symbolic lookup (for performance improvements) for all memory pages via the global setting (Symbolic Address Lookup) under the Memory page preferences (File > Preferences > Global Settings > Memory Page).

Page Features

Open Any Number of Memory Pages

There are no restrictions on the number of Memory pages you can open. You can build a workspace with multiple views containing just the memory blocks you are interested in, displayed the way you want, where you want.

View Memory in the Format You Want

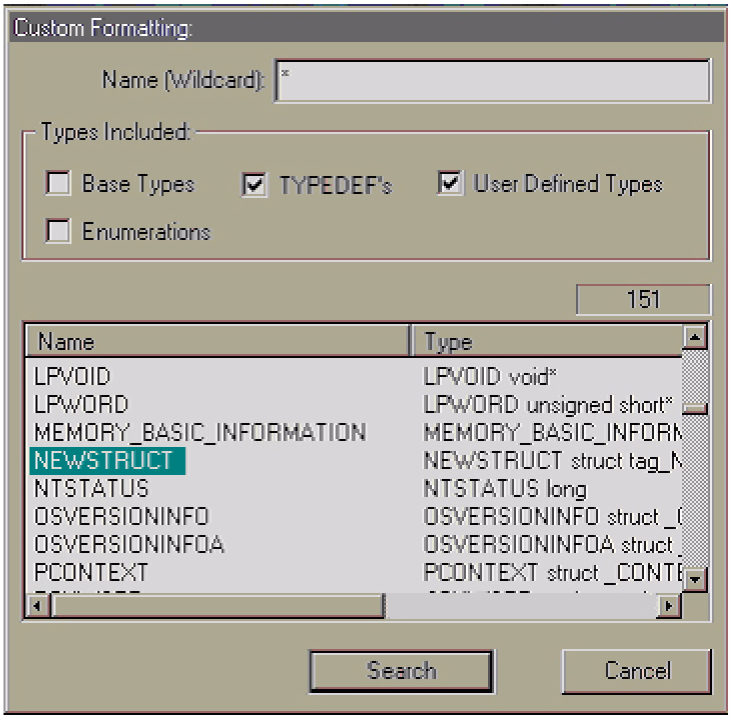
Memory can be interpreted for display in a number of ways, including built-in scalar formats (BYTE, WORD, DWORD, QWORD, ASCII, and UNICODE), and custom types known by the symbol engine (type, structure, or class).

The default view of memory is 2 panes, BYTE view on the left with addresses, ASCII view on the right with no addresses. The Memory page remembers all scalar formatting preferences, the last custom format name you selected (per pane), and last address viewed (on a per-page basis) in the workspace when you save it. The last custom format name and the last address viewed are displayed in the pane pop-menu and address combo box respectively at the time DriverStudio loads the workspace.

You can customize the page to display memory in 2 separate panes or just one via the pop-up menu. If you display memory in 2 panes, the Memory page remembers the splitter location (on a per-page basis) in the workspace when you save it.

You can select a custom format off the pop-up menu, per pane. You should consider this formatting a cast operation on the block of memory. The format must be a known type in the symbol engine (only available in Visual SoftICE).

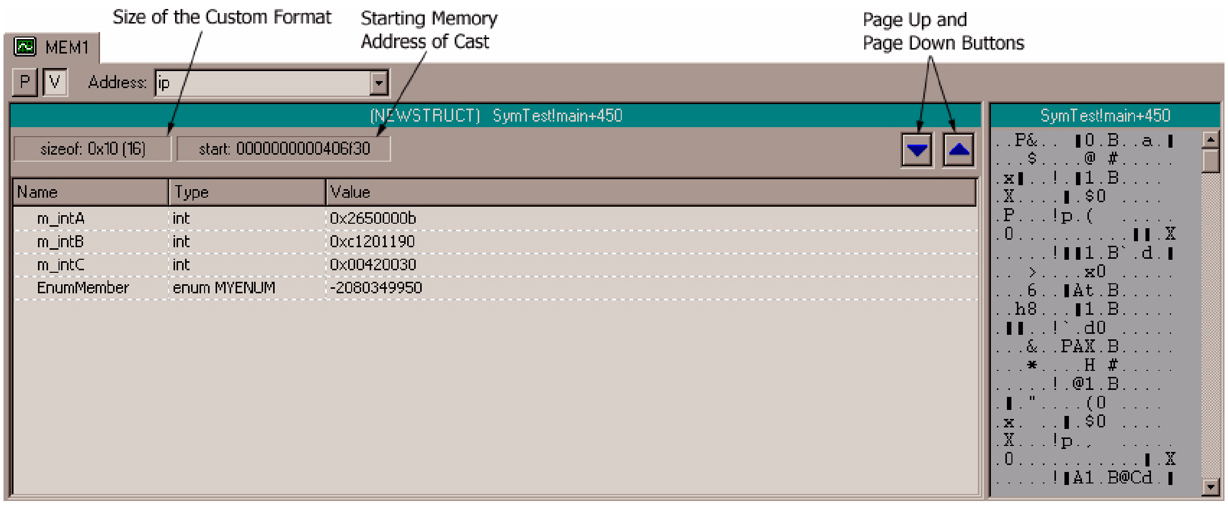
The selection dialog allows you to search using wildcards, and/or limit the types displayed based on your preferences. Additionally, if you know the format name, you can enter it directly and press Enter. If it is a known type in the symbol engine, Visual SoftICE will immediately apply it.



Name	Type
LPVOID	LPVOID void*
LPWORD	LPWORD unsigned short*
MEMORY_BASIC_INFORMATION	MEMORY_BASIC_INFORM
NEWSTRUCT	NEWSTRUCT struct tag_N
NTSTATUS	NTSTATUS long
OSVERSIONINFO	OSVERSIONINFO struct _(
OSVERSIONINFOA	OSVERSIONINFOA struct _
PCONTEXT	PCONTEXT struct _CONTE

Figure 4-9. Custom Formatting Dialog

When a custom format is shown, the name of the format, size of the format, and two additional buttons (**PageUp**, **PageDn**) are displayed to walk through memory.


Size of the Custom Format
Starting Memory Address of Cast
Page Up and Page Down Buttons
MEM1
P
V
Address: ip
(NEWSTRUCT) SymTest!main+450
sizeof: 0x10 (16)
start: 0000000000406f30
SymTest!main+450

Name	Type	Value
m_intA	int	0x2650000b
m_intB	int	0xc1201190
m_intC	int	0x00420030
EnumMember	enum MYENUM	-2080349950

```
..P&.. ▌0.B..a.▌
...$.....@ #.....
.x▌..!.▌1.B....
.X....▌.$0 ....
.P...!p.( .....
.0...........▌▌.X
.....!▌▌1.B`.d.▌
.. >....x0 .....
...6..▌At.B.....
..h8...▌1.B.....
.▌▌..!`.d0 .....
...&..PAX.B.....
...*....H #.....
.....!.@1.B....
.▌."....(0 ....
.x...▌.$0 ....
.X...!p.. .....
.0............▌.X
.....!▌A1.B@Cd.▌
```

Figure 4-10. Custom Memory Formatting

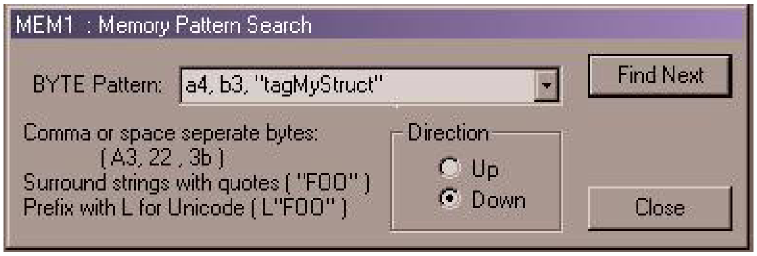
Please note this is a cast operation on existing memory. Therefore these buttons may be useful for walking through simple arrays of fixed size types, structures,or classes, but do not walk lists (linked lists, pointers, or other collection mechanisms).

If you save the workspace with a pane displaying a custom format (type, structure, or class), Visual SoftICE remembers the format name, but does not apply it at the time it loads the workspace. This is because the format may be unknown in the current context (not in the loaded or active symbol tables). Once the workspace is loaded, you may reassign the formatting directly via the pop-up menu.

Search Page Contents

This page provides a utility to conduct byte pattern searching. The Search utility for the Memory page allows you to search the page's contents forward and backward, but restricts the searching to a 32-byte pattern length.

This utility behaves like the S command in Visual SoftICE, however it is restricted to the contents of the memory page, and can search forwards or backwards.


MEM1 : Memory Pattern Search

BYTE Pattern: a4, b3, "tagMyStruct"

Comma or space seperate bytes:
(A3, 22 , 3b)
Surround strings with quotes ("FOO")
Prefix with L for Unicode (L"FOO")

Direction: Up [✓ Down]

Figure 4-11. Memory Page Search Dialog

Edit Memory

This page provides several ways of editing memory data, both in scalar and custom formatting.

- In scalar modes, you can double-click or select a unit of memory and press **Enter,** or the **Backspace** key, to get an in-place edit control. You can cancel editing by pressing the **Esc** key, or you can apply any changes by pressing **Enter**. If your edit fails to be accepted, the Memory page displays an error message. If accepted, the Memory page updates all panes displaying that memory location, using the modified color to highlight the change.

  - In non-string formats, this edit control will be constrained to the unit of memory (BYTE, WORD, DWORD, QWORD), and allows hexadecimal input only.
  - In string formats (ASCII, UNICODE), the edit control will be constrained to a full line of memory (as displayed) and allows string input. The current character at the start of editing will be highlighted within the line. On acceptance of your input, the string will be converted to an appropriate byte stream (ASCII or UNICODE) before being sent to the target.
- In custom format mode, you can directly edit the members of the type, structure, or class by clicking on the member value cell. The Memory page opens an in-place edit control. You can cancel editing by pressing the <**Esc>** key, or you can apply any changes by pressing <**Enter>**.
- You can edit virtual memory from the command line in Visual SoftICE by issuing the E command.
- You can edit physical memory from the command line in Visual SoftICE by issuing the POKE command.

Copy, Paste, Drag, and Drop

The page supports many ways of retrieving its data.

- You can a copy a single scalar unit, or range of lines, by highlighting them and then selecting **Copy** from the pop-up menu.
- You can drag a single scalar data value out of the memory page and drop it on any other page that accepts data this way.
- You can drop a single data value from any other page on the memory page. Visual SoftICE interprets the data value as an address, and places it in the address field (along with keyboard focus). While focus is in the Address field, you can edit the address, or append to it before pressing <**Enter>** to accept and display that address.

Customize the User Interface

There are multiple attributes of the page that you can customize. They are divided into two categories: per-page and application wide settings. Per-page attributes are always remembered by the workspace when you save it.

Per-Page Settings

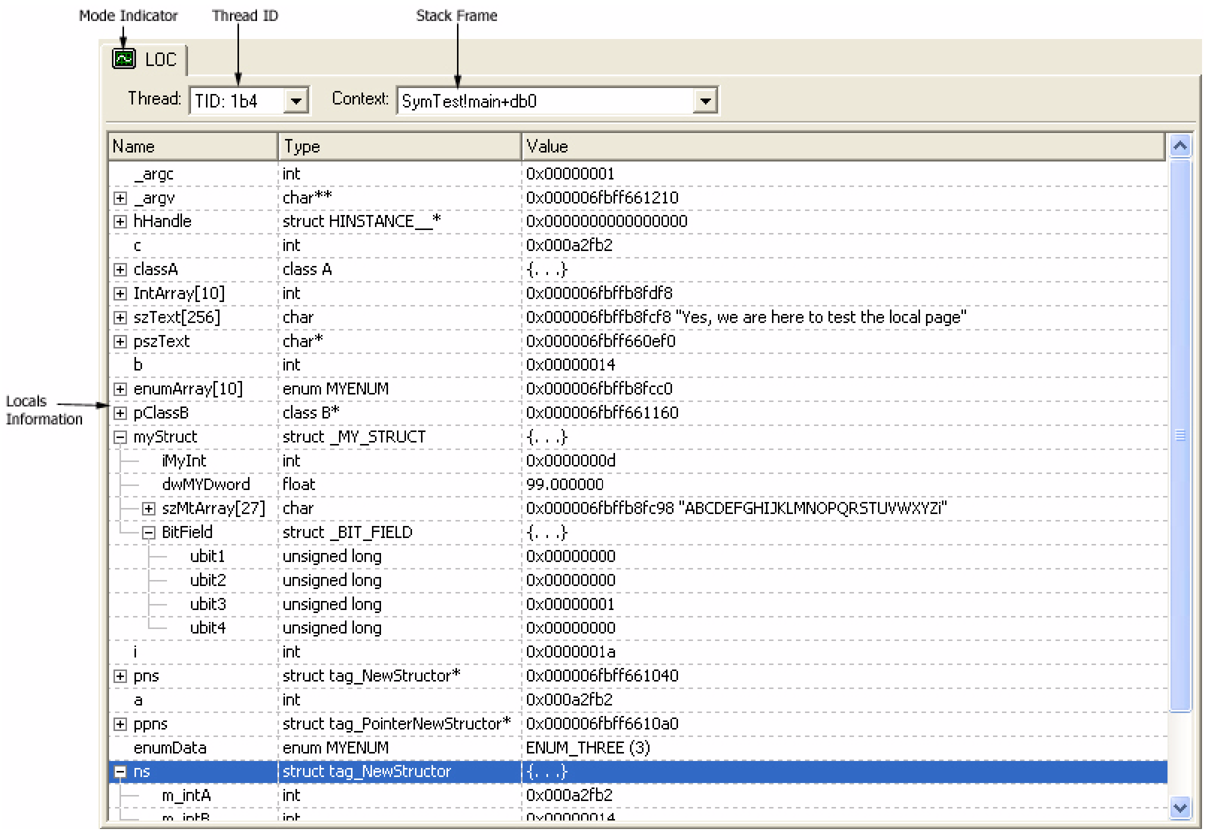
- You can customize the page to display 1 or 2 views of memory (panes).
- You can customize the page to display addresses in each pane.
- You can customize the page to display a range of scalar or custom formats per pane.

Application Wide Settings

- You can select the font used, as well as the background and foreground color, for each pane.
- You can select the color used to indicate a change in the value of memory between updates of the page.
- You can enable or disable symbolic lookup of every address location displayed in the page, as you move the caret through memory.
- You can set the number of bytes per line the Memory page displays in each pane (the valid range is 16 to 128 bytes, and must be evenly divisible by 2).
- You can control the size of memory blocks the page requests from the target when you display memory. Larger memory blocks provide better performance, while smaller memory blocks reduce the chances of hitting paged out conditions (the valid size range is 128 to 8192 bytes).
- You can control whether or not the page switches automatically to Live mode when a connection to a new target is established.

The Locals Page

The Local page is a read-write container for displaying and editing the variables local to current instruction pointer. The Local page can display the variable name, type and value. The local page also has a page-specific context bar, allowing you to change threads or stack frames, which overrides the main context bar.



LOC

Thread: TID: 1b4 Context: SymTest!main+db0

Name	Type	Value
_argc	int	0x00000001
⊞ _argv	char**	0x000006fbff661210
⊞ hHandle	struct HINSTANCE__*	0x0000000000000000
c	int	0x000a2fb2
⊞ classA	class A	{. . .}
⊞ IntArray[10]	int	0x000006fbffb8fdf8
⊞ szText[256]	char	0x000006fbffb8fcf8 "Yes, we are here to test the local page"
⊞ pszText	char*	0x000006fbff660ef0
b	int	0x00000014
⊞ enumArray[10]	enum MYENUM	0x000006fbffb8fcc0
⊞ pClassB	class B*	0x000006fbff661160
⊟ myStruct	struct _MY_STRUCT	{. . .}
iMyInt	int	0x0000000d
dwMYDword	float	99.000000
⊞ szMtArray[27]	char	0x000006fbffb8fc98 "ABCDEFGHIJKLMNOPQRSTUVWXYZi"
⊟ BitField	struct _BIT_FIELD	{. . .}
ubit1	unsigned long	0x00000000
ubit2	unsigned long	0x00000000
ubit3	unsigned long	0x00000001
ubit4	unsigned long	0x00000000
i	int	0x0000001a
⊞ pns	struct tag_NewStructor*	0x000006fbff661040
a	int	0x000a2fb2
⊞ ppns	struct tag_PointerNewStructor*	0x000006fbff6610a0
enumData	enum MYENUM	ENUM_THREE (3)
⊟ ns	struct tag_NewStructor	{. . .}
m_intA	int	0x000a2fb2
m_intB	int	0x00000014

Figure 4-12. Visual SoftICE Locals Page

Concepts and Associated Commands

WL

The WL command opens the Locals page. If a Locals page is already open, WL activates that page.

Page Features

Live Mode Variable Tracking

When the Locals page is in live mode, Visual SoftICE displays the variable values in red when they change. You can expand the variable if it contains child-level components.

Open Only One Locals Page

You can only have one Locals page open.

Toggle Values Displayed Between Decimal and Hexadecimal

You can toggle displaying values in hexadecimal or decimal. To toggle display format for the values, right-click on the Value column and select **Hexadecimal** from the pop-up menu. When it is checked, Visual SoftICE displays values in hexadecimal. When it is unchecked, Visual SoftICE displays values in decimal.

Edit Variable Values

Visual SoftICE allows you to edit the value of a variable displayed in this page. To modify the value of a variable, double-click the value column of the variable to highlight the value, and enter the new value in the field. Press **Enter** or click any place in the Local page to save the value.

Select a Context

Visual SoftICE allows you to select a different stack frame, or context, whose variables you wish to display. To switch contexts for the current thread, click the Context combo-box to access the drop-down list of available stack frames for the current thread, and select the stack frame you want to change to. Visual SoftICE displays the variables for the selected context.

Select a Thread

Visual SoftICE allows you to select a different thread whose variables you wish to display. To select a different thread, click the Thread combo-box to access the drop-down list of available threads for the current process, and select the thread you want to change to. Visual SoftICE automatically updates the stack frames for the selected thread, and displays the latest variables for the first stack frame.

Copy and Drag

The page supports copying and dragging of variable names to other Visual SoftICE pages. How Visual SoftICE handles the variables once you drop them depends on the page you drop them into.

Customize the User Interface

There are multiple attributes of the page that you can customize. They are divided into two categories: per-page and application wide settings. Per-page attributes are always remembered by the workspace when you save it.

Per-Page Settings

There are no per-page settings available for the Locals page.

Application Wide Settings

- You can toggle the display format for variable values between decimal and hexadecimal.
- You can hide or display the page-specific context bar. To toggle display of the context bar on or off, right-click on the page and select View Context Bar from the pop-up menu.

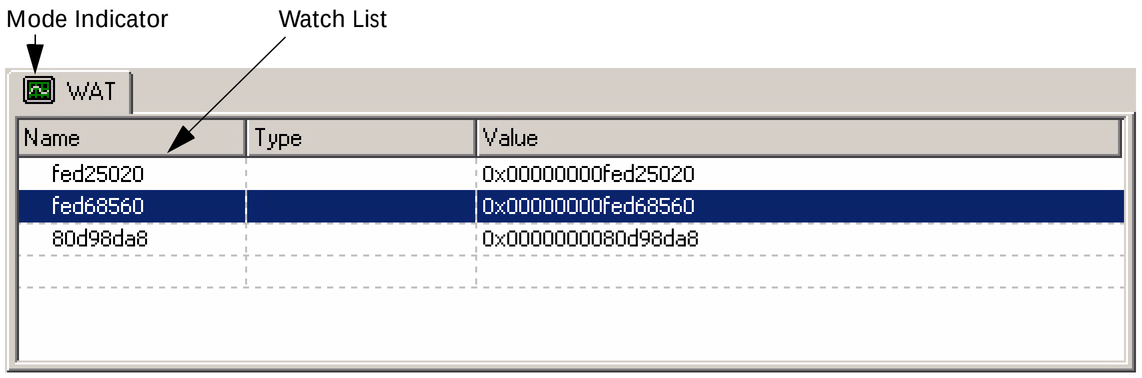
Print

The page supports printing, and print-previewing of its contents.

The Watch Page

The Watch page is a read-write container for displaying and editing any expression. You can enter any expression into this page and Visual SoftICE returns the resulting value. The Watch page displays the expression, result type, and value.

When the you switch the context in the Locals page, the Watch page also switches expression result types or values, remaining synchronized with the Locals page. When this happens, some expressions may not have a value at the current context even though they displayed a value previously.



Name	Type	Value
fed25020		0x00000000fed25020
fed68560		0x00000000fed68560
80d98da8		0x0000000080d98da8

Figure 4-13. Visual SoftICE Watch Page

Concepts and Associated Commands

WATCH

Use the WATCH command to display the results of expressions.

Page Features

Live Mode Variable Tracking

When the Watch page is in live mode, Visual SoftICE displays the variable values in red when they change. You can expand the variable if it contains child-level components.

Open Only One Watch Page

You can only have one Watch page open.

Toggle Values Displayed Between Decimal and Hexadecimal

You can toggle displaying values in hexadecimal or decimal. To toggle display format for the values, right-click on the Value column and select **Hexadecimal** from the pop-up menu. When it is checked, Visual SoftICE displays values in hexadecimal. When it is unchecked, Visual SoftICE displays values in decimal.

Edit Expression Values

Visual SoftICE allows you to edit the value of expressions displayed in this page. To modify the value of an expression, double-click the Value column of the expression to highlight the value, and enter the new value in the Edit field. Press <**Enter**> or click anyplace on the Watch page to save the new value.

Save Expressions

When you have a workspace file opened, Visual SoftICE will store all the expressions on the Watch page in your workspace file, when saved.

Add Expressions

Visual SoftICE allows you to add expressions directly into the Watch page, or by using the WATCH command from the Command page. To add a new expression directly into the Watch page, double-click on an empty row to make the **Name** field editable, and enter your new expression name into the field. Press <**Enter**> or click anyplace on the Watch page to save the new expression.

Modify Expressions

Visual SoftICE allows you to modify expressions directly on the Watch page. Visual SoftICE re-evaluates the expressions and displays the new result types and values in each column. To modify an expression, double-click the **Name** column of the expression you want to modify, and enter the new expression name in the Name field. Press <**Enter**> or click anyplace on the Watch page to save the value.

Delete or Clear Expressions

Visual SoftICE allows you to delete or clear expressions using the Watch page. To delete an expression, select the expression and press <**Delete**>, or right-click on the expression and select **Delete** from the pop-up menu.

Copy, Paste, Drag, and Drop

The Watch supports Copy, Paste, Drag, and Drop operations.

Customize the User Interface

There are multiple attributes of the page that you can customize. They are divided into two categories: per-page and application wide settings. Per-page attributes are always remembered by the workspace when you save it.

Per-Page Settings

There are no per-page settings available for the Watch page.

Application Wide Settings

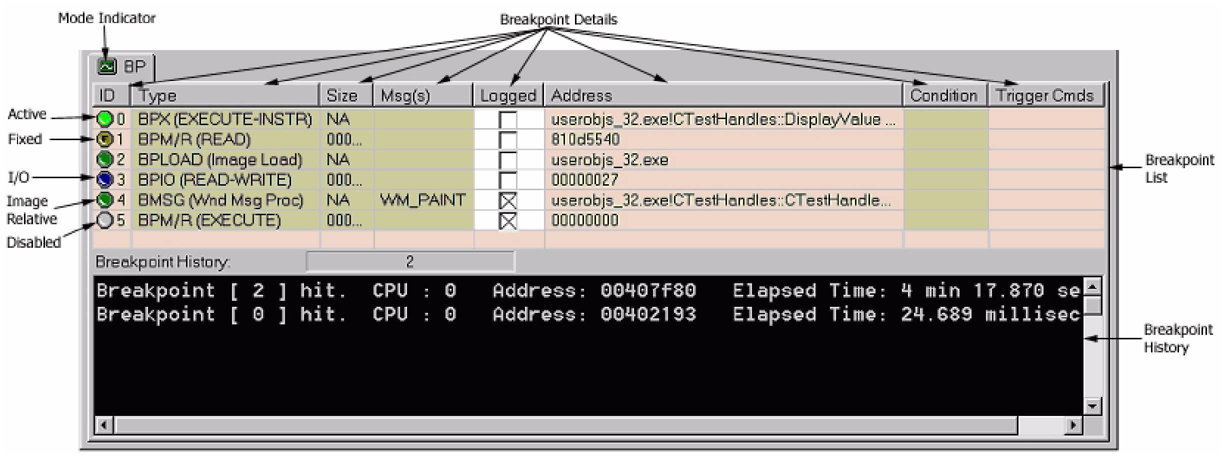
- You can toggle the display format for variable values between decimal and hexadecimal.

Print

The page supports printing, and print-previewing of its contents.

The Breakpoint Page

The Breakpoint page is used to maintain breakpoints. It displays statistics on the breakpoints, breakpoint history, and indicates if any of the breakpoints are currently active.



ID	Type	Size	Msg(s)	Logged	Address	Condition	Trigger Cmds
0	BPX (EXECUTE-INSTR)	NA		☐	userobjs_32.exe!CTestHandles::DisplayValue ...		
1	BPM/R (READ)	000...		☐	810d5540		
2	BPLOAD (Image Load)	NA		☐	userobjs_32.exe		
3	BPIO (READ-WRITE)	000...		☐	00000027		
4	BMSG (Wnd Msg Proc)	NA	WM_PAINT	☒	userobjs_32.exe!CTestHandles::CTestHandle...		
5	BPM/R (EXECUTE)	000...		☒	00000000		

Breakpoint History: 2

```
Breakpoint [ 2 ] hit.  CPU : 0   Address: 00407f80   Elapsed Time: 4 min 17.870 se
Breakpoint [ 0 ] hit.  CPU : 0   Address: 00402193   Elapsed Time: 24.689 millisec
```

Figure 4-14. Visual SoftICE Breakpoint Page

Concepts and Associated Commands

Breakpoint Types

Visual SoftICE supports the following types of breakpoints:

- Code Execution
- Interrupt
- Memory Location Access
- I/O
- Memory Range Access
- Image Load
- HWND and OS Application Message
- Deferred or Virtual

Use the New Breakpoint utility to set any of the breakpoints.

Display Breakpoint Information

There are two commands you can issue at the command line to display breakpoint information.

- You can display Breakpoint information by issuing the BL command.
- You can display the Breakpoint page by issuing the WB command.

Conditions

All breakpoint types, except the Image Load type, support an optional conditional IF statement as an additional trigger. If the condition evaluates to true when the breakpoint is encountered, it triggers the breakpoint (in the manner you specified) by stopping, logging the breakpoint without stopping, or executing a triggered command list.

Triggered Command Lists

All breakpoints support execution of an optional list of commands when the breakpoint is triggered. You can enter a single command, or list of commands, that Visual SoftICE will trigger upon hitting a successful breakpoint.

Global Break

You can configure Visual SoftICE to break upon loading image files by using the SET GLOBALBREAK command.

Page Features

Breakpoint Event History

The Breakpoint page allows you to display breakpoint related event messages as they are generated on the target. The breakpoint history window also supports the following functionality:

- You can save all of the current breakpoint history to a specified text file by right-clicking on the window and selecting **Save Output To File** from the pop-up menu.
- You can clear the breakpoint history by right-clicking on the window and selecting **Clear All** from the pop-up menu.

New / Edit Breakpoint

The Breakpoint page allows you to create a new breakpoint by right-clicking on the breakpoint list and selecting **New Breakpoint** from the pop-up menu. Visual SoftICE opens the New Breakpoint utility, which provides an easy interface to create breakpoints.

You can also edit an existing breakpoint with the New Breakpoint utility by double-clicking on a breakpoint in the breakpoint list.

Enable, Disable, Delete Breakpoint

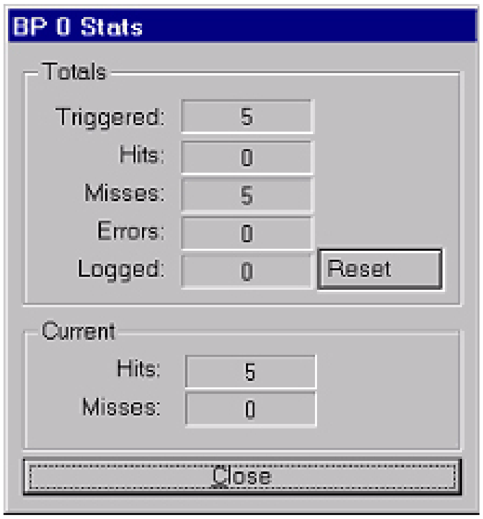
The Breakpoint page allows you to enable, disable, or delete a breakpoint by right-clicking on that breakpoint and selecting the desired operation (**Enable**, **Disable**, or **Delete**) from the pop-up menu. You can also enable, disable, or delete a breakpoint by issuing the following commands at the command line:

- You can enable a breakpoint by issuing the BE command with the appropriate breakpoint index.
- You can disable a breakpoint by issuing the BD command with the appropriate breakpoint index.
- You can delete a breakpoint by issuing the BC command with the appropriate breakpoint index.

Note: To get the appropriate breakpoint index, issue the BL command.

Display Breakpoint Statistics

The Breakpoint page allows you to display the statistics for a breakpoint by right-clicking on the address and selecting Display Statistics. The Breakpoint Statistics dialog displays the following information:


BP 0 Stats

Totals

Triggered:	5
Hits:	0
Misses:	5
Errors:	0
Logged:	0

Current

Hits:	5
Misses:	0

Figure 4-15. Breakpoint Statistics Dialog

- You can reset the breakpoint log count by clicking Reset.
- You can view breakpoint statistics from the command line by issuing the BSTAT command with the appropriate breakpoint index.

Note: To get the appropriate breakpoint index, issue the BL command.

Toggle Breakpoint Logging

The Breakpoint page allows you to toggle whether Visual SoftICE is only logging a breakpoint, or if it will stop the target. To toggle logging, right-click on a breakpoint and select **Toggle Logging** from the pop-up menu.

Open Only One Breakpoint Page

You can only have one Breakpoint page open.

Copy and Drag

The page only supports the Copy and Drag functions to retrieve its data.

- You can select any text in the breakpoint history window and copy it to the clipboard.
- You can drag any text from the breakpoint history window to any page that accepts dropped items.
- You can select any cell and copy its text to the clipboard.
- You can drag any cell to any page that accepts dropped items.

Note: Select a cell by placing the mouse cursor on the cell and then right-clicking on the element you wish to copy.

Customize the User Interface

There are multiple attributes of the page that you can customize. They are divided into two categories: per-page and application wide settings. Per-page attributes are always remembered by the workspace when you save it.

Per-Page Settings

- You can sort the Breakpoint list by any of the displayed columns by clicking on the column heading.
- You can reorganize and resize the individual columns displayed.

Application Wide Settings

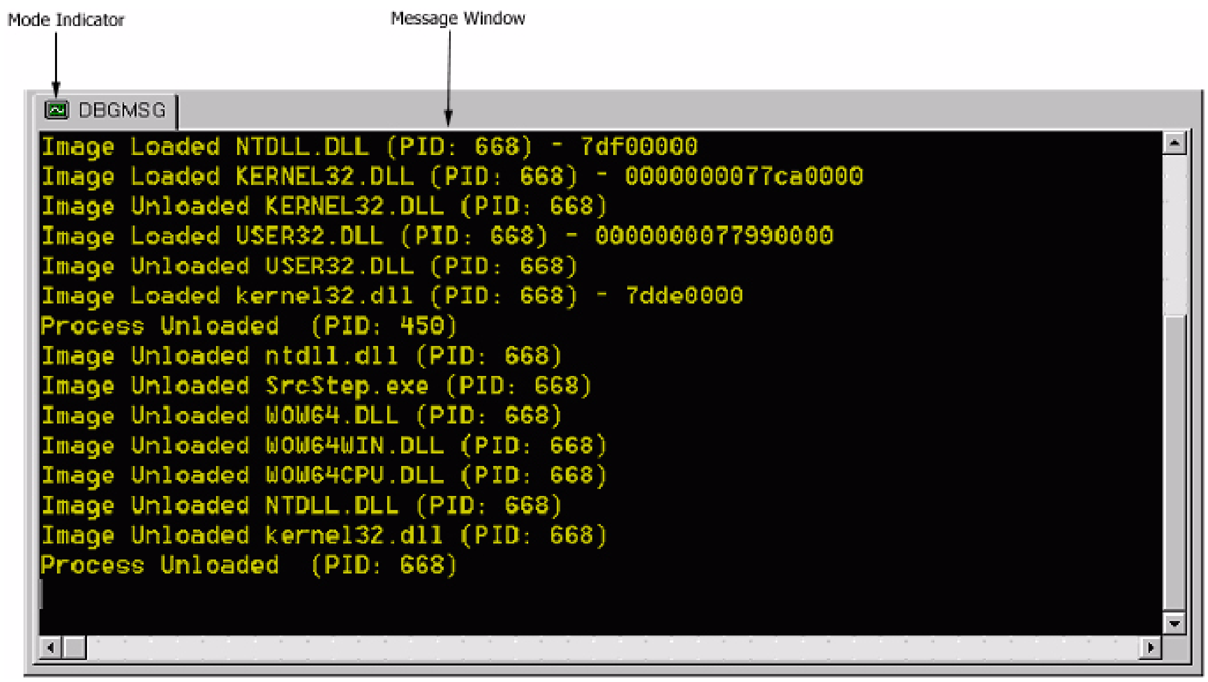
- You can control whether or not the page switches automatically to Live mode when a connection to a new target is established.

Print

The page supports printing, and print-previewing of its contents.

The Debug Message Page

The Debug Message page is a read-only receiver of information from the target and from other sources. The page is composed of a single window, which receives and displays text messages, and a pop-up utility controlling the filtering of the messages that are displayed in the window. There is no limit to the number of lines that can be retained by the page, and the color attributes of all lines, even those no longer visible on screen, are preserved.



```
Image Loaded NTDLL.DLL (PID: 668) - 7df00000
Image Loaded KERNEL32.DLL (PID: 668) - 0000000077ca0000
Image Unloaded KERNEL32.DLL (PID: 668)
Image Loaded USER32.DLL (PID: 668) - 0000000077990000
Image Unloaded USER32.DLL (PID: 668)
Image Loaded kernel32.dll (PID: 668) - 7dde0000
Process Unloaded  (PID: 450)
Image Unloaded ntdll.dll (PID: 668)
Image Unloaded SrcStep.exe (PID: 668)
Image Unloaded WOW64.DLL (PID: 668)
Image Unloaded WOW64WIN.DLL (PID: 668)
Image Unloaded WOW64CPU.DLL (PID: 668)
Image Unloaded NTDLL.DLL (PID: 668)
Image Unloaded kernel32.dll (PID: 668)
Process Unloaded  (PID: 668)
```

Figure 4-16. Visual SoftICE Debug Message Page

Concepts and Associated Commands

Setting Message Level

The Debug Message page can also display event messages from the target if you have set the Message Level to **On** or **Verbose**. For more information on controlling the display of messages in Visual SoftICE, refer to the SET MSGLEVEL command.

Filtering Messages

The Debug Message page supports the filtering of messages output to the page. Message filtering is controlled by the Debug Message Filter utility, which is available by right clicking on the page and selecting **Configure Filtering** from the pop-up menu.

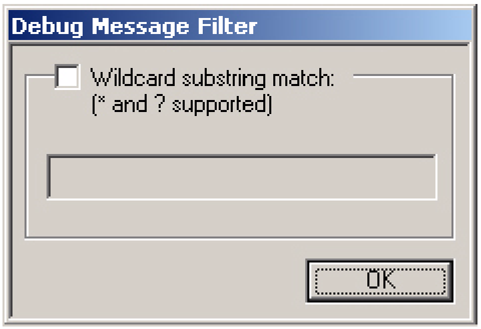


Figure 4-17. Debug Message Filter Utility

You can filter messages based upon a regular expression match of the message content. The regular expression filtering is limited to file system level matching criteria. That is, the ‘?’ and ‘*’ operators are the only supported matching wildcards. For example, to display the debug messages that your driver sends, prefix your DBGPRINT statements with a tag like `MYDRIVER:`, then set the regular expression message matching to `MYDRIVER:*`.

Open Any Number of Debug Message Pages

There are no restrictions on the number of Debug Message pages you can have open, so you can build a workspace with multiple views containing just the debug messages you are interested in, where you want them displayed. You may find this useful when debugging problems with a process on the target, as it allows non-intrusive observation of the behavior of the process.

Copy

The page only supports copy command to retrieve its data.

- You can select any text within the page and copy it to the clipboard.
- You can use **Select All** from the pop-up menu to highlight all the lines on the page before copying them to the clipboard.

Customize the User Interface

There are multiple attributes of the page that you can customize. They are divided into two categories: per-page and application wide settings. Per-page attributes are always remembered in the workspace when you save it.

Per-Page Settings

- ◆ You can filter the messages that are output to a particular page. For more information, refer to "Filtering Messages" on page 83.

Application Wide Settings

- ◆ You can select the font used for all the text, as well as the background and foreground color, for the following items: Normal text, Highlighted text, Title text, Dim text, Target Notifications, and Internal Notifications.

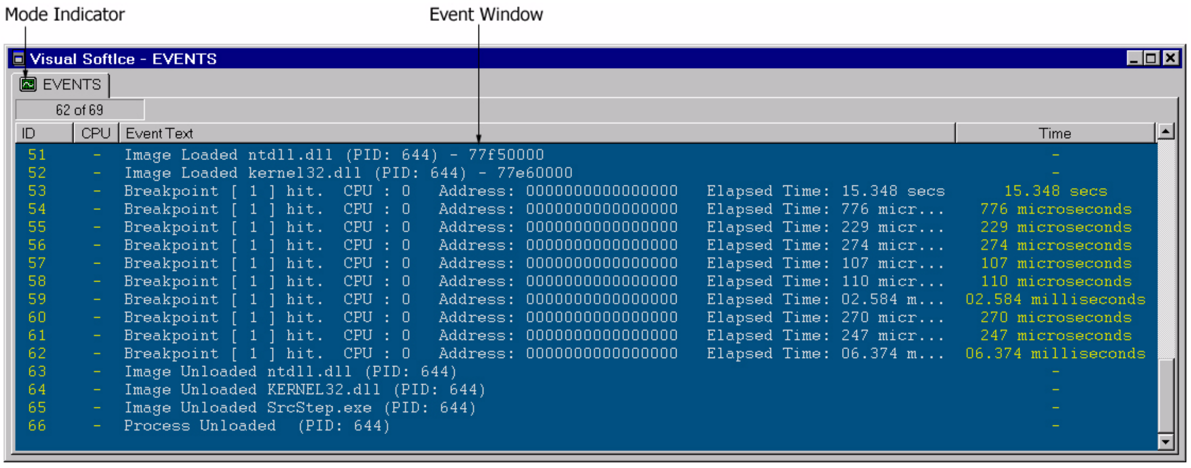
Print

The page supports printing, and print-previewing of its contents.

The Event Page

The Event page is a read-only receiver of information from the target and from other sources. The page is composed of a single window, which receives and displays events, and a pop-up utility controlling the filtering of the events that are displayed in the window. There is no limit to the number of lines that can be retained by the page, and the color attributes of all lines, even those no longervisible on screen, are preserved.

You may only have one Event page per target. If the connection to the target is lost, the collection of events is cleared from the page.



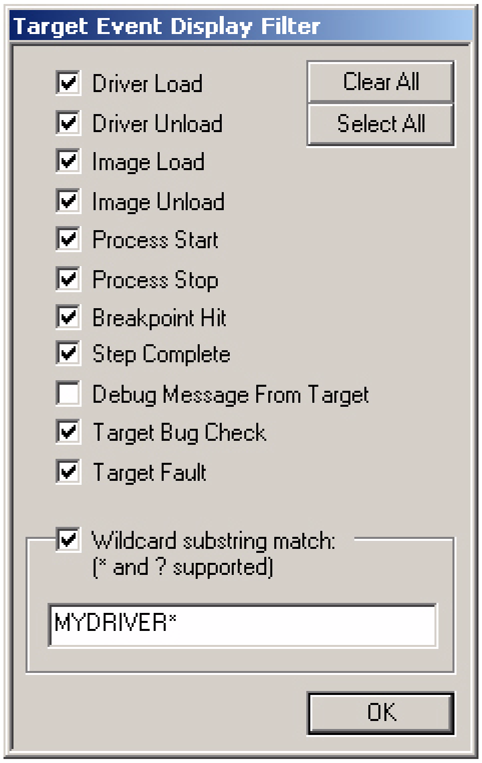
```
51   -   Image Loaded ntdll.dll (PID: 644) - 77f50000                                                     -
52   -   Image Loaded kernel32.dll (PID: 644) - 77e60000                                                  -
53   -   Breakpoint [ 1 ] hit.  CPU : 0   Address: 0000000000000000   Elapsed Time: 15.348 secs     15.348 secs
54   -   Breakpoint [ 1 ] hit.  CPU : 0   Address: 0000000000000000   Elapsed Time: 776 micr...   776 microseconds
55   -   Breakpoint [ 1 ] hit.  CPU : 0   Address: 0000000000000000   Elapsed Time: 229 micr...   229 microseconds
56   -   Breakpoint [ 1 ] hit.  CPU : 0   Address: 0000000000000000   Elapsed Time: 274 micr...   274 microseconds
57   -   Breakpoint [ 1 ] hit.  CPU : 0   Address: 0000000000000000   Elapsed Time: 107 micr...   107 microseconds
58   -   Breakpoint [ 1 ] hit.  CPU : 0   Address: 0000000000000000   Elapsed Time: 110 micr...   110 microseconds
59   -   Breakpoint [ 1 ] hit.  CPU : 0   Address: 0000000000000000   Elapsed Time: 02.584 m...  02.584 milliseconds
60   -   Breakpoint [ 1 ] hit.  CPU : 0   Address: 0000000000000000   Elapsed Time: 270 micr...   270 microseconds
61   -   Breakpoint [ 1 ] hit.  CPU : 0   Address: 0000000000000000   Elapsed Time: 247 micr...   247 microseconds
62   -   Breakpoint [ 1 ] hit.  CPU : 0   Address: 0000000000000000   Elapsed Time: 06.374 m...  06.374 milliseconds
63   -   Image Unloaded ntdll.dll (PID: 644)                                                              -
64   -   Image Unloaded KERNEL32.dll (PID: 644)                                                           -
65   -   Image Unloaded SrcStep.exe (PID: 644)                                                            -
66   -   Process Unloaded  (PID: 644)                                                                     -
```

Figure 4-18. Visual SoftICE Event Page

Page Features

Configuring Filtering

The Event page supports the filtering of events output to the page. Event filtering is controlled by the Target Event Display Filter utility, which is available by right clicking on the page and selecting **Configure Filtering** from the pop-up menu.



[✓] Driver Load
[✓] Driver Unload
[✓] Image Load
[✓] Image Unload
[✓] Process Start
[✓] Process Stop
[✓] Breakpoint Hit
[✓] Step Complete
[ ] Debug Message From Target
[✓] Target Bug Check
[✓] Target Fault

[✓] Wildcard substring match:
(* and ? supported)

MYDRIVER*

Figure 4-19. Target Event Display Filter Utility

- You can enable or disable specific Target Notification events, such as Driver Load, Driver Unload, or Debug messages.
- You can filter events based upon a regular expression match of the event content. The regular expression filtering is limited to file system level matching criteria. That is, the '?' and '*' operators are the only supported matching wildcards. For example, to display the debug messages that your driver sends, prefix your DBGPRINT statements with a tag like `MYDRIVER:`, then set the regular expression event matching to `MYDRIVER:*`.

Note: The Event page will always attempt to apply any filtering you implement to the entire event collection. With extremely large event collections this process can take some time, so if the collection is very large and the target is running, the Event page will halt the target while it synchronizes the event collection.

Open Only One Event Page

You can only have one Event page open.

Copy

The page only supports copy command to retrieve its data.

- You can select any text within the page and copy it to the clipboard.
- You can use **Select All** from the pop-up menu to highlight all the lines on the page before copying them to the clipboard.

Customize the User Interface

There are multiple attributes of the page that you can customize. They are divided into two categories: per-page and application wide settings. Per-page attributes are always remembered in the workspace when you save it.

Per-Page Settings

- You can filter the events that are output to the page. For more information, refer to "Configuring Filtering" on page 85.
- You can display or hide the event counter at the top of the Event page by right-clicking on the page and selecting **View Counter** to toggle it.

Application Wide Settings

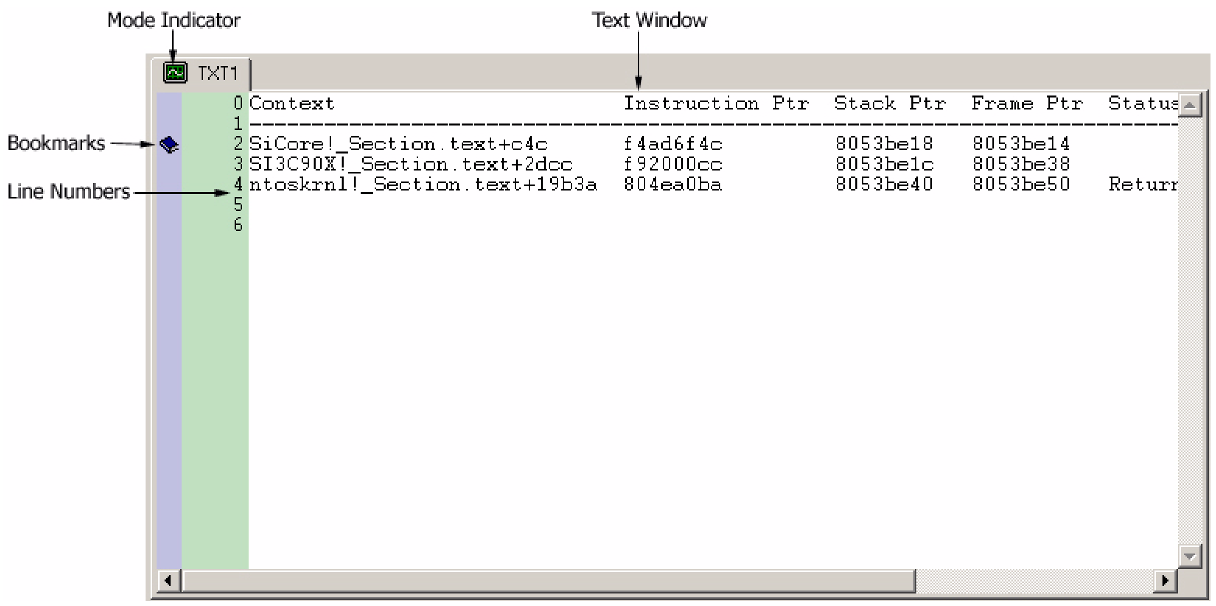
- You can select the background and foreground color for event text and all other columns.
- You can configure an **Auto Save Log Filename** from the Global settings tab by right-clicking on the page and selecting **Page Preferences** from the pop-up menu.
- You can configure the page to **Auto Save Log on Disconnect** from the Global settings tab by right-clicking on the page and selecting **Page Preferences** from the pop-up menu.
- You can configure the page to **Warn on Disconnect Data Loss** from the Global settings tab by right-clicking on the page and selecting **Page Preferences** from the pop-up menu.

Print

The page supports printing, and print-previewing of its contents.

The Text Scratch Page

The Text Scratch page is used to capture redirected output from various commands that can be executed in the Command page. The Text Scratch page is also a convenient place to type, cut, and paste output to.



```
0 Context                        Instruction Ptr   Stack Ptr   Frame Ptr   Status
1 ------------------------------------------------------------------------------
2 SiCore!_Section.text+c4c       f4ad6f4c          8053be18    8053be14
3 SI3C90X!_Section.text+2dcc     f92000cc          8053be1c    8053be38
4 ntoskrnl!_Section.text+19b3a   804ea0ba          8053be40    8053be50    Returr
5
6
```

Figure 4-20. Visual SoftICE Text Scratch Page

Concepts and Associated Commands

Command Redirection

The page supports command redirection from a command page. This allows output (such as from a kd extension, or a script file) to be captured and saved. Refer to Command Redirection in the main help file for more information.

Page Features

Save and Clear Output

You can save all of the output on this page to a specified text file by right-clicking on the page and selecting **Save Output To File** from the pop-up menu. You can clear all of the output on this page by right-clicking on the page and selecting **Clear All** from the pop-up menu.

Open Any Number of Text Scratch Pages

There are no restrictions on the number of Text Scratch pages you can open. You may find this useful when capturing the output from a specific command. Simply specify /> TXT as the page to redirect the output to. If there is no existing Text Scratch page with that name, a new page is created with the command as its name.

Find Text

The Text Scratch page provides a utility for finding a text string value within the page.

- You can search by attempting to match the whole word, only or partial string.
- You can further narrow a search by attempting to match the case of the specified search string.

Go To a Specific Line

You can right-click on the Text Scratch page and select **Go To Line** from the pop-up menu to advance the cursor to any line of text currently contained in the Text Scratch page.

Copy, Paste, Drag, and Drop

The page supports many ways of retrieving its data.

- You can select any text within the page and copy it to the clipboard.
- You can choose **Select All** to select all the lines on the page and copy them to the clipboard.
- You can paste any text from the clipboard into the page.
- You can drag items from other pages and drop them onto the page. For example, you can drag multiple registers from a register page, and Visual SoftICE inserts a table of names and values into the page.

Append a File

You can right-click on the Text Scratch page and select **File Append** from the pop-up menu to append the contents of any text file to the Text Scratch page contents.

View Line Numbers

You can right-click on the Text Scratch page and select **View Line Numbers** from the pop-up menu to display line numbers for every line of text currently contained in the Text Scratch page.

View Bookmarks

You can right-click on the Text Scratch page and select **View Bookmarks** from the pop-up menu to display the bookmarks that may be set for the lines of text currently contained in the Text Scratch page.

Customize the User Interface

There are multiple attributes of the page that you can customize. They are divided into two categories: per-page and application wide settings. Per-page attributes are always remembered by the workspace when you save it.

Per-Page Settings

There are no per-page settings for this page.

Application Wide Settings

- You can use the **Save/Restore Contents** in Workspace settings to configure Visual SoftICE to automatically save any text contained in the Text Scratch pages when you save your workspace. Each Text Scratch page saves and restores its own content.
- You can select the font, foreground color, and background color used for the text on the page.

Print

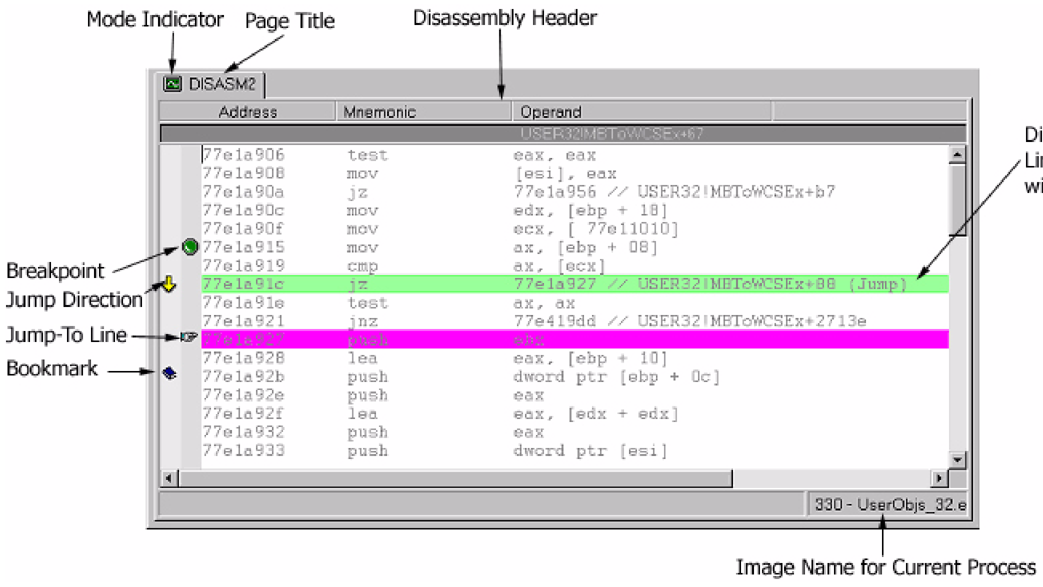
The page supports printing, and print-previewing of its contents.

The Disassembly Page

The Disassembly page is a read-only container to disassemble target memory. By selecting different modes, the page can be used as a live display, for viewing disassembly code on a specified target memory address, or for capturing a snapshot of a chunk of disassembled lines. Use the Disassembly page to:

- Display a chunk of disassembled lines, including current Instruction Pointer (IP), or at a specific target address.
- Set, remove, enable, and disable breakpoints on target addresses.
- Set and remove bookmarks on target addresses.
- Step through instructions.
- Perform **Go To** and **Run To** by target address or function name, current IP, and bookmarks.
- Search for strings inside the current view.
- Print out the page.

You can open a disassembly page by clicking the Disassembly page icon on the toolbar. Visual SoftICE can also automatically open a disassembly page while stepping or receiving WC or T commands if it cannot find a matching source file for the current IP. The automatic behavior of this page is controlled by **AutoFocusOpen** setting under **Source/Disasm Page** on the Global Settings tab of the Preferences dialog.



```
Address          Mnemonic      Operand
                       USER32!MBToWCSEx+67
77e1a906         test          eax, eax
77e1a908         mov           [esi], eax
77e1a90a         jz            77e1a956 // USER32!MBToWCSEx+b7
77e1a90c         mov           edx, [ebp + 18]
77e1a90f         mov           ecx, [ 77e11010]
77e1a915         mov           ax, [ebp + 08]
77e1a919         cmp           ax, [ecx]
77e1a91c         jz            77e1a927 // USER32!MBToWCSEx+88 (Jump)
77e1a91e         test          ax, ax
77e1a921         jnz           77e419dd // USER32!MBToWCSEx+2713e
77e1a927         push          ebx
77e1a928         lea           eax, [ebp + 10]
77e1a92b         push          dword ptr [ebp + 0c]
77e1a92e         push          eax
77e1a92f         lea           eax, [edx + edx]
77e1a932         push          eax
77e1a933         push          dword ptr [esi]
                                                   330 - UserObjs_32.e
```

Figure 4-21. Visual SoftICE Disassembly Page

Associated Commands

T

The T command steps one instruction when the target is stopped. Visual SoftICE searches through the symbols to locate a file containing the current instruction pointer (IP). If it finds the file, and a source page for that file is already available, Visual SoftICE updates the existing source page. If it finds the file and no source page for that file is open, Visual SoftICE creates a new source page.

If Visual SoftICE does not find a source file containing the current IP, and a live disassembly page for the current IP is available, it updates that page. If no live disassembly page is available, Visual SoftICE opens a new disassembly page.

WC

When you type WC [address], Visual SoftICE searches through the symbols to locate the source file containing the specified address. If Visual SoftICE finds the source file, and a source page for the file is already open, it brings the opened page forward. If the source file is not already open, Visual SoftICE creates a new source page for the file and brings it forward.

If the address is not contained in any source file, Visual SoftICE creates a disassembly page and brings it forward.

Page Features

Open Any Number of Disassembly Pages

There are no restrictions on the number of Disassembly pages you can open.

Mode Selection

Page modes have some specific affects to the Disassembly page:

- **Live Mode -** If the target is connected and stopped, it automatically updates the display to include the disassembly line for current IP upon target events. Instruction stepping can be performed.
- **Manual Mode -** The display is not updated upon target events. You can choose to display a chunk of disassembly lines for a specific target address. Instruction stepping is not allowed.
- **Snapshot Mode -** The page is only used to view the current chunk of disassembly lines. It will not update the contents upon target events. Instruction stepping is not allowed.

For more detailed information on Page Modes and how they work in Visual SoftICE, refer to “Page Modes” on page 23.

Breakpoints

When the master is connected to a target, you can set breakpoints on the disassembly page at any viewable target address. You can set breakpoints by locating the cursor on the line and using one of the following methods:

- Selecting **Set Breakpoint** under the Breakpoint group on the pop-up menu for the page
- Pressing the F9 function key
- Clicking the Breakpoint button on the main toolbar
- Right-clicking in the second column to the left of the code to access the Breakpoint pop-up menu.

Once you set a breakpoint, you can disable or remove it via one of the following methods:

- Selecting the applicable options from the breakpoint pop-up menu
- Using the function keys
- Using the buttons on the main toolbar

Any breakpoint changes made elsewhere in the system are synchronously updated on the disassembly page.

Instruction Stepping

When the master is connected to a target that is stopped at an address, the disassembly line of the current IP is highlighted. You can choose to step-over, step-into, or step-out of the current instruction. Use the options under the Debug group of the pop-up menu for the page, the function keys as defined in the Preference settings, or the stepping buttons on the main toolbar, to perform your stepping functions.

If the current instruction has a jump-to address to branch to, the yellow arrow on the left of the highlighted line changes its direction to up or down, depending on the branch address location, indicating that the next instruction will be at a specific address. To view the jump-to line, you can click the arrow, press <**Ctrl**><**J**>, or select **Go To Jump** from the pop-up menu for the page. Visual SoftICE highlights the jump-to line when it displays it.

You can always view the current IP line by pressing <**Ctrl**><**I**>, or by selecting **Go To Current IP** from the pop-up menu for the page.

To run-to and stop at a specified line, you can set the cursor to the line where you want to stop and press **<Ctrl><R>**, or select **Run To Here** from the pop-up menu for the page.

Bookmarks

You can set bookmarks at any viewable disassembly line, but only if the target is connected. Bookmarks in the disassembly page are represented by a target address. To set a bookmark:

- Right-click on the line and select **Set Bookmark** from the pop-up menu for the page
- Right-click in first column to the left of the code and select **Set Bookmark** from the pop-up menu

To remove bookmarks:

- You must have focus on a line with a bookmark on it. You can then remove the bookmark for the line, or all bookmarks in the file, by right-clicking and selecting the applicable option from the page's pop-up menu.
- You can locate the cursor in the first column to the left of the code and right-click to access the Bookmark pop-up menu.

You can view the bookmark list in the Go To dialog. From that dialog you can view or run-to the specified bookmark via the **Go To** and **Run To** buttons.

Visual SoftICE automatically removes all bookmarks when the target is disconnected.

Go To

You can use the Go To dialog to:

- View or run-to a disassembly line using its target address or function name
- View a line containing the current IP
- View or run-to a selected bookmarked line

Go To Current IP

You can right-click on the page and select **Go To Current IP** from the pop-up menu to view the line containing the current IP.

Go To Jump

When you are on a jump statement, you can right-click on the page and select **Go To Jump** from the pop-up menu to move to the line where the jump will go.

Run To Here

You can right-click on the page and select **Run To Here** from the pop-up menu, and Visual SoftICE will run from the current IP to the line that currently contains the cursor.

Set Next Statement

The Set Next Statement command moves the IP to the next statement. You can execute the Set Next Statement command via one of the following methods:

- Right-clicking to access the pop-up menu for the page
- Pressing <**Ctrl**><**Shift**><**F10**>
- Selecting the command from the Source Page drop-down menu
- Clicking the **Set Next Statement** toolbar button

If you attempt to execute the Set Next Statement command on a specified line with an address outside of the function scope of the current IP, and the warning level is not set to off, then Visual SoftICE displays a warning message asking you to confirm that you want to set the IP to the new address.

Find

You can use the Find dialog to search for a string in the current disassembly page.

Copy

The page only supports copy command to retrieve its data.

- You can select any text within the page and copy it to the clipboard.
- You can use **Select All** from the pop-up menu to highlight all the lines on the page before copying them to the clipboard.

AutoFocus

You can use the AutoFocusOpen setting to customize the behavior of the disassembly page when a target event occurs. If you select **Source or Disassembly** or **Source or Disassembly – No Focus**, then when Visual SoftICE can locate a source file for the symbols, it creates a source page. If Visual SoftICE cannot locate a source file, it creates a disassembly page.

For more information on available settings and how they behave, refer to AutoFocusOpen Settings in the on-line help.

Customize the User Interface

There are multiple attributes of the page that you can customize. They are divided into two categories: per-page and application wide settings. Per-page attributes are always remembered in the workspace when you save it.

Per-Page Settings

- There are no per-page settings for the Disassembly page.

Application Wide Settings

- You can use the AutoFocusOpen settings to control whether Visual SoftICE opens a new page when none exists (or brings an existing page to the top of the pad it's on), and places input focus on that page.
- You can configure Visual SoftICE to place new Disassembly pages on the largest pad with the **New Src/Disasm Pages to Largest Pad** setting.
- You can configure Visual SoftICE to place new Disassembly pages on a named pad with the **New Src/Disasm Pages to NAMED Pad** setting.
- You can configure Visual SoftICE to put Disassembly pages into live mode upon connection to a live target with the **Live Mode on Connection** setting.
- You can configure Visual SoftICE to restrict the minimum column width on the Disassembly page to the actual field length using the **Minimum Column Widths** setting.
- You can configure Visual SoftICE to display or hide the machine language (op-code) bytes column by right clicking on the Disassembly page, selecting **Disassembly**, and toggling **Show Op-Code Bytes** in the sub-menu. You can also toggle this setting under the **Disassembly** element in the **Global Settings** tab of the Preferences dialog.

- You can configure Visual SoftICE to display or hide the address column by right clicking on the Disassembly page, selecting **Disassembly**, and toggling **Show Addresses** in the sub-menu. You can also toggle this setting under the **Disassembly** element in the **Global Settings** tab of the Preferences dialog.
- You can configure Visual SoftICE to display or hide the instruction template field (on IA-64 only) by right clicking on the Disassembly page, selecting **Disassembly**, and toggling **Show Instruction Template** in the sub-menu. You can also toggle this setting under the **Disassembly** element in the **Global Settings** tab of the Preferences dialog.

Print

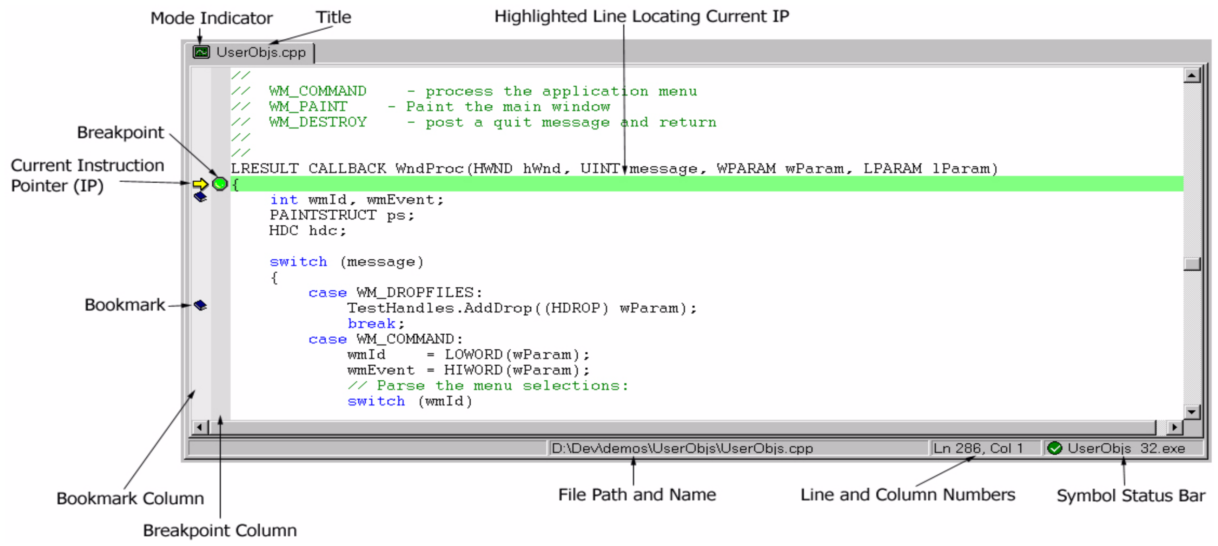
The page supports printing, and print-previewing of its contents.

The Source Page

The Source page is a read-only container for displaying source code files. This page tracks the current context on the target, and whether its source file is associated with an image of the current process when symbol information is available. Once the symbol information for the source file is available, this page can be used to do the following:

- Disassemble the entire source file, or individual source lines that have associated disassembly code
- Set, remove, enable, and disable breakpoints
- Set and remove bookmarks
- Step through source code or associated disassembly
- Perform Go To and Run To by line number, target address, function name, current Instruction Pointer (IP), and bookmarks
- View or hide line numbers
- Perform string searches
- Print out the page

You can open a source file by using the FILE command in the command page, or by clicking the Source page icon in the toolbar. You can have any number of source pages open at any given time. The automatic behavior of this page is controlled by **AutoFocusOpen** setting under **Source/ Disasm Page** on the Global Settings tab of the Preferences dialog.



```
//
//  WM_COMMAND    - process the application menu
//  WM_PAINT    - Paint the main window
//  WM_DESTROY    - post a quit message and return
//
//
LRESULT CALLBACK WndProc(HWND hWnd, UINT message, WPARAM wParam, LPARAM lParam)
{
    int wmId, wmEvent;
    PAINTSTRUCT ps;
    HDC hdc;

    switch (message)
    {
        case WM_DROPFILES:
            TestHandles.AddDrop((HDROP) wParam);
            break;
        case WM_COMMAND:
            wmId    = LOWORD(wParam);
            wmEvent = HIWORD(wParam);
            // Parse the menu selections:
            switch (wmId)
```

```
D:\Dev\demos\UserObjs\UserObjs.cpp    Ln 286, Col 1    UserObjs 32.exe
```

Figure 4-22. Visual SoftICE Source Page

Associated Commands

FILE

The FILE command is often useful when setting a breakpoint on a line that has no associated symbol. For information on using FILE to set a breakpoint refer to the Command Reference.

- If you specify a file name, that file becomes the current file and Visual SoftICE creates a new source page for it.
- If you do not specify a file name, Visual SoftICE opens a source page for the file that contains current IP. If no source file for the current IP is available, Visual SoftICE displays a message stating that no file can be opened for the current IP.
- If you specify the * (asterisk), Visual SoftICE displays all the files in the current symbol table.

LOAD

The LOAD command loads symbols for a specified image-name. If you specify the * wildcard in place of image-name, Visual SoftICE opens the Symbol Files utility.

SS

The SS command searches all the opened source pages for a specified string, starting from an optional line number. If Visual SoftICE finds the string in a source page, it highlights the first matching string together with the line it's on. The string may be found in multiple open source pages.

T

The T command steps one instruction when the target is stopped. Visual SoftICE searches through the symbols to locate a file containing the current instruction pointer (IP). If it finds the file, and a source page for that file is already available, Visual SoftICE updates the existing source page. If it finds the file and no source page for that file is open, Visual SoftICE creates a new source page.

If Visual SoftICE does not find a source file containing the current IP, and a live disassembly page for the current IP is available, it updates that page. If no live disassembly page is available, Visual SoftICE opens a new disassembly page.

WC

When you type WC [address], Visual SoftICE searches through the symbols to locate the source file containing the specified address. If Visual SoftICE finds the source file, and a source page for the file is already open, it brings the opened page forward. If the source file is not already open, Visual SoftICE creates a new source page for the file and brings it forward.

If the address is not contained in any source file, Visual SoftICE creates a disassembly page and brings it forward.

Page Features

Open Only One Source Page Per File

You can only open one Source page per file.

File or Line Disassembly

When symbol information is available, Visual SoftICE displays the process ID and image name on the status bar at the bottom right of the page. When this is the case, the Disassemble File and Disassemble Line options become enabled under the Disassembly group of the pop-up menu for the page.

- You can view or hide the disassembly code for the entire source file by toggling the Disassemble File option on or off.
- You can view or hide a single source line by toggling the Disassemble Line option on or off.

When disassembly code is visible in the source page, the header bar appears at the top of the page. You can choose to view or hide the Address and Op-code fields of the disassembly lines by toggling the **Show Address** and **Show Opcode Bytes** options under the Disassembly group of the pop-up menu for the page. You can also use the Page Preference option to set the default mode for header display.

If the target is an IA64 CPU, Visual SoftICE displays the Predicate/Prefix field on the disassembly header in addition to the other fields.

Source and Instruction Stepping

When the master is connected to a target that is stopped at the address contained in the source page, Visual SoftICE highlights the line of the current instruction pointer (IP). You can choose to step-over, step-into, or step-out of the current source line.

If the page is in the mixed mode in which disassembly lines are available, you can elect to do source stepping or instruction stepping. Use the options under the Debug group of the pop-up menu for the page, the function keys as defined in the Preference Settings, or the stepping buttons on the main toolbar, to perform your stepping functions.

If you are source stepping and attempt a step-into when the related source or .asm file is available, Visual SoftICE opens the file and automatically highlights the line containing current IP. If no source file is available, the step-into action behaves the same as step-over.

If you are instruction stepping in the mixed mode, Visual SoftICE highlights the disassembly line for the current IP. If the current instruction has a jump-to address to branch to, the yellow arrow on the left of the highlighted line changes its direction to up or down, depending on the branch address location, indicating that the next instruction will be at a specific address. To view the jump-to line, you can click the arrow, press <**Ctrl**><**J**>, or select **Go To Jump** from the pop-up menu for the page. Visual SoftICE highlights the jump-to line when it displays it.

You can always view the current IP line by pressing <**Ctrl**><**I**>, or by selecting **Go To Current IP** from the pop-up menu for the page.

To run-to and stop at a specified source or disassembly line, you can set the cursor to the line where you want to stop and press <**Ctrl**><**R**>, or select **Run To Here** from the pop-up menu for the page.

Breakpoints

With symbol information available, you can set breakpoints at a source line containing actual code via one of the following methods:

- Selecting Set Breakpoint under the Breakpoint group on the pop-up menu for the page
- Pressing the F9 function key
- Clicking the Breakpoint button on the main toolbar
- Locating the cursor in the second column to the left of the code, and right-clicking to access the Breakpoint pop-up menu.

If you attempt to set a breakpoint on a line that does not contain actual code, Visual SoftICE sets the breakpoint at the next line containing code. If a line is in the source/disassembly mixed mode, you can set the breakpoint at any of the visible disassembly lines.

Once you set a breakpoint, you can disable or remove it via one of the following methods:

- Selecting the applicable options from the breakpoint pop-up menu
- Using the function keys
- Using the buttons on the main toolbar

Any breakpoint changes made elsewhere in the system are synchronously updated on the source page.

View Line Numbers

You can right-click on the Source page and select **View Line Numbers** from the pop-up menu to display line numbers for every line currently contained in the Source page.

Bookmarks

In the source page, you can set a bookmark at any line in the source file, but you cannot set bookmarks at the disassembly lines in mixed mode. Bookmarks in the source page are represented by line numbers. You can view or hide line numbers by toggling the **Show Line Numbers** option on and off from the pop-up menu for the page, or by using <**Ctrl**><**L**> to toggle the selection.

To remove bookmarks:

- You must have focus on a line with a bookmark on it. You can then remove the bookmark for the line, or all bookmarks in the file, by right-clicking and selecting the applicable option from the page's pop-up menu.
- You can locate the cursor in the first column to the left of the code and right-click to access the Bookmark pop-up menu.

You can view the bookmark list in the Go To dialog. From that dialog you can view or run-to the specified bookmark via the **Go To** and **Run To** buttons.

Go To

You can use the Go To dialog to:

- View or run-to a source line using its target address or function name
- View or run-to a line using its line number
- View a line containing the current IP
- View or run-to a selected bookmarked line

Go To Current IP

You can right-click on the page and select **Go To Current IP** from the pop-up menu to view the line containing the current IP.

Go To Jump

When you are on a jump statement, you can right-click on the page and select **Go To Jump** from the pop-up menu to move to the line where the jump will go.

Run To Here

You can right-click on the page and select **Run To Here** from the pop-up menu, and Visual SoftICE will run from the current IP to the line that currently contains the cursor.

Set Next Statement

The Set Next Statement command moves the IP to the next statement. You can execute the Set Next Statement command via one of the following methods:

- Right-clicking to access the pop-up menu for the page
- Pressing **<Ctrl><Shift><F10>**
- Selecting the command from the Source Page drop-down menu
- Clicking the **Set Next Statement** toolbar button

If you attempt to execute the Set Next Statement command on a specified line with an address outside of the function scope of the current IP, and the warning level is not set to off, then Visual SoftICE displays a warning message asking you to confirm that you want to set the IP to the new address.

Find

You can use the Find dialog to search for a string in the current source page.

Copy

The page only supports copy command to retrieve its data.

- You can select any text within the page and copy it to the clipboard.
- You can use **Select All** from the pop-up menu to highlight all the lines on the page before copying them to the clipboard.

AutoFocus

You can use the AutoFocusOpen setting to customize the behavior of the source page when a target event occurs. If you select **Source or Disassembly** or **Source or Disassembly – No Focus**, then when Visual SoftICE can locate a source file for the current IP, it creates a source page. If Visual SoftICE cannot locate a source file, it creates a disassembly page.

For more information on available settings and how they behave, refer to AutoFocusOpen Settings in the on-line help.

Customize the User Interface

There are multiple attributes of the page that you can customize. They are divided into two categories: per-page and application wide settings. Per-page attributes are always remembered in the workspace when you save it.

Per-Page Settings

- There are no per-page settings for the Source page.

Application Wide Settings

- You can use the AutoFocusOpen settings to control whether Visual SoftICE opens a new page when none exists (or brings an existing page to the top of the pad it's on), and places input focus on that page.
- You can configure Visual SoftICE to place new Source pages on the largest pad with the **New Src/Disasm Pages to Largest Pad** setting.
- You can configure Visual SoftICE to place new Source pages on a named pad with the **New Src/Disasm Pages to NAMED Pad** setting.

Print

The page supports printing, and print-previewing of its contents.

Chapter 5

Visual SoftICE Symbol Management

Visual SoftICE Symbol Management

Due to the Visual SoftICE dual machine nature, symbol management is very different from SoftICE, the single machine system debugger. In some ways, symbol management is simplified, allowing dynamic table loading and unloading, as well as automatic symbol retrieval. In other ways (for those familiar with SoftICE) some preparation is required, in that the system must be prepared properly to provide access to data that is normally packaged up in the SoftICE NMS file creation process.

Where to Put the Symbols

With Visual SoftICE, the master loads and manages all symbols, exports, and image data (PE headers, etc...) locally. This can be on a network drive, or via a company MS Symbol Server, but the clear distinction to understand is that symbols are used on the master side, not on the target.

Note: The one exception to this general rule is providing data to the target core debugger about OS structures and locations for boot mode operation. This is handled via a dedicated target side driver for the target platform.

What Symbols are Supported

Visual SoftICE supports PDB files (version 5 or later), and DBG files, for each target platform and OS it supports. This includes support and identification of images that have been optimized (usually kernel components), support for DDK compiler/linker generated symbols, and application generated symbols. Modern compilers and linkers support features like incremental compilation, incremental linking, and “edit and continue”. These useful rapid application development enhancements complicate symbol file information, and sometimes can generate files with missing data, or data that is out of date.

Visual SoftICE also supports user-defined symbols, of address or value type. These are considered global in nature, are always accessible, and always loaded. To learn more about user-defined symbols, refer to the NAME command in the Visual SoftICE Command Reference.

Images — Why you need access to them

Symbols are only half the story, and can be considered meta-data about the image that will be loaded into memory by the operating system. The image itself contains information about the type and location of accurate symbol files generated for it, section layout (memory partitioning within the image), and export data (what publicly visible functions there are, and their locations). Depending on the target architecture, and the type of image, this file may also contain UNWIND data (useful stack-walking information on IA64 and x86-64 platforms) and other useful information.

Local Copies of Images

Keeping local copies of images you are debugging on the target is highly recommended. This allows the master to look things up locally, instead of attempting to retrieve data from the target over a transport. Also, keep in mind that any image or parts of its memory may be paged out on the target. This generally happens just when you need to access the paged out data on a stopped machine.

Additionally, when preloading symbols to set breakpoints in drivers or applications that have not yet been loaded into memory, the section table of the image is required to fix up image relative addresses (especially with symbols that contain OMAP data on x86 platforms).

Keeping a local copy accessible on the master allows access to the necessary data, regardless of the targets current state.

Note: Refer to the FGET command in the Visual SoftICE Command Reference for retrieving files from the target to the master.

Tables – Active Table, Loading, and Unloading Commands

Visual SoftICE, like SoftICE, uses the metaphor of a table to manage symbols to images. The TABLE command, without parameters, will display the currently loaded symbol tables, and highlight the current or active table. Entering the TABLE command with the verbose switch (-v) shows that there is a table entry for every image in the active/current process on the target.

Active Table

Many symbol commands will restrict their search to the active table for performance reasons, unless user input specifically references another table, or indicates a request to search all known tables (refer to the following commands in the Visual SoftICE Command Reference: FILE, SYM, TYPES).

The active or current table will automatically change as the user changes the current UI context, either through the context bar, or the ADDR command.

The active table can be directly changed by entering the TABLE command with the name of one of the loaded tables.

Loading and Unloading

The following commands and actions apply to dynamic tables (those symbol resources that are needed on demand, and that the system can unload if not necessary).

- ◆ You can load a table for any in-memory image by use of the LOAD command. You can optionally specify the exact path to the file (otherwise the MS Symbol Server and/or search paths are used). The file will only be loaded for images that exist, and for which a table has not yet been loaded.
- ◆ You can force a RELOAD of a table (equivalent to an UNLOAD followed by a LOAD command) for any image that currently exists.
- ◆ You can force a table to be unloaded by issuing the UNLOAD command.

The following commands and actions apply to persistent tables (those symbol resources that you have hand loaded, and stay loaded until you unload them by hand).

- You can persistently load a table by use of the ADDSYM command.
- You can remove a persistent table by use of the DELSYM command.

Matching

Tables that are loaded (dynamically, or persistently) have matching criteria to images in the operating system. The match between a given symbol file and an image is achieved differently depending on the image, the type of symbol file loaded, and by user image match settings.

Images have a name, extension, target architecture type (64,32,16bit), signature type (PDB6, PDB7, etc...), timestamp, and in some cases, age and/or GUID data for reference to symbol files. The image data will be retrieved and compared to the symbol file to be loaded, using the IMAGEMATCH setting, to conclude if the symbol file meets the minimum criteria of a match.

Differences in target architecture are not allowed:

- 32bit symbols that match a 64bit image of the same name will not be loaded.
- Symbols for a 64bit Itanium of the same name of that for a 64bit x86-64 platform will not be loaded.

The IMAGEMATCH setting (refer to SET IMAGEMATCH in the Visual SoftICE Command Reference) can be **BEST** (default), or **EXACT**.

- When this setting is **BEST**, symbols that are out of date, mismatched on timestamp, GUID, signature, or only partially match the image name and extension, are still allowed. This supports having symbols that are “close” but not exact, which is sometimes good enough for debugging.
- When **BEST** is not good enough, (or confusing), **EXACT** mode is available, which means that image name, extension, signature, and version (age, GUID, timestamp) must match exactly, or the table will not load. This is useful to ensure you absolutely have the right symbols for a given situation.

Additionally, the TABLE command will display other pertinent information that is discovered about loaded tables for an image; for example, showing the “(PERF)” indicator in the status field, which informs you that the image has been performance optimized since the symbol file was generated, and some lookups may not match exactly (refer to the TABLE command in the Visual SoftICE Command Reference for more details).

Automatic, On-Demand Loading and Unloading

Automatic Loading

Visual SoftICE loads only the tables on-demand that are necessary.

- Tables are loaded for “current” images (images referenced in the current process, and/or referenced in the current stack). As the OS loads images, and those images are part of the current context, the Symbol Engine will automatically load them.
- As debugger commands are issued that reference other tables, these are loaded on-demand as well.
- All known images are tracked, and those not currently loaded yet are considered in a “Deferred” state. Once loaded, the table remains loaded until the image itself unloads, or the user forces it to unload (refer to the UNLOAD command in the Visual SoftICE Command Reference).
- Automatic, dynamic loading presumes that the path information has been setup so it can retrieve symbols in local (or accessible) directories, or from any configured MS Symbol Server, and that it may find local exports and images as well.

Automatic Unloading

When the OS unloads an image, if the Symbol Engine has a matching table, that table is dynamically unloaded as well.

Controls

To disable or enable automatic symbol table loading, use the SET SYMTABLEAUTOLOAD on/off command. Automatic loading is enabled by default.

Pre-Loading/Persistent Loading

It is often desirable to set breakpoints in images that have not yet been loaded by the operating system. You can accomplish this in Visual SoftICE by preloading symbols. These symbols are considered persistent. Their match state is updated whenever image collections change, but these tables are loaded into the symbol engine, even if there is no match. Persistent symbols do not participate in automatic load or unload behavior.

Commands

- To load persistent symbols, use the ADDSYM command.
- To unload persistent symbols, use the DELSYM command.

Integrated MS Symbol Server Access

Retrieval of operating system symbols using the Microsoft Symbol Server technology is a great boon to productivity when working with a heterogeneous set of target hardware and OS versions. This same technology is reusable by end users to provide their own symbol files (and file types) in private servers.

Visual SoftICE has directly integrated MS SymbolServer access into its automatic, on-demand loading logic. If so configured, when Visual SoftICE does not find the file it is looking for in the local search directories, it will attempt to retrieve the file from a symbol server. The paths to use, symbol server to attempt a connection to, and whether this behavior is enabled or not, is fully configurable.

GUI Configuration

You can configure the MS Symbol Server Access and the server and local directory information from the **Paths** element under **Global Settings** in the Preferences dialog.

- To set the server and local directory Visual SoftICE will bring files into, configure the **MS Symbol Server Paths** element.

- To enable symbol server retrieval (disabled by default), configure the **MS Symbol Server(s) Enabled** element.

Equivalent Commands

You can configure the MS Symbol Server Access and the server and local directory information by issuing some Visual SoftICE commands at the command line in the Command page.

- To modify or add a symbol server, use the SET SYMPATH command.
- To enable symbol server retrieval (disabled by default), use the SET SYMSRVSEARCH on/off command.

Using Exports

It is often useful to have exports in the absence of, or in addition to, full or partial symbolic data for images. Visual SoftICE supports a means to use exports, when available, in addition to any symbol files.

- A path to the top of an exports tree is specified (refer to the SET EXPORTPATH command in the Visual SoftICE Command Reference), and exports can then be extracted on demand from the connected target (refer to the GETEXP command in the Visual SoftICE Command Reference).
- Exports are stored with as much matching information as is available in the image on the target, in a local directory structure similar to the storage mechanism MS Symbol Servers uses for symbols
- You can specify a wildcard "*" in the GETEXP command, getting exports for everything in the specified directory. You can also specify the flag (-s) to walk subdirectories in the retrieval. Please note that retrieving exports for every image file on a target can be a very lengthy process (but is possible).

Symbols — Getting Setup in Visual SoftICE

This section provides a couple steps to get you up and running quickly with symbols and Visual SoftICE.

Once these are complete, you should be able to easily manage the tables loaded into Visual SoftICE via the TABLE, ADDSYM, DELSYM, LOAD, UNLOAD, and RELOAD commands.

Once proper symbols are loaded, you should be able to explore available data via the EXP, FILE, LOCALS, SYM and TYPES commands.

Setting Up General Paths

Configure the following Global General Path Settings:

- **Image Search** — Where you will put copies of the executables you will debug on the target.
- **Source Search** — Where your source code lives.
- **User Symbol Search Path** — Where your symbols live.
- **MS Symbol Server Paths** — Where the MS Symbol Server should be found, and where it should store things its retrieves.
- **KD Extension Paths** (optional) — Where to find KD extensions when no absolute path is specified.

Setting Up Visual SoftICE Paths

Configure the following Global Visual SoftICE Path Settings:

- **Export** — Where you want the top of the retrieved export directory structure to grow.
- **Script Search Path** (optional) — Where to find scripts when no absolute path is specified.

Settings Notes

Path Settings

Path settings come in two types, a path list (which is an ordered list of paths to search, each one semicolon separated) and a single path entry. All path types for Visual SoftICE accept the ellipses (...) syntax at the end of a directory name to indicate Visual SoftICE should search all directories below the one specified.

Thus to get the source search path to include all your projects and source code, the following example might be useful:

```
SET SRCPATH c:\development\driver_projects\…
```

In this case, Visual SoftICE searches all the files in `c:\development\driver_projects`, and any subdirectories underneath that directory, for a matching source file.

Per-Workspace Settings Notes

Workspaces are intended to be containers not only of GUI layout, but also of user preferences. These might be used for particular targets, or types of targets (one for x86, one for IA64, etc...), or even one workspace for debugging, and one for development.

Whatever way they are used, the per-workspace settings provide a means to augment or override global (application wide) settings. The case of interest here is for Path Settings, specifically **Per-Workspace General Path** settings.

Under this section of preferences, the Image, Script, Source, and User Symbol paths are repeated. These entries are provided so that you may *prepend* additional search paths to the global search paths already specified under Global General Path Settings. Optionally (and on a per-path basis), you can replace the global settings of the same name. Visual SoftICE provides a checkbox allowing you to choose whether the path you specify here will prepend or replace the Global General Path setting.

To evaluate the effect these settings have, after modifying them, issue the SET SYMPATH command. The output will allow you to see how Visual SoftICE is using your configuration.

Toolbars & Status Bar Settings Notes

Visual SoftICE provides two optional status bar fields to give you quick updates of the Symbol Engine's current status and actions.

VSI System Activity Messages

This field will be updated when the Symbol Engine takes an action to retrieve, load, or make a given table active. This status bar field is shown in the default workspaces provided with the product.

VSI Symbol Table Status

This field shows:

- The current table name, within the set of loaded tables
- The status of all active tables for the current context
    - A green icon indicates all tables for the current context are loaded and matched.
    - A yellow warning icon indicates that some of the tables for the current context may have mismatch or other pertinent warning information.
    - A red error icon indicates some tables for the current context are missing.
- The status of automatic table loading
    - A folder and magnifying glass icon is displayed when enabled, the same icon with a red circle and line over it indicates it is disabled.
- The status of symbol server retrieval (enabled or disabled)
    - A server icon is displayed when enabled, the same icon with a red circle and line over it indicates it is disabled

To display these fields on your status bar, click the **Customize** button under the Status Bar section of the **Toolbars & Status Bar** tab in the Preferences dialog.

Visual SoftICE will save your status bar configuration in the active workspace, and restore it any time you open that workspace.

Chapter 6
Using Breakpoints

Introduction

You can use Visual SoftICE to set breakpoints on program execution, memory location reads and writes, interrupts, and reads and writes to I/O ports. Visual SoftICE assigns a breakpoint index to each breakpoint. You can use this breakpoint index to identify breakpoints when you delete, disable, enable, or reference them.

All Visual SoftICE breakpoints are sticky, which means that Visual SoftICE tracks and maintains a breakpoint until you intentionally clear or disable it using the BC or the BD command. Breakpoints on a target machine are removed when you disconnect, or the connection times out.

The number of breakpoints you can set on memory location (BPMs) and I/O ports (BPIOs) is limited by the number of debug registers the target processor supports (e.g. x86 supports 4).

Where symbol information is available, you can set breakpoints using function names. When in a disassembly, source, or stack page, you can set breakpoints via popup context menus, or at anytime from the breakpoint page, or the debug main menu. A valuable feature is that you can set breakpoints in an image before it is loaded.

Types of Breakpoints Supported by Visual SoftICE

Visual SoftICE provides a powerful array of breakpoint capabilities that take full advantage of the target architecture, as follows:

- **Execution Breakpoints:** Visual SoftICE replaces existing instructions with architecture appropriate breaks. You use the BPX command to set execution breakpoints.
- **Memory Breakpoints:** Visual SoftICE uses the target processor debug registers to break when a certain byte/word/dword/qword of memory is read, written, or executed. You use the BPM command to set memory breakpoints on scalar data types. You use the BPR command (where supported) to set memory breakpoints on a larger range of memory.
- **Interrupt Breakpoints:** Visual SoftICE allows setting breakpoints on OS fault handlers or driver provided interrupt service routines (see INTOBJ command). You use the BPINT command to set interrupt breakpoints.
- **I/O Breakpoints:** Visual SoftICE uses target processor debug registers to watch for an IN or OUT instruction going to a particular port address. You use the BPIO command to set I/O breakpoints.
- **Image Load Breakpoints:** Visual SoftICE supports two mechanisms to stop on an image load – either when an image matching a name is loaded (using the BPLOAD command), or when any image in the system loads (using the SET GLOBALBREAK command).
- **Window Message Breakpoints:** Visual SoftICE triggers these breakpoints when a particular message or range of messages arrives at a window. This is not a fundamental breakpoint type; it is just a convenient feature built on top of the other breakpoint primitives. You use the BMSG command to set window message breakpoints.

Breakpoint Options

Visual SoftICE supports logging, conditions and actions (triggered) on most breakpoint types.

You can qualify each type of breakpoint with the following options:

- An optional logged flag can be specified [-l] which indicates that statistics should be updated, but execution continued when the breakpoint is encountered.
- A conditional expression [IF expression]: The expression must evaluate to non-zero (TRUE) for the breakpoint to trigger. Refer to "Conditional Breakpoints" on page 125.
- A breakpoint action [DO "command1;command2;…"]: A series of Visual SoftICE commands can automatically execute when the breakpoint triggers. You can use this feature in concert with user-defined macros to automate tasks that would otherwise be tedious. Refer to "Setting a Breakpoint Action" on page 125.

Note: For complete information on each breakpoint command, refer to the *Visual SoftICE Command Reference*.

Execution Breakpoints

An execution breakpoint traps executing code such as a function call or language statement. This is the most frequently used type of breakpoint. By replacing an existing instruction with an architecture appropriate break instruction, Visual SoftICE takes control when execution reaches the breakpoint.

Visual SoftICE provides many ways for setting execution breakpoints: using the BPX command, the breakpoint page, the main debug menu, and off numerous context menus on other pages. The following section describes how to use the commands for setting breakpoints.

Using the BPX Command to Set Breakpoints

Use the BPX command with any of the following parameters to set an execution breakpoint:

```
BPX [-l] address [IF conditional-expression] [DO
"command1;command2;…"]
```

`IF expression`	Refer to "Conditional Breakpoints" on page 125.
`DO "command1;command2;…"`	Refer to "Setting a Breakpoint Action" on page 125.

To set a breakpoint on your application's WinMain function, use this command:

```
BPX WinMain
```

Memory Breakpoints

A memory breakpoint uses the debug registers found on the target processor to monitor access to a certain memory location. This type of breakpoint is extremely useful for finding out when and where a program variable is modified, and for setting an execution breakpoint in read-only memory. You are limited by the number of processor debug registers to how many breakpoints may be set at one time.

Use the BPM command to set memory breakpoints on scalar data type sized memory blocks:

```
BPM[size] [-l] address [ R | W | RW | X ] [IF conditional-
expression][DO “command1;command2;…”]
```

`size`	B (byte) W (word) D (dword) Q (qword).
`BPM and BPMB`	Set a byte-size breakpoint.
`R, W, and RW`	Break on reads, writes, or both.
`X`	Breaks on execution; this is more powerful than a BPX-style breakpoint because memory does not need to be modified, enabling such options as setting breakpoints in ROM or setting breakpoints on addresses that are not present.
`IF expression`	Refer to “Conditional Breakpoints” on page 125.
`DO “command1;command2;…”`	Refer to “Setting a Breakpoint Action” on page 125.

The following example sets a memory breakpoint to trigger when a value of 5 is written to the Dword (4-byte) variable MyGlobalVariable.

```
BPMD MyGlobalVariable W IF MyGlobalVariable==5
```

If the target location of a BPM breakpoint is frequently accessed, performance can be degraded regardless of whether the conditional expression evaluates to FALSE.

Use the BPR command to set a memory breakpoint on a larger range of memory.

Note: The BPR command is currently available only on the IA64 processor.

```
BPR [-l] start-address end-address [R|W|RW|X] [IF conditional-
expression] [DO “command1;command2;…”]
BPR [-l] start-address L length [R|W|RW|X] [IF conditional-
expression] [DO “command1;command2;…”]
```

The following example sets a memory breakpoint to trigger when a value of 5 is written to the Dword (4-byte) variable MyGlobalVariable.

```
BPMD MyGlobalVariable W IF MyGlobalVariable==5
```

If the target location of a BPM breakpoint is frequently accessed, performance can be degraded regardless of whether the conditional expression evaluates to FALSE.

Interrupt Breakpoints

Sets an execution breakpoint on an interrupt handler, proivided by either the OS or Driver.

Use the BPINT command to set interrupt breakpoints:

```
BPINT [-l] interrupt-number [service-address] [IF conditional-
expression] [DO “command1;command2;…”]
```

`interrupt-number`	Architecture supported number.
`service-address`	Often an interrupt is shared between different handler services. This optional parameter allows for setting the breakpoint on the exact service. If excluded, the breakpoint will be set on all service handlers associated with the interrupt.
`IF expression`	Refer to “Conditional Breakpoints” on page 125.
`DO “command1;command2;…”`	Refer to “Setting a Breakpoint Action” on page 125.

Visual SoftICE will stop either on the OS fault handler, or the entry point to the driver provided Interrupt Service Routine. You can list all interrupts and their handlers by using the IT/IDT command.

I/O Breakpoints

An I/O breakpoint monitors reads and writes to a port address. The breakpoint traps when an IN or OUT instruction accesses the port.

Use the BPIO command to set I/O breakpoints:

```
BPIO [-l] port [R|W|RW] [IF conditional-expression] [IF
expression] [DO “command1;command2;…”]
```

`R, W, and RW`	Break on reads (IN instructions), writes (OUT instructions), or both, respectively.
`IF expression`	Refer to “Conditional Breakpoints” on page 125.
`DO “command1;command2;…”`	Refer to “Setting a Breakpoint Action” on page 125.

When an I/O breakpoint triggers and Visual SoftICE stops, the current instruction is the instruction following the IN or OUT that caused the breakpoint to trigger. Unlike BPM breakpoints, there is no size specification; any access to the port-number, whether byte, word, dword, or larger triggers the breakpoint. Any I/O that spans the I/O breakpoint will also trigger the breakpoint. For example, if you set an I/O breakpoint on port 2FF, a word I/O to port 2FE would trigger the breakpoint.

Use the following command to set a breakpoint to trigger when a value is read from port 3FEH with the upper 2 bits set:

```
BPIO 3FE R IF (AL & C0)== C0
```

The condition is evaluated after the instruction completes. The value will be in AL, AX, or EAX because all port I/O, except for the string I/O instructions (which are rarely used), use the EAX register.

I/O breakpoints appear as blue throughout the GUI.

Image Load Breakpoints

An image load breakpoint is a stop request when a given image loads in the operation system and the module or dll contains an entry point that gets executed. This can be filtered to when a named image is loaded, or when ANY image loads, with the following commands:

```
BPLOAD [-once] image-name [DO “command1;command2;…”]
```

`-once`	One-shot breakpoint. Once this breakpoint is hit, it is automatically deleted.
`image-name`	Name to match for the image or module. This can be a partial match, but does not support wildcards.
`DO “command1;command2;…”`	Refer to “Setting a Breakpoint Action” on page 125.

```
SET GLOBALBREAK [off/load]
```

This is a general setting for the debugger, and if set to LOAD, the target will stop on every image loaded into the system.

Window Message Breakpoints

Use a window message breakpoint to trap a certain message or range of messages delivered to a window procedure. Although you could implement an equivalent breakpoint yourself using BPX with a conditional expression, the following BMSG command is easier to use:

```
BMSG [-l] window-handle [begin-msg [end-msg]] [IF conditional-
expression] [DO “command1;command2;…”]
```

`window-handle`	Value returned when the window was created; you can use the HWND command to get a list of windows with their handles.
`begin-message`	Single Windows message or the lower message number in a range of Windows messages. If you do not specify a range with an end-message, then only the begin-message will cause a break. For both begin-message and end-message, the message numbers can be specified either in hexadecimal or by using the actual ASCII names of the messages, for example, WM_QUIT.
`end-message`	Higher message number in a range of Windows messages.
`IF expression`	Refer to “Conditional Breakpoints” on page 125.
`DO “command1;command2;…”`	Refer to “Setting a Breakpoint Action” on page 125.

When specifying a message or a message range, you can use the symbolic name, for example, WM_NCPAINT. Use the WMSG command to get a list of the window messages that Visual SoftICE understands. If no message or message range is specified, any message will trigger the breakpoint.

To set a window message breakpoint for the window handle `1001E`, use the following command:

```
BMSG 1001E WM_NCPAINT
```

Visual SoftICE is smart enough to take into account the address context of the process that owns the window, so it does not matter what address context you are in when you use BMSG.

Understanding Breakpoint Contexts

A breakpoint context consists of the address context in which the breakpoint was set, and in what image the breakpoint is in, if any. The concept of breakpoint context applies to all breakpoints, and there are two fundamental types of breakpoints supported by Visual SoftICE, Fixed Address, and Image-Relative.

Image-Relative Breakpoints

Image-Relative breakpoints are the most common, and most useful type — where the address is understood to be in the context of a given image (even if that image has not loaded, or the its memory been paged in). Any image-relative breakpoint will trigger in any instance of the image that loads, regardless of its address or process (e.g. a breakpoint set in an OS image like `KERNEL32.DLL` breaks in every process context that has the image loaded, regardless of what context the breakpoint was initially set in). Image relative breakpoints require accurate symbols to be available in order to set them. These types of breakpoints appear as green throughout the GUI.

Fixed Address Breakpoints

Fixed Address breakpoints are uncommon, and not as useful. They are set at an address in a particular process context. They stay at that address, and are not shared unless their placement accidentally coincides with another OS mapping. Fixed addresses are not recommended, as they can be system fatal if used incorrectly. These types of breakpoints appear as yellow or brown throughout the GUI. Their appearance generally means missing or mismatched symbols are in use.

Virtual Breakpoints

In Visual SoftICE, you can set breakpoints in images before they load, and it is not necessary for a page to be present in physical memory for a breakpoint to be set. In such cases, the breakpoint is virtual; it will be automatically planted when the image loads or the page becomes present. Virtual breakpoints can only be set when symbols are available.

Setting a Breakpoint Action

You can set a breakpoint to execute a series of Visual SoftICE commands, including user-defined macros, after the breakpoint is triggered. You define these breakpoint actions with the DO option, which is available with every breakpoint type:

```
DO “command1;command2;…”
```

The body of a breakpoint action definition is a sequence of Visual SoftICE commands, or other macros, separated by semicolons. You need not terminate the final command with a semicolon.

Breakpoint actions are closely related to macros. Breakpoint actions are essentially unnamed macros that do not accept command-line arguments. Breakpoint actions, like macros, can call upon macros. In fact, a prime use of macros is to simplify the creation of complex breakpoint actions.

If a breakpoint is marked as log-only, the action will not be executed.

The following examples illustrate the basic use of breakpoint actions:

```
BPX EIP DO “dd eax”
BPX EIP DO “data 1;dd eax”
BPMB dataaddr if (byte(*dataaddr)==1) do “? IRQL”
```

Conditional Breakpoints

Conditional breakpoints provide a fast and easy way to isolate a specific condition or state within the system (or application) you are debugging. By setting a breakpoint on an instruction or memory address and supplying a conditional expression, Visual SoftICE will only trigger if the breakpoint evaluates to non-zero (TRUE). Because the Visual SoftICE expression evaluator handles complex expressions easily, conditional expressions take you right to the problem or situation you want to debug with ease.

Most Visual SoftICE breakpoint commands accept conditional expressions using the following syntax:

```
breakpoint-command [breakpoint options] [IF conditional
expression][DO “commands”]
```

The IF keyword, when present, is followed by any expression that you want to be evaluated when the breakpoint is triggered. The breakpoint will be ignored if the conditional expression is FALSE (zero). When the conditional expression is TRUE (non-zero), Visual SoftICE stops and displays the reason for the break, which includes the conditional expression.

Most of the following x86 examples contain system-specific values that vary depending on the exact version of Windows NT/2000/XP you are running.

- Watch CSRSS HWND objects (type 1) being created:

  ```
  bpx winsrv!HMAllocObject IF (esp+c == 1)
  ```

- Watch CSRSS thread info objects (type 6) being destroyed:

  ```
  bpx winsrv!HMFreeObject+0x25 IF (byte(esi+8) == 6)
  ```

- Watch process object-handle-tables being created:

  ```
  bpx ntoskrnl!ExAllocatePoolWithTag IF (esp+c == 'Obtb')
  ```

- Watch a heap block (230CD8) get freed:

  ```
  bpx ntddl!RtlFreeHeap IF (esp+c == 230CD8)
  ```

Conditional Breakpoint Count Functions

Visual SoftICE supports the ability to monitor and control breakpoints based on the number of times a particular breakpoint has or has not been triggered. You can use the following count functions in conditional expressions:

- BPCOUNT
- BPMISS
- BPTOTAL
- BPINDEX

BPCOUNT

The value for the BPCOUNT function is the current number of times that the breakpoint has been evaluated as TRUE.

Use this function to control the point at which a triggered breakpoint causes a popup to occur. Each time the breakpoint is triggered, the conditional expression associated with the breakpoint is evaluated. If the condition evaluates to TRUE, the breakpoint instance count (BPCOUNT) increments by one. If the conditional evaluates to FALSE, the breakpoint miss instance count (BPMISS) increments by one.

The fifth time the breakpoint triggers, the BPCOUNT equals five, so the conditional expression evaluates to TRUE and Visual SoftICE stops.

```
BPX myaddr IF (bpcount==5)
```

Use BPCOUNT only on the righthand side of compound conditional expressions for BPCOUNT to increment correctly:

```
BPX myaddr if (eax==1) && (bpcount==5)
```

Due to the early-out algorithm employed by the expression evaluator, the `BPCOUNT==5` expression will not be evaluated unless `EAX==1` (The C language works the same way). Therefore, by the time `BPCOUNT==5` gets evaluated, the expression is TRUE. BPCOUNT will be incremented and if it equals five, the full expression evaluates to TRUE and Visual SoftICE stops. If `BPCOUNT != 5`, the expression fails, BPMISS is incremented and Visual SoftICE will not stop (although BPCOUNT is now 1 greater).

Once the full expression returns TRUE, Visual SoftICE stops, and all instance counts (BPCOUNT and BPMISS) are reset to 0.

Note: Do NOT use BPCOUNT before the conditional expression, otherwise BPCOUNT will not increment correctly:

```
BPX myaddr if (bpcount==5) && (eax==1)
```

BPMISS

The value for the BPMISS expression function is the current number of times that the breakpoint was evaluated as FALSE.

The expression function is similar to the BPCOUNT function. Use it to specify that Visual SoftICE stop in situations where the breakpoint is continually evaluating to FALSE. The value of BPMISS will always be one less than you expect, because it is not updated until the conditional expression is evaluated. You can use the (>=) operator to correct this delayed update condition.

```
BPX myaddr if (eax==43) || (bpmiss>=5)
```

Due to the early-out algorithm employed by the expression evaluator, if the expression `eax==43` is ever TRUE, the conditional evaluates to TRUE and Visual SoftICE stops. Otherwise, BPMISS is updated each time the conditional evaluates to FALSE. After five consecutive failures, the expression evaluates to TRUE and Visual SoftICE stops.

BPTOTAL

The value for the BPTOTAL expression function is the total number of times that the breakpoint was triggered.

Use this expression function to control the point at which a triggered breakpoint causes a stop to occur. The value of this expression is the total number of times the breakpoint was triggered (refer to the Hits field in the output of the BSTAT command) over its lifetime. This value is never cleared.

The first 50 times this breakpoint is triggered, the condition evaluates to FALSE and Visual SoftICE will not pop up. Every time after 50, the condition evaluates to TRUE, and Visual SoftICE stops on this and every subsequent trap.

```
BPX myaddr if (bptotal > 50)
```

You can use BPTOTAL to implement functionality identical to that of BPCOUNT. Use the modulo "%" operator as follows:

```
if (!(bptotal%COUNT))
```

The COUNT is the frequency with which you want the breakpoint to trigger. If COUNT is four, Visual SoftICE stops every fourth time the breakpoint triggers.

BPINDEX

Use the BPINDEX expression function to obtain the breakpoint index to use with breakpoint actions.

This expression function returns the index of the breakpoint that caused Visual SoftICE to stop. This index is the same index used by the BL, BC, BD, BE, BPE, BPT, and BSTAT commands. You can use this value as a parameter to any command that is being executed as an action.

The following example of a breakpoint action causes the BSTAT command to be executed with the breakpoint that caused the action to be executed as its parameter:

```
BPX myaddr do “bstat bpindex”
```

This example shows a breakpoint that uses an action to create another breakpoint:

```
BPX myaddr do “t;bpx @esp if(tid==_tid) do \“bc bpindex\”;g”
```

BPINDEX is intended to be used with breakpoint actions, and causes an error if it is used within a conditional expression. Its use outside of actions is allowed, but the result is unspecified and you should not rely on it.

Using Local Variables in Conditional Expressions

Visual SoftICE lets you use local variable names in conditional expressions as long as the type of breakpoint is an execution breakpoint (BPX or BPM X). Visual SoftICE does not recognize local symbols in conditional expressions for other breakpoint types, such as BPIO or BPMD RW, because they require an execution scope. This type of breakpoint is not tied to a specific section of executing code, so local variables have no meaning.

When using local variables in conditional expressions, functions typically have a prologue where local variables are created and an epilogue where they are destroyed. You can access local variables after the prologue code completes execution and before the epilogue code begins execution. Function parameters are also temporarily inaccessible using symbol names during prologue and epilogue execution, because of adjustments to the stack frame.

To avoid these restrictions, set a breakpoint on either the first or last source code line within the function body. We will use the following “foobar function” to explain this concept.

Foobar Function

```
1:DWORD foobar ( DWORD foo )
2:{
3:DWORDfooTmp=0;
4:
5:if(foo)
6:{
7:fooTmp=foo*2;
8:}else{
9:fooTmp=1;
10:}
11:
12:return fooTmp;
13:}
```

Source code lines 1 and 2 are outside the function body. These lines execute the prologue code. If you use a local variable at this point, you receive the following symbol error:

```
>BPX foobar if(foo==1)
error: Undefined Symbol (foo)
```

Set the conditional on the source code line 3, where the local variable fooTmp is declared.

Source code line 13 marks the end of the function body. It also begins epilogue code execution; thus, local variables and parameters are out of scope.

Although it is possible to use local variables as the input to a breakpoint command, such as BPMD RW, you should avoid doing this. Local variables are relative to the stack, so their absolute address changes each time the function scope where the variable is declared executes. When the original function scope exits, the address tied to the breakpoint no longer refers to the value of the local variable.

Referencing the Stack in Conditional Breakpoints

If you create your symbol file with full symbol information, you can access function parameters and local variables through their symbolic names, as described in "Using Local Variables in Conditional Expressions" on page 129. If, however, you are debugging without full symbol information, you need to reference function parameters and local variables on the stack.

Note: The following section is specific to x86 32-bit flat application or system code.

Function parameters are passed on the stack, so you need to de-reference these parameters through the ESP or EBP registers. Which one you use depends on the function's prologue and where you set the actual breakpoint in relation to that prologue.

Most 32-bit functions have a prologue of the following form:

```
PUSH   EBP
MOV    EBP,ESP
SUB    ESP,size (locals)
```

Which sets up a stack frame as follows:

- Use either the ESP or EBP register to address parameters. Using the EBP register is not valid until the PUSH EBP and MOV EBP, ESP instructions are executed. Also note that once space for local variables is created (SUB ESP, size) the position of the parameters relative to ESP needs to be adjusted by the size of the local variables and any saved registers.
- Typically you set a breakpoint on the function address, for example:

  ```
  BPX IsWindow
  ```

	PARAM n	ESP+(n*4), or EBP+(n*4)+4	**Pushed by caller**
Stack Top	PARAM #2	ESP+8, or EBP+C	
	PARAM #1	ESP+4, or EBP+8	
	RET EIP	⇐ Stack pointer on entry	**Call prologue**
Current EBP →	SAVE EBP	⇐ Base pointer (PUSH EBP, MOV EBP,ESP)	
	LOCALS+size-1 LOCALS+0	⇐ Stack pointer after prologue (SUB ESP, size (locals))	
Stack Bottom	SAVE EBX	optional save of 'C' registers	**Registers saved by compiler**
Current ESP →	SAVE ESI		
	SAVE EDI	⇐ Stack pointer after registers are saved	

When this breakpoint is triggered, the prologue has not been executed, and parameters can easily be accessed through the ESP register. At this point, use of EBP is not valid.

Note: This assumes a stack-based calling convention with arguments pushed right-to-left.

To be sure that de-referencing the stack in a conditional expression operates as you would expect, use the following guidelines.

- If you set a breakpoint at the exact function address, for example, BPX IsWindow, use `ESP+(param# * 4)` to address parameters, where param# is 1…n.

- If you set a breakpoint inside a function body (after the full prologue has been executed), use `EBP+(param# * 4)+4` to address parameters, where param# is 1…n. Be sure that the routine does not use the EBP register for a purpose other than a stack-frame.

- Functions that are assembly-language based or are optimized for frame-pointer omission may require that you use the ESP register, because EBP may not be set up correctly.

Note: For x86, once the space for local variables is allocated on the stack, the local variables can be addressed using a negative offset from EBP. The first local variable is at `EBP-4`. Simple data types are typically Dword sized, so their offset can be calculated in a manner similar to function parameters. For example, with two pointer local variables, one will be at `EBP-4` and the other will be at `EBP-8`.

Performance

Conditional breakpoints have some overhead associated with run-time evaluation. Under most circumstances you see little or no effect on performance when using conditional expressions. In situations where you set a conditional breakpoint on a highly accessed data variable or code sequence, you may notice slower system performance. This is due to the fact that every time the breakpoint is triggered, the conditional expression is evaluated. If a routine is executed hundreds of times per second (such as ExAllocatePool), the fact that any type of breakpoint with or without a conditional is trapped and evaluated with this frequency results in some performance degradation.

Duplicate Breakpoints

Once a breakpoint is set on an address, you cannot set another breakpoint on the same address. With conditional expressions, however, you can create a compound expression using the logical operators (&&) or (||) to test more than one condition at the same address.

Elapsed Time

Visual SoftICE supports using the target processor time stamp counter. Every time Visual SoftICE stops due to a breakpoint, the elapsed time displays since the last time Visual SoftICE stopped. The time displays after the break reason in seconds, milliseconds, or microseconds, and this can be seen in the command page, or breakpoint page history log.

Most processor time stamp counters are highly accurate, but you must keep the following issues in mind:

- There is overhead involved in stopping Visual SoftICE, which may impact the results very slightly.
- If a hardware interrupt occurs before the breakpoint goes off, all the interrupt processing time is included.
- Certain processors will vary the clock rate dynamically, for power management reasons.

Breakpoint Statistics

Visual SoftICE collects statistical information about each breakpoint, including the following:

- Total number of hits, breaks, misses, and errors
- Current hits and misses

Use the BSTAT command to display this information. Refer to the *Visual SoftICE Command Reference* for more information on the BSTAT command.

Referring to Breakpoints in Expressions

You can combine the prefix “BP” with the breakpoint index to use as a symbol in an expression. This works for all BPX and BPM breakpoints. Visual SoftICE uses the actual address of the breakpoint.

To disassemble code at the address of the breakpoint with index 0, use the command:

```
U BP0
```

Manipulating Breakpoints

Visual SoftICE provides a variety of commands for manipulating breakpoints such as listing, modifying, deleting, enabling, and disabling breakpoints. Breakpoints are identified by breakpoint index numbers, which are uniquely assigned. Table describes the breakpoint manipulation commands.

Table 6-1. Visual SoftICE Breakpoint Manipulation Commands

Command	Description
BD	Disable a breakpoint.
BE	Enable a breakpoint.
BL	List current breakpoints.
BC	Clear (remove) a breakpoint.

Note: Refer to the *Visual SoftICE Command Reference* for more information on each of these commands.

Using Embedded Breakpoints

It may be helpful for you to embed a breakpoint in your program source rather than setting a breakpoint with Visual SoftICE. To embed a breakpoint in your program, do the following:

- Place an INT 1 or INT 3 instruction at the desired point in the program source.
- To enable Visual SoftICE to pop up on such embedded breakpoints, use one of the following commands:
  - **`I1HERE ON`** for INT 1 breakpoints
  - **`I3HERE ON`** for INT 3 breakpoints

Chapter 7
Using Expressions

Expression Values

The Visual SoftICE expression evaluator determines the values of expressions used with Visual SoftICE commands and conditional breakpoints. It provides full operator precedence; support for standard C language arithmetic, bit-wise, and logical operators; predefined macros, functions, and casts for data type conversion; and access to common Visual SoftICE and operating system values.

The Visual SoftICE expression evaluator parses and evaluates expressions similarly to the way a C or C++ language compiler translates expressions. If you are comfortable with either language, you are already familiar with the grammar and syntax of Visual SoftICE expressions.

There are no limitations on the complexity of an expression. You can combine multiple operators, operands, and expressions to create compound expressions for conditional breakpoints or expression evaluation.

This example uses a compound expression to trigger a breakpoint if the first parameter (ESP+4) passed to the `IsWindow( )` API function is an HWND with the value of `0x10022` or `0x1001E`. If either of the two expressions is TRUE, the conditional expression is TRUE, and the breakpoint triggers:

```
BPX IsWindow if (esp+4 == 10022) || (esp+4 == 1001E)
```

Supported Operators

The Visual SoftICE expression evaluator supports the following operators sorted by type:

Note: Use the SET EE_IMPL_DEREF command to control the expression evaluator's behavior regarding dereferencing. If you have set EE_IMPL_DEREF to *on*, and the expression evaluator encounters an expression containing a symbol that is a pointer, it will use the value it points to for evaluation. If you have set EE_IMPL_DEREF to *off*, and the expression evaluator encounters an expression containing a symbol that is a pointer, it will use the address of the pointer for evaluation.

Table 7-1. Visual SoftICE Indirection Operators

Indirection Operators	Example
*	`*eax (gets the Dword value pointed to by eax)`
&*symbol-name*	`&Foo (gets the address of the symbol Foo)`
[ ] (array subscript)	`Foo[2] (gets the second element of the array Foo)`

Table 7-2. Visual SoftICE Math Operators

Math Operators	Example
+	`eax + 1`
-	`ebp - 4`
*	`ebx * 4`
/	`Symbol / 2`
% (modulo)	`eax % 3`
<< (logical shift left)	`bl << 1 (result is bl shifted left by 1)`
>> (logical shift right)	`eax >> 2 (result is eax shifted right by 2)`

Table 7-3. Visual SoftICE Bitwise Operators

Bitwise Operators	Example
& (bitwise AND)	`eax & F7`
| (bitwise OR)	`Symbol | 4`
^ (bitwise XOR)	`ebx ^ 0xFF`

Table 7-4. Visual SoftICE Logical Operators

Logical Operators	Example
! (logical NOT)	`!eax`
&& (logical AND)	`eax && ebx`
|| (logical OR)	`eax || ebx`
== (compare equality)	`Symbol == 4`
!= (compare inequality)	`Symbol != al`
<	`eax < 7`
>	`bx > cx`
<=	`ebx <= Symbol`
>=	`Symbol >= Symbol`

Operator Precedence

Operator precedence within the Visual SoftICE expression evaluator is equivalent to the C language operator precedence. Operator precedence plays a crucial part in evaluating expressions, so the order in which you input expression operators can have a dramatic result on the final result of the expression. To override the default operator precedence to produce a desired result, use parentheses to force the order of evaluation.

Forming Expressions

Tip: *Use the* ***?*** *or EVAL (evaluate expression) command to display the result of any expression.*

The Visual SoftICE expression evaluator accepts a variety of operands, such as symbols, register names, user-defined symbols, and numbers, that you can combine with any Visual SoftICE operator. Visual SoftICE places an emphasis on providing flexibility of expression, so input is as natural as possible.

Numbers

The Visual SoftICE expression evaluator accepts the following numeric inputs.

Table 7-5. Visual SoftICE Expression Inputs

Input	Description
Hexadecimal	The default radix for input and output is controlled by the SET RADIX command. The valid character set for hexadecimal numbers is [0-9, A-F]. Hexadecimal input can be optionally preceded by the standard C language radix identifier: 0x. Examples of valid hexadecimal numbers include: `FF, ABC, 0x123, 0xFFFF0000` The symbolic form of a valid hexadecimal number could conflict with a symbol name. For example, ABC. Use the 0x form to ensure that the number is not misinterpreted as a symbol name. Prefixing 0x to input forces it to be evaluated as hexadecimal input regardless of the default radix. You can also use the HEX(n) and DEC(n) functions to force interpretation to a given radix.

Registers

Visual SoftICE supports multiple names for the target register sets. You control the target register set name using the SET REGNAME command.

A register name or alias might conflict with a symbol. To force input to be evaluated as a register name, use the REG(n) function.

Symbols

Symbol names are the symbolic representation of an address or value. They are defined in symbol tables, export tables, or via Visual SoftICE's NAME command, during debugging.

Symbol names in Visual SoftICE differ from symbols defined in C or C++ programs. All compilers add some form of decoration to the names defined in a program, and this decoration often includes characters which are not valid in C/C++ symbol names. Visual SoftICE therefore accepts a wider range of characters in symbol names than a compiler. Table 7-6 shows the characters which may be found in a legal symbol name. Symbols must begin with one of the characters marked valid as first symbol characters in the table.

Table 7-6. "Legal" Symbol Characters

Characters	
A..Z and a..z	Yes
0..9	No
dollar sign ($)	Yes
underscore (_)	Yes
exclamation point (!)	No
scope operator (::)	Yes

The scope operator (`::`) is allowed in symbols. However, note that the "operator" is in this context simply part of the symbol name, and is not functioning as a true operator. Any number of scope operators are allowed in a symbol name, so namespaces and nested classes will function properly.

Each symbol file loaded into Visual SoftICE is placed in a separate table, and only one symbol table can be "active" at a time. (Refer to the TABLE command in the Visual SoftICE Command Reference for more information on changing the active table.)

To specify a symbol from an inactive symbol table in an expression, you may precede the symbol with the table name, followed by an exclamation point, followed by the symbol name. For example:

```
table-name!symbol-name
```

Symbols that are defined by the NAME command are always active, because Visual SoftICE treats these symbol sources as global.

Built-in Casts and Functions

Visual SoftICE predefines a number of casts and functions for use in expressions. They can be used within expressions to modify values or translate data types.

Use casts (or functions that do not take arguments) just like symbols from a symbol table. Functions that accept arguments operate on user-specified values, looking and behaving like C language functions and have the following form:

```
FUNC (arg-list)
```

The following casts are supported by Visual SoftICE:

Table 7-7. Visual SoftICE Casts

Name	Description	Example
Byte	Get low-order byte	`? (Byte) 0x1234 = 0x34`
UChar	Convert to unsigned Char	`? (UChar) 0x12345678 = 0x78`
Bool	Get 1 bit value	`? (Bool) 0x1234 = 0x1`
Word	Get low-order word	`? (Word) 0x12345678 = 0x5678`
Dword	Get low-order dword	`? (Dword) 0xFF =0x000000FF`
Qword	Convert to quad word	`? (Qword) 0x12345678 = 0x12345678`
UlongLong	Convert to unsigned double long	`? (UlongLong) 0x12345678 = 0x12345678`
Short	Convert to short (INT16)	`? (Short) 0x12345678 = 0x5678`
UShort	Convert to unsigned short	`? (UShort) 0x12345678 = 0x5678`

Long	Convert byte or word to signed long	`? (Long) 0xFF = 0x00000000000000FF`
		`? (Long) 0xFFFF = 0x000000000000FFFF`

ULong	Convert to unsigned long	`? (ULong) 0xFF = 0x00000000000000FF`

Visual SoftICE also has a few useful variables:

Table 7-8. Visual SoftICE Predefined Values

Name	Description	Example
IRQL	Windows NT/2000/XP OS IRQ Level	`? IRQL = unsigned-char`
Process	KPEB (Kernel Process Environment Block) of the Active OS process	`? process`
Thread	KTEB (Kernel Thread Environment Block) of the Active OS thread	`? thread`
PID	Active process Id	`? pid == Test32Pid`
TID	Active thread Id	`? tid == Test32MainTid`
BPCount	Breakpoint instance count. For these BP functions, refer to “Conditional Breakpoint Count Functions” on page 114	`bp <bp params> IF bpcount==0x10`
BPTotal	Breakpoint total count	`bp <bp params> IF bptotal>0x10`
BPMiss	Breakpoint instance miss count	`bp <bp params> IF bpmiss==0x20`
BPIndex	Current Breakpoint Index #	`bp <bp params> DO “bd bpindex”`

Visual SoftICE also has a few predefined macro-like functions:

Table 7-9. Visual SoftICE Predefined Macro-like Functions

Name	Description	Example
HiByte	Get high-order byte	`? HiByte(0x1234) = 0x12`
HiWord	Get high-order word	`? HiWord(0x12345678) = 0x1234`

Expression Evaluator Type System

The Visual SoftICE expression evaluator uses a very basic type system that categorizes all expression values into one of the following types:

Table 7-10. Visual SoftICE Expression Types

Type	Example
Literal	`1, 0x80000000, 'ABCD'`
Register	`EAX, DS, ESP`
Symbol-type	`PoolHitTag, IsWindow`
Address-type	`FS:18, &Symbol, WIN32k!CreateCompatiblePublicDC`

In most cases, you can ignore the distinction between types as it is only important to Visual SoftICE. In the cases of symbol-type and address-type, however, there are important semantics or restrictions.

Symbol Type

The symbol-type is used for symbol names that are in export or symbol tables. In general, the type represents the linear address of a symbol within a code or data segment. The symbol type also represents the contents of memory at that linear address. This is similar to the use of a variable in a C program, but because Visual SoftICE is a debugger and not a compiler, there are a few semantic differences. Visual SoftICE determines whether you mean *contents-of* or *address-of* based on the context of how you use the symbol/variable in an expression. In general, the way Visual SoftICE treats a symbol seems completely natural, not unlike that of the C compiler; but, in cases where you are not sure how Visual SoftICE interprets the symbol, you can explicitly state:

```
    address-of (&Symbol) or contents-of (*Symbol)
```

Note: Refer also to the SET EE_IMPL_DEREF command.

When symbol-types are used in expressions, Visual SoftICE will, in most cases, present the result of the expression in the correct type. For example, given an array of integers declared like this:

```
int TinyArray[] = { 1, 2, 3, 4 };
```

The expression:

```
?TinyArray[ 1 ]
```

will cause Visual SoftICE to display the second element of the array, which will be of type **int**.

Alternately, if you have a pointer-to-char expression declared like this:

```
char *str = "Twas Brillig"
```

the expression

```
*str
```

will result in the following display:

```
<char> = 0x54, 'T', 84
```

Address Type

Visual SoftICE treats a symbol as an address-type if you use it in an expression where an address-type is legal and it makes sense to use an address. Otherwise, Visual SoftICE automatically indirects the symbol, taking the contents of the memory the symbol represents. You can also control this behavior with the SET EE_IMPL_DEREF command.

There are many operations that are illegal or do not make sense for address-types such as multiplication and division, so a majority of the operators used with the symbol-type act like a C compiler and automatically take the contents-of at the address for the symbol.

Evaluating Symbols

When data type information is available, using the ? (evaluate expression) command with a symbol yields the contents of the symbol rather than the address of the symbol. For example, MyVariable is an integer variable containing the value 5, so you get the following:

```
? MyVariable
int=0x5,"\0\0\0\x05"
```

To get the address of MyVariable, use the following:

```
? &MyVariable
```

If you use a symbol in conjunction with a command other than ‘?,’ be sure to add the address of the ‘&’ operator where needed. For example, the data display command (D) takes an address as a parameter, so to display the contents of a symbol, you should add the ‘&’ operator:

```
dd &MyVariable
```

Pointer Arithmetic with Symbols

When Visual SoftICE performs arithmetic on a symbol whose type is an address, it will perform C-style pointer arithmetic by scaling the second operand by the size of the first. So, given this declaration:

```
long Numbers[] = { 1, 2, 3, 4 };
long *ptr = Numbers;
```

The Visual SoftICE command

```
? ptr + 1
```

will be equivalent to the same expression in C. Thus, the offset (1) will be scaled by the size of the type pointed to by ptr; in this case, 4 bytes. This causes Visual SoftICE to display the second element of the Numbers array.

Array Symbols In Expressions

Visual SoftICE's array operator allows you to evaluate and display individual members of arrays. It has a couple of limitations, however. First of all, Visual SoftICE does not allow multi-dimensional array expressions. Entering `? mychars[1][1]`, for example, will produce an error.

Secondly, unlike C and C++, Visual SoftICE does not treat pointers and arrays as equivalent. Using an array operator on a pointer type will therefore produce unpredictable results.

Chapter 8
Exploring Windows NT

- Overview
- Inside the Windows Kernel
- Win32 Subsystem

Overview

If you are going to write Windows NT family applications, you should explore what lies beneath your application code: the operating system. The knowledge you gain from the time you invest to go beneath your application and into the depths of the system will benefit both you and the application or driver that you are creating.

This chapter provides a quick overview of the more pertinent and interesting aspects of the basic Windows NT Operating System. By combining this information with available reference material and a little practical application using Visual SoftICE, you should be able to gain a basic understanding of how the components of Windows NT fit together.

Resources for Advanced Debugging

Microsoft provides several resources for advanced Windows NT debugging including: the Windows NT DDK, .DBG files, and Kernel Debugger Extensions.

Windows DDK

The Windows DDK contains header files, sample code, on-line help, and special tools that let you query various kernel components. The most obvious and useful resource is NTDDK.H. Although there is quite a bit of information missing from this header file, enough pertinent information is available to make it worth studying. Besides the basic data structures needed for device driver development, system data structures are described (some completely, others briefly, many not at all). There are many API prototypes and type enumerations that are useful for both exploration and development. There are also useful comments about the system design, as well as restrictions and limitations. Most of the other header files in the DDK are specific to the more esoteric aspects of the system, but `NTDEF.H`, `BUGCODES.H`, and `NTSTATUS.H` are generally useful.

The Windows DDK includes a few utilities that are of general interest. For example, `POOLMON.EXE` allows you to monitor system pool usage, and `OBJDIR.EXE` provides information on the Object Manager hierarchy and information about a specific object within the hierarchy. Visual SoftICE provides similar functionality with the OBJDIR, DEVICE, and DRIVER commands. The utility `DRIVERS.EXE`, like the Visual SoftICE DRIVER command, lists all drivers within the system, including basic information about the driver. Some versions of the Windows DDK include a significantly more powerful version of the standard `PSTAT.EXE` utility. PSTAT is a Win32 console application that provides summary information on processes and threads. Included with the Win32 SDK and the Visual C++ compiler, are two utilities worth noting: PVIEW and SPY++. Both provide information on processes and threads, and SPY++ provides HWND and CLASS information.

The Windows DDK also includes help files and reference manuals for device driver development, as well as sample code. The sample code is most useful, because it provides you with the information necessary for creating actual Windows device drivers. Simply find something in your area of interest, build that sample, and step through it with Visual SoftICE.

.DBG/.PDB Files

***Tip:** Using symbol files is probably the most important aspect of setting up your development and debugging environment.*

Microsoft provides a separate symbol (.PDB/.DBG) file for every distributed executable file with both the checked and free builds of the Windows operating system. This includes the systems components that make up the kernel executive, device drivers, Win32 system DLLs, subsystem processes, control panel applets, and even accessories and games. The symbol files contain basic debug information similar to the PUBLIC definitions of a .MAP file. Every API and global variable, exported or otherwise, has a basic definition (for example, name, section and offset). Advanced type information such as structures and locals may be provided, but having access to a public definition for each API makes debugging through system calls a lot easier.

Regardless of your specific area of interest, obtain symbols for the following key system components. The most important components are listed in bold typeface.

Table 8-1. Key System Component Symbols

Component	
`NTOSKRNL.EXE`	The Windows NT Kernel. (Most of the operating system resides here.)
`HAL.DLL`	The Hardware Abstraction Layer. Important primitives for NTOSKRNL.
`NTDLL.DLL`	Basic implementation of the Win32 API, and functionality traditionally attributed to KERNEL. Also the interface between USER and SYSTEM mode. Essentially replaces `KERNEL32.DLL`.
`CSRSS.EXE`	The Win32 subsystem server process. Most subsystem calls are routed through this process.
`WINSRV.DLL`	Under Windows NT 3.51, the core implementation of USER and GDI functionality. Only loaded in the context of CSRSS.
`WIN32K.SYS`	A system device driver that replaces `WINSRV.DLL` and minimizes inter-process communication between applications and CSRSS. May not be available for all versions of the OS.

Table 8-1. Key System Component Symbols (Continued)

Component	
USER32.DLL	Basic implementation of USER functionality. Mostly stubs to WINSRV.DLL (via LPC to CSRSS). More recent versions contain more implementation to minimize context switches.
KERNEL32.DLL.	Some basic implementation of traditional KERNEL functionality, but mostly stubs to NTDLL.DLL.

Resources

The following resources provide extensive information for developing drivers and applications for Windows NT:

◆ *Microsoft Developers Network* (MSDN)
MSDN is published quarterly, on CD-ROM, and it contains a wealth of information and articles on all aspects of programming Microsoft operating systems. This is one of the only places where you can find practical information on writing Windows NT device drivers.

◆ *Inside Windows NT* - Helen Custer, Microsoft Press
Inside Windows NT provides a high-level view of the design for the Windows NT operating system. Each of the major sub-systems is thoroughly discussed, and many block diagrams illuminate internal data structures, policies, and algorithms. Although the contents of this book may seem highly abstracted from the actual operating system implementation, once you step into OS code with Visual SoftICE, many of the higher level relationships become clear. Currently, this is the most valuable set of information on Windows NT operating system internals. You will gain the most benefit from the information in this book if you use Visual SoftICE to explore the actual implementation of the system design.

◆ *Advanced Windows* - Jeffery Richter, Microsoft Press
Advanced Windows is an excellent resource for the systems programmer developing Win32 applications and system code. Richter presents extensive discussions of processes, threads, memory management, and synchronization objects. Relevant sample code and utilities are also provided.

Inside the Windows Kernel

To gain a basic understanding of Windows, look at the platform from many different perspectives. A general knowledge of how Windows works at different levels enables you to understand the constraints and assumptions involved in designing other aspects of the operating system.

This section explains the most critical component of the operating system, the Windows Kernel. It describes how Windows configures the core operating system data structures, such as the IDT and TSS, and how to use corresponding Visual SoftICE commands to illustrate the Windows configuration of the CPU. It also examines a general map of the Windows system memory area on x86, describing important system data structures and examining the critical role they play within the operating system.

A majority of the information in this section is based on the implementation details of the following two modules:

- Hardware Abstraction Layer (HAL.DLL)
  HAL is the Windows hardware abstraction layer. Its purpose is to isolate as many hardware platform dependencies as possible into one module. This makes the Windows kernel code highly portable. Various parts of the kernel use platform dependent code, but only for performance considerations.

  The primary responsibility of the HAL is to deal with very low-level hardware control such as Interrupt controller programming, hardware I/O, and multiprocessor inter-communication. Many of the HAL routines are dedicated to dealing with specific bus types (PCI, EISA, ISA) and bus adapter cards. HAL also controls basic fault handling and interrupt dispatch.

- The Kernel (NTOSKRNL.EXE)
  The vast majority of the Windows operating system resides in the Windows Kernel, or Kernel Executive. This is the kernel-level functionality that all other system components, such as the Win32 subsystem, are built upon. The Kernel Executive Services cover a broad range of functionality, including:
  - Memory Management
  - Object Manager
  - Process and Thread creation and manipulation
  - Process and Thread scheduling
  - Local Procedure Call (LPC) facilities
  - Security Management
  - Exception handling
  - VDM hardware emulation
  - Synchronization primitives, such as Semaphores and Mutants
  - Run Time Library
  - File System
- I/O subsystems

Managing the Intel x86 Architecture

One of the fundamental requirements of starting a protected-mode operating system is the setup of CPU architecture, policies, and address space that the operating system will use. System initialization is coordinated between NTLDR, NTDETECT, NTOSKRNL, and HAL. Use the following Visual SoftICE commands to obtain a general idea of how Windows uses the Intel x86 architecture to provide a secure and robust environment.

Table 8-2. Visual SoftICE Architecture Commands

Command	Description
IT/IDT	Display information on the Interrupt Descriptor Table
GDT	Display information on the Global Descriptor Table

Note: The *Visual SoftICE Command Reference* provides detailed information about using each command.

IT/IDT (Interrupt Descriptor Table)

Windows creates an IDT for 255 interrupt vectors and maps it into the system linear address space. The first 48 interrupt vectors are generally used by the kernel to trap exceptions, but certain vectors provide operating system services or other special features. Use the Visual SoftICE IDT command to view the Windows Interrupt Descriptor Table.

Table 8-3. Interrupt Descriptor Table

Interrupt #	Purpose
2	NMI. A Task gate is installed here so the OS has a clean set of registers, page-tables, and level 0 stack. This enables the operating system to continue processing long enough to throw a *Blue Screen*.
8	Double Fault. A Task gate is installed here so the OS has a clean set of registers, page-tables, and level 0 stack. This enables the operating system to continue processing long enough to throw a *Blue Screen*.
2A	Service to get the current tick count.
2B,2C	Direct thread switch services.
2D	Debug service.
2E	Execute System Service. Windows NT transitions from user mode to system mode using INT 2E. For more information, refer to the NTCALL command in the *Visual SoftICE Command Reference*.
The following are Non-APIC only	
30-37	Primary Interrupt Controller (IRQ0-IRQ7). 30 - HAL clock interrupt (IRQ0).
38-3F	Secondary Interrupt Controller (IRQ8-IRQ15).

Non-APIC only: Interrupt vectors 0x30 - 0x3F are mapped by the primary and secondary interrupt controllers, so hardware interrupts for IRQ0 through IRQ15 are vectored through these IDT entries. In many cases, these hardware interrupt vectors are not hooked, so the system assigns default stub routines for each one. As devices require the use of these hardware interrupts, the device driver requests to be connected. When the interrupt is no longer needed, the device driver requests to be disconnected.

The default stubs are named KiUnexpectedInterrupt#, where # represents the unexpected interrupt. To determine which interrupt vector is assigned to a particular stub, add `0x30` to the UnexpectedInterrupt#. For example, KiUnexpectedInterrupt2 is actually vectored through IDT vector 32 (`0x30 + 2`).

Drivers may install and uninstall interrupt handlers as necessary, using IoConnectInterrrupt and IoDisconnectInterrupt. These routines create special thunk objects, allocated from the Non-Pageable Pool, which contain data and code to manage simultaneous use of the same interrupt handler by one or more drivers.

TSS (Task State Segment)

The purpose of the TSS is to save the state of the processor during task or context switches. For performance reasons, Windows does not use this architectural feature and maintains one base TSS that all processes share. As noted in the previous section on the Windows IDT, other TSS data types exist, but are only used during exceptional conditions to ensure that the system will not spontaneously reboot before Windows can properly crash itself.

GDT (Global Descriptor Table)

Windows on x86 is a flat, 32-bit architecture. Thus while it still needs to use selectors, it uses them minimally. Most Win32 applications and drivers are completely unaware that selectors even exist. The following is abbreviated output from the Visual SoftICE GDT command that shows the selectors in the Global Descriptor Table.

```
Global Descriptor Table - Base Address: 8003f000, Limit: 3ff
Count: 24

Selector    Type                                    Address     Limit       DPL   Granularity    Present
---------------------------------------------------------------------------------------------------------
8           Code: Execute/Readable (accessed)       0           ffffffff    0     Page           P
10          Data: Read-Write (accessed)             0           ffffffff    0     Page           P
1B          Code: Execute/Readable (accessed)       0           ffffffff    3     Page           P
23          Data: Read-Write (accessed)             0           ffffffff    3     Page           P
28          32bit TSS (busy)                        80042000    20ab        0     Byte           P
30          Data: Read-Write (accessed)             ffdff000    1fff        0     Page           P
3B          Data: Read-Write (accessed)             0           fff         3     Byte           P
43          Data: Read-Write                        400         ffff        3     Byte           P
50          32bit TSS (available)                   8053c180    68          0     Byte           P
58          32bit TSS (available)                   8053c1e8    68          0     Byte           P
60          Data: Read-Write (accessed)             22f20       ffff        0     Byte           P
68          Data: Read-Write                        b8000       3fff        0     Byte           P
70          Data: Read-Write                        ffff7000    3ff         0     Byte           P
78          Code: Execute/Readable                  80400000    ffff        0     Byte           P
80          Data: Read-Write                        80400000    ffff        0     Byte           P
88          Data: Read-Write                        0           0           0     Byte           P
a0          32bit TSS (available)                   80f728a8    68          0     Byte           P
e0          Code: Conform,Exec/Rdble (accessed)     f92c1000    ffff        0     Byte           P
e8          Data: Read-Write                        0           ffff        0     Byte           P
f0          Code: Execute-Only                      804fc26c    f0baa       0     Byte           P
f8          Data: Read-Write                        0           ffff        0     Byte           P
100         Data: Read-Write (accessed)             f8ff7040    ffff        0     Byte           P
108         Data: Read-Write (accessed)             f8ff7040    ffff        0     Byte           P
110         Data: Read-Write (accessed)             f8ff7040    ffff        0     Byte           P
```

Note that the first four selectors address the entire 4GB linear address range. These are flat selectors that Win32 applications and drivers use. The first two selectors have a DPL of zero and are used by device drivers and system components to map system code, data, and stacks. The selectors 1B and 23 are for Win32 applications and map user level code, data, and stacks. These selectors are constant values and the Windows NT system code makes frequent references to them using their literal values.

The selector value `30h` addresses the Kernel Processor Control Region and is always mapped at a base address of `0xFFDFF000`. When executing system code, this selector is stored in the FS segment register. Among its many other purposes, the Processor Control Region maintains the current kernel mode exception frame at offset 0.

Similarly, the selector value `3Bh` is a user-mode selector that maps the current user thread environment block (UTEB). This selector value is stored in the FS segment register when executing user level code and has the current user-mode exception frame at offset 0. The base address of this selector varies depending on which user-mode thread is running. When a thread switch occurs, the base address of this GDT selector entry is updated to reflect the current UTEB.

Windows NT System Memory Map

Windows reserves the upper 2GB of the linear address space for system use. The address range `0x80000000 - 0xFFFFFFFF` maps system components such as device drivers, system tables, system memory pools, and system data structures such as threads and processes. While you cannot create an exact map of the Windows system memory space, you can categorize areas that are set aside for specific usage. The following System Memory Map diagram gives you a rough idea of where operating system information is located. Remember that a majority of these system areas could be mapped anywhere within the system address space, but are generally in the address ranges shown.

- System Code area
  Boot drivers and the NTOSKRNL and HAL components are loaded in the System Code address space. Non-boot drivers are loaded in the NonPaged system address space near the top of the linear address space. You can use the Visual SoftICE IMAGE/MOD and IMAGEMAP/MAP32 commands to examine the base address and extents of boot drivers loaded in this memory area. This is also where the TSS, IDT, and GDT system data structures are mapped.

- System View area
  The System View address space is symbolically referenced, but does not ever seem to be mapped under Windows NT 3.51. Under newer versions of Windows , the System View address space maps the global tables for GDI and USER objects. You can use the Visual SoftICE OBJTAB command to view information about the USER object table.

◆ System Tables Area

This region of linear memory maps process page tables and related data structures. This is one of the few areas of system memory that is not truly global, in that each process has unique page tables. When Windows executes a process context switch, the physical address of the process Page Directory is extracted from the kernel process environment block (KPEB) and loaded into the CR3 register. This causes the process page tables to be mapped in this memory area. Although the linear addresses remain the same, the physical memory used to back this area contains process-specific values. When you use the Visual SoftICE ADDR command to change to a specific process context, you are referencing the Page Directory information for this process.

To manage the mapping of linear memory to physical memory, Windows reserves a 4MB region of the system linear address space for Page Tables. This region represents the entire range of memory necessary to fully define a Page Directory and complete set of page tables. The need for a 4MB region can be calculated given that there is one Page Directory structure which contains entries for 1024 Page Tables. To map a 4GB linear address space, each Page Table must map a 4MB region of linear address space (4GB /1024). Each Page Table is a multiple of the CPU page size (4KB under Windows on x86), so multiplying 1024 by 4096 (the page size) yields the expected 4MB value. An operating system using paging and a 4KB page size requires 4MB of memory to map the entire address space. The following diagram shows the system memory map for Windows on x86.

In this design, the Page Directory is actually performing two functions. In addition to being the Page Directory, representing 4GB, it also serves as a page table, representing 4MB in the address range `0xC0000000 - 0xC03FFFFF`. The Page Directory maps the 4MB region where the process page tables are mapped (`0xC0000000-0xC03FFFFF`), so the Page Directory entry that maps this area must point to itself. If you use the Visual SoftICE PAGE command, the physical address of the Page Directory displayed at the top of the command output matches the physical address for the entry that maps the `0xC0000000 - 0xC03FFFFF` memory range. If you use the Visual SoftICE ADDR command to obtain the CR3 (the CR3 register contains the physical address of the Page Directory) value for the current process and supply this value as input to the Visual SoftICE PHYS command, all the linear addresses that are mapped to the physical address of the Page Directory are displayed. One of the addresses is `0xC0300000`.

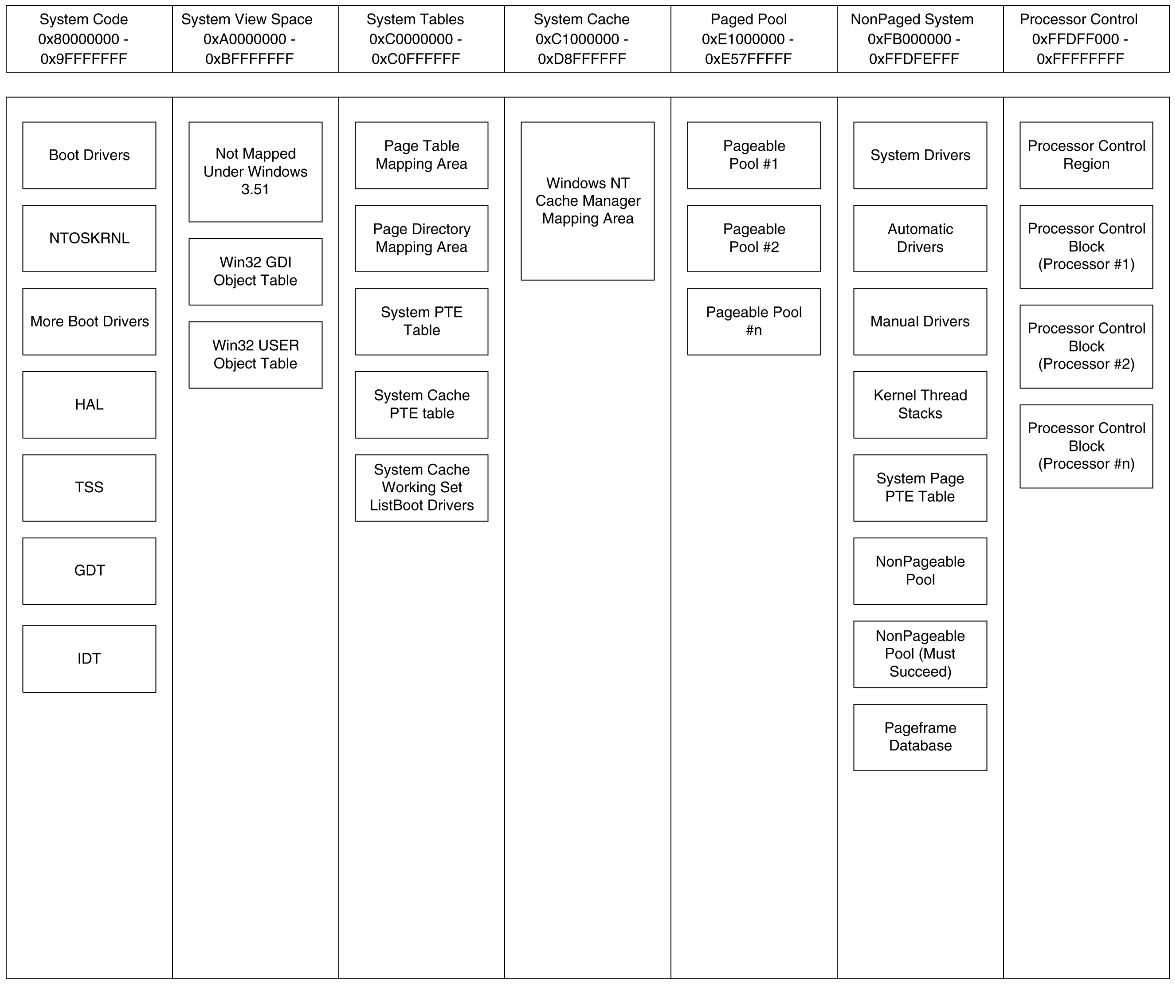

System Code 0x80000000 - 0x9FFFFFFF
System View Space 0xA0000000 - 0xBFFFFFFF
System Tables 0xC0000000 - 0xC0FFFFFF
System Cache 0xC1000000 - 0xD8FFFFFF
Paged Pool 0xE1000000 - 0xE57FFFFF
NonPaged System 0xFB000000 - 0xFFDFEFFF
Processor Control 0xFFDFF000 - 0xFFFFFFFF
Boot Drivers
NTOSKRNL
More Boot Drivers
HAL
TSS
GDT
IDT
Not Mapped Under Windows 3.51
Win32 GDI Object Table
Win32 USER Object Table
Page Table Mapping Area
Page Directory Mapping Area
System PTE Table
System Cache PTE table
System Cache Working Set ListBoot Drivers
Windows NT Cache Manager Mapping Area
Pageable Pool #1
Pageable Pool #2
Pageable Pool #n
System Drivers
Automatic Drivers
Manual Drivers
Kernel Thread Stacks
System Page PTE Table
NonPageable Pool
NonPageable Pool (Must Succeed)
Pageframe Database
Processor Control Region
Processor Control Block (Processor #1)
Processor Control Block (Processor #2)
Processor Control Block (Processor #n)

Figure 8-1. Windows NT System Memory Map

The following example illustrates how all these values interrelate. Important values are show in bold typeface.

1 Use the ADDR command to obtain the *physical* address of the Page Directory (Page Dir).

```
:addr
Page Dir     KPEB Addr    PID     Name
00030000     FF116020     0002    System
0115A000     FF0AAA80     0051    RpcSs
0073B000     FF083020     004E    nddeagnt
00653000     FF080020     0061    ntvdm
00AEE000     FF07A600     0069    Explorer
01084000     FF06ECA0     0077    FINDFAST
010E9000     FF06CDE0     007B    MSOFFICE
*01F6E000    FF088C60     006A    WINWORD
01E0A000     FF09CCA0     008B    4NT
017D3000     FF09C560     006D    ntvdm
00030000     80140BA0     0000    Idle
```

2 Use the physical address as input to the PHYS command to obtain all linear addresses that map to that physical page. (One physical page may be mapped to more than one linear address, and one linear address may be mapped to more than one page.)

```
:phys 1F6E000
C0300000
```

3 Use the linear address (C0300000) and run it through the PAGE command to verify the physical page for that linear address.

```
:page C0300000
Linear      Physical    Attributes
C0300000    01F6E000    P D A S RW
```

System Page Table Entries and ProtoPTEs

The acronym, PTE, which appears in various places on the system map, stands for Page Table Entry. A Page Table Entry is one of the 1024 entries that is contained in a Page Table. Each PTE describes one page of memory, including its physical address and attributes. Because Windows also runs on non-Intel platforms, and because the operating system may need to extend the types of page-level protection beyond what any particular CPU may provide, Windows virtualizes the CPU PTE with what is referred to as a ProtoPTE. The ProtoPTE is similar to the Intel Architecture PTE, but includes attributes that are not provided by the Intel PTE. By overloading the meaning of an attribute bit within an Intel PTE, the operating system can gain control on a page fault, and examine the extended attributes of the corresponding ProtoPTE to determine why the operating system requested that the fault occur. Throughout NTOSKRNL, manipulations are performed on the ProtoPTE abstraction, and translated to the actual CPU PTE type. Note that the operating system also compares the ProtoPTE to its corresponding CPU PTE to ensure their consistency. This effectively prevents an application or device driver from directly manipulating the page table entries.

- **Paged Pool Area:** The Paged Pool system memory area is where `ntoskrnl!ExAllocatePool` and its related functions allocate memory that can be paged to disk. This is in direct contrast to the Non-Paged pool area. Non-Paged pool allocations are never paged to disk and are designed for routines such as Interrupt Handlers that need high performance or need a guarantee that a piece of information is always available for use.

  Windows makes extensive use of the Paged pools, as this is where most operating system objects are created. Note that the starting address and the size and number of paged pools is determined dynamically during system initialization. Only use the addresses presented here as a guideline. For the actual addresses, load the symbols for NTOSKRNL and examine the appropriate variables that describe the paged pool configuration. (To see several of them, use the Visual SoftICE SYM command with the Parameter “MmPaged*”.)

Although there is one Paged Pool area, there are multiple paged pools. The number is determined during system initialization. Paged pool allocations occur with relatively high frequency and those accesses must be thread safe, so having one data structure which must be owned exclusively by one thread during memory allocation or deallocation creates a bottleneck. To avoid potential traffic jams and reduced system performance, multiple pool descriptors are created, each with its own private data structures, including an executive spinlock for thread synchronization. Thus, the more paged pools created, the more threads that can perform paged pool allocations simultaneously, increasing the throughput of the system. In case you plan on using similar techniques in your driver or application, an important design note to remember is that the overhead for a Paged Pool (or Non-Paged Pool) descriptor is very minimal. Thus it is practical for four or five of them to exist. However, determine that an actual bottleneck exists before creating elaborate schemes to solve a non-existent problem.

- **Non-Paged System Area:** This linear region is intended for system components and data structures that need to be present in memory at all times. This includes non-boot drivers, kernel mode thread stacks, two Non-Paged memory pools, and the Page Frame Database. Although it is contradictory to say that items in the Non-Paged System area can become "not present", the truth is that they can be. Specifically, kernel thread stacks and process address spaces can be made not present, and often are.

  The Non-Paged pool is similar to the Paged Pool with the exception that objects created in the Non-Paged pool are not discarded from memory for any reason. The Non-Paged pool is used to allocate key system data structures such as kernel process and thread environment blocks. There is a second Non-Paged pool used for memory allocations that *must succeed.* At system initialization, NTOSKRNL reserves a small amount of physical memory for critical allocations, and saves this memory for use by the must succeed pool. The size of an allocation from the must succeed pool must be less than one page (4KB). If the must succeed allocation cannot be satisfied, or the requested allocation size is larger than 4KB, the system throws a *Blue Screen*.

- **Processor Control Region:** At the high end of the system memory area is the Processor Control Region. Here, Windows maintains Processor Control Block (PCRB) data structures for each processor within the system and a global data structure, the Processor Control Region that reflects the current state of the system. The Processor Control Region (PCR) contains key pieces of information about the current state of the system, such as the currently running kernel thread; the current interrupt request level (IRQL); the current exception frame; base addresses of the IDT, TSS, and GDT; and kernel thread stack pointers. Small portions of the PCR and PCRB data structures are documented in `NTDDK.H`.

  In many cases, device driver writers need to know the current IRQL at which they are executing. Although you could look inside the PCR data structure at offset `0x24`, it is simpler to look at the status bar or use the Visual SoftICE intrinsic function, *IRQL*, as follows:

  ```
  ? IRQL
  00000002h
  ```

  The most common piece of data accessed from the PCRB is the current kernel thread pointer. This is at offset 4 within the PCRB, but is generally referenced through the PCR at offset `0x124`. This works because the PCRB is nested within the PCR at offset `0x120`. Code that accesses the current thread is usually of the form:

  ```
  mov reg, FS:[124].
  ```

  Remember that while executing in system mode, the FS register is set to a GDT selector whose base address points to the beginning of the PCR. Visual SoftICE makes it much easier to get the current thread pointer or thread id by viewing the context bar, or using the intrinsic functions *thread or tid*:

  ```
  ? thread
  FF088E90h
  ```

  ```
  ? tid
  71h
  ```

For more extensive information on the current thread use the following commands:

```
:thread tid
TID      Krnl TEB     StackBtm     StkTop       StackPtr     User TEB     Process(Id)
0071     FF0889E0     FC42A000     FC430000     FC42FE5C     7FFDE000     WINWORD(6A)

:thread thread
TID      Krnl TEB     StackBtm     StkTop       StackPtr     User TEB     Process(Id)
0071     FF0889E0     FC42A000     FC430000     FC42FE5C     7FFDE000     WINWORD(6A)
```

The current process is not stored as part of the PCR or PCRB. Windows references the current process through the current thread. Code such as the following obtains the current process pointer:

```
mov    eax,    FS:[124]     ; get the current thread (KTEB)
mov    esi,    [eax+40h]    ; get the threads process pointer (KPEB)
```

Note: All this information is available in the Visual SoftICE context bar.

Win32 Subsystem

Inside CSRSS

The Win32 subsystem server process CSRSS implements the Win32 API. The Win32 API provides many different types of service, including functionality traditionally attributed to the original Windows components KERNEL, USER, and GDI. Although these standard modules exist in the form of 32-bit DLLs under Windows NT 3.51, and to a lesser degree under new versions of the operating system, most of the core functionality is actually implemented in `WINSRV.DLL` within the CSRSS process. Calls that are traditionally associated with one of the standard Windows components are typically implemented as stubs that call other modules, for example, `NTDLL.DLL`, or use inter-process communication to CSRSS for servicing.

Most USER and GDI API calls are routed through the appropriate 32-bit module in the process address space. There, they are packaged as Local Procedure Call (LPC) messages and routed to CSRSS for processing. As you might imagine, this LPC mechanism, although much more optimized than a true Remote Procedure Call (RPC), has much more overhead than a simple function call. It is surprising to think that every time your application calls the IsWindow function in USER32.DLL, it must be packaged for LPC and sent as a subsystem message to CSRSS. For CSRSS to be able to process this message, a process switch must occur and a worker thread must be awoken and dispatched. The specific service must be determined, parameters must be validated, and finally, the service must be executed. When everything is complete on the CSRSS side, a LPC reply must be made to the client (your application), which involves another process switch and unpackaging of the LPC reply. All that just to determine if a handle represents a valid window.

In their design of a forthcoming version of Windows NT, Microsoft is working to remove as much of this overhead as possible. First, they are moving much of the functionality of WINSRV.DLL into the actual USER32 and GDI32 modules that are loaded into your application's address space. This allows the most common services to execute as simple function calls; no LPC is necessary. Second, rather than making a context switch into CSRSS to access functionality in WINSRV.DLL, a new system driver, WIN32K.SYS allows USER and GDI services to execute more efficiently through a simple transition from user to system mode. Having WIN32K.SYS as a device driver that provides application services allows Windows NT to maintain a high level of encapsulation and robustness, while providing a much more efficient pseudo client-server service architecture.

Although CSRSS executes as a separate process, it still has a big impact on the address space of every Win32 application. If you use the Visual SoftICE HEAP command on your process, you will notice at least two heaps that your application did not specifically create, but were created on its behalf. The first is the default process heap that was created during process initialization. The second is a heap specifically created by CSRSS. There may be other heaps in your application address space that were not created by your process. These heaps are generally located very high in the user-mode address space and appear if you use the Visual SoftICE QUERY command, but do not appear in the output of the HEAP command. The reason for this is quite simple: for each user-mode process, a list of process heaps is maintained and the Visual SoftICE HEAP command uses this list to enumerate the heaps for a process. If the heap was not created by or on behalf of your application, it does not appear in the process heap list. The Visual SoftICE QUERY command traverses the user-mode address space for your application.

Heaps that exist in the process address space, but that are not enumerated in the process heap list, were mapped into the process address space by another process. In most cases, this mapping is done by CSRSS. During subsystem initialization, CSRSS creates a heap at a well-known base address. When new processes are created, this heap is mapped into their address spaces at the same well-known base address. Theoretically, mapping the heap of one process at the same base address of another process allows both processes to use that heap. In practice, there are issues that might prevent this from working under all circumstances — synchronization being one such issue. Note that under newer versions of Windows , more than one heap may be mapped into the process address space, and those heaps may be mapped at different base addresses in different processes. The Visual SoftICE QUERY command notes this condition in its output. Also, new versions of the operating system use heaps that are created in the system address space, and these heaps are sometimes mapped into the user address space. Windows allows the creation of heaps within the system address space using APIs exported from NTOSKRNL. These APIs are similar to the same APIs exported from the user-mode module, NTDLL.DLL.

USER and GDI Objects

Under Windows NT 3.51, the protected Win32 subsystem process, CSRSS, provides a majority of the traditional USER functionality. APIs and data structures provided by the `WINSRV.DLL` module manage window classes, and window data structures, as well as many other USER data types.

Under Windows NT 3.51, the following USER object types exist.

FREE	Free handle
HWND	Window handle
HMENU	Menu handle
HCURSOR	Cursor handle
HDWP	Deferred window position handle
HHOOK	Window hook callback handle
CLIPDATA	Clipboard data handle
QUEUE	Call procedure handle
HACCEL	Accelerator table handle
DDEACCESS	DDE access handle
HCONV	DDE conversion handle
HDDEDATA	DDE data handle
HMONITOR	Display monitor handle
HKL	Keyboard layout handle
HKF	Keyboard layout file handle
HWINEVENTHOOK	Window event hook callback handle
HWINSTA	Window station handle
HIMC	Input context handle
HHID	Human interface device data handle
HDEVINFO	Device information set handle
DESKTOP	Window handle that is a Desktop type window

Rather than maintaining per-process data structures for USER and GDI object types, CSRSS maintains a master handle table for all processes. The USER and GDI objects are segregated into two different tables that have the same basic structure and semantics. WINSRV provides distinct Handle Manager APIs for managing the two different tables. You can identify the handle manager API names by the HM prefix in front of the API name, and the GDI specific routines by the “g” appended to this prefix. The routine HMAllocObject creates USER object types, while HmgAlloc is a GDI object type API that creates GDI object types.

The management of USER and GDI handles is relatively straightforward, and its design is a good example of how to implement basic management of abstract object types. Specifically, this API uses a simple, but robust, technique for creating unique handles and managing reference counts. The design also provides for handle opaqueness which prevents applications, including USER32 and CSRSS, from directly manipulating the objects outside the handle manager. Preventing clients, including itself, from directly manipulating the object data allows the handle manager to ensure that reference counts and synchronization issues are managed correctly.

The master object tables maintained by the Handle Manager are growable arrays of fixed size entries. The following table lists the fields for an object table. Only columns with **bold** field headers are part of the entry. The columns with *italicized* headers are for illustration only.

Entry	**Object Pointer (DWORD)**	**Owner (DWORD)**	**Type (BYTE)**	**Flags (BYTE)**	**Instance Count (WORD)**	*Handle Value*
0	NULL	NULL	FREE (0)	00	0001	00010000
1	HEAP *	HEAP *	DESKTOP (0C)	00	0001	00010001
2	HEAP *	HEAP *	HWND (04)	01	0003	00030002

The Object Pointer field points to the actual object data. This pointer is generally from one of the CSRSS heaps or the Paged Pool. The type field is the enumeration for the object type. The Instance Count field creates unique handles. The Flags field is used by the Handle Manager to note special conditions, such as when a thread locks an object for exclusive use.

How Handle Values Are Created

Initially, all object table Instance counts are set to one. When a new Object Entry is allocated, the Instance Count is combined with the table index to create a unique handle value. When references are made to an object, the table entry portion of the handle is extracted and used to index into the table. As part of the handle validation, the instance count is extracted from the table entry and compared to the handle being validated. If the instance count does not match the table entry instance count, the handle is bogus. The following example illustrates these concepts:

To create an object handle from an object table entry:

```
Object Handle = Table Entry Index + (InstanceCount << 16);
```

To validate an object handle:

```
ObjectTable [LOWORD(handle)]. InstanceCount == HIWORD(handle);
```

When an object is destroyed, all fields are reinitialized to zero and the current Instance Count for that entry is incremented by one. Thus, when the object table entry is reused, it generates a different handle value for the new object.

Note: The actual object type is not part of the object handle value. This means that given an object handle, an application cannot directly determine its type. It is necessary to dereference the object table entry to obtain the object type.

This technique for creating unique handle values is simple and efficient, and makes validation trivial. Imagine the case where a process creates a window and obtains a handle to that window. During subsequent program execution, the process destroys the window but retains the handle value. If the process uses the handle after the window is destroyed, the handle value is invalid and the type it points to has an object type of FREE. This condition is caught, and the program is not be able to use the handle successfully. In the meantime, if another process creates a new object, it is likely that the entry originally for the now destroyed window will be reused. If the original program uses the invalid window handle, the handle instance counts no longer match, and the validation fails.

Object tables are not process specific, so USER and GDI object handles values are not unique to a specific process. HWND handles are unique across the entire Win32 subsystem. One process never has an HWND handle value that is duplicated in any other process.

USER Object Table

Use the Visual SoftICE OBJTAB command to display all the object entries within the USER object table. The OBJTAB command is relatively flexible, allowing a handle or table entry index to be specified. It also supports the display of objects by type using abbreviations for the object type names. To see a list of object type names that the OBJTAB command can use, specify the -H option on the OBJTAB command line.

The Object Pointer field can reference the object specific data for an object table entry. All objects have a generic header that is maintained by the object manager, which includes the object handle value and a thread reference count. Most object types also contain a pointer to a desktop object and/or a pointer to its owner.

The following example shows an object table entry for a window handle and a data dump of the object header maintained by the handle manager. Key information from the command output is listed in bold.

1 Use the Visual SoftICE OBJTAB command to find an arbitrary window handle and obtain the object pointer. In this example, the handle value is 0x1000C and the owner field is 0xE12E7008:

```
:objtab hwnd
Object          Type                 Id        Handle          Owner           Flags
E12E9EA8        Hwnd                 01        0001001C        E12E7008        00
```

2 Dumping 0x20 bytes of the object data reveals the following:

```
:dd e12e9ea8 l 20
0010:E12E9EA8   0001001C        00000006        00000000        FF0E45D8
0010:E12E9EB8   00000000        E12E7008        00000000        00000000
```

The value 0x1001C, at offset 0, is the object handle value. The field at offset 4, which contains the value six (6), is the object reference count. The value at offset 0x0C, of 0xFF0E45D8, is a pointer to the window's desktop object.

3 Verify this using the Visual SoftICE HWND command as follows:

```
:hwnd 0001001c
-----------------------------------------------------
Handle      : 1001c
CLASS       : System : Desktop
Module      : 3b7de6b41a690482; win32k.sys
WindowProc  : bf8624b8 (win32k!_Section.text+62138)
Title       :
Parent      : 00000000
Next        : 00000000
1st Child   : bcab06e8
Style       : 96000000;
WS_POPUP,WS_VISIBLE,WS_CLIPSIBLINGS,WS_CLIPCHILDREN
ExStyle     : 0;
Window Area : -16384, -16384, 16383, 16383 (32767 x 32767)
Client Area : -16384, -16384, 16383, 16383 (32767 x 32767)
```

4 Dumping `0x20` bytes at the address of the owner data reveals the following:

```
:dd e12e7008 l 20
0010:E12E7008  0001001B      00000000      00000000      E12E9C34
0010:E12E7018  E17DB714      00000000      00000000      00000000
```

5 The value (0x1001B) at offset 0 of the owner data looks like an object handle, but it is a thread information object. The following example uses the OBJTAB command with 0x1001B as the parameter to show the type for the owner data.

```
:objtab 1001b
Object        Type                 Id       Handle      Owner       Flags
E12E7008      Thread Info          06       0001001B    00000000    00
```

Monitoring USER Object Creation

If you do a considerable amount of Win32 application development, the HMAllocObject API is a convenient place to monitor creation of object types such as windows. Use the Visual SoftICE MACRO command to create a breakpoint template that can trap creation of specific object types as follows:

```
:MACRO obx = “bpx winsrv!HMAllocObject if (esp+c == %1)”
```

The HMAllocObject API is implemented in `WINSRV.DLL` and the object type being created is the third parameter, which translates to `Dword ptr esp [ 0Ch ]`. The “%1” portion of the conditional expression is a place holder for argument replacement. When you execute the OBX macro, the argument provided is inserted into the macro stream at the “%1”:

```
   :OBX 1 -> bpx winsrv!HMAllocObject if (esp+c == 1)
```

When this breakpoint is instantiated, it traps all calls to HMAllocObject that creates window object types.

Process Address Space

The address space for a user-mode process is mapped into the lower 2GB of linear memory at addresses `0x00000000 - 0x7FFFFFFF`. The upper 2GB of linear memory is reserved for the operating system kernel and device drivers.

In general, each Win32 application’s process address space has the following regions of linear memory mapped for the corresponding purpose.

Table 8-4. Process Address Space

Linear Address Range	Purpose
`0x00000000 - 0x0000FFFF`	Protected region. Useful for detecting NULL pointer writes.
`0x00010000`	Default load address for Win32 processes.
`0x70000000 - 0x78000000`	Typical range for Win32 subsystem DLLs to be loaded.
`0x7FFB0000 - 0x7FFD3FFF`	ANSI and OEM code pages. Unicode translation table(s).
`0x7FFDE000 - 0x7FFDEFFF`	Primary user-mode thread environment block.
`0x7FFDF000 - 0x7FFDFFFF`	User-mode process environment block (UPEB).
`0x7FFE0000 - 0x7FFE0FFF`	Message queue region.
`0x7FFFF000 - 0x7FFFFFFF`	Protected region.

Under Windows, the lowest and highest 64KB regions in the user-mode address space are reserved and are never mapped to physical memory. The 64KB at the bottom of the linear address space is designed to help catch writes through NULL pointers.

The default load address for processes under Windows is 0x10000. Processes often change their load address to a different base address.

The linear range at 0x70000000 is an approximation of the area where Win32 subsystem modules load. Use the Visual SoftICE IMAGE/MOD, IMAGEMAP/MAP32, or QUERY/ADDRESSMAP commands to obtain information on modules loaded in this range.

The user process environment block is always mapped at 0x7FFDF000, while the process's primary user-mode thread environment block is one page below that at 0x7FFDE000. As a process creates other worker threads, they are mapped on page boundaries at the current, highest unused linear address.

The following use of the Visual SoftICE THREAD command shows how each subsequent thread is placed one page below the previous thread:

```
:thread winword
TID     Krnl TEB   StackBtm  StkTop    StackPtr  User TEB   Process(Id)
006B    FFA7FDA0   FEAD7000  FEADB000  FEADAE64  7FFDE000   WINWORD(83)
007C    FF0A0AE0   FEC2A000  FEC2D000  FEC2CE18  7FFDD000   WINWORD(83)
009C    FF04E4E0   FC8F9000  FC8FC000  FC8FBE18  7FFDC000   WINWORD(83)
```

To find out more about the user-mode address space of a process, use the Visual SoftICE QUERY command. The QUERY command provides a high-level view of the linear regions that were reserved and/or committed. It uses the Visual SoftICE WHAT engine to identify the contents of a linear range. From its output you see the process heaps, modules, and memory-mapped files, as well as the thread stacks and thread environment blocks.

Heap API

Heap Architecture

Every user-mode application directly or indirectly uses the Heap API routines, which are exported from KERNEL32 and NTDLL. Heaps are designed to manage large areas of linear memory and sub-allocate smaller memory blocks from within this region. The core implementation of the Heap API routine is contained within NTDLL, but some of the application interfaces such as HeapCreate and HeapValidate are exported from KERNEL32. For some API routines, such as HeapFree, there is no code implementation within KERNEL32, so they are fixed by the loader to point at the actual implementation within NTDLL.

Note: The technique of fixing an export in one module to the export of another module is called 'Snapping'.

Although the Heap API routines used by applications are relatively straightforward and designed for ease of use, the implementation and data structures underneath are quite sophisticated. The management of heap memory has come quite a long way from the standard C run-time library routines `malloc()` and `free()`. Specifically, the Heap API handles allocations of large, non-contiguous regions of linear memory, which are used for sub-allocation and to optimize coalescing of adjacent blocks of free memory. The Heap API also performs fast look-ups of best-fit block sizes to satisfy allocation requests, provides thread-safe synchronization, and supplies extensive heap information and debugging support.

The primary heap data structure is large, at approximately 1400 bytes, for a free build and twice that for a checked build. This does not include the size of other data structures that help manage linear address regions. A vast majority of this overhead is attributed to 128 doubly-linked list nodes that manage free block chains. Small blocks, less than 1KB in size, are stored with other blocks of the same size in doubly linked lists. This makes finding a best-fit block very fast. Blocks larger than 1KB are stored in one sorted, doubly-linked list. This is an obvious example of a time versus space trade-off, which could be important to the performance of your application.

To understand the design and implementation of the Heap API, it is important to realize that a Win32 heap is not necessarily composed of one section of contiguous linear memory. For growable heaps, it might be necessary to allocate many linear regions, using VirtualAlloc, which will generally be non-contiguous. Special data structures track all the linear address regions that comprise the heap. These data structures are call Heap Segments. Another important aspect of the Heap API design is the use of the two-stage process of reserving and committing virtual memory that is provided by the VirtualAlloc and related APIs. Managing which memory is reserved and which memory is committed requires special data structures known as Uncommitted Range Tables, or UCRs for short.

The `Ntdll!RtlCreateHeap()` API implements heap creation and initialization. This routine allocates the initial virtual region where the heap resides and builds the appropriate data structures within the heap. The heap data structure and Heap Segment #1 reside within the initial 4KB (one page) of the virtual memory that is initially allocated for the heap. Heap Segment #1 resides just beyond the heap header. Heap Segment #1 is initialized to manage the initial virtual memory allocated for the heap. Any committed memory beyond Heap Segment #1 is immediately available for allocation through HeapAlloc(). If any memory within Heap Segment #1is reserved, a UCR table entry is used to track the uncommitted range.

Note: `Kernel32!HeapAlloc()` is 'Snapped' to Ntdll!RtlAllocateHeap.

Besides the 128 free lists mentioned above, the heap header data structure contains eight UCR table entries, which should be sufficient for small heaps, although as many UCRs as are necessary can be created. It also contains a table for sixteen (16) Heap Segment pointers. A heap can never have more than sixteen segments, as no provision is made for allocating extra segments entries. If the heap requires thread synchronization, the heap header appends a critical section data structure to the end of the fixed size portion of the heap header preceding Heap Segment #1.

The diagram on the next page is a high-level illustration of how a typical heap is constructed, and how the most important pieces relate to each other.

The left side of the diagram represents a region of virtual memory that is allocated for the heap. The heap header appears at the beginning of the allocated memory and is followed by Heap Segment #1. The first entry within the heap's segment table points to this data structure. Committed memory immediately follows Heap Segment #1. This memory is initially marked as a free block. When an allocation request is made, assuming this block of memory is large enough, a portion is used to satisfy the allocation and the remainder continues to be marked as a free block. Beyond the committed region is an area of memory that is reserved for future use. When an allocation request requires more memory than is currently committed, a portion of this area is committed to satisfy the request.

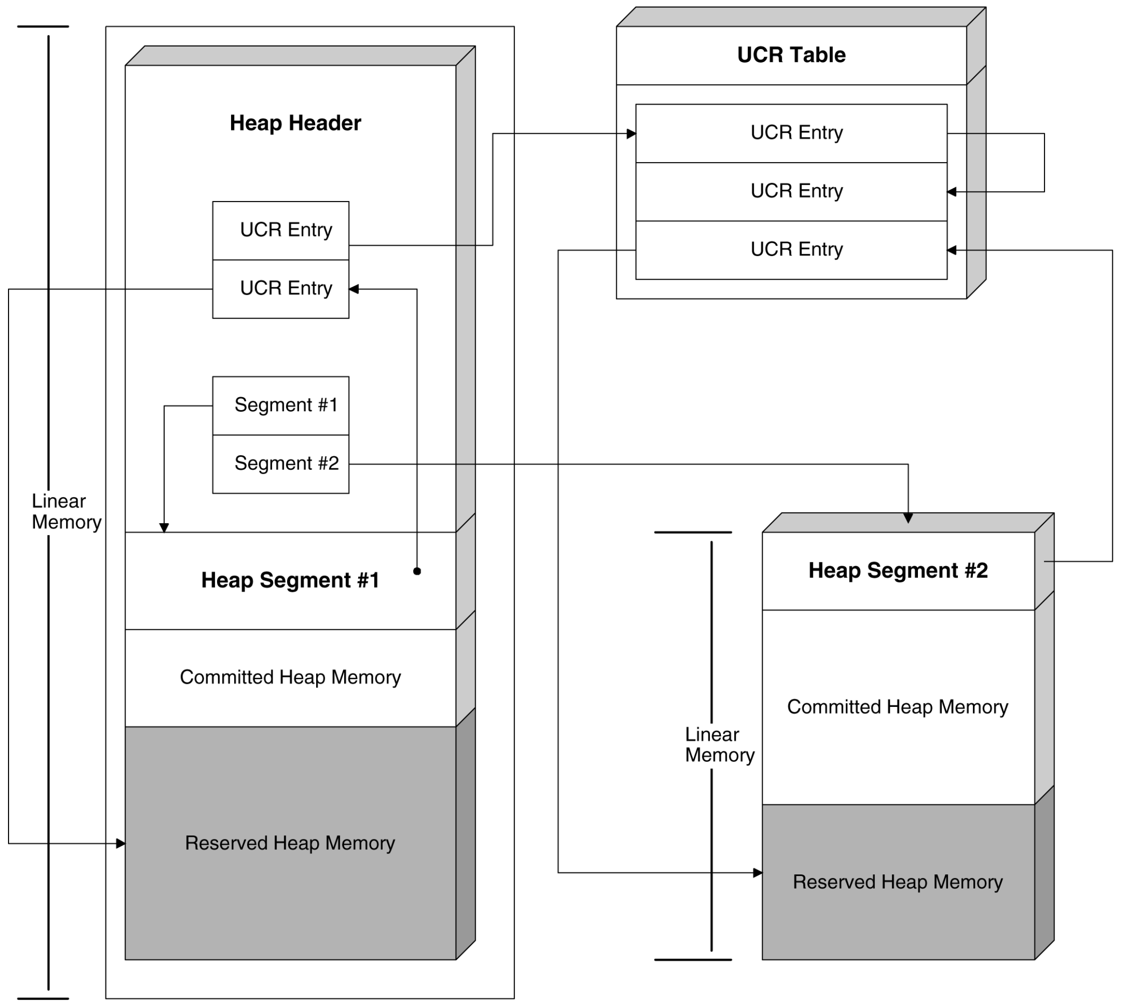


Figure 8-2. Typical Heap Construction

Heap Segment #1 tracks the virtual memory region initially allocated for the heap. The starting address for the heap segment equals to the base address of the heap and the end range points to the end of the allocated memory. A portion of the heap in the diagram is in a reserved state, that is, it has not been committed, so the heap segment uses an available UCR entry to track the area. When memory must be committed to satisfy an allocation request, all UCR entries maintained by a particular segment are examined to determine if the size of the uncommitted range is large enough to satisfy the allocation. To increase performance, the heap segment tracks the largest available UCR range and the total number of uncommitted pages within the virtual memory region of the heap segment.

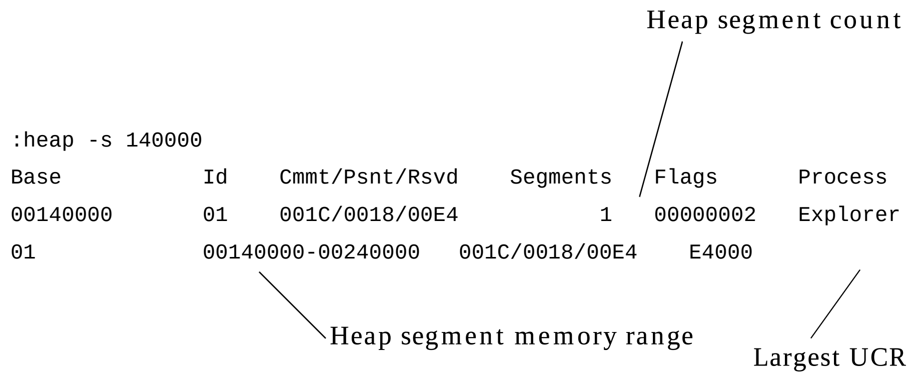
On the right side of the diagram, a second area of virtual memory was allocated and is managed by Heap Segment #2. Additional heap segments are created when an allocation request exceeds the size of the largest uncommitted range within the existing segment. This is only true if the size of the requested allocation is less than the heap's VMthreshold. When the requested allocation size exceeds the VMThreshold, the heap block is directly allocated through VirtualAlloc and a new heap segment is not created.

As mentioned previously, a small number of UCR entries are provided within the heap header. For illustration purposes, this diagram shows a UCR TABLE entry that was allocated specifically to increase the number of UCR entries that are available. The need to create an extra UCR table is generally rare, and is usually a sign that a large number of segments were created or that the heap segments are fragmented.

Fragmentation of virtual memory can occur when the Heap API begins decommitting memory during the coalescing of free blocks. Decommitting memory is the term used to describe reverting memory from a committed state to a reserved or uncommitted state. When a free block spans more than one physical page (4k), that page becomes a candidate for being decommitted. If certain decommit threshold values are satisfied, the Heap manager begins decommitting free pages. When those pages are not contiguous with an existing uncommitted range, a new UCR entry must be used to track the range.

The following examples use the Visual SoftICE HEAP command to examine the default heap for the Explorer process.

1 Use the -S option of the HEAP command to display segment information for the default heap:



```
:heap -s 140000
Base            Id    Cmmt/Psnt/Rsvd     Segments  Flags       Process
00140000        01    001C/0018/00E4            1  00000002    Explorer
01              00140000-00240000   001C/0018/00E4    E4000
```

2 Use the -X option of the HEAP command to display extended information about the default heap:

```
:heap -x 140000

        Extended Heap Summary for heap 00140000 in Explorer

  Heap Base:      140000      Heap Id:          1   Process:    Explorer
  Total Free:        6238     Alignment:        8   Log Mask:   10000
  Seg Reserve:    100000      Seg Commit:   2000
  Committed:         112k     Present:        96k   Reserved:   912k
  Flags: GROWABLE
  DeCommit:          1000     Total DeC:    10000   VM Alloc:   7F000
```

Default size of a heap segment

Default size for commits

VM threshold

3 Use the -B option of the HEAP command to display the base addresses of heap blocks within the default heap:

```
:heap -b 140000
       Base        Type        Size        Seg#        Flags
       00140000    HEAP        580         01
       00140580    SEGMENT     38          01
       001405B8    ALLOC       30          01
```

In the above output, you can see how the heap header is followed by Heap Segment #1 and that the first allocated block is just beyond the Heap Segment data structure.

Managing Heap Blocks

As discussed in the preceding section, the Heap API uses the Win32 Virtual Memory API routines to allocate large regions of the linear address space and uses heap segments to manage committed and uncommitted ranges. The actual sub-allocation engine that manages the allocation and deallocation of the memory blocks used by your application is built on top of this functionality. To track allocated and free blocks, the Heap API creates a header for each block.

The diagram on the next page illustrates how the heap manager tracks blocks of *contiguous* memory. The heap manager also tracks non-contiguous free blocks in doubly-linked lists, but the node pointers for the next and previous links are not stored in the block header. Instead, the heap manager uses the first two Dwords within the heap block memory area.

As shown in the diagram, each block stores its unit size as well as the unit size of the previous block. The unit size represents the number of heap units occupied by the heap block. The previous unit size is the number of heap units occupied by the previous heap block. Using these two values, the heap manager is able to walk contiguous heap blocks.

Heap units represent the base granularity of allocations made from a heap. The size of an allocation request is rounded upwards as necessary, so that it is an even multiple of this granularity. Rather than using a granularity of 1 byte, the heap manager uses a granularity of 8 bytes. This means that all allocations are an even multiple of 8 bytes, and that allocation sizes can be converted to units by round up and dividing by 8.

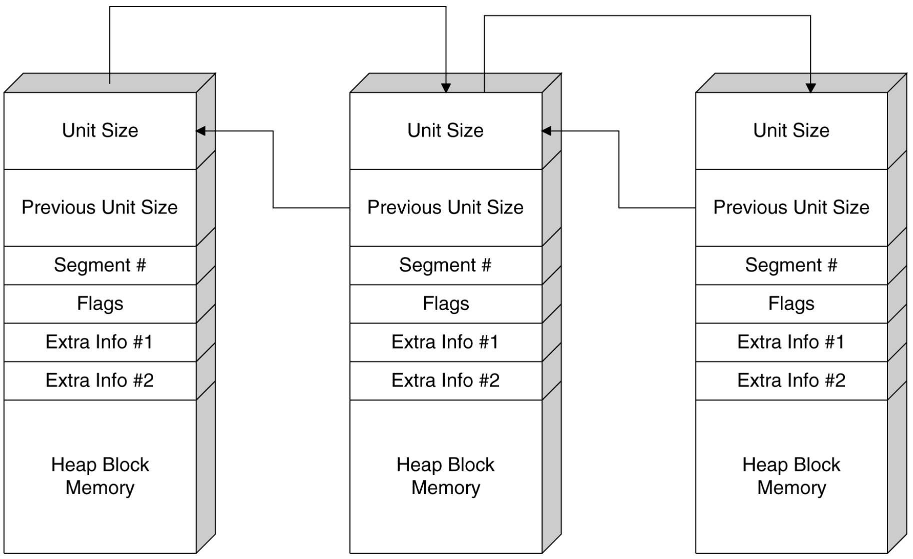


Figure 8-3. *Contiguous* Memory Tracking

For example, if a process requests an allocation of 32 bytes, the number of units is 32 / 8 = 4. If the allocation request was 34 bytes, the allocation size is rounded upward to an even multiple of 8. In this example, the 34 bytes requested would be rounded to an allocation of 40 bytes, or 5 units. The process requesting the allocation is unaware of any rounding to satisfy unit granularity and proceeds as if the allocation request of 34 bytes was actually 34 bytes.

By using a unit size of 8, the types of allocation made by most applications can be recorded using one word value with the restriction that the maximum size of a heap block, in units, is the largest unsigned short or `0xFFFF`. This makes the theoretical maximum size of a heap block in bytes, `0xFFFF * 8`, or 524,280 bytes. (This limitation is documented in the Win32 HeapAlloc API documentation.) Does that mean that a program cannot allocate a heap block greater than 512k? Well, yes and no. A heap block larger than 512k cannot be allocated, but there is nothing to prevent the Heap API from using VirtualAlloc to allocate a region of linear memory to satisfy the request. This is exactly what the heap manager does if the size of the requested allocation exceeds the heaps VMThreshold. The value of VMThreshold is stored in the heap header and by default is 520,192 bytes (or `0xFE000` units). When the heap manager allocates a large heap block using VirtualAlloc, the resulting structure is referred to as a Virtually Allocated Block (VAB).

The heap manager walks contiguous heap blocks by converting the current heap block's unit size into bytes and adding that to the heap block's base address. The address of the previous heap block is calculated in a similar manner, converting the unit size of the previous block to bytes and subtracting it from the heap block's base address. The heap manager walks contiguous heap blocks during coalescing free blocks, sub-allocating a smaller block from a larger free block, and when validating a heap or heap entry.

Unit sizes are important for free block list management as the array of 128 doubly-linked lists inside the heap header track free blocks by unit size. Free blocks that have a unit size in the range from 1 to 127 are stored in the free list at the corresponding array index. Thus, all free blocks of unit size 32 are stored in Heap->FreeLists[32]. Because it is not possible to have a heap block that is 0 units, the free list at array index zero stores all heap blocks that are larger than 127 units; these entries are sorted by size in ascending order. Because a majority of allocations made by a process are less than 128 units (1024 bytes or 1K), this is a fast way to find an exact or best fit block to satisfy an allocation. Blocks of 128 units or greater are allocated much less frequently, so the overhead of doing a linear search of one free list does not have a large impact on the overall performance of most applications.

The flags field within the heap block header denotes special attributes of the block. One bit is used to mark a block as allocated versus free. Another is used if it is a VAB. Another is used to mark the last block within a committed region. The last block within a committed region is referred to as a sentinel block, and indicates that no more contiguous blocks follow. Using this flag is much faster than determining if a heap block address is valid by walking the heap segment's UCR chain. Another flag is used to mark a block for free or busy-tail checking. When a process is debugged, the heap manager marks the block in certain ways. Thus, when an allocated block is released or a free block is reallocated, the heap manager can determine if the heap block was overwritten in any way.

The extra info fields of the heap block header have different usage depending on whether the block is allocated or free. In an allocated block, the first field records the number of extra bytes that were allocated to satisfy granularity or alignment requirements. The second field is a pseudo-tag. Heap tags and pseudo tags are beyond the scope of this discussion.

For a free block, the extra info fields hold byte and bit-mask values that access a free-list-in-use bit-field maintained within the heap header. This bit-field provides quicker lookups when a small block needs to be allocated. Each bit within the bit-field represents one of the 127 small block free lists, and if the corresponding bit is set, that free list contains one or more free entries. A zero bit means that a free entry of that size is not available and a larger block will need to be sub-allocated from. The first extra info field holds the byte index into the bit-field array. The second extra info field holds the inverted mask of the bit position within the bit-field. Note that this applies to Windows NT 3.51 only. Newer versions of Windows NT still use the free list bit-field, but do not store the byte index or bit-mask values. The heap block memory array is also different depending on the allocated state of the free block. For allocated blocks, this is the actual memory used by your application. For free blocks, the first two Dwords (1 unit) are used as next and previous pointers that link free blocks together in a doubly-linked list. If the process that allocated the heap block is being debugged, an allocated heap block also contains a busy-tail signature at the end of the block. Free blocks are marked with a special tag that can detect if a stray pointer writes into the heap memory area, or the process continues to use the block after it was deallocated.

The following diagram shows the basic architecture of an allocated heap block.

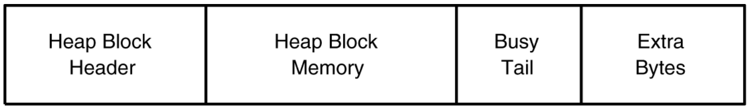


Figure 8-4. Basic Architecture of an Allocated Heap Block

The portion labeled *Extra Bytes* is memory that was needed to satisfy the heap unit size or heap alignment requirements. This memory area should not be used by the allocating process, but the heap manager does not directly protect this area from being overwritten. The busy-tail signature appears just beyond the end of the memory allocated for use by the process. If an application writes beyond the size of the area requested, this signature is destroyed and the heap manager signals the debugger with a debug message and an INT 3. It is possible for a process to write into the extra bytes area without disturbing the busy-tail signature. In this case, the overwrite is not caught. The Heap API provides an option for initializing heap memory to zero upon allocation. If this option is not specified when debugging, the heap manager fills the allocated memory block with a special signature. You can use this signature to determine if the memory block was properly initialized in your code.

The following diagram shows the basic architecture of a free heap block.

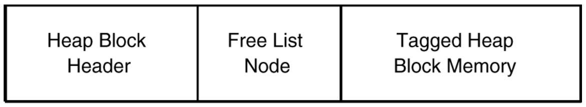


Figure 8-5. Basic Architecture of a Free Heap Block

When a block is deallocated and the process is being debugged, the heap manager writes a special signature into the heap memory area. When the block is allocated at some point in the future, the heap manager checks that the tag bytes are intact. If any of the bytes was changed, the heap manger outputs a debug message and executes an INT 3 instruction. This is a good thing if the debugger you are using traps INT 3, but most debuggers ignore this debug-break because it was not set by the debugger. As an aside, having the Free List Node pointers at the beginning of the memory block is somewhat flawed, because a program that continues to use a free block is more likely to overwrite data at the beginning of the block than data at the end. Because these pointers are crucial to navigating the heap, an invalid pointer eventually causes an exception. When this exception occurs, it can be quite difficult to track this overwrite back to the original free block.

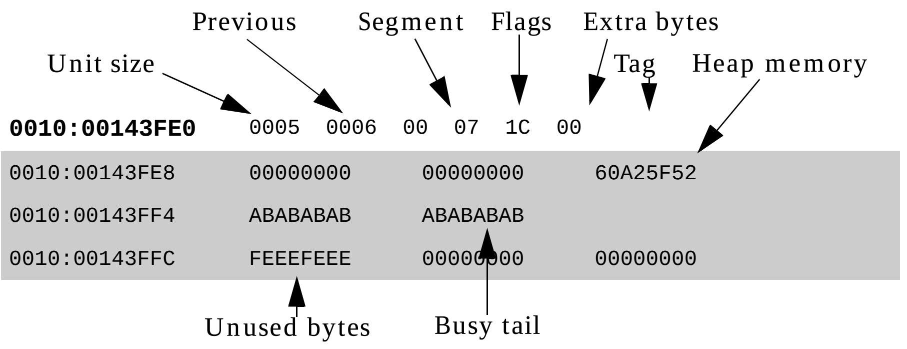
The following two examples show how to use the Visual SoftICE HEAP command to aid in monitoring and debugging Win32 heap issues.

The first example uses the HEAP command to walk all the entries for the heap based at `0x140000`. The -B option of the HEAP command causes the base address and size information to display as the heap manager would view the information. Without the -B option, the HEAP command shows base addresses and sizes as viewed by the application that allocated the memory. The output is abbreviated for clarity and the two heap blocks that appear in bold type are used to examine the heap block header in the second example.

```
:HEAP -b explorer.exe
Count:11
Base               Heap Base        Id   Committed     Present  Reserved  SegmentCount     Flags       Mapped
---------------------------------------------------------------------------------------------------------------
-------------------------------------
explorer.exe       00080000         1    9b            85       100       1                00000002    no
explorer.exe       00180000         2    6             4        10        1                00001002    no
explorer.exe       00190000         3    1             0        10        1                00008000    no
explorer.exe       00260000         4    d             b        10        1                00001002    no
explorer.exe       00460000         5    10            c        10        1                00001002    no
explorer.exe       01240000         6    3             1        10        1                00001002    no
explorer.exe       01250000         7    3             1        40        1                00001002    no
explorer.exe       012a0000         8    4             3        10        1                00001002    no
explorer.exe       00a10000         9    3             1        100       1                00000002    no
explorer.exe       00b10000         a    c             8        10        1                00001002    no
explorer.exe       00b60000         b    3             2        10        1                00001002    no
```

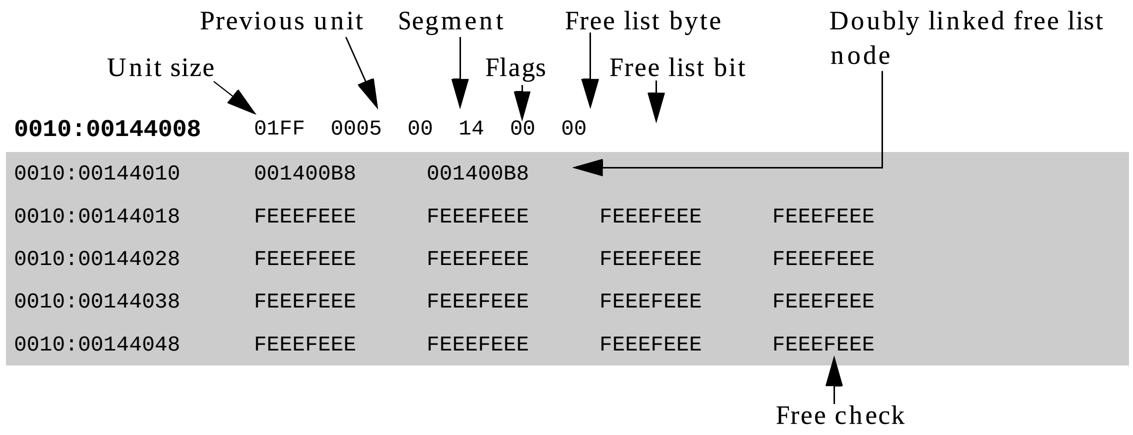
To examine the contents of an allocated heap block and a free block, the second example dumps memory at the base address of the heap block at `0x143FE0`. Enough memory is dumped to show the subsequent block, which is a free block at address `0x144008`.

The heap block header fields from the memory dump at address `0x143FE0` are identified with call-outs. This heap block is five units in size (40 bytes) and `0x1C` bytes of that size is overhead for the heap block header (1 unit), busy-tail (1 unit), unit alignment (1 Dword), and an extra unit left over from a previous allocation.



```
0010:00143FE0    0005  0006  00  07  1C  00
0010:00143FE8    00000000    00000000    60A25F52
0010:00143FF4    ABABABAB    ABABABAB
0010:00143FFC    FEEEFEEE    00000000    00000000
```

The heap block immediately following this is a free block that begins at address 0x144008. This block is 0x1FF units and the size of the previous block is five units. For free blocks 1KB or larger (80+ units), the Free List byte position and bit-mask values are not used and are zero. The flag for this heap block indicates that it is a sentinel (bit 4, or 0x10). Immediately following the heap header is the location where the heap manager has placed a doubly-linked list node for tracking free blocks. The pointer values for the next and previous fields of the node are both 0x1400B8. After the free list node, the heap manager tagged all the blocks memory with a special signature that is validated the next time the block is allocated, coalesced with another block, or a heap validation is performed.



```
0010:00144008    01FF  0005  00  14  00  00
0010:00144010    001400B8    001400B8
0010:00144018    FEEEFEEE    FEEEFEEE    FEEEFEEE    FEEEFEEE
0010:00144028    FEEEFEEE    FEEEFEEE    FEEEFEEE    FEEEFEEE
0010:00144038    FEEEFEEE    FEEEFEEE    FEEEFEEE    FEEEFEEE
0010:00144048    FEEEFEEE    FEEEFEEE    FEEEFEEE    FEEEFEEE
```

Appendix A

Troubleshooting Visual SoftICE

Troubleshooting

If you encounter the following problems, try the corresponding solutions. If you encounter further difficulties, contact the Technical Support Center.

Problem	Solution
In Visual SoftICE, I want to connect to the target machine but I do not see it in the list of network target machines.	There may be several reasons for this. First, you may not have installed the target portion of Visual SoftICE on that target machine. If you are confident that you have properly installed the Visual SoftICE target components on the target machine, then go to the configuration tool and make certain that you have properly configured a connection. If you have a properly configured connection on that target machine, and it is active, then you will need to do more extensive troubleshooting. Contact the Technical Support Center with all applicable details.
I want to set a break point on something in my source, but when I open the source code and attempt to set the break point, it fails to set.	This is because the source is not mapped where Visual SoftICE can determine the location of the image file to break on. To fix this, you can either pre-load symbols into Visual SoftICE using the ADDSYM command, or you can set the breakpoint on the loading of that module (using the BPLOAD command) and have them found automatically. Once the module is in memory, you can set breakpoints on other locations.
I am noticing that the values in the Command page do not appear to be correct. For example, I entered the CPU command and Visual SoftICE returned the value for my processor speed as 585.	Most values returned in the Command page are displayed in hexadecimal format by default. Try converting the value you receive to decimal, and reinterpret the results. The 585 processor speed in your example would actually be 1400 decimal. Use the SET RADIX command to configure the displayed and input radix to whatever format you are expecting to see.
After installing Visual SoftICE on my target machine, it will no longer boot. It reaches a specific part of the boot sequence and then goes no further.	Try connecting with the master and check the status of the target to see if it has reached a breakpoint. A major cause of this behavior is the “stop on boot” setting. Once the target is up and running, use DSConfig to check the “stop on boot” setting for the target.

Problem	Solution
I want to connect to the target machine, but do not want to wait for it to finish booting. How do I accomplish this?	Use the WCONNECT command.
In the Command page, I receive far too much information regarding the steps and other trivialities I do not care about. How do I stop or decrease these messages?	Use the SET command by itself to review your settings. You probably have a more detailed message level reporting configured. Use the SET MSGLEVEL command to decrease or turn off message reporting in the Command page.
I'm trying to stack-walk on an IA64/AMD64 machine, and I keep getting a message that **Unwind Information is Unavailable**.	This is a very common issue when trying to stack-walk on an IA64/AMD64 machine. This results from one of three problems: Either no pdb file is loaded for the specific image, the unwind data section of the image is paged out, or you set your EXEPATH and need to reload the table. First, make certain that the correct pdb file is loaded for the image. If the error persists, try copying the image file into a folder, and then set your executable path (EXEPATH) to include that folder. If you set your EXEPATH and still cannot get unwind information, reload the table.
After installing the Visual SoftICE network debugging, I lost my Ethernet connection.	This may not be a problem. It may be just a misunderstanding of the Visual SoftICE network debugging. Visual SoftICE uses a dedicated network that becomes the connection for debugging only. So if your connection to the debugger is working, but connection to the internet is not, then this is normal. We suggest that the target machine have two network cards: one PCI network card for normal network operation, and another PCI network card for the Visual SoftICE debugger. The only other alternative is to use the VNIC driver in addition to the installed connection.

Appendix B

Kernel Debugger Extensions

Debugger Extension Overview and VSI Support

There are two different types of KD extension DLLs:

- “Old style” WinDbg Extensions: Extensions that call routines in `wdbgexts.h`
- “New style” DbgEng extensions: Extensions that call routines in `dbgeng.h` and `wdbgexts.h`

Visual SoftICE supports both KD extension DLLS.

Controlling Debugger Extension DLLs

There are several commands for controlling the debugger extension DLLs:

- **!Module.load** (Load Extension DLL) loads a new DLL.
- **!Module.unload** (Unload Extension DLL) unloads a DLL.
- **kdlist** (List Debugger Extensions) displays all loaded debugger extension modules in their default search order.

You can also load an extension DLL by using the full **!module.extension** syntax the first time you issue a command from that module, but you have to manually unload a KD extension DLL that has been loaded. Even disconnecting from a target will not unload KD extension DLLs automatically. Refer to the “Using Debugger Extension Commands” section later in this appendix for details.

The extension DLLs that you are using must match the operating system of the target computer. The extension DLLs that ship with WinDbg are each placed in a different subdirectory of the installation directory according to the OS version and target mode (Release version or Debug version). For example, nt4fre, w2kchk, and winxp. You must make sure you use the right version, and you must set the right KD extension search path (using either the path definitions in the **Settings** dialog, or via the **SET KDEXTPATH** command at the command line).

You should be as specific as possible when setting the KD extension search path, such as using `C:\Program Files\Debugging Tools for Windows\winxp` instead of `C:\Program Files\Debugging Tools for Windows`, otherwise Visual SoftICE will look for the first name-matched KD Extension DLL and load it.

Note: If you write your own debugger extensions, you should place them in a new directory and add that directory to the debugger extension path.

Using Debugger Extension Commands

The use of debugger extension commands is very similar to the use of other commands. A debugger extension command is an entry point in a KD Extensions DLL called by the debugger.

You invoke debugger extensions via the following syntax:

![module.]extension [arguments]

Required: The module name should not be followed with the .dll file name extension.

If the module has not already been loaded, it will be loaded into the debugger using a call to LoadLibrary(module). After the debugger has loaded the extension library, it calls the GetProcAddress function to locate the extension name in the extension module. The extension name is case-sensitive and must be entered exactly as it appears in the extension module's `.def` file. If the extension address is found, the extension is called.

If the module name is not specified, the debugger will search the loaded extension modules for this export. The latest loaded DLL is searched first.

When an extension module is unloaded, it is removed from the search order. When an extension module is loaded, it is added to the beginning of the search order.

You can use the **kdlist** (List Debugger Extensions) command to display a list of all loaded extension modules in their current search order.

If you attempt to execute an extension command that is not in any of the loaded extension modules, you will get an SI_E_NAME_NOT_FOUND error message.

Glossary

Datum	Artificial data elements contrived to make the user's life easier. For example, BPID, and _TID (current thread ID). Datums are time and target context sensitive.
Interrupt Descriptor Table (IDT)	Table pointed to by the IDTR register, which defines the interrupt/ exception handlers. Use the IDT command to display the table.
MAP file	Human-readable file containing debug data, including global symbols and usually line number information.
MMX	Multimedia extensions to the Intel Pentium and Pentium-Pro processors.
Object	Represents any hardware or software resource that needs to be shared as an object. Also, the term section is sometimes called an object. Refer to Section.
One-Shot Breakpoint	Breakpoint that only goes off once. It is cleared after the first time it goes off or the next time Visual SoftICE pops up for any reason.
Ordinal Form	When a symbol table is not relocated, it is said to be in its ordinal form; in this state, the selectors are section numbers or segment numbers (for 16 bit).
Relocate	Adjust program addresses to account for the program's actual load address.
Section	In the PE file format, a chunk of code or data sharing various attributes. Each section has a name and an ordinal number.
Sticky Breakpoint	Breakpoint that remains until you remove it. It remains even through unloading and reloading of your program.
SYM File	File containing debug data, including global symbols and usually line number information. The SYM file is usually derived from a MAP file.
Symbol Table	Visual SoftICE-internal representation of the debugging information, for example, symbols and line numbers associated with a specific module.

Virtual Breakpoint	Breakpoint that can be set on a symbol or a source line that is not yet loaded in memory.

Index