Using SoftICE®

Version 3.2

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. 11577a
December 10, 2004

Table of Contents

Preface

Chapter 1

Choosing Your SoftICE Version

Chapter 2

Welcome to SoftICE

Chapter 3

SoftICE Tutorial

Chapter 4

Loading Code into SoftICE

Chapter 5

Navigating Through SoftICE

Chapter 6

Using SoftICE

Chapter 7

Using Breakpoints

Chapter 8

Using Expressions

Chapter 9

Loading Symbols for System Components

Chapter 10

Remote Debugging with SoftICE

Chapter 11

Customizing SoftICE

Chapter 12

Exploring Windows NT

Appendix A

Appendix B

Appendix C

Appendix D

Appendix E

SoftICE and VMware

Appendix F

SoftICE API Specification

Preface

Purpose of This Manual

Note: Unless stated otherwise, this document will use “Windows® 9x” to refer to the Windows 95, Windows 98, and Windows Millennium (Windows ME) operating systems (treated as a group); “the Windows NT® family” will refer to the Windows NT, Windows 2000, Windows XP, Windows Server 2003, and Longhorn operating systems. (Also, unless stated otherwise, characteristics of Windows NT described in this manual also apply to Windows 2000, Windows XP, Windows Server 2003, and Longhorn.)

SoftICE® is an advanced, all-purpose debugger that can debug virtually any type of code including applications, device drivers, EXEs, DLLs, OCXs, and dynamic and static VxDs. Since many programmers prefer to learn through hands on experience, this manual includes a tutorial that leads you through the basics of debugging code.

This manual is intended for programmers who want to use SoftICE to debug code for Windows 9x and the WINDOWS NT family platforms.

Users of previous versions of SoftICE should read the Release Notes/ Readme documentation to see how this version of SoftICE differs from previous versions.

This manual assumes that you are familiar with the Microsoft® Windows interface and with software debugging concepts.

What This Manual Covers

This manual contains the following chapters and appendixes:

The *Using SoftICE* manual is organized as follows:

- Chapter 1, "Choosing Your SoftICE Version"

  Explains the differences between SoftICE and its companion two-machine debugger, Visual SoftICE.
- Chapter 2, "Welcome to SoftICE"

  Briefly describes SoftICE components and features. Chapter 2 also explains how to contact the Compuware Technical Support Center.
- Chapter 3, "SoftICE Tutorial"

  Provides a hands-on tutorial that demonstrates the basics for debugging code. Topics include tracing code, viewing the contents of locals and structures, setting a variety of breakpoints, and viewing the contents of symbol tables.
- Chapter 4, "Loading Code into SoftICE"

  Explains how to use SoftICE Symbol Loader to load various types of code into SoftICE.
- Chapter 5, "Navigating Through SoftICE"

  Describes how to use the interface that SoftICE provides for code debugging.
- Chapter 6, "Using SoftICE"

  Provides information about trapping faults, address contexts, using INT 0x41.DOT commands, and transitions from Ring-3 to Ring-0.
- Chapter 7, "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 8, "Using Expressions"

  Explains how to form expressions to evaluate breakpoints.

- Chapter 9, “Loading Symbols for System Components”

  Explains how to load export symbols for DLLs and EXEs and how to use symbol files with SoftICE.

- Chapter 10, “Remote Debugging with SoftICE ”

  Explains how to establish a remote connection to operate SoftICE from a remote PC.

- Chapter 11, “Customizing SoftICE”

  Explains how to use the SoftICE configuration settings to customize your SoftICE environment, pre-load symbols and exports, configure remote debugging, modify keyboard mappings, create macro-definitions, and set troubleshooting options.

- Chapter 12, “Exploring Windows NT”

  Provides a quick overview of the Windows NT operating system.

- Appendix A, “Error Messages”

  Explains the SoftICE error messages.

- Appendix B, “Supported Display Adapters”

  Lists the display adapters that SoftICE supports.

- Appendix C, “Troubleshooting SoftICE”

  Explains how to solve problems you might encounter.

- Appendix D, “Kernel Debugger Extensions”

  Explains how to prepare a Kernel Debugger Extension for use with SoftICE.

- Appendix E, “SoftICE and VMWare”

  Explains the restrictions, limitations, and differences between SoftICE running on a “virtual machine” and SoftICE running on a real machine.

- Appendix F, “SoftICE API Specification”

  Explains the process for defining a public interface allowing you to add conditional code to your driver that will execute if SoftICE is present.

- Glossary

- Index

Conventions Used In This Manual

This book uses the following conventions to present information:

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

Accessibility

Prompted by federal legislation introduced in 1998 and Section 508 of the U.S. Rehabilitation Act enacted in 2001, Compuware launched an accessibility initiative to make its products accessible to all users, including people with disabilities. This initiative addresses the special needs of users with sight, hearing, cognitive, or mobility impairments.

Section 508 requires that all electronic and information technology developed, procured, maintained, or used by the U.S. Federal government be accessible to individuals with disabilities. To that end, the World Wide Web Consortium (W3C) Web Accessibility Initiative (WAI) has created a workable standard for online content.

Compuware supports this initiative by committing to make its applications and online help documentation comply with these standards. For more information, refer to:

- W3C Web Accessibility Initiative (WAI) at www.W3.org/WAI
- Section 508 Standards at www.section508.gov
- Microsoft Accessibility Technology for Everyone at www.microsoft.com/enable/

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	Perform the tutorial in Chapter 3.
Experience with other debuggers	Read Chapter 4, “Loading Code into SoftICE.” Then read Chapter 5, “Navigating Through SoftICE.”
Experience using a previous release of SoftICE	Read Chapter 1, “Product Overview,” to learn about this version of SoftICE.

Other Useful Documentation

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

- SoftICE Command Reference
- Describes all the 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.
- SoftICE on-line Help
- SoftICE provides context-sensitive help for Symbol Loader and a help line for SoftICE commands in the debugger.
- On-line documentation

- Both the *Using SoftICE* manual and the *SoftICE Command Reference* are available on line. To access the on-line version of these books, start Acrobat Reader and open the *Using SoftICE* or the *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 as well as the Readme files for this product. You can contact Technical Support by:

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

  `mailto:nashua.support@compuware.com`

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

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

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

  `1-603-578-8401`

- **Telephone:** Telephone support is available as a paid* Priority Support Service from 8:30am to 5:30pm EST, Monday through Friday. Have product version and serial number ready.
  - In the U.S. and Canada, call: 1-888-686-3427
  - International customers, call: +1-603-578-8100

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

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

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

Chapter 1

Choosing Your SoftICE Version

SoftICE or Visual SoftICE?

DriverStudio and SoftICE Driver Suite 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 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
Intel x86 and compatibles	Windows 2000, Windows XP, Advanced Server, Windows Server 2003	Yes	Yes
Intel Itanium1 and Itanium2 (IA64)	Windows XP 64bit Ed., Windows Server 2003 IA64 64-bit Ed.	No	Yes

Table 1-1. Supported Platforms (Continued)

Processor	Operating System	SoftICE	Visual SoftICE
AMD Opteron, Athlon64 (AMD64 / K8)	Windows XP 64bit Ed., Windows Server 2003 Extended 64-bit Ed.	No	Yes
Intel EM64T (Preliminary)	Windows XP 64bit Ed., Windows Server 2003 Extended 64-bit 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 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 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.

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.

But Which One Should I Use?

If your project falls into the wide overlap between SoftICE and Visual SoftICE, 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, a Direct3D or streaming app or driver, or input device driver, try Visual SoftICE.
- If you want direct access to BoundsChecker events from within the debugger, use SoftICE.
- 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.

Chapter 2

Welcome to SoftICE

Product Overview

SoftICE is available for Windows 9x and the Windows NT family. SoftICE consists of the SoftICE kernel-mode debugger and the Symbol Loader utility. The SoftICE debugger (SoftICE) is an advanced, all-purpose debugger that can debug virtually any type of code including interrupt routines, processor level changes, and I/O drivers. The Symbol Loader utility (Symbol Loader) loads the debug information for your module into SoftICE, maintains the SoftICE initialization settings, and lets you save the contents of the SoftICE history buffer to a file. The following sections briefly describe SoftICE and Symbol Loader.

Benefits of SoftICE

SoftICE combines the power of a hardware debugger with the ease of use 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. SoftICE displays your source code as you debug, and lets you access your local and global data through their symbolic names.

Some of the major benefits SoftICE provides include the following:

- Source level debugging of 32-bit (Win32) applications, the Windows NT family device drivers (both kernel and user mode), Windows 9x drivers, VxDs, 16-bit Windows programs, and DOS programs.

- Debugging virtually any code, including interrupt routines and the Windows 9x and the Windows NT family kernels.
- Setting real-time breakpoints on memory reads/writes, port reads/writes, and interrupts.
- Setting breakpoints on Windows messages.
- Setting breakpoints on module loads and unloads.
- Setting conditional breakpoints and breakpoint actions.
- Displaying elapsed time to the breakpoint trigger using the Pentium clock counter.
- Kernel-level debugging on one machine.
- Displaying internal Windows 9x and Windows NT family 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.
- Popping up the SoftICE screen automatically when an unhandled exception occurs.
- Using SoftICE to connect by modem, network, serial, or Internet to a remote user. This enables you to diagnose a remote user’s problem, such as a system crash.
- Supporting the MMX, SSE, and SSE2 instruction set extensions.
- Creating user-defined macros.

How SoftICE is Implemented

SoftICE for Windows 9x and SoftICE for the Windows NT family are implemented in slightly different ways. SoftICE for Windows 9x comprises two VxDs, while SoftICE for the Windows NT family comprises several NT kernel device drivers. This is shown in Table 2-1 on page 9.

Table 2-1. SoftICE Implementation Methods

Windows 9x (VxD)	Windows ME	Windows NT family (NT Family Kernel Device Driver)	Description
WINICE.EXE	WINICE.EXE	NTICE.SYS	Provides the debugger.
SIWVID.386	SIWVID.386	SIWVID.SYS	Provides video support for your PC.
	WINICE.VXD		
	DEBUGGER.EXE		

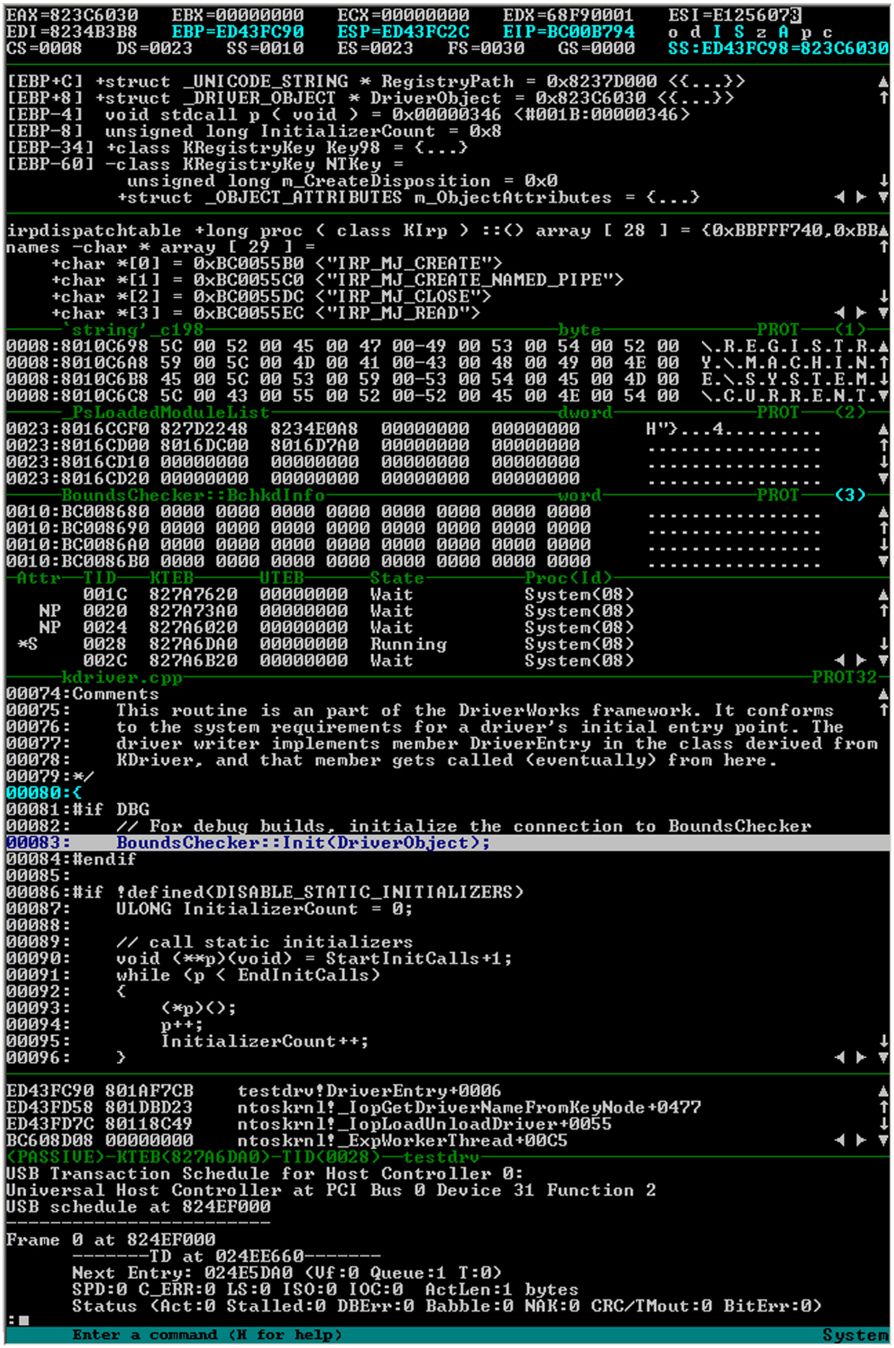
Note: SoftICE for the Windows NT family must be loaded by the operating system because it is implemented as a device driver. If you need to debug a boot mode driver, you will need to take an additional step of setting up Siwsym and manually changing the load order of SoftICE. You will not be able to debug the NTOSKRNL initialization code,and any Windows NT family loader or NTDETECT code. For additional information on Siwsym, please read the included siwsym.txt file.

SoftICE User Interface

SoftICE provides a consistent interface for debugging applications across all platforms. The SoftICE user interface is designed to be functional without compromising system robustness. For SoftICE to pop up at any time without disturbing the system state, it must access the hardware directly to perform its I/O.

SoftICE uses a full-screen character-oriented display window, as shown in Figure 2-1 on page 10.

Refer to *Chapter 4: Navigating Through SoftICE* on page 47 for more information about using the SoftICE screen.



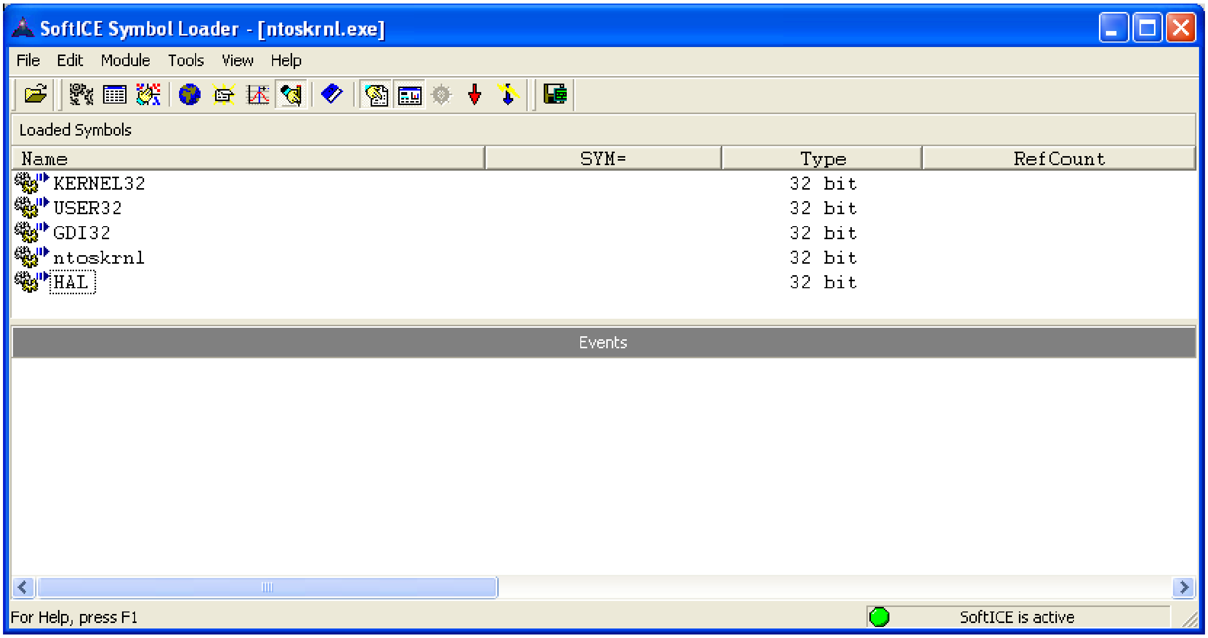
```
EAX=823C6030   EBX=00000000   ECX=00000000   EDX=68F90001   ESI=E1256073
EDI=8234B3B8   EBP=ED43FC90   ESP=ED43FC2C   EIP=BC00B794   o d I S z A p c
CS=0008   DS=0023   SS=0010   ES=0023   FS=0030   GS=0000   SS:ED43FC98=823C6030

[EBP+C] +struct _UNICODE_STRING * RegistryPath = 0x8237D000 <{...}>
[EBP+8] +struct _DRIVER_OBJECT * DriverObject = 0x823C6030 <{...}>
[EBP-4]  void stdcall p ( void ) = 0x00000346 <#001B:00000346>
[EBP-8]  unsigned long InitializerCount = 0x8
[EBP-34] +class KRegistryKey Key98 = {...}
[EBP-60] -class KRegistryKey NTKey =
          unsigned long m_CreateDisposition = 0x0
         +struct _OBJECT_ATTRIBUTES m_ObjectAttributes = {...}

irpdispatchtable +long proc ( class KIrp ) ::() array [ 28 ] = {0xBBFFF740,0xBB
names -char * array [ 29 ] =
    +char *[0] = 0xBC0055B0 <"IRP_MJ_CREATE">
    +char *[1] = 0xBC0055C0 <"IRP_MJ_CREATE_NAMED_PIPE">
    +char *[2] = 0xBC0055DC <"IRP_MJ_CLOSE">
    +char *[3] = 0xBC0055EC <"IRP_MJ_READ">
-----`string'_c198-------------------------------byte--------------PROT---(1)--
0008:8010C698 5C 00 52 00 45 00 47 00-49 00 53 00 54 00 52 00  \.R.E.G.I.S.T.R.
0008:8010C6A8 59 00 5C 00 4D 00 41 00-43 00 48 00 49 00 4E 00  Y.\.M.A.C.H.I.N.
0008:8010C6B8 45 00 5C 00 53 00 59 00-53 00 54 00 45 00 4D 00  E.\.S.Y.S.T.E.M.
0008:8010C6C8 5C 00 43 00 55 00 52 00-52 00 45 00 4E 00 54 00  \.C.U.R.R.E.N.T.
-----_PsLoadedModuleList-------------------------dword-------------PROT---(2)--
0023:8016CCF0 827D2248  8234E0A8  00000000  00000000     H"}...4.........
0023:8016CD00 8016DC00  8016D7A0  00000000  00000000     ................
0023:8016CD10 00000000  00000000  00000000  00000000     ................
0023:8016CD20 00000000  00000000  00000000  00000000     ................
-----BoundsChecker::BchkdInfo--------------------word--------------PROT---(3)--
0010:BC008680 0000 0000 0000 0000 0000 0000 0000 0000    ................
0010:BC008690 0000 0000 0000 0000 0000 0000 0000 0000    ................
0010:BC0086A0 0000 0000 0000 0000 0000 0000 0000 0000    ................
0010:BC0086B0 0000 0000 0000 0000 0000 0000 0000 0000    ................
-Attr--TID---KTEB------UTEB------State------------Proc(Id)---------------------
       001C  827A7620  00000000  Wait             System(08)
   NP  0020  827A73A0  00000000  Wait             System(08)
   NP  0024  827A6020  00000000  Wait             System(08)
  *S   0028  827A6DA0  00000000  Running          System(08)
       002C  827A6B20  00000000  Wait             System(08)
-----kdriver.cpp------------------------------------------------------PROT32--
00074:Comments
00075:    This routine is an part of the DriverWorks framework. It conforms
00076:    to the system requirements for a driver's initial entry point. The
00077:    driver writer implements member DriverEntry in the class derived from
00078:    KDriver, and that member gets called (eventually) from here.
00079:*/
00080:{
00081:#if DBG
00082:    // For debug builds, initialize the connection to BoundsChecker
00083:    BoundsChecker::Init(DriverObject);
00084:#endif
00085:
00086:#if !defined(DISABLE_STATIC_INITIALIZERS)
00087:    ULONG InitializerCount = 0;
00088:
00089:    // call static initializers
00090:    void (**p)(void) = StartInitCalls+1;
00091:    while (p < EndInitCalls)
00092:    {
00093:        (*p)();
00094:        p++;
00095:        InitializerCount++;
00096:    }

ED43FC90 801AF7CB    testdrv!DriverEntry+0006
ED43FD58 801DBD23    ntoskrnl!_IopGetDriverNameFromKeyNode+0477
ED43FD7C 80118C49    ntoskrnl!_IopLoadUnloadDriver+0055
BC608D08 00000000    ntoskrnl!_ExpWorkerThread+00C5
(PASSIVE)-KTEB(827A6DA0)-TID(0028)--testdrv-----------------------------------
USB Transaction Schedule for Host Controller 0:
Universal Host Controller at PCI Bus 0 Device 31 Function 2
USB schedule at 824EF000
------------------------
Frame 0 at 824EF000
      -------TD at 024EE660-------
      Next Entry: 024E5DA0 (Vf:0 Queue:1 T:0)
      SPD:0 C_ERR:0 LS:0 ISO:0 IOC:0  ActLen:1 bytes
      Status (Act:0 Stalled:0 DBErr:0 Babble:0 NAK:0 CRC/TMout:0 BitErr:0)
:■
      Enter a command (H for help)                                      System
```

Figure 2-1. SoftICE Display Window

About Symbol Loader

Symbol Loader (Figure 2-2) is a graphical utility that extracts debug symbol information from your device drivers, EXEs, DLLs, OCXs, and dynamic and static VxDs and loads it into SoftICE. This utility lets you do the following:

- Customize the type and amount of information it loads to suit your debugging requirements.
- Provides a Workspace and Session environment.
- Load and unload entire groups of symbol files, translations, and links.
- Automatically start your application and set a breakpoint at its entry point.
- Save your debugging session to a file.



Name	SYM=	Type	RefCount
KERNEL32		32 bit	
USER32		32 bit	
GDI32		32 bit	
ntoskrnl		32 bit	
HAL		32 bit	

For Help, press F1

SoftICE is active

Figure 2-2. SoftICE Symbol Loader

Symbol Loader also supports a command line interface that lets you use many of its features from a DOS prompt. Thus, you can automate many of the most common tasks it performs. Additionally, SoftICE provides a separate command-line utility (NMSYM) that lets you automate the creation of symbol information from a batch file.

Chapter 3

SoftICE Tutorial

Introduction

This tutorial gives you hands-on experience debugging Windows software, and teaches you the fundamental steps for debugging applications and drivers. During this debugging session, you will learn how to:

- Load SoftICE
- Build the sample code

- Load the source and symbol files
- Trace and step through source code and assembly language
- View local data and structures
- Change memory
- Set point-and-shoot breakpoints
- Use SoftICE informational commands to explore the state of the application/machine/OS
- Work with symbols and symbol tables
- Modify a breakpoint to use a conditional expression

Each section in the tutorial builds upon the previous sections, so you should perform them in order.

This tutorial uses the NMDEMO application. Once you install DriverStudio, NMDEMO is located in:

```
\program files\compuware\driverstudio\softice\examples
```

You can substitute a different sample application or an application of your own design. The debugging principles and features of SoftICE used in this tutorial apply to most applications.

Loading SoftICE

If you are running SoftICE in Boot, System, or Automatic mode, it automatically loads when you start or reboot your PC. If you are running SoftICE in Manual or Disabled mode, it does not load automatically. To change the mode in which you have SoftICE configured to load, access the **Startup** screen in the Settings utility, and select the desired mode from the SoftICE drop-down list (See Figure 4-1 on page 46). If SoftICE does not load automatically, do one of the following:

- Select **START SOFTICE** from the Compuware DriverStudio/Debug group
- At the Windows command prompt, enter the command:
  `NET START NTICE`

Note: Once you load SoftICE, you cannot deactivate it until you reboot your PC.

Controlling the SoftICE Screen

Before you start debugging, take a few minutes to explore and customize the SoftICE screen (see Figure 3-2 on page 17).

You can pop SoftICE up at any time by pressing **<Ctrl>-D**. The SoftICE screen consists of a single panel, split horizontally into a number of windows which display different information. There are several types of windows you can open and close within the SoftICE screen, but when you first pop into SoftICE the screen is split into a Registers window at the top, a Code window below it, and a Command window at the bottom along with the Help line. The Register window will always display the contents of the machine's registers when SoftICE was popped up, and the Code window will display the next instruction to be executed. The Command window contains a command prompt at the bottom, and the SoftICE output. It also captures any DebugPrint output from drivers and applications in the system.

If you have installed SoftICE with the default configuration, you will probably find that the screen is a little small. There are a few ways to change this. First, you can use the LINES and WIDTH commands to change the number of lines and the number of characters in the SoftICE screen. You can use the SET MAXIMIZE command to automatically make the SoftICE screen as large as possible. If you are running in UVD mode, the SET FONT command can be used to change the font, which can make the screen more readable. Table 3-1 describes the function of each command.

Table 3-1 SoftICE Window Resizing Commands

Command	Action
LINES *N*	Sets the maximum number of lines for the window to *N*.
WIDTH *N*	Sets the maximum character width of the window to *N*.
SET MAXIMIZE on/off	Maximizes the window length and width when enabled.
SET FONT *N*	Changes the font used on the SoftICE screen.

Note: You can also make these changes permanent through the **Settings** application by adding semi-colon (;) separated entries in the **Init String** field.

While in UVD mode, you can reposition and resize the entire SoftICE window. To reposition the window, hold down **<Ctrl>-<Alt>** while using the numeric keypad to move the window. Figure 3-1 on page 16 shows the numeric keypad functions.

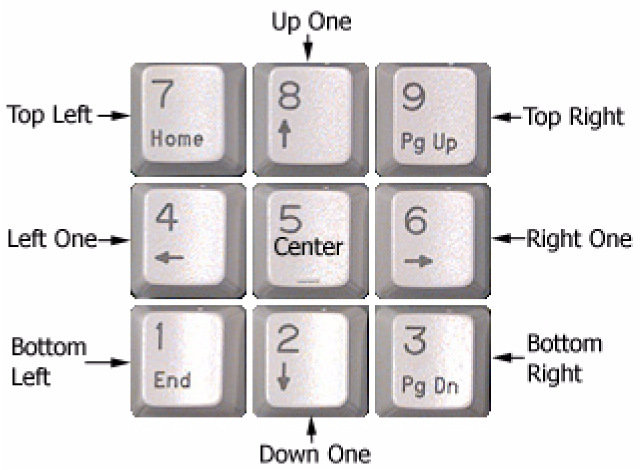


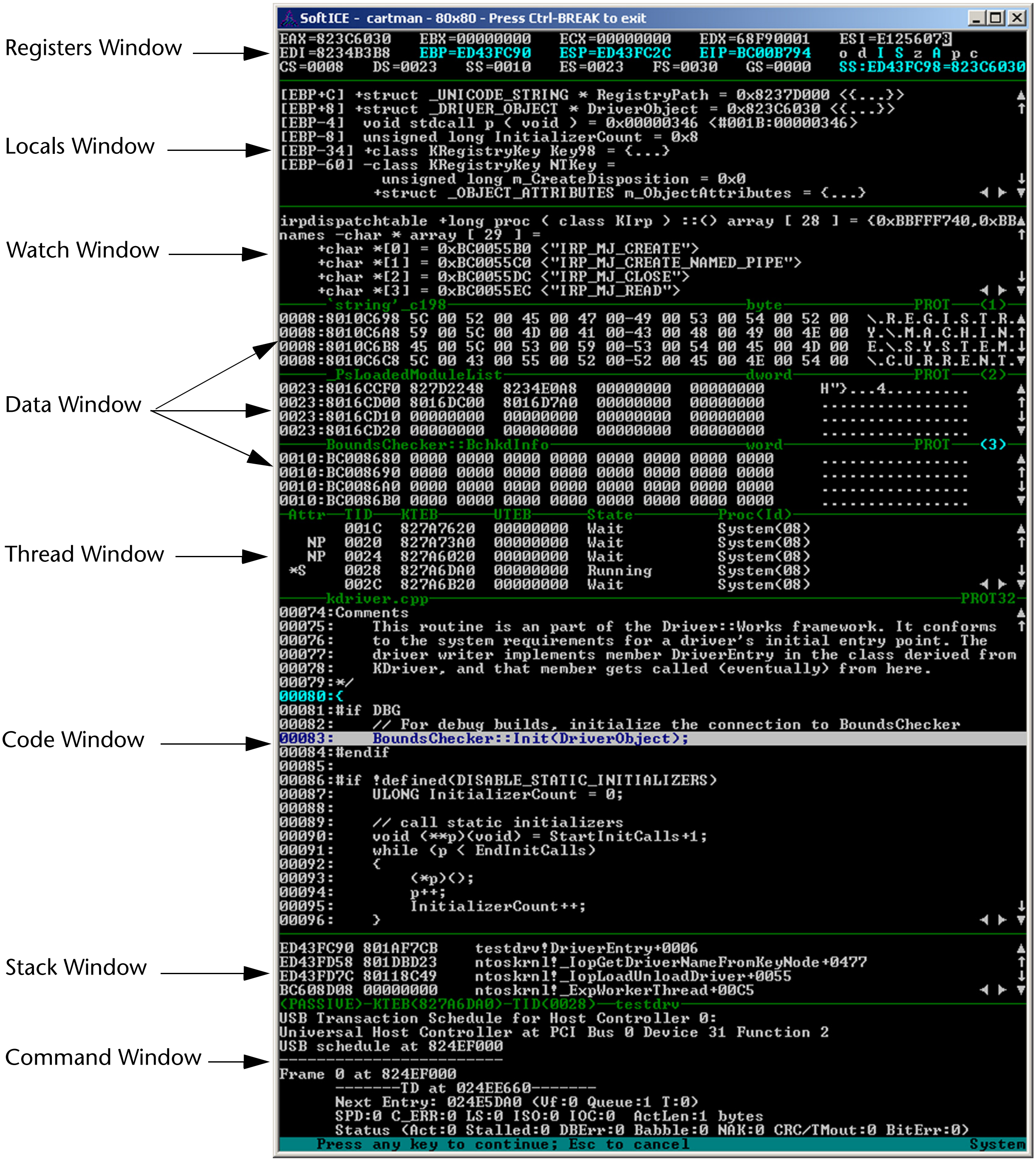
Figure 3-1. Numeric Keypad Functions

The size of the SoftICE screen in UVD mode will be limited by the amount of graphics memory allocated by SoftICE. If the screen cannot be made large enough, you can increase the amount of graphics memory with the **Settings** utility (**Start>Programs>Compuware DriverStudio>Settings**).

Table 3-2 on page 16 describes the SoftICE windows that are available, and the commands and keystrokes you can use to display and/or switch to each.

Table 3-2. Available SoftICE Windows

Name	Command	Hot Key	Description
Registers	WR	**<Alt>-R**	Shows the state and values of the hardware register set.
Locals	WL	**<Alt>-L**	Displays the local information (variables, etc.) for the current stack frame.
Watch	WW	**<Alt>-W**	Allows you to monitor the values of expressions that you have set using the WATCH command.
Data	WD.*n*	**<Alt>-D**	Allows you to view and edit the contents of memory. Up to four Data windows can be opened (0-3) specified by *n*. **<Alt>-D** functions on the current Data window only.
Thread	WT	**<Alt>-T**	Displays information on the threads within a given process.
Code	WC	**<Alt>-C**	Displays source code, disassembled instructions, or both (mixed).
Stack	WS	**<Alt>-S**	Displays the call stacks.
Command	N/A	N/A	Enter user commands here, and receive command execution information.
Help Line	N/A	N/A	Provides information about SoftICE commands and shows the active address context



```
EAX=823C6030   EBX=00000000   ECX=00000000   EDX=68F90001   ESI=E1256078
EDI=8234B3B8   EBP=ED43FC90   ESP=ED43FC2C   EIP=BC00B794   o d I S z A p c
CS=0008   DS=0023   SS=0010   ES=0023   FS=0030   GS=0000   SS:ED43FC98=823C6030

[EBP+C] +struct _UNICODE_STRING * RegistryPath = 0x8237D000 <{...}>
[EBP+8] +struct _DRIVER_OBJECT * DriverObject = 0x823C6030 <{...}>
[EBP-4]  void stdcall p ( void ) = 0x00000346 <#001B:00000346>
[EBP-8]  unsigned long InitializerCount = 0x8
[EBP-34] +class KRegistryKey Key98 = {...}
[EBP-60] -class KRegistryKey NTKey =
            unsigned long m_CreateDisposition = 0x0
           +struct _OBJECT_ATTRIBUTES m_ObjectAttributes = {...}

irpdispatchtable +long proc ( class KIrp ) ::() array [ 28 ] = {0xBBFFF740,0xBB
names -char * array [ 29 ] =
    +char *[0] = 0xBC0055B0 <"IRP_MJ_CREATE">
    +char *[1] = 0xBC0055C0 <"IRP_MJ_CREATE_NAMED_PIPE">
    +char *[2] = 0xBC0055DC <"IRP_MJ_CLOSE">
    +char *[3] = 0xBC0055EC <"IRP_MJ_READ">
------`string'_c198----------------------------------byte------------PROT---(1)--
0008:8010C698 5C 00 52 00 45 00 47 00-49 00 53 00 54 00 52 00  \.R.E.G.I.S.T.R.
0008:8010C6A8 59 00 5C 00 4D 00 41 00-43 00 48 00 49 00 4E 00  Y.\.M.A.C.H.I.N.
0008:8010C6B8 45 00 5C 00 53 00 59 00-53 00 54 00 45 00 4D 00  E.\.S.Y.S.T.E.M.
0008:8010C6C8 5C 00 43 00 55 00 52 00-52 00 45 00 4E 00 54 00  \.C.U.R.R.E.N.T.
------_PsLoadedModuleList----------------------------dword-----------PROT---(2)--
0023:8016CCF0 827D2248  8234E0A8  00000000  00000000      H"}...4.........
0023:8016CD00 8016DC00  8016D7A0  00000000  00000000      ................
0023:8016CD10 00000000  00000000  00000000  00000000      ................
0023:8016CD20 00000000  00000000  00000000  00000000      ................
------BoundsChecker::BchkdInfo-----------------------word------------PROT---(3)--
0010:BC008680 0000 0000 0000 0000 0000 0000 0000 0000     ................
0010:BC008690 0000 0000 0000 0000 0000 0000 0000 0000     ................
0010:BC0086A0 0000 0000 0000 0000 0000 0000 0000 0000     ................
0010:BC0086B0 0000 0000 0000 0000 0000 0000 0000 0000     ................
-Attr--TID---KTEB-----UTEB------State-------Proc(Id)----------------------------
       001C  827A7620  00000000  Wait        System(08)
   NP  0020  827A73A0  00000000  Wait        System(08)
   NP  0024  827A6020  00000000  Wait        System(08)
  *S   0028  827A6DA0  00000000  Running     System(08)
       002C  827A6B20  00000000  Wait        System(08)
------kdriver.cpp----------------------------------------------------PROT32-
00074:Comments
00075:    This routine is an part of the Driver::Works framework. It conforms
00076:    to the system requirements for a driver's initial entry point. The
00077:    driver writer implements member DriverEntry in the class derived from
00078:    KDriver, and that member gets called (eventually) from here.
00079:*/
00080:{
00081:#if DBG
00082:    // For debug builds, initialize the connection to BoundsChecker
00083:    BoundsChecker::Init(DriverObject);
00084:#endif
00085:
00086:#if !defined(DISABLE_STATIC_INITIALIZERS)
00087:    ULONG InitializerCount = 0;
00088:
00089:    // call static initializers
00090:    void (**p)(void) = StartInitCalls+1;
00091:    while (p < EndInitCalls)
00092:    {
00093:        (*p)();
00094:        p++;
00095:        InitializerCount++;
00096:    }

ED43FC90 801AF7CB    testdrv!DriverEntry+0006
ED43FD58 801DBD23    ntoskrnl!_IopGetDriverNameFromKeyNode+0477
ED43FD7C 80118C49    ntoskrnl!_IopLoadUnloadDriver+0055
BC608D08 00000000    ntoskrnl!_ExpWorkerThread+00C5
(PASSIVE)-KTEB(827A6DA0)-TID(0028)--testdrv-------------------------------------
USB Transaction Schedule for Host Controller 0:
Universal Host Controller at PCI Bus 0 Device 31 Function 2
USB schedule at 824EF000
------------------------
Frame 0 at 824EF000
      -------TD at 024EE660-------
      Next Entry: 024E5DA0 (Vf:0 Queue:1 T:0)
      SPD:0 C_ERR:0 LS:0 ISO:0 IOC:0  ActLen:1 bytes
      Status (Act:0 Stalled:0 DBErr:0 Babble:0 NAK:0 CRC/TMout:0 BitErr:0)
    Press any key to continue; Esc to cancel                          System
```

Figure 3-2 SoftICE Window

Many SoftICE windows can be scrolled if you have a “wheel” mouse. Otherwise, you can click on the scroll arrows. SoftICE also provides key sequences that let you scroll specific windows. Press **<Ctrl>-D** to pop-up SoftICE and press **<Alt>-C** to switch to the Code window (if not using a mouse) and try these methods for scrolling:

Table 3-3. Scrolling Methods

Scroll Code Window	Key Sequence	Mouse Action
Scroll to the previous page.	PageUp	Click the innermost up scroll arrow
Scroll to the next page.	PageDown	Click the innermost down scroll arrow
Scroll to the previous line.	UpArrow	Click the outermost up scroll arrow
Scroll to the next line.	DownArrow	Click the outermost down scroll arrow
Scroll left one character.	Ctrl-LeftArrow	Click the left scroll arrow
Scroll right one character.	Ctrl-RightArrow	Click the right scroll arrow

To disassemble the instructions for the current instruction pointer, enter the U command with EIP as a parameter.

```
:U EIP
```

You can also use the . (dot) command to accomplish the same thing:

```
:.
```

Moving and Resizing SoftICE Windows

You can resize the windows within SoftICE , using either a mouse or the command line, to best fit your specific debugging scenario.

Using a Mouse

To resize a window using a mouse, click on the bottom of the window or on the title bar, and drag the window border to the desired position.

Using the Command Line

To set a window to an exact number of lines, use the W*X* command followed by the number of lines you want, where *X* is the letter identifying the window (see the commands in Table 3-2 on page 16).

You can also resize windows by a number of lines relative to the current size by issuing the W*X* command followed by +/- and the number of lines. This will increase or decrease the window size by the number of lines specified. For example:

```
:WL +10
```

This command would increase the Locals window by ten lines.

When you have finished adjusting the screen size, exit SoftICE by typing X and pressing **<Enter>** or pressing **<Ctrl>-D**.

Overview of the Sample Software

The sample software consists of several binaries encompassing most of the debugging situations that you routinely encounter. Table 3-4 provides descriptions of the files comprising the sample application set.

Table 3-4. Sample Files

Filename	Description
NMDemo1a	Main application serving as a front end to the sample application set
DemoDLL.dll	Supporting DLL file for NMDemo1a
DSDemo1d	Driver component
GDIDemo	Secondary sample application

The sample applications are presented as a collection of short contrived tests to illustrate specific features of SoftICE, as well as certain common programming errors.

Caution: **The sample applications consist of code that is deliberately buggy. Most code is written to demonstrate a single topic. Many programming rules have been broken and there are a large number of bugs that can crash your machine. This is by design. When a Blue Screen crash is expected, you will be warned ahead of time and given the option to abort the impending crash.**

Building the Sample Code

Note: We have provided several mechanisms for building the samples. For the purposes of this tutorial, we suggest that you begin by using the pre-built binaries shipped with the samples. This will get you comfortable with debugging without the added complexity of compiler settings. If you want to experiment with making changes to the sample programs, you can return to this section for instructions on building them.

The first step in preparing to debug a Windows application is to build it with debug information. SoftICE uses NMS files to obtain symbols and other debug information. NMS files are a superset of the DBG and PDB files used in Windows. In addition to debug information, NMS files are optimized for quick lookups, and can contain embedded source files. NMS files are generated and loaded into SoftICE using Loader32, a GUI-based tool, or NMSYM, a command-line tool. SoftICE can also be set up to load a set of NMS files whenever it is started.

You can build the sample applications using:

- Microsoft Visual Studio 6.0 (and later)
- Microsoft Visual Studio.NET 2003
- A Windows XP DDK Free or Debug Build Environment

Note: We have provided pre-built binaries and symbol files for those users who may not have a compiler installed. These files are located in the Output directory.

Note: All build environments will generate .nms symbol files and copy them to the Output directory along with their associated binaries. This is done via post-build steps in the IDE, and with sources and makefiles from the build command line.

Table 3-5. Building the Sample Code

Building with...	Follow these steps...
Visual Studio 6	1 Open `Numega DS Demo 1.dsw` in the **`NmDemo1`** root directory. 2 Choose the appropriate project and build environment. 3 Load the symbols into SoftICE (optional). 4 Execute the program from the **Output** directory.

Table 3-5. Building the Sample Code (Continued)

Building with...	Follow these steps...
Visual Studio.NET 2003	1 Open the `NuMega DS Demo1.SLN` solution in the **NmDemo1** root directory. 2 Choose the appropriate project and build environment. 3 Load the symbols into SoftICE (optional). 4 Execute the program from the **Output** directory.
DDK Build Utility	1 Open up the approriate DDK Build Environment. 2 From the root **Examples** directory issue the `build` command. Note: You could also change into each of the subprojects and issue the build command for each.

Launching the Application

To debug an application with SoftICE, you will need to load the application's symbol information first, and then launch the application from Windows. For this first demonstration, you will load the NMS files for `nmdemo1a`, set a breakpoint in SoftICE, and then launch the application from Windows. Later we will show you how to use the SoftICE Symbol Loader to do all this in one step.

Complete the following steps to load the symbols into SoftICE .

1 Open the **Output** folder in the Examples directory where SoftICE was loaded on your system.

2 Right-click on each of the four NMS files, and select **Load into SoftICE** from the pop-up menu.

Now you must set a breakpoint so SoftICE will pop-up when the application is started.

1 Press **<Ctrl>-D** to pop-up the SoftICE window.

2 Use the **<Ctrl>** and **<Alt>** keys in conjunction with the keypad to position the SoftICE window where you want it on your screen.

3 Use the **SET MAXIMIZE** command to fully expand it.

```
:SET MAXIMIZE on
```

4 Issue the **TABLE** command.
SoftICE displays the symbol tables you loaded. When you load symbols for an application, SoftICE creates a symbol table containing all the symbols defined for that module. The current table is highlighted. To switch to another table, and make that the current table, issue the **TABLE** command with the new table-name as a parameter.

Note: SoftICE will do best-case matching for the table or file name. You need only enter enough of the name to uniquely identify it.

Note: SoftICE allows you to auto-complete table and file names by pressing <**Tab**> after typing a partial name.

5 Issue the **FILE *** command.
SoftICE displays the source files associated with the current table. Switch to a different table and issue the **FILE *** command again. The source files will have changed. Table 3-6 displays the source files associated with each table for the sample aplications.

Table 3-6. Sample Application Tables and Associated Source Files

Current Table	Source Files
DemoDLL	demodll.cpp
GDIdemo	xform.c, wininfo.c, poly.c, maze.c, init.c, gdidemo.c, draw.c, dialog.c, bounce.c
NMdemo1a	service.cpp, nmdemo1app.cpp
DSdemo1d	tt.c, tc.c, si.c, nmdemo1drv.c, bc.c

Note: The Command window varies in size depending upon the number of lines used by open windows, so you might not see all these file names. To display the remaining file names, press any key (Refer to *Chapter 5:* on page 75 for information about resizing windows).

6 Issue the **TABLE** command again and specify `NMdemo1a`.

```
:TABLE nmdemo1a
```

Tip: *Use the AutoCompletion feature by pressing <**Tab**> after entering a partial symbol, table, or file name throughout this section.*

7 Issue the **FILE** command and specify `nmdemo1app`.

```
:FILE nmdemo1app
```

8 Jump to the `WinMain` function using the U command.

```
:U winmain
```

9 Set a breakpoint on WINMAIN by issuing the **BPX** command.

```
:BPX winmain
```

Note: You can also set the breakpoint by pressing the **F9** key or double-clicking on a location within the **Code** window; SoftICE will place the breakpoint at your current location.

10 Set a breakpoint on openConnectionToTestDriver by issuing the **BPX** command.

```
:BPX openConnectionToTestDriver
```

This breakpoint will be used later, in an example of setting conditional breakpoints.

11 Enter the **X** (exit) command, or press **F5** to pop-down SoftICE.

12 Start the NMdemo1a application by double-clicking on the executable in the **Output** folder under the **Examples** directory. SoftICE pops up when the breakpoint you set on WINMAIN is hit.

Tracing and Stepping through the Source Code

The following steps show you how to use SoftICE to trace through source code:

1 Enter the **T** (trace) command or press the **F8** key to trace one instruction.

```
:T
```

Note: The F8 key is the default key for the T (trace) command.

Tip: To change the default behavior of function keys, refer to the FKEY command in the SoftICE Command Reference.

Execution proceeds to the next source line and highlights it. At this point, the following source line should be highlighted:

```
gApphInstance = hInstance;
```

The Code window is currently displaying source code. However, it can also display disassembled code or mixed (both source and disassembled) code.

2 To view mixed code, use the **SRC** command (or press the **F3** key).

```
:SRC
```

Now each source line is followed by its assembler instructions.

3 Press **F3** once to see disassembled code, then again to return to source code.

4 Enter the **T** command (**F8**) to trace one instruction.

```
:T
```

Execution proceeds to the next source line and highlights it. At this point, the following source line should be highlighted:

```
if (!IsCorrectOSVersion())
```

As demonstrated in these steps, the **T** command executes one source statement or assembly language instruction. You can also use the **P** command (or press the **F10** key) to execute one program step.

Note: Stepping differs from tracing in one crucial way — if you are stepping and the statement or instruction is a function call, control is not returned until the function call is complete. Use **P** (**F10**) to step over a function call, or **T** (**F8**) to step into it.

Tip: The T command does not trace into a function call if the source code is not available. A good example of this is Win32 API calls. To trace into a function call when source code is not available, use the ***SRC*** *command (****F3****) to switch into mixed or assembly mode and then continue tracing into the call statement.*

Viewing and Editing Local Data

The Locals window displays the current stack frame. The following steps illustrate how to use the Locals window:

1 Enter the **WL** command to open the Locals window.

```
:WL
```

In this case, the Locals window contains the local data for the `WINMAIN` function.

2 Enter the **T** command (**F8**) to enter the `IsCorrectOSVersion()` function.

```
:T
```

The Locals window is now empty because local data is not yet allocated for the function.

Note: Local data will not appear until you step into a function. If you examine the assembly code you will note that the stack frame is not yet set up when a breakpoint is set on the start of the function.

The `IsCorrectOSVersion()` function is implemented in the source file `nmdemo1app.cpp`. SoftICE displays the current source file in the upper left corner of the Code window.

3 Enter the **P** command (**F10**) to step into the `IsCorrectOSVersion()` function.

```
:P
```

The Locals window now contains the locals that are available in the `IsCorrectOSVersion()` function.

The structure tag `osvi` is marked with a plus sign (+). This indicates that you can expand the structure to view its contents.

Note: You can also expand character strings and arrays.

4 If you have a mouse, double-click the structure `osvi` to expand it. To collapse the structure `osvi`, double-click it again.

To use the keyboard to expand the structure: press <**Alt**>-**L** to move the cursor to the Locals window, use the UpArrow or DownArrow to move the highlight bar to the structure, and press <**Enter**>.
Press <**Enter**> a second time to collapse it.

Editing Local Data

You can edit the local variables that appear in the Local window using inline editing mode.

Note: You can also use inline editing mode to edit variables in the Watch window.

1 Use the **SS** command to search for `EX_3`.

```
:SS EX_3
```

The **SS** command is used to search for a string in the current source, and `EX_3` is a comment we placed in the code to allow you to find the dummy variables we provided for you to practice editing.

2 Press <**Alt**>-**C** to enter the Code window.
The cursor is placed at the top of the Code window, on the line containing the EX_3 comment.

3 Set a breakpoint on the line by issuing the **BPX** command.

```
:BPX
```

SoftICE knows that the current line is a comment, and it sets the actual breakpoint on the next executable line of code. The line is highlighted to indicate the breakpoint is set.

4 Press <**Alt**>-**C** to exit the Code window.

5 Press **F5** to pop-down SoftICE and execute to the breakpoint.
The locals window now displays several dummy variables we supplied in the code for you to edit.

6 Press <**Alt**>-**L** to enter the Locals window, and use the down-arrow to move focus to the `bIsWin9x` local.

7 Press **<Alt>-E** to enter inline editing mode.
The cursor enters the Locals window and flashes on the value of `bIsWin9x`, waiting for you to input a new value.

8 Enter `1` and press **<Enter>**.
SoftICE accepts the new value for the `bIsWin9x` local, and exits inline editing mode.

9 Clear the breakpoint we hit for this example (identified as number 02) using the **BC** command.

```
:BC 02
```

Setting Point-and-Shoot Breakpoints

This section shows you how to set two useful types of point-and-shoot breakpoints: one-shot and sticky breakpoints.

Setting a One-Shot Breakpoint

The following steps demonstrate how to set a one-shot breakpoint. A one-shot breakpoint clears after the breakpoint is triggered.

1 Shift focus to the Code window by pressing **<Alt>-C**, or by clicking in the window.

Note: When you want to shift focus back to the Command window, press **<Alt>-C** again, or simply start typing a command.

2 Either use the **SS** command to search for the comment `EX_5`, or use the **U** command to place the cursor on line 145.
This location is a call to the `TempDriverHandle` function. If you use the **U** command, specify the source line 145 as follows:

```
:U .145
```

SoftICE places source line 145 at the top of the Code window.

3 Use the **HERE** command (**F7**) to execute to line 145.

The **HERE** command executes from the current instruction to the instruction that contains the cursor. The **HERE** command sets a one-shot breakpoint on the specified address or source line and continues execution until that breakpoint triggers. When the breakpoint is triggered, SoftICE automatically clears the breakpoint so that it does not trigger again.

The following current source line should be highlighted:

```
if(TempDriverHandle)
```

Note: You can do the same thing by using the G (go) command and specifying the line number or address to which to execute:

```
:G .145
```

Setting a Sticky Breakpoint

The following steps demonstrate another type of point-and-shoot breakpoint: the sticky breakpoint, which does not clear until you explicitly clear it.

1 Find the next call to `OpenConnectionToTestDriver` by entering the **SS** command to search for `EX_6`.

```
:SS EX_6
```

Tip: The F9 key is the default key for the BPX command.

2 Enter the **BPX** command (**F9**) to set an execution breakpoint. Note that the line is highlighted when you set the breakpoint.

3 Press the **F9** key to clear the breakpoint.

Note: If you are using a mouse, you can double-click on a line in the Code window to set or clear a breakpoint.

4 Set the breakpoint again, then use the **G** or **X** command (**F5**) to execute the instructions until the breakpoint triggers:

```
:G
```

At this point SoftICE pops up on an unexpected location. This is due to the breakpoint you set on the `openConnectionToTestDriver` function earlier in the tutorial. This demonstrates a situation where setting a conditional breakpoint would be useful, as you could have avoided hitting this breakpoint while trying to run to the breakpoint we just set. Conditional breakpoints are explained in “Setting a Conditional Breakpoint” on page 29.

5 Use the **G** or **X** command (**F5**) once again.

```
:G
```

The application starts up, and presents you with the main screen. Since the breakpoint we want to hit is set on a test driver function, you will need to start the driver in order for us to encounter it.

6 Click the **Start Driver** button.
When the code line containing the breakpoint is hit, SoftICE pops up. Unlike the **HERE** command, which sets a one-shot breakpoint, the **BPX** command sets a sticky breakpoint. A sticky breakpoint remains until you clear it.

7 To view information about breakpoints that are currently set, use the **BL** command:

```
:BL
00)  BPX WinMain
01)  BPX openConnectionToTestDriver
02)  BPX MainDlgFunc+01BC
```

Note: The address you see might be different.

From the output of the **BL** command, breakpoints are set on `WinMain` and `openConnectionToTestDriver` from the first two breakpoints we set during this tutorial. The third breakpoint is set on `MainDlgFunc+01BC`. This equates to the call to `bDriverRunning=openConnectionToTestDriver(TRUE)` in the current source file.

8 Use the **U** command to jump to the location of the first breakpoint in the list (0).

```
:U BP0
```

BP0 evaluates to the address of breakpoint 0, and the **U** command quickly jumps to that location.

9 Use the dot (.) command to return to the EIP.

```
:.
```

10 You can use the SoftICE expression evaluator to translate a line number into an address. To find the address for line 183, use the **?** command.

```
:? .183
<ulong> = 0x40133C, 4199228, "@!!<"
```

Note: The actual address displayed may be different on your system.

The expression evaluator displays results in three formats: hexadecimal, decimal, and ASCII characters. In this case, the hex value is the one we are interested in to see the address.

11 The `OpenConnectionToTestDriver` function has a relatively straightforward implementation, so it is unnecessary to trace every single source line. Use the **P** command with the **RET** parameter (**F12**) to return to the point where this function was called.

```
:P RET
```

Focus should be on the following source line.

```
UpdateButtonsOnMainDlg(hwndDlg);
```

12 Enter the **BC** command to clear the very first and very last breakpoints, preserving the breakpoint set on openConnectionToTestDriver.

```
:BC 0
:BC 2
```

13 Enter the **BL** command to confirm that the breakpoints have been cleared.

```
:BL
01)  BPX openConnectionToTestDriver
```

Setting a Conditional Breakpoint

One of the symbols defined for the NMDemo1a application is the openConnectionToTestDriver function. The purpose of this routine is to initialize the connection to the test driver in the application.

To learn about conditional breakpoints, you will edit the current BPX breakpoint (set on the openConnectionToTestDriver function) to use a conditional expression, thus making it a conditional breakpoint.

Editing a Breakpoint

If you examine the openConnectionToTestDriver function, you can see that the function accepts one parameter of type BOOL (gWarnings) and returns an INT type. At this point, you want to modify the existing breakpoint, adding a condition to isolate a specific instance.

1 Because you already have a breakpoint set on the openConnectionToTestDriver function, use the **BPE** command to modify (or edit) the existing breakpoint.

```
:BPE 1
```

When you use the **BPE** command to modify an existing breakpoint, SoftICE places the definition of that breakpoint onto the command line so that it can be easily edited. The output of the **BPE** command appears.

```
:BPX openConnectionToTestDriver
```

The cursor appears at the end of the command line, and SoftICE is ready for you to type in the conditional expression.

2 Add an *IF* condition to the breakpoint to cause the breakpoint to fire only when the local bGlobalWarnings evaluates to TRUE. The conditional expression appears in bold type.

```
:BPX openConnectionToTestDriver IF (gWarnings == TRUE)
```

3 Press <**Enter**> when you are done editing the breakpoint.
SoftICE accepts the changes and exits breakpoint editing mode.

4 Verify that the breakpoint and conditional expression are correctly set by using the **BL** command.

```
:BL
01)  BPX openConnectionToTestDriver IF (gWarnings == TRUE)
```

5 Exit SoftICE using the **G** or **X** command (**F5**).
The NMDemo1a application starts up. Click the **Start Driver** button at the bottom of the application's main screen. SoftICE pops-up when the conditional expression is TRUE.

Note: If the **Stop Driver** button is active, it means you have previously started the driver at some point. Click the **Stop Driver** button and continue with the tutorial.

Setting a Read-Write Memory Breakpoint

Setting a memory breakpoint allows you to pop-up SoftICE when memory is accessed to read, write, read/write, or execute. This allows you to use SoftICE to locate and resolve any number of memory related bugs. For the purposes of this example, we are going to use the sample application's built-in ring-3 memory overrun. The sample application sets up a simple scenario where you are accessing memory to change the name of an employee from Chris to Christopher.

1 Exit SoftICE using the **G** or **X** command (**F5**).
You are returned to the main screen of the sample application.

2 Click **Ring-3 Main**.
The sample application opens the set of Ring-3 tests.

3 Click **Memory Overrun**.
The memory overrun test is selected.

4 Click **OK** to begin running the test.
The Old Salary Report for Chris shows that as of last week he made $20,000.00 a year.

5 Click **OK**.
The New Salary Report for Chris shows that after the name change, his salary suddenly also changed and he now makes $7,497,064.00 a year.

Obviously, since we only planned on changing the employee name from Chris to Christopher, there is a memory overwrite problem somewhere. SoftICE can help you locate the source of the memory overwrite error by popping-up when a piece of memory is accessed.

1. Press **<Ctrl>-D** to pop-up SoftICE.
2. Use the **TABLE** command to make sure `nmdemo1a` is the active symbol table.

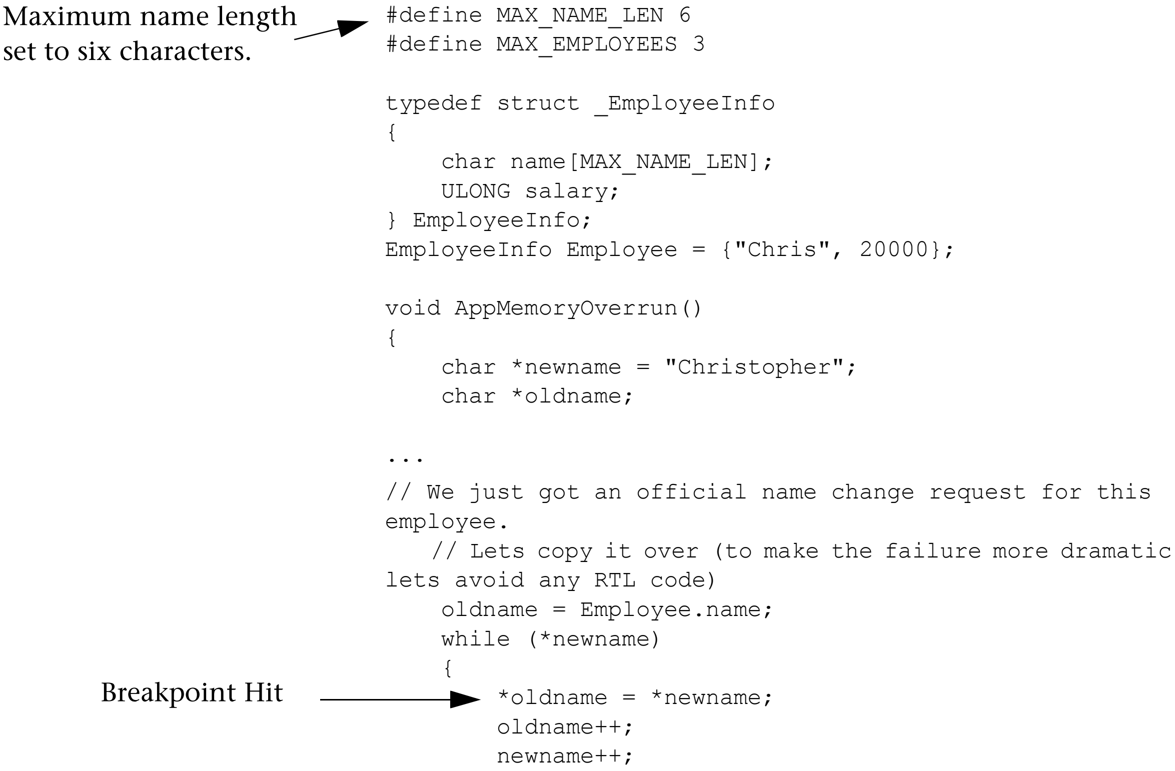
   ```
   :TABLE nmdemo1a
   ```

3. Use the **BPMD** command to set a dword memory breakpoint on the `employee.salary` location, to be triggered by a write to that location.

   ```
   :BPMD employee.salary w
   ```

4. Exit SoftICE using the **G** or **X** command (**F5**).
   You are returned to the sample application.
5. Click **Memory Overrun**.
   The memory overrun test is selected.
6. Click **OK** to begin running the test.
   The Old Salary Report for Chris shows that as of last week he made $20,000.00 a year.
7. Click **OK**.
   SoftICE pops-up when memory is written to `employee.salary`.

By examining the section of code where SoftICE popped-up (Figure 3-3 on page 32), we can see that the source of the memory overwrite error is that we have allocated a maximum of six characters for `employee.name`. When we try and change Chris to Christopher, the excess characters overrun the allocated space for `employee.name` and write into `employee.salary`.



Maximum name length set to six characters.

Breakpoint Hit

```
#define MAX_NAME_LEN 6
#define MAX_EMPLOYEES 3

typedef struct _EmployeeInfo
{
    char name[MAX_NAME_LEN];
    ULONG salary;
} EmployeeInfo;
EmployeeInfo Employee = {"Chris", 20000};

void AppMemoryOverrun()
{
    char *newname = "Christopher";
    char *oldname;

...
// We just got an official name change request for this
employee.
   // Lets copy it over (to make the failure more dramatic
lets avoid any RTL code)
    oldname = Employee.name;
    while (*newname)
    {
        *oldname = *newname;
        oldname++;
        newname++;
```

Figure 3-3 Code to Change Employee Name

Using SoftICE Informational Commands

SoftICE provides a wide variety of informational commands that detail the state of an application or the system. This section teaches you about two of them: **H** (help) and **CLASS**. The **H** command provides general help on all the SoftICE commands, or detailed help on a specific command. The **H** and **CLASS** commands work best when you have more room to display information. Closing the Locals window automatically increases the size of the Command window.

1 Use the **WL** command to close the Locals window.

```
:WL
```

2 To view detailed help about the **CLASS** command, enter **CLASS** as the parameter to the **H** command.

```
:H CLASS
Display window class information
CLASS [-x] [process | thread | module | class-name]
ex: CLASS USER
```

The first line of help provides a description of the command. The second line is the command syntax, including any options and/or parameters it accepts. The third line is an example of use.

3 Use the **TABLE** command to switch the current symbol table to `GDIDemo`.

```
:TABLE GDIDemo
```

4 Set a breakpoint on `CreateAppWindow` using the **BPX** command.

```
:BPX CreateAppWindow
```

5 Exit SoftICE using the **G** or **X** command (**F5**).
You are returned to the sample application.

6 Click **Other Demos** from the main screen.

7 Click **Launch GDIDemo**.
A dialog informs you that GDIDemo is about to launch.

8 Click **OK**.
SoftICE pops-up. The `RegisterAppClass` function registers window class templates that are used by the GDIDemo application to create windows. When we hit the current breakpoint, classes have been registered and we are about to create a window for GDIDemo.

9 Use the **CLASS** command to examine the classes registered by GDIDemo.

```
:CLASS GDIDEMO
```

Table 3-1: Classes Used by GDIDEMO Application

Class Name	Handle	Owner	Wndw Proc	Styles
• ------------------Application Private------------------				
BOUNCEDEMO	A018A3B0	GDIDEMO	004015A4	00000003
DRAWDEMO	A018A318	GDIDEMO	00403CE4	00000003
MAZEDEMO	A018A280	GDIDEMO	00403A94	00000003
XFORMDEMO	A018A1E8	GDIDEMO	00403764	00000003
POLYDEMO	A018A150	GDIDEMO	00402F34	00000003
GDIDEMO	A018A0C0	GDIDEMO	004010B5	00000003

Note: This example shows only those classes specifically registered by the GDIDEMO application. Classes registered by other Windows modules, such as USER32, are omitted.

The output of the CLASS command provides summary information for each window class registered on behalf of the GDIDEMO process. This includes the class name, the address of the internal WINCLASS data structure, the module which registered the class, the address of the default window procedure for the class, and the value of the class style flags.

Note: For more specific information on window class definitions, use the CLASS command with the -X option, as follows:

```
:CLASS -X
```

Using Symbols and Symbol Tables

When you load symbols for an application, SoftICE creates a symbol table that contains all the symbols defined for that module. Symbol tables now do time and date checking to ensure that the symbol files are up to date with the binary in use. Correct any discrepancies by retranslating and reloading the symbol table.

1 Use the **TABLE** command to see all the symbol tables that are loaded.

```
:TABLE
demodll [NM32]
dsdemo1d [NM32]
GDIdemo [NM32]
nmdemo1a [NM32]
0000503629 Bytes Of Symbol Memory Available
```

The currently active symbol table is listed in bold. This is the symbol table used to resolve symbol names. If the current table is not the table from which you want to reference symbols, use the **TABLE** command and specify the name of the table to make active.

2 Use the **TABLE** command to switch the current symbol table to `nmdemo1a`.

```
:TABLE nmdemo1a
```

3 Use the **TABLE** command again to see that the current symbol table has changed.

```
:TABLE
demodll [NM32]
dsdemo1d [NM32]
GDIdemo [NM32]
nmdemo1a [NM32]
0000503629 Bytes Of Symbol Memory Available
```

4 Use the **TABLE** command again to switch the current symbol table back to `GDIDemo`.

```
:TABLE GDIDemo
```

5 Use the **SYM** command to display the symbols from the current symbol table.

```
:SYM
.text(001B:00401000,000153E6 bytes)
001B:00401000 WinMain
001B:004010B5 WndProc
001B:004011DB CreateProc
001B:00401270 CommandProc
001B:00401496 PaintProc
001B:004014D2 DestroyProc
001B:004014EA lRandom
001B:00401530 CreateBounceWindow
001B:004015A4 BounceProc
001B:004016A6 BounceCreateProc
001B:00401787 BounceCommandProc
001B:0040179C BouncePaintProc
```

This list of symbol names is from the `.text` section of the executable. The `.text` section is typically used for procedures and functions. The symbols displayed in this example are all functions of GDIDemo.

The actual output from the SYM command will include all sections of the executable. SYM can also be used with a search string to look for a particular symbol.

```
:SYM paint*
.text(001B:00401000,000153E6 bytes)
001B:00403790 PaintProc
001B:00402430 PaintWindow
```

Congratulations on completing your first SoftICE debugging session! Your world will never be the same again. In this session, you traced through source code, viewed locals and structures, and set point-and-shoot, conditional, and read-write memory breakpoints. SoftICE provides many more advanced features. The SoftICE commands **ADDR**, **HEAP**, **LOCALS**, **QUERY**, **THREAD**, **TYPES**, **WATCH**, and **WHAT** are just a few of many that will help you debug smarter and faster. Refer to the *SoftICE Command Reference* for a complete list, and an explanation of all of the SoftICE commands.

Using Symbol Loader

Symbol Loader is an important tool within SoftICE. It allows you to translate and load the symbols required to do meaningful debugging. SoftICE uses NMS files as the source for its symbolic information, as we mentioned earlier in the tutorial. Compilers, on the other hand, emit symbolic information into PDB or (for older compilers) DBG files. Symbol Loader is used to translate the compiler format to the SoftICE NMS format, and load the resulting files into SoftICE.

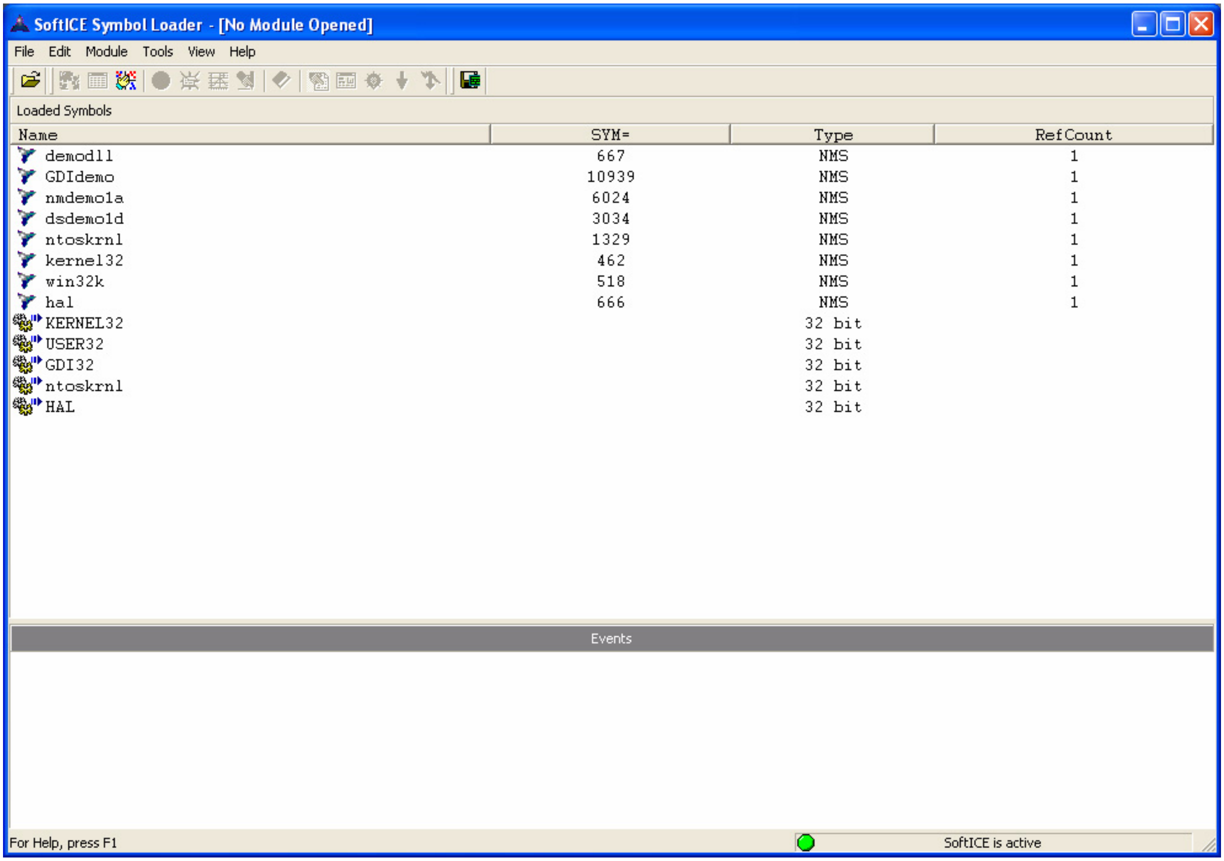
There are two basic ways you can use Symbol Loader to bring symbol files into SoftICE: Translating one file at a time in *Single File* mode, or using *Workspace View* mode, which allows you to manipulate collections of symbol files.

Single File Mode

In Single File mode, you use Symbol Loader to load symbol information into SoftICE one NMS file at a time. You can accomplish this from within the directory containing the NMS file (as you did in the beginning of this tutorial) by right-clicking on the NMS file and selecting **Load into SoftICE** from the pop-up menu. However, Symbol Loader can do much more for you than simply loading an NMS file. It can also regenerate the NMS file if it is older than the binary it is associated with, and for applications it can execute the binary and cause SoftICE to pop-up on the main (or WinMain) routine.

From within Symbol Loader, load a single symbol file into SoftICE by completing the following steps.

1 Start Symbol Loader (**Start>Programs>Compuware DriverStudio>Debug>Symbol Loader**).
The Symbol Loader window appears (Figure 3-4 on page 37).



Name	SYM=	Type	RefCount
demodll	667	NMS	1
GDIdemo	10939	NMS	1
nmdemo1a	6024	NMS	1
dsdemo1d	3034	NMS	1
ntoskrnl	1329	NMS	1
kernel32	462	NMS	1
win32k	518	NMS	1
hal	666	NMS	1
KERNEL32		32 bit	
USER32		32 bit	
GDI32		32 bit	
ntoskrnl		32 bit	
HAL		32 bit	

For Help, press F1

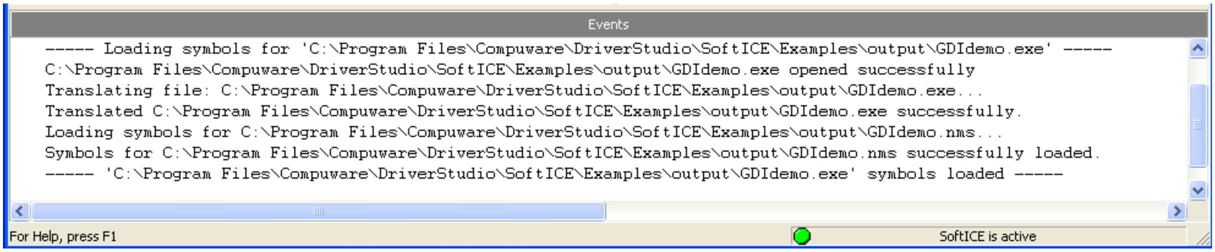
SoftICE is active

Figure 3-4 Symbol Loader Window

2. Select **Open** from the **File** menu.
   The Open File dialog appears.

3. Locate `GDIDEMO.EXE` in the **Output** directory and click **Open**.

4. Select **Load** from the **Module** menu to load GDIDEMO.
   Symbol Loader translates the debug information into an NMS file, loads the symbol and source files, starts GDIDEMO, pops up the SoftICE screen, and displays the source code for the file `GDIDEMO.C`. (Figure 3-5 on page 38). SoftICE popped up because the option Stop at WinMain was checked. To toggle this setting, select **Settings** from the **Module** menu, and click on the **Debugging** tab.



```
----- Loading symbols for 'C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\output\GDIdemo.exe' -----
C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\output\GDIdemo.exe opened successfully
Translating file: C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\output\GDIdemo.exe...
Translated C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\output\GDIdemo.exe successfully.
Loading symbols for C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\output\GDIdemo.nms...
Symbols for C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\output\GDIdemo.nms successfully loaded.
----- 'C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\output\GDIdemo.exe' symbols loaded -----
```

Figure 3-5 GDIDEMO Symbols Loaded

Workspace View Mode

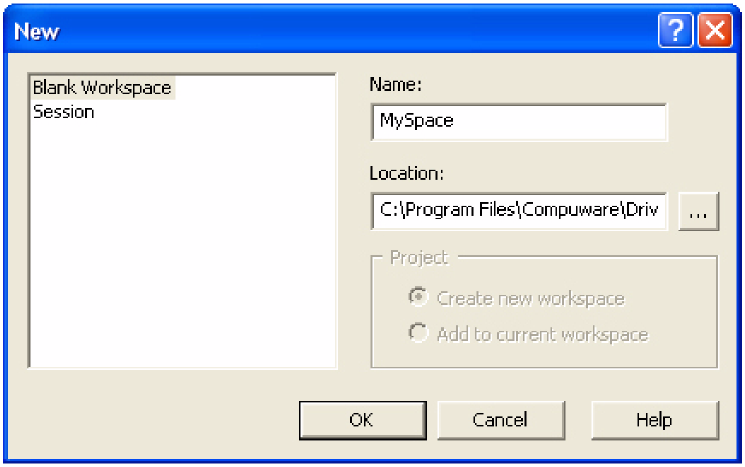
Workspace View mode allows you to create a workspace and associate a number of symbol files with it. Then, the act of loading the workspace automatically loads and translates all of the applicable symbols you need for your debugging scenario.

To create a workspace and associate symbols files with it, complete the following steps.

1. Select **New** from the **File** menu.
   Symbol Loader opens the New Workspace/Sessions dialog (see Figure 3-6 on page 39).

2. Select **Blank Workspace** from the left-hand list.

3. Enter a name for the workspace into the **Name** field, and click **OK**.
   If you have not created any workspaces previously, then Loader32 will confirm that you want to create a directory for them.
   Loader32 shifts to Workspace View mode (see Figure 3-7 on page 39).



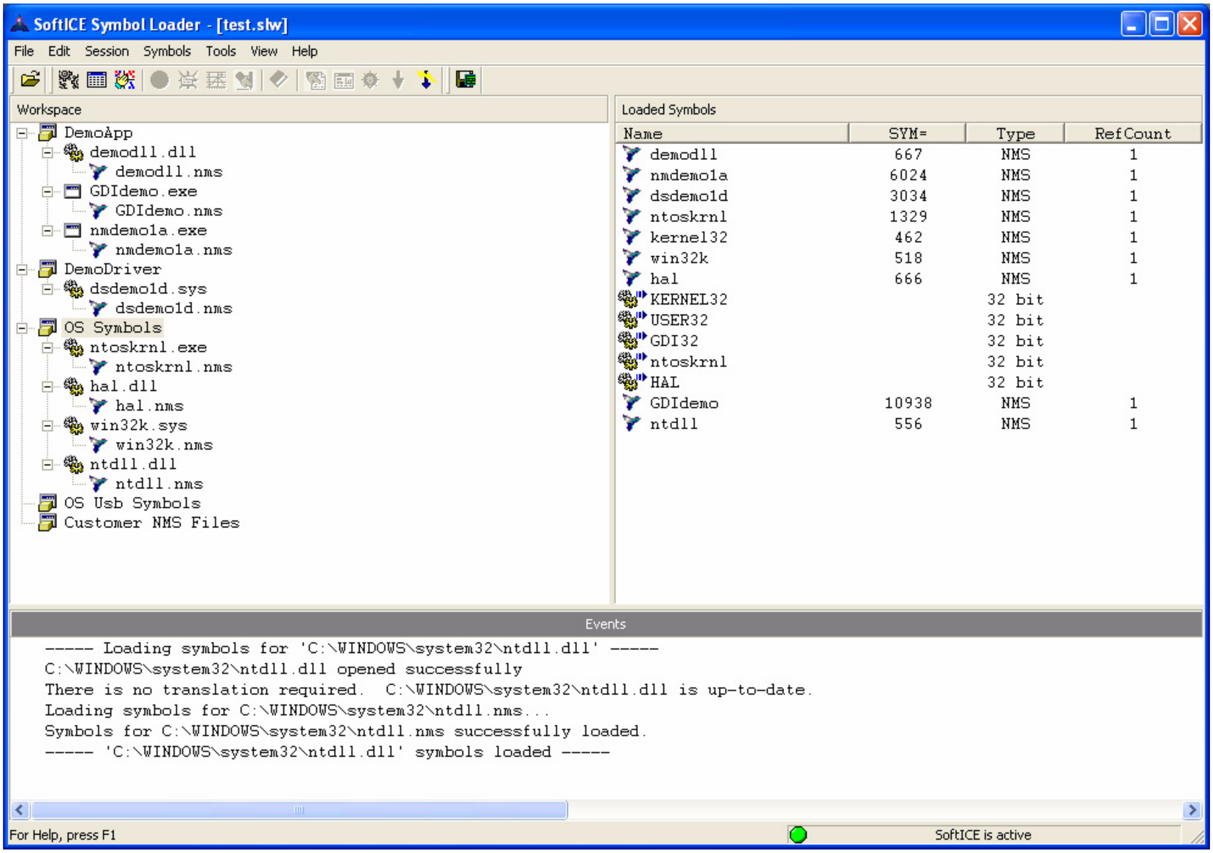
Blank Workspace
Session

Name: MySpace

Location: C:\Program Files\Compuware\Driv

Project
● Create new workspace
○ Add to current workspace

Figure 3-6 New Workspace/Sessions Dialog



```
DemoApp
  demodll.dll
    demodll.nms
  GDIdemo.exe
    GDIdemo.nms
  nmdemo1a.exe
    nmdemo1a.nms
DemoDriver
  dsdemo1d.sys
    dsdemo1d.nms
OS Symbols
  ntoskrnl.exe
    ntoskrnl.nms
  hal.dll
    hal.nms
  win32k.sys
    win32k.nms
  ntdll.dll
    ntdll.nms
OS Usb Symbols
Customer NMS Files
```

Name	SYM=	Type	RefCount
demodll	667	NMS	1
nmdemo1a	6024	NMS	1
dsdemo1d	3034	NMS	1
ntoskrnl	1329	NMS	1
kernel32	462	NMS	1
win32k	518	NMS	1
hal	666	NMS	1
KERNEL32		32 bit	
USER32		32 bit	
GDI32		32 bit	
ntoskrnl		32 bit	
HAL		32 bit	
GDIdemo	10938	NMS	1
ntdll	556	NMS	1

```
----- Loading symbols for 'C:\WINDOWS\system32\ntdll.dll' -----
C:\WINDOWS\system32\ntdll.dll opened successfully
There is no translation required.  C:\WINDOWS\system32\ntdll.dll is up-to-date.
Loading symbols for C:\WINDOWS\system32\ntdll.nms...
Symbols for C:\WINDOWS\system32\ntdll.nms successfully loaded.
----- 'C:\WINDOWS\system32\ntdll.dll' symbols loaded -----
```

For Help, press F1

SoftICE is active

Figure 3-7 Loader32 Workspace View

4. Select **New** from the **File** menu.
   Symbol Loader opens the New Workspace/Sessions dialog (see Figure 3-6 on page 39).

5. Select **Session** from the left-hand list.

6. Enter a name for the session into the **Name** field.

7. Select **Add to Current Workspace** in the Project section, and click **OK**.
   If you have not created any sessions previously, then Loader32 will confirm that you want to create a directory for them.

   Loader32 adds your session to the left-hand pane in the Workspace View.

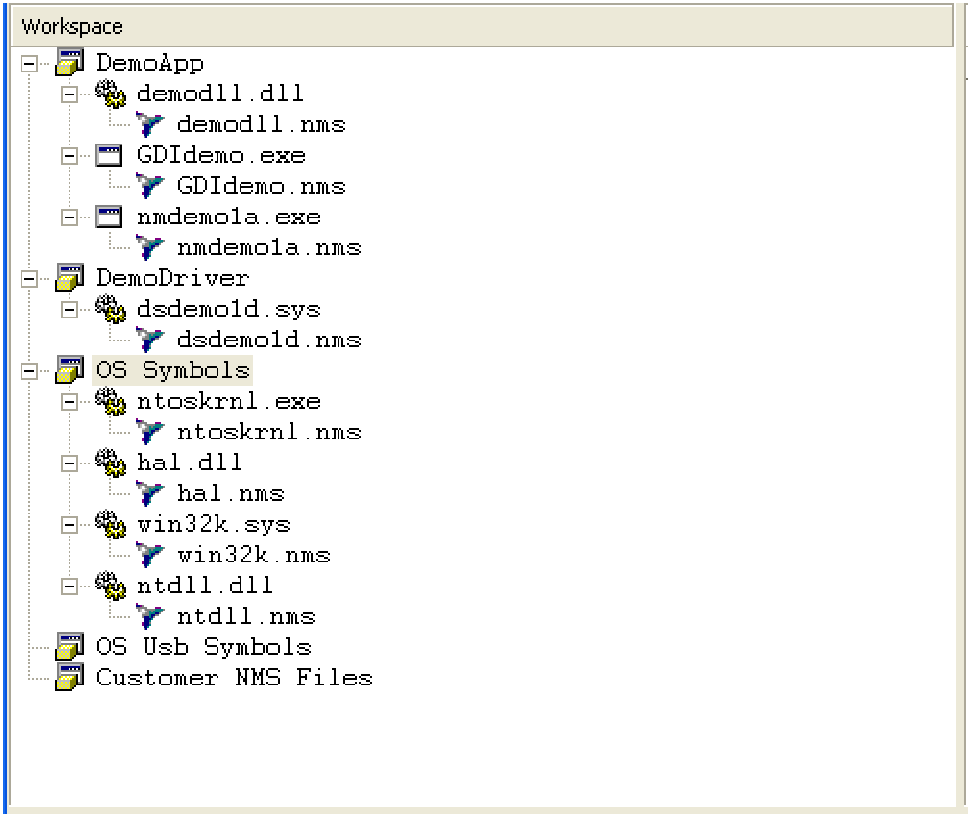
8. Repeat steps 4 through 7 to add another session to the workspace for OS symbols.
   The left-hand pane will now list two sessions in your workspace (see Figure 3-8 on page 41).

9. Right-click on the first session in the list, and select **Add Files** from the pop-up menu.
   Loader32 opens the Add Files dialog.

10. Select the `GDIDemo.exe` file and click **OK**.
    Loader32 automatically builds the symbol dependency chain and adds the `GDIDemo.exe` and `GDIDemo.nms` files to the session.

11. Repeat step 9 to access the Add Files dialog and add each of the remaining Demo Application files: nmdemo1a, dsdemo1d, and demodll (see Figure 3-8 on page 41).

12. Right-click on the second session in the list, and select **Add Files** from the pop-up menu.
    Loader32 opens the Add Files dialog.



```
Workspace
⊟ DemoApp
    ⊟ demodll.dll
        demodll.nms
    ⊟ GDIdemo.exe
        GDIdemo.nms
    ⊟ nmdemo1a.exe
        nmdemo1a.nms
⊟ DemoDriver
    ⊟ dsdemo1d.sys
        dsdemo1d.nms
⊟ OS Symbols
    ⊟ ntoskrnl.exe
        ntoskrnl.nms
    ⊟ hal.dll
        hal.nms
    ⊟ win32k.sys
        win32k.nms
    ⊟ ntdll.dll
        ntdll.nms
  OS Usb Symbols
  Customer NMS Files
```

Figure 3-8 Files Loaded into Sessions

13 Browse to the **Windows | System32** directory, and select and add each of the following Operating System symbol files: `ntdll.dll`, `ntoskrnl.exe`, `user32.dll`, `shell32.dll`, and `win32k.sys`.

14 Right-click on `ntdll.dll` in the Sessions list, and select **Settings** from the pop-up menu.
Loader32 opens the **Project Settings** dialog, and presents the General tab.

15 Select **Translate and Load** in the **Symbol Retriever** section, and Click **OK**.
This allows Loader32 to use the MS Symbol Server that is pre-defined in the **Symbol Site URL** text box. The MS Symbol Server serves as a repository for the most up-to-date OS symbols when you are debugging.

16 Repeat steps 14 and 15 for each of the remaining files in the sessions list (`ntoskrnl.exe`, `user32.dll`, `shell32.dll`, and `win32k.sys`).

A Word on Symbol Server Technology

Microsoft has introduced a very useful technology, known as Symbol Server, that helps you to ensure you are using the correct debugging symbols for your particular build, patch, service pack, or hotfix of the operating system. A symbol server is a repository for debug files that are uniquely identified and tied to a specific build of a binary.

Microsoft also supplies the `symstore.exe` utility, allowing you to build your own symbol server repository for your own binaries. SoftICE can be configured to retrieve symbols from any symbol server site. This can be done through the Symbol Loader, as seen in this tutorial chapter, and also through the stand-alone Symbol Retriever utility (which is located under the **Debug** menu, and is freely available for download).

Chapter 4

Loading Code into SoftICE

- Debugging Concepts
- Loading SoftICE
- Using Symbol Loader to Translate and Load Files
- Modifying Module Settings
- Specifying Modules and Files
- Deleting Symbol Tables
- Downloading Symbols from a Symbol Server
- Using Symbol Loader From an MS-DOS Prompt
- Using the Symbol Loader Command-Line Utility

Debugging Concepts

SoftICE allows you to debug Windows applications and device drivers at the source level. To accomplish this, SoftICE uses the *Symbol Loader* utility to translate the debug information from your compiled module into an .NMS symbol file. When this is done, Symbol Loader can load the .NMS file and, optionally, the source code into SoftICE, where you can debug it.

The point in time at which you need to load the .NMS file depends on whether you are debugging a module that runs after the operating system boots or a device driver or static VxD that loads before the operating system initializes. If you are loading a device driver or VxD, SoftICE pre-loads the module's symbols and source when it initializes. If you are debugging a module or component that runs after the operating system boots, you can use Symbol Loader to load symbols when you need them.

This chapter explains how to use Symbol Loader to load your module into SoftICE. It also describes how to use Symbol Loader from a DOS prompt to automate many of the most common tasks it performs and how to use the Symbol Loader command-line utility (NMSYM) to create a batch process to translate and load symbol information.

Note: Symbol Loader only supports Windows applications. To debug MS-DOS applications use the tools in the UTIL16 directory.

Preparing to Debug Applications

The following general steps explain how to prepare to debug modules and components that run after the operating system boots. These modules include EXEs, DLLs, dynamic VxDs, and OCXs. The sections that follow explain how to perform these steps in detail.

1. Build the module with debug information.
2. If SoftICE is not already loaded, load SoftICE.
3. Start Symbol Loader.
4. Select **File** > **Open** and open the module that you want to debug.
5. Use Symbol Loader to translate the debug information into a .NMS symbol file and load the source and symbol files into SoftICE for you.

Preparing to Debug Device Drivers and VxDs

The following general steps explain how to prepare to debug device drivers or static VxDs that load before the operating system fully initializes. The sections that follow explain how to perform these steps in detail.

1. Build the application with debug information.
2. If SoftICE is not already loaded, load SoftICE.
3. Start Symbol Loader.
4. Click the OPEN button to open the module you want to debug.
5. Select the PACKAGE SOURCE WITH SYMBOL TABLE setting within the Symbol Loader translation settings. Refer to *Modifying Module Settings* on page 49.
6. Click the TRANSLATE button to create a new .NMS symbol file.

7 Modify the SoftICE initialization settings to pre-load the debug information for the VxD or device driver on startup. Refer to *Pre-Loading Symbols and Source Code* on page 196.

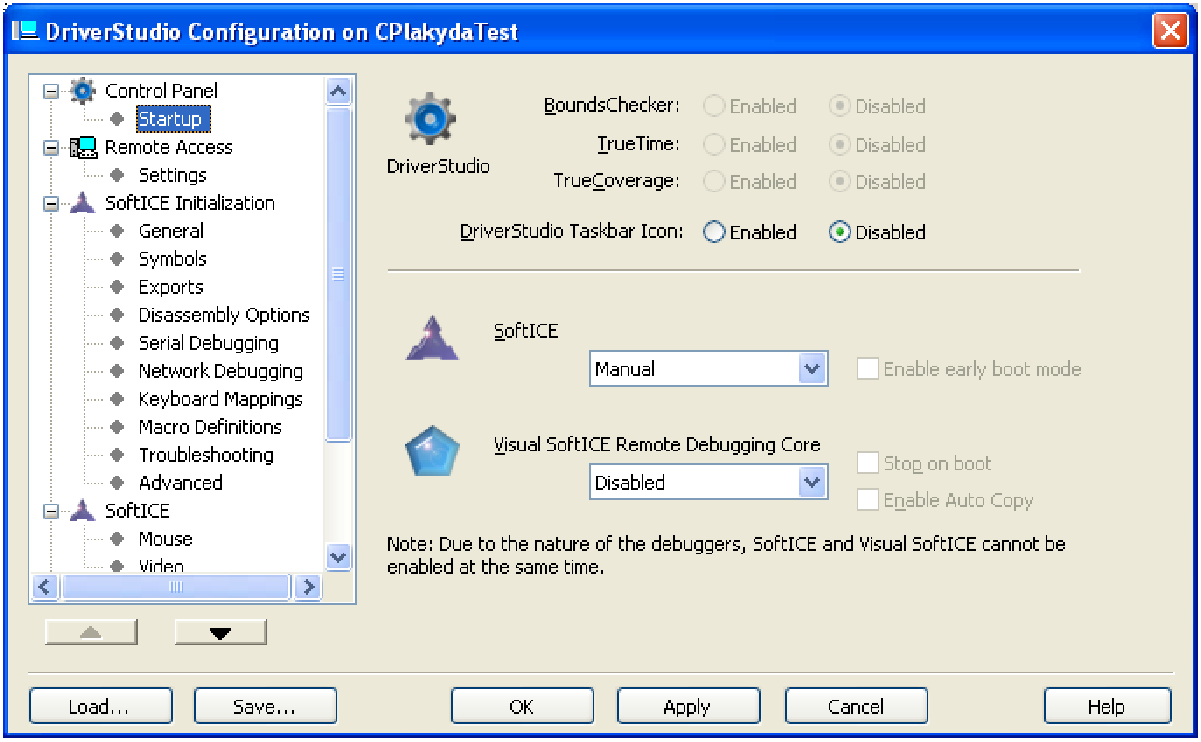
8 Reboot your machine.

Loading SoftICE

If you are running SoftICE in Boot, System, or Automatic mode, it automatically loads when you start or reboot your PC. If you are running SoftICE in Manual or Disabled mode, it does not load automatically. To change the mode in which you have SoftICE configured to load, access the Startup screen in the DSConfig utility, and select the desired mode from the SoftICE drop-down list (See Figure 4-1 on page 46).

Early Loading of SoftICE

When set to **Boot Mode**, SoftICE loads near the end of the list of boot drivers. This is sufficient for most situations, and is done so that we have easier access to a number of hardware devices, such as hard drives. It is possible to configure SoftICE to be the first item loaded after NTOSKRNL and HAL. To Enable/Disable early boot mode, set SoftICE for **Boot Mode** then check or clear the **Enable Early Boot Mode** check-box. For additional information read the `siwsym.txt` readme in the SoftICE installation directory.



BoundsChecker: ○ Enabled ● Disabled

TrueTime: ○ Enabled ● Disabled

TrueCoverage: ○ Enabled ● Disabled

DriverStudio Taskbar Icon: ○ Enabled ● Disabled

SoftICE: Manual

□ Enable early boot mode

Visual SoftICE Remote Debugging Core: Disabled

□ Stop on boot

□ Enable Auto Copy

Note: Due to the nature of the debuggers, SoftICE and Visual SoftICE cannot be enabled at the same time.

Figure 4-1 Startup Configuration Screen

Loading SoftICE Manually

SoftICE does not load automatically when you configure the startup mode to Manual or Disabled. If you have SoftICE startup mode set to Manual or Disabled, you need to load SoftICE manually. To load SoftICE manually, do one of the following:

- Select **START SOFTICE** from the Compuware/SICE group
- Enter the command: **`NET START NTICE`**

Note: Once you load SoftICE, you cannot deactivate it until you reboot your PC.

Building Applications with Debug Information

The following compiler-specific information is provided as a guideline. If you are building an application with debug information, consult your compiler or assembler documentation for more information.

Table 4-1. Compiler-specific Debugging Information

Compiler	Generating Debugging Information
Borland C++ 4.5 and 5.0	To generate Borland's standard debug information: Compile with `/v` `Link with /v`

Delphi 2.0	To generate Delphi's standard debug information:
	Compile with the following:
	-V to include debug information in the executable
	`-$W+ to create stack frames`
	`-$D+ to create debug information`
	`-$L+ to create local debug symbols`
	-$O- to disable optimization

MASM 6.11	To generate Codeview debug information: Assemble with `/Zi /COFF` Use Microsoft's 32-bit LINK.EXE to link with `/DEBUG /DEBUGTYPE:CV /PDB:NONE`
Microsoft Visual C++ 2.x, 4.0, 4.1, 4.2, 5.0, 6.0, and Visual Studio .NET	To generate Program Database (PDB) debug information: Compile with Program Database debug information, using the command-line option `/Zi` Use Microsoft's linker to link with `/DEBUG /DEBUGTYPE:CV` *Note:* `VxDs require you to generate PDB debug information.` To generate Codeview debug information: Compile with C7-compatible debug information, using the command-line option `/Z7` Use Microsoft's linker to link with `/DEBUG /DEBUGTYPE:CV /PDB:NONE` *Note:* If you are using the standard Windows NT DDK make procedure, use the following environment variables: NTDEBUG=ntsd and NTDEBUG-TYPE=windbg.

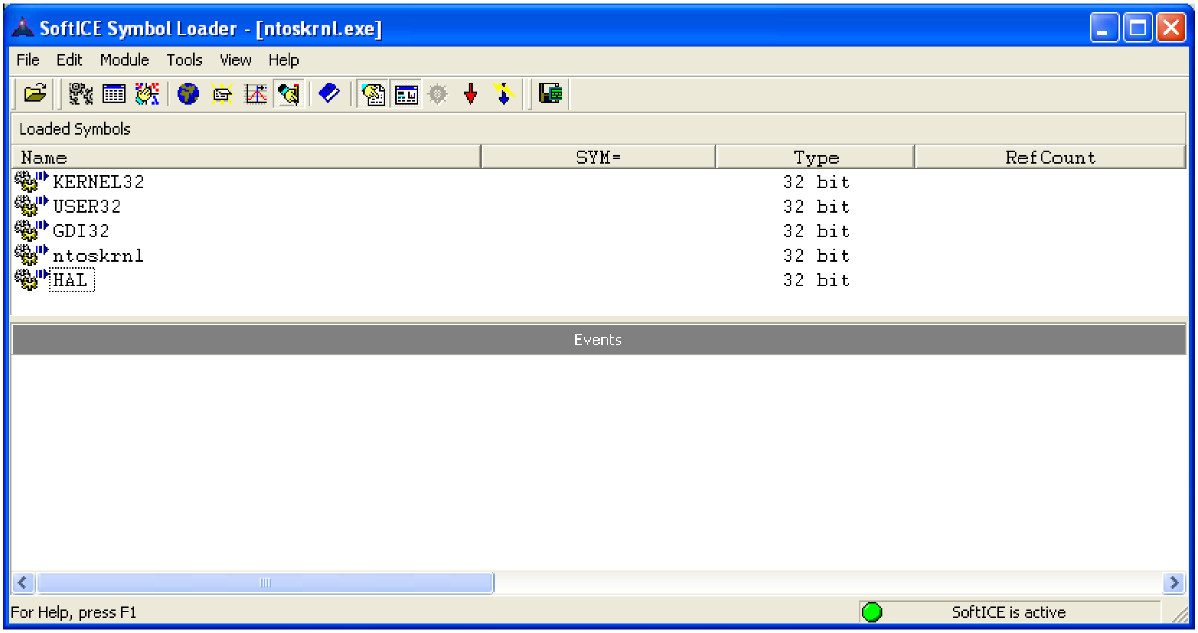
Note: SoftICE supports other compilers that may not appear in the above table. In general, SoftICE provides symbolic debugging for any compiler that produces Codeview compatible debug information.

Using Symbol Loader to Translate and Load Files

Before SoftICE can debug your application, DLL, or driver, you need to create a symbol file for each of the modules you want to debug, and load these files into SoftICE. Symbol Loader makes this procedure quick and easy. Symbol Loader lets you identify the module you want to load, then automatically creates a corresponding symbol file. Finally, Symbol Loader loads the symbol, source, and executable files into SoftICE. By default, Symbol Loader loads all the files referenced in the debug information. To limit the source files Symbol Loader loads, refer to *Specifying Modules and Files* on page 55.

To use Symbol Loader to load a module, do the following:

1 Start Symbol Loader.



Name	SYM=	Type	RefCount
KERNEL32		32 bit	
USER32		32 bit	
GDI32		32 bit	
ntoskrnl		32 bit	
HAL		32 bit	

For Help, press F1

SoftICE is active

Figure 4-2. Symbol Loader Window

2 Choose **File** > **Open** from the File menu.

3 Select your translation options.

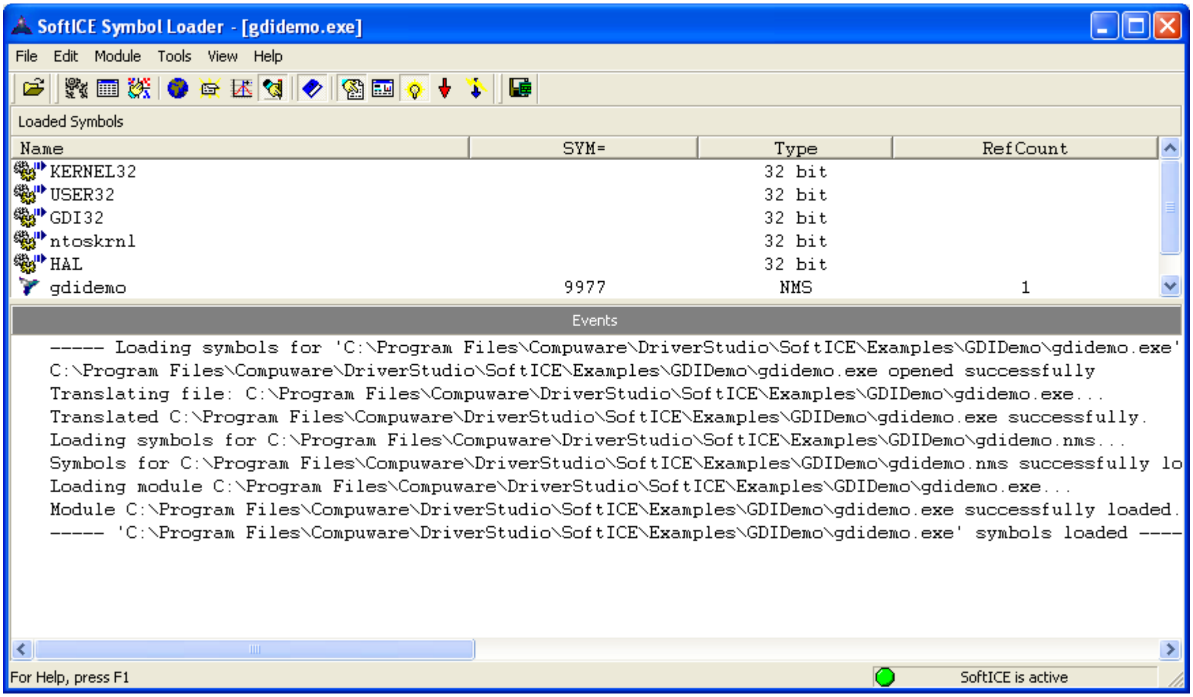
4 If you open a .SYM file, Symbol Loader displays a dialog box that asks you whether or not the file is a 32-bit file. If it is a 32-bit file, click YES; otherwise, click NO.

5 Choose **Module** > **Load** from the Module menu.

Symbol Loader translates your application’s debug information to an .NMS symbol file. Then, Symbol Loader loads the symbol and source files into SoftICE. (See Figure 4-2.)

If you are loading an .EXE file, SoftICE starts the program and sets a breakpoint at the first main module (WinMain, Main, or DllMain) it encounters.

The information Symbol Loader loads depends on the Translation and Debugging settings. Refer to *Modifying Module Settings* for more information about modifying Translation and Debugging settings.



Name	SYM=	Type	RefCount
KERNEL32		32 bit	
USER32		32 bit	
GDI32		32 bit	
ntoskrnl		32 bit	
HAL		32 bit	
gdidemo	9977	NMS	1

```
----- Loading symbols for 'C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\GDIDemo\gdidemo.exe'
C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\GDIDemo\gdidemo.exe opened successfully
Translating file: C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\GDIDemo\gdidemo.exe...
Translated C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\GDIDemo\gdidemo.exe successfully.
Loading symbols for C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\GDIDemo\gdidemo.nms...
Symbols for C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\GDIDemo\gdidemo.nms successfully lo
Loading module C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\GDIDemo\gdidemo.exe...
Module C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\GDIDemo\gdidemo.exe successfully loaded.
----- 'C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\GDIDemo\gdidemo.exe' symbols loaded ----
```

Figure 4-3. Symbols Loaded

Modifying Module Settings

The Symbol Loader uses a series of settings to control how it translates and loads files. These settings are categorized as follows:

- **General** — Specifies command-line arguments and symbol server configuration information.
- **Debugging** — Specifies the types of files (symbols and executables) Symbol Loader loads into SoftICE, as well as any default actions SoftICE performs at load time.

- **Translation** — Specifies which combination of symbols (publics, type information, symbols, or symbols and source) Symbol Loader translates.
- **Modules and Files** — Associates additional debug files to load when loading the base file.
- **Source Files** — Specifies local and global source file paths.

These settings are available on a per-module basis. Thus, changing a particular setting applies to the current module only. When you open a different module, Symbol Loader uses the pre-established defaults.

Tip The name of the current open file is listed in the Symbol Loader title bar.

To change the default file settings for a module, do the following:

1. Open the file if it is not already open.
2. Select **Module** > **Settings**.
3. Click the tab that represents the settings you want to modify.
4. See the sections that follow for more information about specific settings for each tab.
5. When you are done modifying the settings, click OK.
6. Load the file to apply your changes.

Modifying General Settings

The General tab (Figure 4-4) allows you to set command-line arguments and configure the symbol retriever.

The following paragraphs describe the General settings selections.

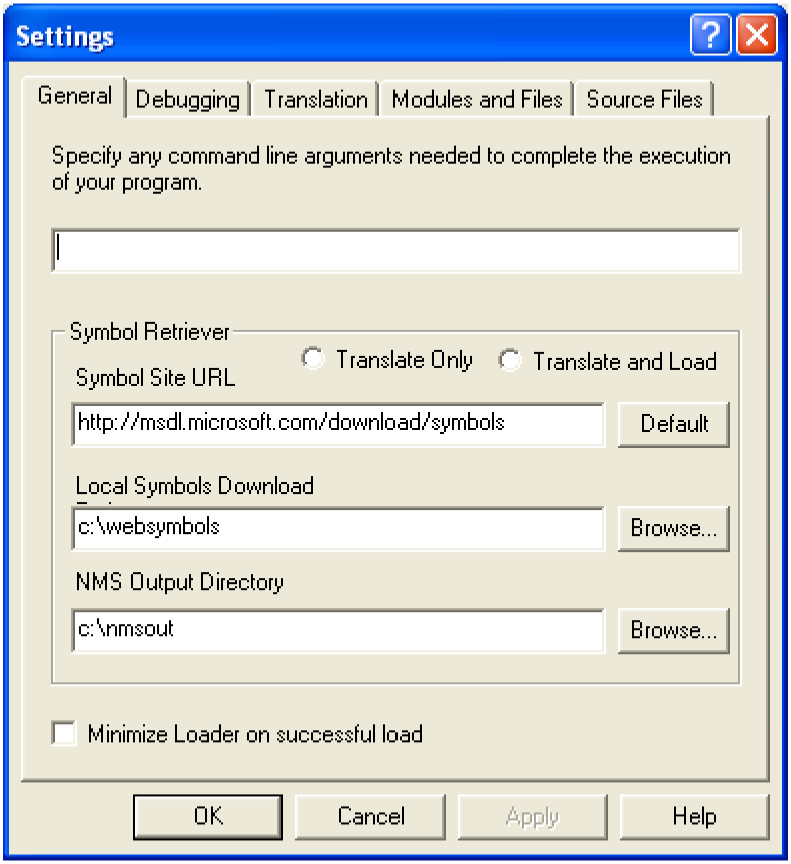
Command Line Arguments

Use **Command line arguments** to specify command-line arguments to pass to your program.

Symbol Retriever Configuration

Symbol loader has the ability to download symbols from any symbol server site. By default it points to the Microsoft public symbol server site. You can easily reconfigure it to point to a private or corporate site. For more information on setting up a symbol server site, refer to the documentation in the DDK for SYMSERVER.

The options allow you to specify where to store the `.pdb` or `.dbg` files, and also to specify the location of the resulting `.nms` file.



Settings

General | Debugging | Translation | Modules and Files | Source Files

Specify any command line arguments needed to complete the execution of your program.

Symbol Retriever

○ Translate Only ○ Translate and Load

Symbol Site URL: http://msdl.microsoft.com/download/symbols

Local Symbols Download: c:\websymbols

NMS Output Directory: c:\nmsout

□ Minimize Loader on successful load

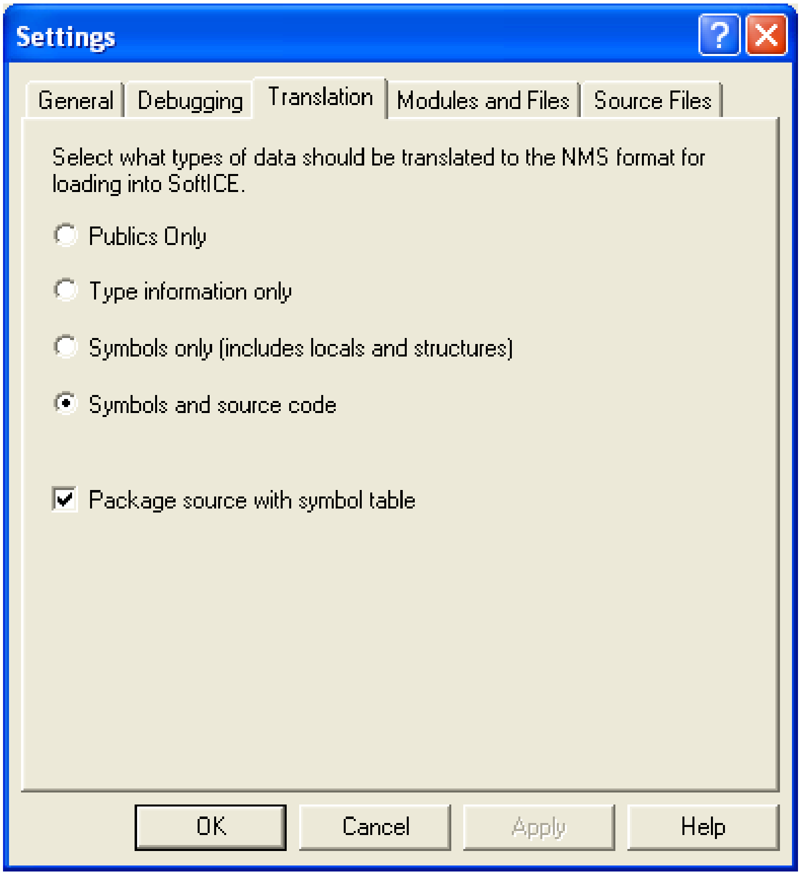
Figure 4-4. General Tab

Prompt for Missing Source Files

Check the **Prompt for missing source files** check box to determine if Symbol Loader is to prompt you when it cannot find a source file. This setting is global and is turned on by default.

Modifying Translation Settings

The Translation tab settings (Figure 4-4) determine the type of information Symbol Loader translates when it creates .NMS symbol files and specifies if your source code is stored in the symbol file. These settings determine how much memory is needed to debug your program and they are listed in order from least to most amount of symbol memory required. The following paragraphs describe the Translation settings selections.



Select what types of data should be translated to the NMS format for loading into SoftICE.

○ Publics Only

○ Type information only

○ Symbols only (includes locals and structures)

● Symbols and source code

☑ Package source with symbol table

Figure 4-5. Translation Tab

Publics Only

Publics Only provides public (global) symbol names. Neither type information nor source code are included.

Type information only

This setting provides type information only. Use this setting to provide type information for data structures that are *reverse engineered*.

Symbols only

Symbols only provides global, static, and local symbol names in addition to type information. Source code is not included.

Symbols and source code

Symbols and source code provides all available debugging information, including source code and line number information. This setting is enabled by default.

Package source with symbol table

This setting saves your source code with the symbol information in the .NMS file. You might want to include your source file in the symbol file under the following circumstances:

- Loading source code at boot time.
- SoftIce does not look for code files at boot time. If you need to load source code for a VxD or Windows NT device driver, select **Package source with symbols table.** Then, modify the SoftICE initialization settings to load the debug information for the VxD or device driver on startup. Refer to *Pre-Loading Symbols and Source Code* on page 196.
- Debugging on a system that does not have access to your source files.
- If you want to debug your application on a system that does not have access to your source files, select PACKAGE SOURCE WITH SYMBOLS and copy the .NMS file to the other system.

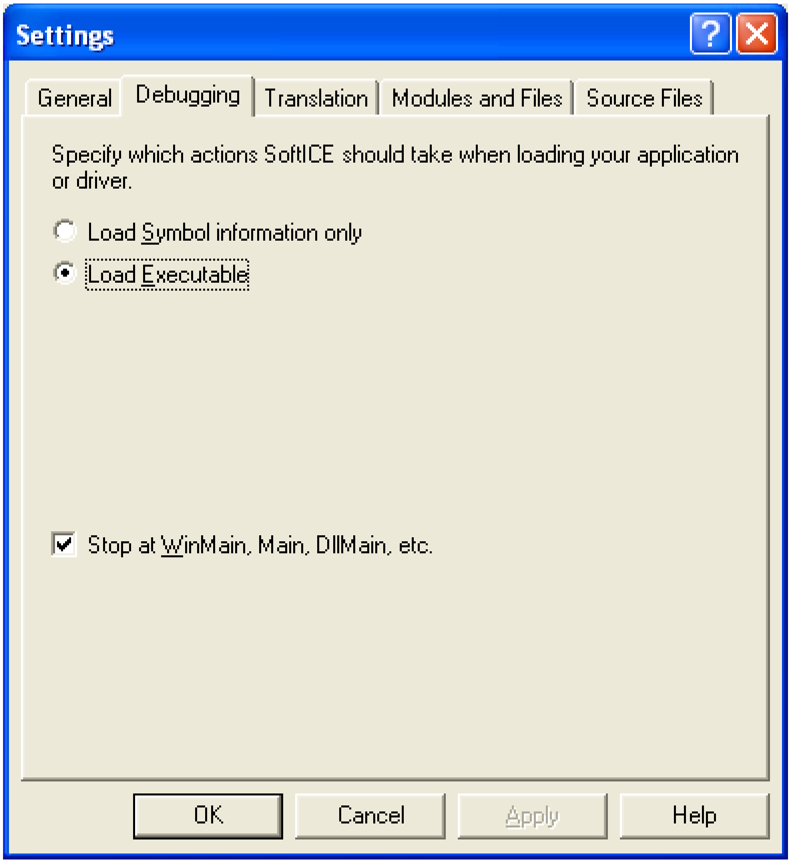
Caution: **If you select** *Package source with symbol table*, **your source code is available to anyone who accesses the symbol table. If you do not want others to have access to your source code and you provide the .NMS file with your application, turn off this option.**

Modifying Debugging Settings

The Debugging tab settings (Figure 4-5) determine what type of information to load and whether or not to stop at the module entry point. The following paragraphs describe the Debugging settings selections.

Load symbol information only

Load symbol information only loads the .NMS symbol file, but does not load the executable image. It also loads the associated source files if you selected Symbols and Source Code in the Translation options. By default, Symbol Loader selects this setting for .DLL, .SYS, and VxD file types.



Specify which actions SoftICE should take when loading your application or driver.

○ Load Symbol information only

● Load Executable

☑ Stop at WinMain, Main, DllMain, etc.

Figure 4-6. Debugging Tab

Load executable

Load executable loads your executable and .NMS file. It also loads the associated source files if you selected **Symbols and Source Code** in the Translation options. By default, Symbol Loader makes this selection for .EXE files.

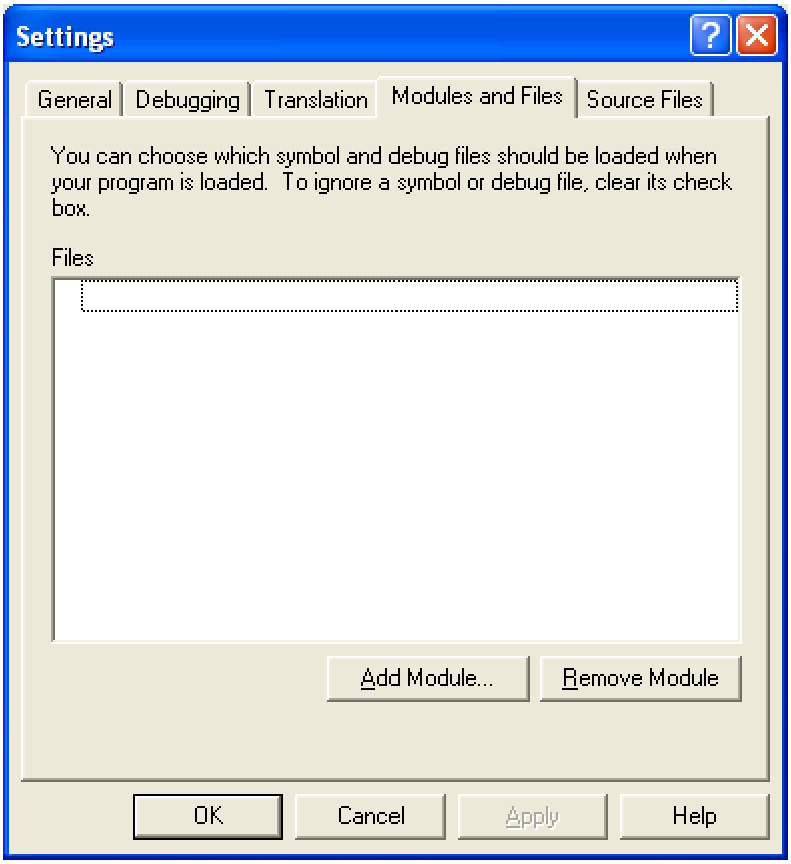
Stop at WinMain, Main, DllMain, etc.

This setting creates a breakpoint at the first main module SoftICE encounters as it loads your application.

Specifying Modules and Files

By default, all program source files that are referenced in the debug information are loaded. Depending on your needs, loading all program source files may not be necessary. Also, if the number of source files is large, loading all source files may not be practical.

The **Modules and Files** tab settings (Figure 4-6) determine which symbol and debug files should be loaded when your program is loaded. To ignore a symbol or debug file, clear its check box. To add or remove a module use the buttons provided.



You can choose which symbol and debug files should be loaded when your program is loaded. To ignore a symbol or debug file, clear its check box.

Figure 4-7. Modules and Files Tab

SoftICE also lets you use a .SRC file to specify which source files to load for an executable module. A .SRC file is a text file that you create in the directory where your executable resides. The filename of the .SRC file is the same as the filename of the executable, but with a .SRC extension. The .SRC file contains a list of the source files that are to be loaded, one per line.

If you have an executable named PROGRAM.EXE, you would create a .SRC file, PROGRAM.SRC. The contents of the PROGRAM.SRC file might look like the following:

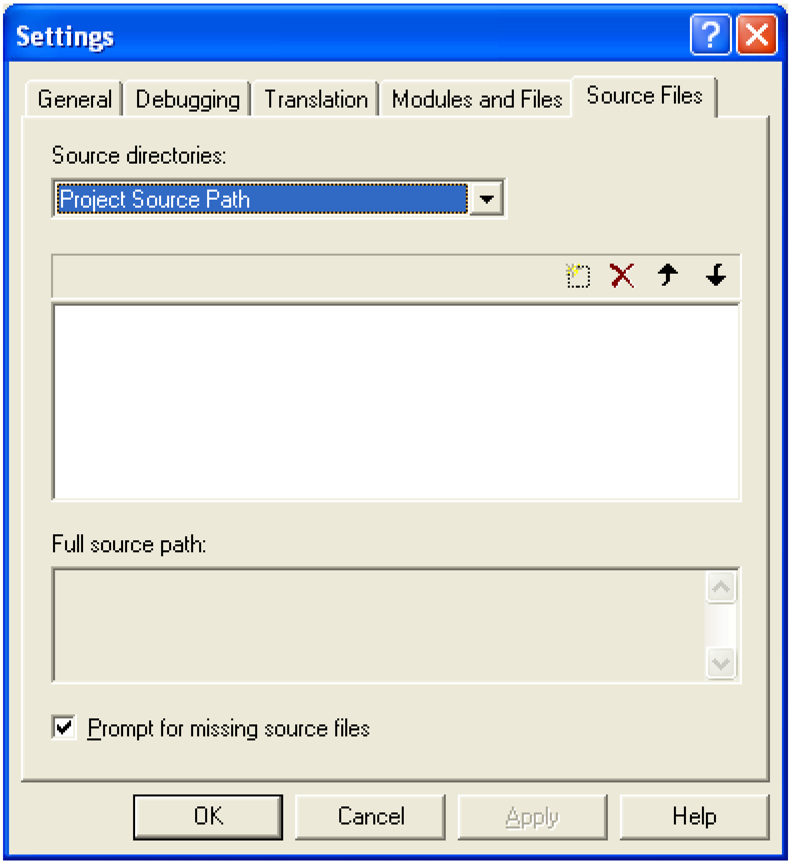
```
FILE1.C
FILE3.CPP
FILE4.C
```

Assuming that FILE2.C was a valid program source file, it would not be loaded because it does not appear in the .SRC file. FILE1.C, FILE3.CPP, and FILE4.C would be loaded.

Modifying Source Files

The Source Files tab (Figure 4-8.) allows you to specify source file search paths.

The following paragraphs describe the Source File selections.



Source directories: Project Source Path

Full source path:

☑ Prompt for missing source files

Figure 4-8. Source Files Tab

Source File Search Path

Use **Source file search path** to determine the search path SoftICE uses to locate files associated with this application. If Symbol Loader cannot locate the files within this search path, it uses the contents of the **Global source file search path** to expand its search.

Global Source File Search Path

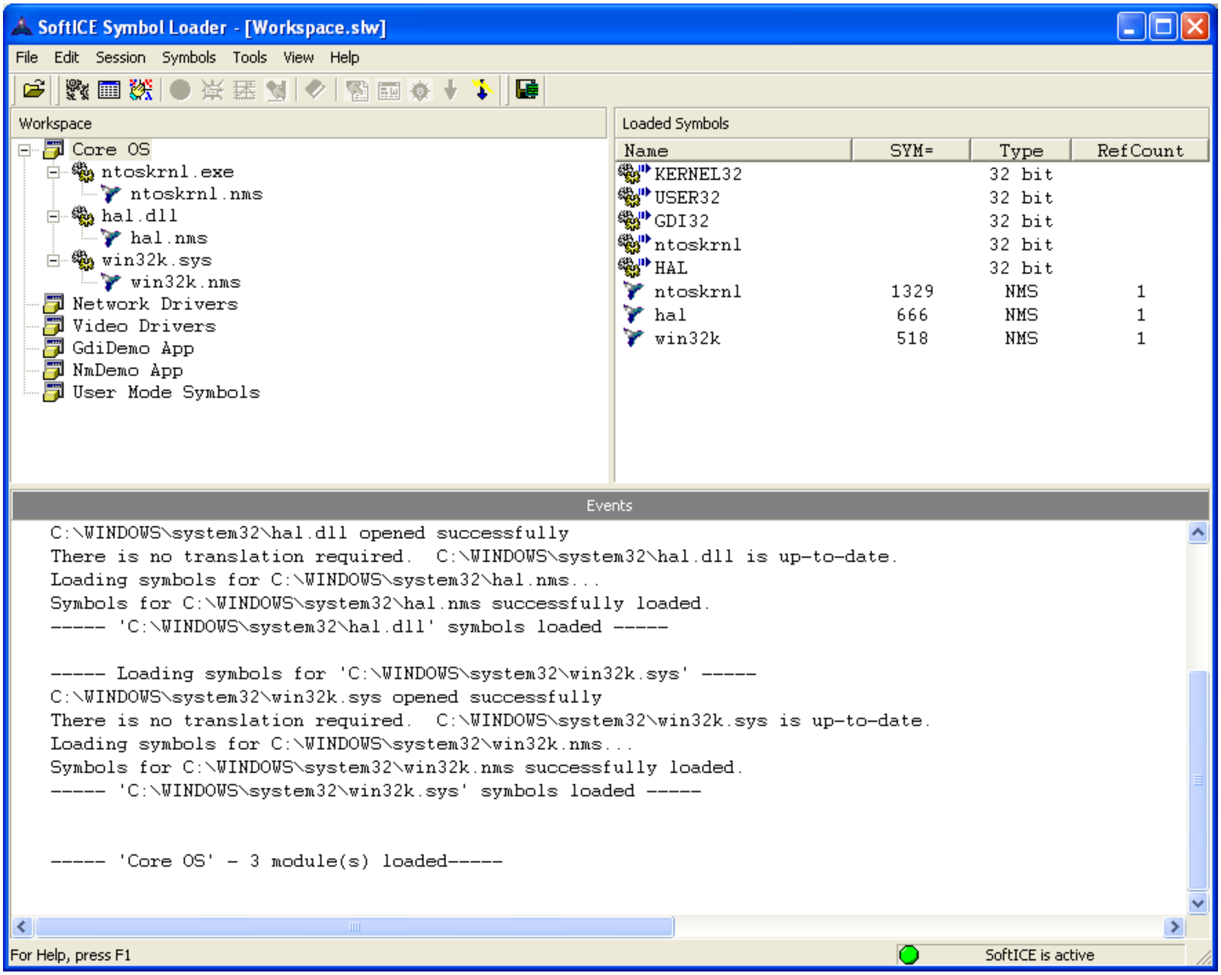
Use **Global source file search path** to determine the search path SoftICE uses to locate files in general. This setting is a global setting.

Note: If you use the **Source file search path** setting to specify the search path for a specific program, Symbol Loader uses the search path you specified for the application before looking at the global search path.

Deleting Symbol Tables

Every time you translate your source code, Symbol Loader creates a .NMS symbol file in the form of a symbol table. When you load your module, Symbol Loader stores the table in memory until you either delete the table or reboot your machine. To delete a symbol table, complete the following steps.

1 Choose **Symbol Tables** from the Edit menu.


SoftICE Symbol Loader - [Workspace.slw]
File Edit Session Symbols Tools View Help
Workspace
Loaded Symbols
Events
For Help, press F1
SoftICE is active

```
Core OS
    ntoskrnl.exe
        ntoskrnl.nms
    hal.dll
        hal.nms
    win32k.sys
        win32k.nms
Network Drivers
Video Drivers
GdiDemo App
NmDemo App
User Mode Symbols
```

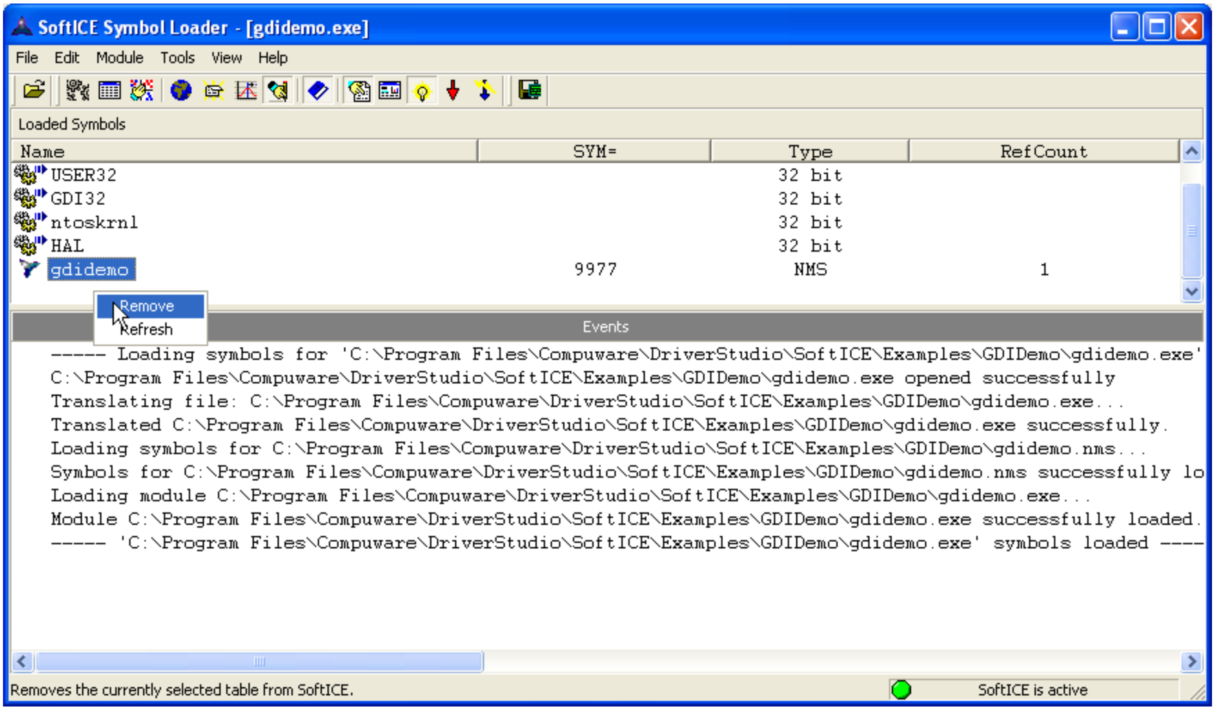
Name	SYM=	Type	RefCount
KERNEL32		32 bit	
USER32		32 bit	
GDI32		32 bit	
ntoskrnl		32 bit	
HAL		32 bit	
ntoskrnl	1329	NMS	1
hal	666	NMS	1
win32k	518	NMS	1

```
C:\WINDOWS\system32\hal.dll opened successfully
There is no translation required.  C:\WINDOWS\system32\hal.dll is up-to-date.
Loading symbols for C:\WINDOWS\system32\hal.nms...
Symbols for C:\WINDOWS\system32\hal.nms successfully loaded.
----- 'C:\WINDOWS\system32\hal.dll' symbols loaded -----

----- Loading symbols for 'C:\WINDOWS\system32\win32k.sys' -----
C:\WINDOWS\system32\win32k.sys opened successfully
There is no translation required.  C:\WINDOWS\system32\win32k.sys is up-to-date.
Loading symbols for C:\WINDOWS\system32\win32k.nms...
Symbols for C:\WINDOWS\system32\win32k.nms successfully loaded.
----- 'C:\WINDOWS\system32\win32k.sys' symbols loaded -----


----- 'Core OS' - 3 module(s) loaded-----
```

Figure 4-9. Symbol Loader with Workspace Pane

2 Right-click on the .NMS file in the Loaded Symbols list and select **Remove** from the pop-up menu.

Note: You can also right-click on an item in the Loaded Symbols view (Figure 4-10.) and select **Remove** from the pop-up menu for an individual file. The selected symbol table is removed (Figure 4-11.).


SoftICE Symbol Loader - [gdidemo.exe]
File Edit Module Tools View Help
Loaded Symbols
Remove
Refresh
Events

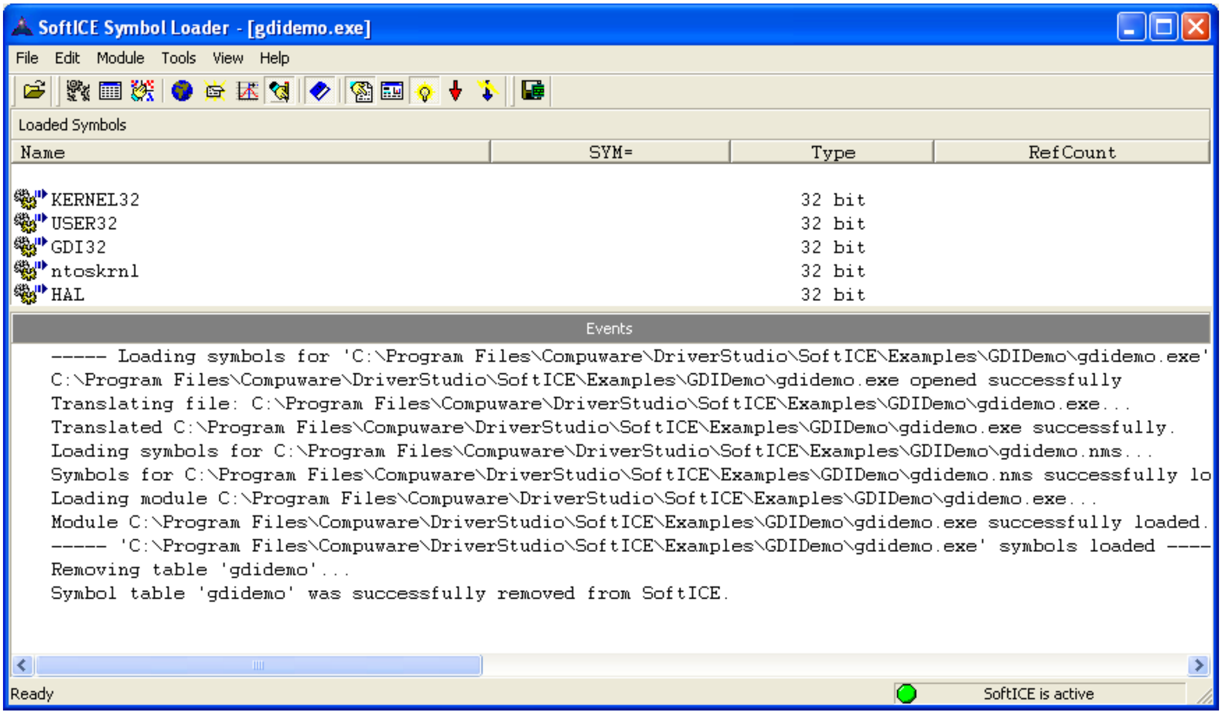
Name	SYM=	Type	RefCount
USER32		32 bit	
GDI32		32 bit	
ntoskrnl		32 bit	
HAL		32 bit	
gdidemo	9977	NMS	1

```
----- Loading symbols for 'C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\GDIDemo\gdidemo.exe'
C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\GDIDemo\gdidemo.exe opened successfully
Translating file: C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\GDIDemo\gdidemo.exe...
Translated C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\GDIDemo\gdidemo.exe successfully.
Loading symbols for C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\GDIDemo\gdidemo.nms...
Symbols for C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\GDIDemo\gdidemo.nms successfully lo
Loading module C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\GDIDemo\gdidemo.exe...
Module C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\GDIDemo\gdidemo.exe successfully loaded.
----- 'C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\GDIDemo\gdidemo.exe' symbols loaded ----
```

Removes the currently selected table from SoftICE.

SoftICE is active

Figure 4-10. Removing a Symbol Table



SoftICE Symbol Loader - [gdidemo.exe]
File Edit Module Tools View Help
Loaded Symbols
Events
Ready
SoftICE is active

Name	SYM=	Type	RefCount
KERNEL32		32 bit	
USER32		32 bit	
GDI32		32 bit	
ntoskrnl		32 bit	
HAL		32 bit	

```
----- Loading symbols for 'C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\GDIDemo\gdidemo.exe'
C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\GDIDemo\gdidemo.exe opened successfully
Translating file: C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\GDIDemo\gdidemo.exe...
Translated C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\GDIDemo\gdidemo.exe successfully.
Loading symbols for C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\GDIDemo\gdidemo.nms...
Symbols for C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\GDIDemo\gdidemo.nms successfully lo
Loading module C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\GDIDemo\gdidemo.exe...
Module C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\GDIDemo\gdidemo.exe successfully loaded.
----- 'C:\Program Files\Compuware\DriverStudio\SoftICE\Examples\GDIDemo\gdidemo.exe' symbols loaded ----
Removing table 'gdidemo'...
Symbol table 'gdidemo' was successfully removed from SoftICE.
```

Figure 4-11. Symbol Table Removed Statement

Downloading Symbols from a Symbol Server

To download OS symbol files, complete the following steps.

1 Open the OS binary.

2 Change the **Symbol Retriever** option from the general page to **Translate and Load.**

3 Select **Download** from the **Module** menu or click the associated toolbar icon.

Using Symbol Loader From an MS-DOS Prompt

Symbol Loader (LOADER32.EXE) supports a command-line interface that lets you use many of its features from a DOS prompt without viewing Symbol Loader's graphical interface. Thus, you can automate many of the most common tasks it performs.

Before you use LOADER32.EXE from a DOS prompt, use Symbol Loader's graphical interface to set the default search paths and to specify translation and debugging settings for each module you plan to load. Symbol Loader save these settings for each file and uses them when you use LOADER32 to load or translate the files from a DOS prompt. Refer to *Modifying Module Settings* on page 49.

To run LOADER32.EXE, either set your directory to the directory that contains LOADER32.EXE or specify the SoftICE directory in your search path.

Command Syntax

Use the following syntax for LOADER32.EXE:

```
LOADER32 [[option(s)] file-name]
```

Where `file-name` is the name of the file you want to translate or load and `options` are as shown in Table 3-3.

Table 4-2. Symbol Loader Command-Line Options

Option	Definition
/EXPORTS	Loads exports for a file.
/LOAD	Translates the module into a .NMS file, if one does not already exist, and loads it into SoftICE. If you previously set Translation and Debugging settings for this file, LOADER32.EXE uses these settings. If you did not specify these settings, LOADER32.EXE uses the defaults for the module type.
/LOGFILE	Saves the SoftICE history buffer to a log file.
/NOPROMPT	Instructs LOADER32.EXE not to prompt you if it cannot find a source file.
/PACKAGE	Saves your source code with the symbol information in the .NMS file.
/TRANSLATE	Translates the module into a .NMS file using the Translation settings you set the last time you translated the file or, if none exist, the default translation for the module type.

Follow these guidelines when specifying the command syntax:

- Options are not required. If you specify a file name without an option, LOADER32.EXE starts the Symbol Loader graphical interface and opens the file.
- Specify both the /TRANSLATE and /LOAD options to force LOADER32.EXE to translate the module before loading it.
- Do not use the /EXPORTS or the /LOGFILE options with any other option.

Note: If you specify an option, LOADER32.EXE does not display the Symbol Loader graphical interface unless it encounters an error. If LOADER32.EXE encounters an error, it displays the error in the Symbol Loader window.

Using the Symbol Loader Command-Line Utility

NMSYM is a utility program that lets you create a batch process to translate and load symbol information for use with SoftICE or other programs that use the NM32™ symbol table file format. NMSYM provides a series of command options analogous to features within SoftICE Symbol Loader (Loader32.exe) that perform the following functions:

Table 4-3. NMSYM Command-Line Options

Function	NMSYM Options
Translate and load symbol information for an individual module	/TRANSLATE or /TRANS /LOAD /SOURCE /ARGS /OUTPUT or /OUT /PROMPT
Load and unload groups of symbol tables and module exports	/SYMLOAD or /SYM /EXPORTS or /EXP /UNLOAD
Save the SoftICE history buffer to a file	/LOGFILE or /LOG
Obtain product version information and help	/VERSION or /VER /HELP or /H

NMSYM Command Syntax

Use the following syntax for NMSYM.EXE:

```
NMSYM [option(s)] <module-name>
```

Where:

- Options are specified by using a slash (/) followed by the option name.
- `<module-name>` is the name of the module you want to translate or load.

The following example shows a valid command line:

```
NMSYM /TRANSLATE C:\MYPROJ\MYPROJECT.EXE
```

Using Option and File-list Specifiers

Many options include additional option and file-list specifiers. Option specifiers modify an aspect of the option and file-list specifiers specify operations on a group of files.

The syntax for option specifiers is as follows:

```
/option:<option-specifier>[,<option-specifier>]
```

The option is followed by a colon (:), which, in turn, is followed by a comma delimited list of specifiers. The following example uses the /TRANSLATE option with the SOURCE and PACKAGE specifiers to instruct NMSYM to translate source and symbols, then package the source files with the NMS symbol table:

```
/TRANSLATE:SOURCE,PACKAGE
```

The syntax for file-list specifiers is as follows:

```
/option:<filename|pathname>[;<filename|pathname>]
```

The following example uses the /SOURCE option with three path-list specifiers. NMSYM searches the paths in the path-list specifiers to locate source code files during translation and loading:

```
/SOURCE:c:\myproj\i386;c:\myproj\include;c:\msdev\include;
```

The option and file list specifiers are listed here and described on the pages that follow.

- /TRANSLATE
- /LOAD
- /OUTPUT
- /SOURCE
- /ARGS
- /PROPMT
- /SYM(LOAD)
- /EXP(ORTS)
- /UNLOAD
- /LOG(FILE)
- /VER(SION)

Using NMSYM to Translate Symbol Information

The primary purpose of NMSYM is to take compiler generated debug information for a module and translate it into the NM32 symbol format, then place that information into a .NMS symbol file. To accomplish this, use the following options and parameters on the NMSYM command line:

- Use the /TRANSLATE option to specify the type of symbol information you want to generate.
- Use the /SOURCE option to specify the source paths that NMSYM searches to locate source code files.
- If you want to specify an alternate filename for the .NMS file, use the /OUTPUT option.
- Specify the name of the module that you want to translate.

```
NMSYM /TRANSLATE C:\MYPROJ\MYPROJECT.EXE
```

The following paragraphs describe the translation options. Use these options to translate symbol information for an individual module.

/TRANSLATE Option

The **/TRANSLATE :<translation-specifier-list>** option lets you specify the type of symbol information you wish to produce, as well as whether source code is packaged with the symbol file. Other options include the ability to force the translation to occur, even if the symbol file is already up to date.

The /TRANSLATE option takes a variety of option specifiers, including symbol-information, source code packaging, and a miscellaneous specifier, ALWAYS. The following sections describe these specifiers.

Symbol-information Specifiers

The following table lists optional symbol-information specifiers that determine what symbol information is translated. Use one symbol-information specifier only. If you do not use a specifier, NMSYM defaults to SOURCE.

Table 4-4. Optional Symbol-information Specifiers

Symbol-information Specifier	Description
PUBLICS	Only public (global) symbols are included. Static functions and variables are excluded. This option is similar to the symbol information that can be found in a MAP file. It produces the smallest symbol tables.
TYPEINFO	Only the type information is included. Symbol information is excluded. Use this option when you produce advanced type information without the original source code or debug information.
SYMBOLS	Includes all symbol and type information. Source code and line-number information is excluded. This specifier produces smaller symbol tables.
SOURCE	This is the default translation type. All symbol, type, and source code information is included.

Note: Source code information does not include the source files themselves. It is information about the source code files, such as their names and line-number information.

Source Code Packaging Specifiers

Optional source code packaging specifiers determine whether or not NMSYM attaches source code to the .NMS symbol file. By default, NMSYM does the following:

- Packages the source code with the .NMS symbol files for device driver modules, because they load before the operating system fully initializes.
- Does not package the source code for applications that run after the operating system boots.

Use the following source code packaging specifiers to override these defaults:

Table 4-5. Optional Source Code Packaging Specifiers

Source Code Packaging Specifier	Description
PACKAGE	Include source files with the .NMS symbol file.
NOPACKAGE	Do not include source files with the .NMS symbol file.

Note: If you package the source code with the .NMS symbol file, your code is available to anyone who accesses the symbol table.

ALWAYS Specifier

By default, NMSYM does not translate the symbol information if it is current. Use the ALWAYS specifier to force NMSYM to translate the symbol information regardless of its status.

Examples: Using the /TRANSLATE Option

The following example specifies a module name without the /TRANSLATON option. Thus, the translation is performed using the default options for the module type.

```
NMSYM myproj.exe
```

Note: For Win32 applications or DLLs, the default is /TRANSLATE:SOURCE,NOPACKAGE.
For driver modules, the default is /TRANSLATE:SOURCE:PACKAGE.

The following example translates symbol information for a VxD. It uses the SYMBOLS specifier to exclude information related to the source code and the /NOPACKAGE specifier to prevent NMSYM from packaging source code.

```
NMSYM /TRANSLATE:SYMBOLS,NOPACKAGE c:\myvxd.vxd
```

The following example uses the default options for the module type and uses the /ALWAYS specifier to force NMSYM to translate the symbol information into a .NMS symbol file.

```
NMSYM /TRANSLATE:ALWAYS myproj.exe
```

/SOURCE Option

Use the **/SOURCE :<path-list>** option to specify the source paths that NMSYM should search to locate source code files. At translation time (PACKAGE only) or module load time (/LOAD or /SYMLOAD), NMSYM will attempt to locate all the source files specified within the NMS symbol table. It will do a default search along this path to locate them.

The path-list specifier is one or more paths concatenated together. Each path is separated from the previous path by a semi-colon ';'. The /SOURCE option may be specified one or more times on a single command-line. The order of the /SOURCE statements, and the order of the paths within the path-list determines the search order.

Examples: Using the /SOURCE Option

The following example specifies two paths for locating source files.

```
NMSYM /TRANSLATE:PACKAGE /
SOURCE:c:\myproj\i386;c:\myproj\include; myproj.exe
```

The following example specifies two sets of source paths.

```
NMSYM /TRANS:PACKAGE /SOURCE:c:\myproj\i386;c:\myproj\include;
/SOURCE:c:\msdev\include; myproj.exe
```

The following example specifies the base project source path and uses the DOS replacement operator % to take the path for include files from the standard environment variable INCLUDE=. The path-list expands to include c:\myproj\i386 and every path listed in the INCLUDE= environment variable.

```
NMSYM /TRANS:PACKAGE /SOURCE:c:\myproj\i386;%INCLUDE%
myproj.exe
```

Note: In the event that a source code file cannot be found, the /PROMPT switch determines whether the file will be skipped, or if you will be asked to help locate the file.

/OUTPUT Option

NMSYM derives the output file name for the NMS symbol table by taking the root module name and appending the standard file extension for NM32 symbol tables, NMS. Secondly, the path for the NMS file is also the same as path to the module being translated. If you need to change the default name or location of the NM32 symbol table file, then use the **/OUTPUT:<filename>** option to specify the location and name. If you specify a name, but do not specify a path, the path to the module will be used.

Examples: Using the /OUTPUT Option

In the following example, the path of the NMS file is changed to a common directory for NM32 symbol tables.

```
NMSYM /OUTPUT:c:\NTICE\SYMBOLS\myproj.nms
c:\myproj\myproject.exee
```

/PROMPT Option

NMSYM is a command-line utility designed to allow tasks of symbol translation and loading to be automated. As such, you probably do not desire to be prompted for missing source files, but there are cases where it might be useful. Use the /PROMPT option to specify that NMSYM should ask for your help in locating source code files when you use the / TRANSLATE:PACKAGE, /LOAD, or /SYMLOAD options.

Using NMSYM to Load a Module and Symbol Information

Like translation, the /LOAD functionality of NMSYM is designed to work on a specific module that is specified using the module-name parameter. This module is one which will be translated and loaded. If you do not need to translate or load and execute a module, then the /SYMLOAD option may be a better choice.

The following example shows how to use NMSYM to translate, load, and execute a module:

```
NMSYM /TRANS:PACKAGE /LOAD:EXECUTE myproj.exe
```

The next example shows the alternate functionality of loading a group of pre-translated symbol files using the /SYMLOAD option:

```
NMSYM /SYMLOAD:NTDLL.DLL;NTOSKRNL.NMS;MYPROJ.EXE
```

In the preceding example, three symbol tables will be loaded, but translation will not be performed, even if the modules corresponding NMS is out of date. Also, MYPROJ.EXE will not be executed so that it can be debugged.

/LOAD Option

The **/LOAD: <load-specifier-list>** option allows you to load a modules NM32 symbol table into SoftICE, and optionally, execute the module so it can be debugged.

You can use the following specifiers with the /LOAD option.

Load-Type Specifiers

One of the following options may be selected to determine how the module and its symbol information will be loaded. The default specifier is dependent on the type of the module, and for executables is EXECUTE. For non-executable module types, the default is SYMBOLS.

Table 4-6. Load-Type Specifiers

Load Type Specifiers	Definition
SYMBOLS	Only symbol information for the module will be loaded. You may set breakpoints using this symbol information, and when the module is loaded the breakpoints will trigger as appropriate.
EXECUTE	Symbol information is loaded and the executable is loaded as a process so that it may be debugged.

Break-On-Load Specifiers

To enable or disable having a breakpoint set at the modules entry-point, use one of the following specifiers.

Table 4-7. Break-On-Load Specifiers

Break on Load Specifiers	Definition
BREAK	Set a breakpoint on the module's entry-point (WinMain, DllMain, or DriverEntry).
NOBREAK	Do not set a breakpoint on the modules entry-point.

The ability to explicitly turn module entry breakpoints on or off is provided because the default setting of this option is dependent upon the type of the module. For applications the BREAK option is the default. For other module types NOBREAK is the default.

NOSOURCE Specifier

NOSOURCE prohibits the load of source code files, even if the symbol table includes a source package or line-number information.

Examples: Using the /LOAD Option

In the following example NMSYM will load (and by default) execute the module MYPROJ.EXE. If the symbol table is not current, then a default translation for the module type will be performed:

```
NMSYM /LOAD MYPROJ.EXE
```

The next example specifies that the program is to be executed, but a breakpoint should not be set on the program entry-point. Once again, if a translation needs to be performed, it will be the default translation for the module type.

```
NMSYM /LOAD:NOBREAK MYPROJ.EXE
```

The next example specifies that only symbol information should be loaded, and explicitly specifies the PUBLICS translation type:

```
NMSYM /TRANS:PUBLIC /LOAD:SYMBOLS MYPROJ.DLL
```

/ARGS Option

The **/ARGS:<program-argumens>** option is used to specify the program arguments that will be passed to an executable module. This option is only useful when used with the /LOAD:EXECUTE option.

The string *program-arguments* defines the program arguments. If it contains white-space, then you should surround the entire option in double quotes (").

Examples: Using the /ARGS Option

In the following example, the MYPROJ.EXE module is going to be loaded for debugging, and the arguments passed to the application are TEST.RTF.

```
NMSYM /LOAD:EXECUTE /ARGS:test.rtf myproj.exe
```

In the next example, the command-line is a bit more complicated, so we are going to wrap the entire option in double-quotes ("):

```
NMSYM /LOAD:EXECUTE "/ARGS:/PRINT /NOLOGO test.rtf" myproj.exe
```

Using the double quotes around the option prevents NMSYM from becoming confused by the white-space that appears within the program arguments: /PRINT^/NOLOGO^test.rtf.

Using NMSYM to Load Symbol Tables or Exports

In addition to the translation and loading functions, NMSYM also supplies options that allow for batch loading and unloading of both symbol tables and exports. This is extremely useful for loading an "environment" or related set of symbol table files. For example, if you start SoftICE manually you can use NMSYM to give you the equivalent functionality of the SoftICE Initialization Settings for Symbols and Exports.

For example, you could use a batch file similar to the following to control which symbol tables are loaded. The batch file takes one optional parameter that determines whether the files to be loaded are for driver or application debugging (application is the default). In both cases we are loading exports for the standard Windows modules.

```
net start ntice
echo off

if "%1" == "D" goto dodriver
if "%1" == "d" goto dodriver

REM *** These are for debugging applications *** set
SYMBOLS=ntdll.dll;shell32.dll;ole32.dll;win32k.sys goto doload

:dodriver REM *** These are for debugging drivers *** set
SYMBOLS=hal.dll;ntoskrnl.exe;

:doload

NMSYM /SYMLOAD:%SYMBOLS% /
EXPORTS:kernel32.exe;user32.exe;gdi32.exe
```

Another benefit of using NMSYM is that it does not require explicit path information to find NMS files or modules. If you do not specify a path, and the specified module or NMS file cannot be found within the current directory or the symbol table cache, then a search will be executed along the current path.

/SYMLOAD Option

The **/SYMLOAD: <module-list>** option is used to load one or more symbol tables into SoftICE. The symbol tables must have been previously translated since this function does not perform translation.

The module-list specifier may specify NMS files or their associated modules, with or without explicit paths to the files. If you do not specify an explicit path for the module, then NMSYM will attempt to find the file in the current directory, in the symbol table cache, or on the system path. If you specify an absolute or relative path for the module then no search will be performed.

Examples: Using the /SYMLOAD Option

The following example uses the /SYMLOAD option to load the symbol tables typically used for debugging OLE programs. It does not specify any paths, so a search will be performed (as necessary).

```
    NMSYM /SYMLOAD:ole32.dll;oleaut32.dll;olecli32.dll
```

/EXPORTS Option

The **/EXPORTS: <module-list>** option is used to load exports for one or more modules into SoftICE. Exports are lightweight symbol information for API's exported from a module (usually a DLL, but EXEs can also contain exports).

The module-list specifier may specify modules with or without explicit paths. If you do not specify an explicit path for the module, then NMSYM will attempt to find the file in the current directory, in the system directory, or on the system path. If you specify a absolute or relative path for the module then no search will be performed.

Examples: Using the /EXPORTS Option

The following example uses the /EXPORTS option to load the exports for modules typically used when debugging OLE programs. It does not specify any paths, so a search will be performed, as necessary.

```
    NMSYM /EXPORTS:ole32.dll;oleaut32.dll;olecli32.dll
```

Using NMSYM to Unload Symbol Information

NMSYM provides the /UNLOAD option so that you can programmatically remove symbol information for a related set of symbol tables and/or exports. This can be used to save memory used by unneeded symbol tables.

/UNLOAD Option

The **/UNLOAD: <module-list>** option may specify either symbol tables or export table names. The name of a symbol table or export table is derived from the root module-name, without path or extension information. For flexibility and to support future table naming conventions you should specify any path or extension information that is relevant to uniquely distinguish the table.

Examples: Using the /UNLOAD Option

The following example is the reverse of the examples provided in the /SYMLOAD and /EXPORTS sections:

```
NMSYM /UNLOAD:ole32.dll;oleaut32.dll;olecli32.dll
```

SoftICE will find the table that corresponds to the specified module name and remove the table (if possible) and free any memory in use by that symbol table.

Note: SoftICE attempts to unload a symbol table by default. If the specified symbol table does not exist then SoftICE attempts to unload an export table with that name.

Using NMSYM to Save History Logs

NMSYM provides the ability to save the SoftICE history buffer to a file using the /LOGFILE option. This operation is equivalent to the Symbol Loader 'Save SoftICE History As..." option. NMSYM supports the ability to append to an existing file using the APPEND specifier.

/LOGFILE Option

The **/LOGFILE: <filename>[,logfile-specifier-list]** option is the path and filename of the file the history buffer will be written to. If no path is specified the current directory will be assumed.

LogFile Specifiers

APPEND lets you append the current contents of the History buffer to an existing file. The default is to overwrite the file.

Examples: Using the /LOGFILE Option

The following example will create/overwrite the MYPROJ.LOG file with the current contents of the SoftICE history buffer:

```
NMSYM /LOGFILE:myproj.log
```

The next example will create/append the current contents of the SoftICE history buffer to the file MYPROJ.LOG:

```
NMSYM /LOGFILE:myproj.log,APPEND
```

Caution: NMSYM will not ask you if you want to overwrite an existing file. It will automatically do so.

Getting Information about NMSYM

To get information about NMSYM, use the /VERSION and /HELP options.

/VERSION Option

Use the **/VERSION** option to obtain version information for NMSYM, SoftICE, as well as the translator and symbol engine version numbers. For SoftICE, Loader32 and NMSYM to work together correctly, these versions must be compatible. Each product negotiates and verifies version numbers with the other products to insure that each can work together.

/HELP Option

Use the **/HELP** option to obtain command-line syntax, options, specifiers and option/specifier syntax.

Chapter 5

Navigating Through SoftICE

Introduction

This chapter describes how to use the SoftICE screen and its windows. The SoftICE windows are described in order of importance.

If you are new to SoftICE, read this chapter thoroughly, then use it as a reference.

Universal Video Driver

SoftICE uses a Universal Video Driver (UVD) to display on the user's desktop. The UVD allows SoftICE to draw directly in linear frame memory. To use the UVD, SoftICE requires that the video hardware and video driver support Direct Draw. Table 5-1 describes the commands and key sequences you can use to move, size, and customize the SoftICE display window. Figure 5-12 displays the keypad functions that reposition the SoftICE window when you hold down the **<Ctrl>-<Alt>** keys.

Table 5-1 SoftICE Commands and Keystrokes

Command/Keystrokes	Result
LINES n	Where n is 25-128, selects the number of lines in the SoftICE window.
WIDTH n	Where n is 80-160, selects the number of columns in the SoftICE window.
SET FONT n	Where n is 1, 2, or 3, selects a font.
SET ORIGIN x y	Where x and y are pixel coordinates, locates the window
SET FORCEPALETTE [ON|OFF]	When On, SoftICE will prevent the system colors (palette indices 0-7 and 248-255) from being changed in 8-bpp mode. This ensures that the SoftICE display can always be seen. This is OFF by default.
SET MAXIMIZE [ON | OFF]	When On, SoftICE resizes its window to the maximum possible size, based on font, number of lines, and video memory size. When Off, changing a display format parameter (font, number of lines, etc.) will not cause SoftICE to resize its window.
SET MONITOR n	Where n is 0 to the number of UVD-enabled video cards installed. Used without supplying an n-value, this command returns the list of video drivers that SoftICE is aware of, and tells you which one is active. Passing in an n-value tells SoftICE to switch the output to the specified monitor. This command can only be used for UVD displays, not VGA or Mono.
Control-Alt- cursor key	Moves the SoftICE window by a character increment.
Control-Alt-Home	Resets the SoftICE window position to (0, 0)
Control-Alt-End	Moves the SoftICE window to the bottom left.
Control-Alt-Page Up	Moves the SoftICE window to the top right.

Table 5-1 SoftICE Commands and Keystrokes (Continued)

Command/Keystrokes	Result
Control-Alt-Page Down	Moves the SoftICE window to the bottom right.
Control-Alt-5	Moves the SoftICE window to the center of the screen.
Control-L	Refreshes the SoftICE display. Useful in the rare case where the part of the display used by SoftICE is overlapped by a bitblt operation that was running when SoftICE popped up.
Control-C	Centers the SoftICE display window.

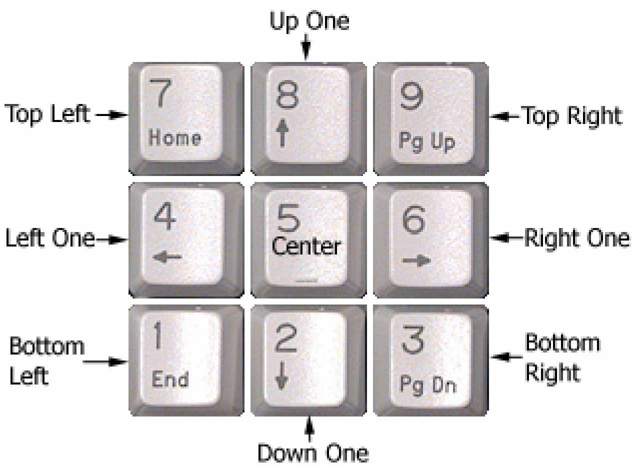


Figure 5-12 SoftICE Repositioning Keypad Functions

Setting the Video Memory Size

When using the UVD, SoftICE must save the existing contents of the frame buffer so it can be restored later. The amount of memory required depends on the video mode, the number of lines used by SoftICE. In any case, the amount of memory required cannot exceed the amount of memory on your video card. By default, SoftICE reserves 2MB, but you can modify this using the Symbol Loader (go to **Edit -> SoftICE Initialization Settings** and change the “Video memory size” setting).

Popping Up the SoftICE Screen

Once loaded, the SoftICE screen will automatically pop up in the following situations:

- When SoftICE loads. By default, the SoftICE initialization string contains the X (Exit) command, so it immediately closes after opening. Refer to *Modifying SoftICE Initialization Settings* on page 191.
- When you press Ctrl-D. This hot-key sequence toggles the SoftICE screen on and off.
- When breakpoint conditions are met.
- When SoftICE traps a system fault.
- When a system crash in the Windows NT family results in "Blue Screen" Mode.

Tip Use the ALTKEY command to change the SoftICE default pop-up key (Ctrl-D).

When the SoftICE screen pops up, all background activity on your computer comes to a halt, all interrupts are disabled, and SoftICE performs all video and keyboard I/O by accessing the hardware directly.

Disabling SoftICE at Startup

If SoftICE was installed as a boot or system driver with the Windows NT family, you can disable it at startup. Press the Escape key when the following message appears at the bottom of the display:

```
Press Esc to Modify DriverStudio Startup Environment
```

Stopping SoftICE at Startup

If SoftICE was installed as a boot or system driver with the Windows NT family, you can stop it at startup. Press the **ESC** key when the DriverStudio environment message appears, check the **Stop SoftICE On Load** check-box.

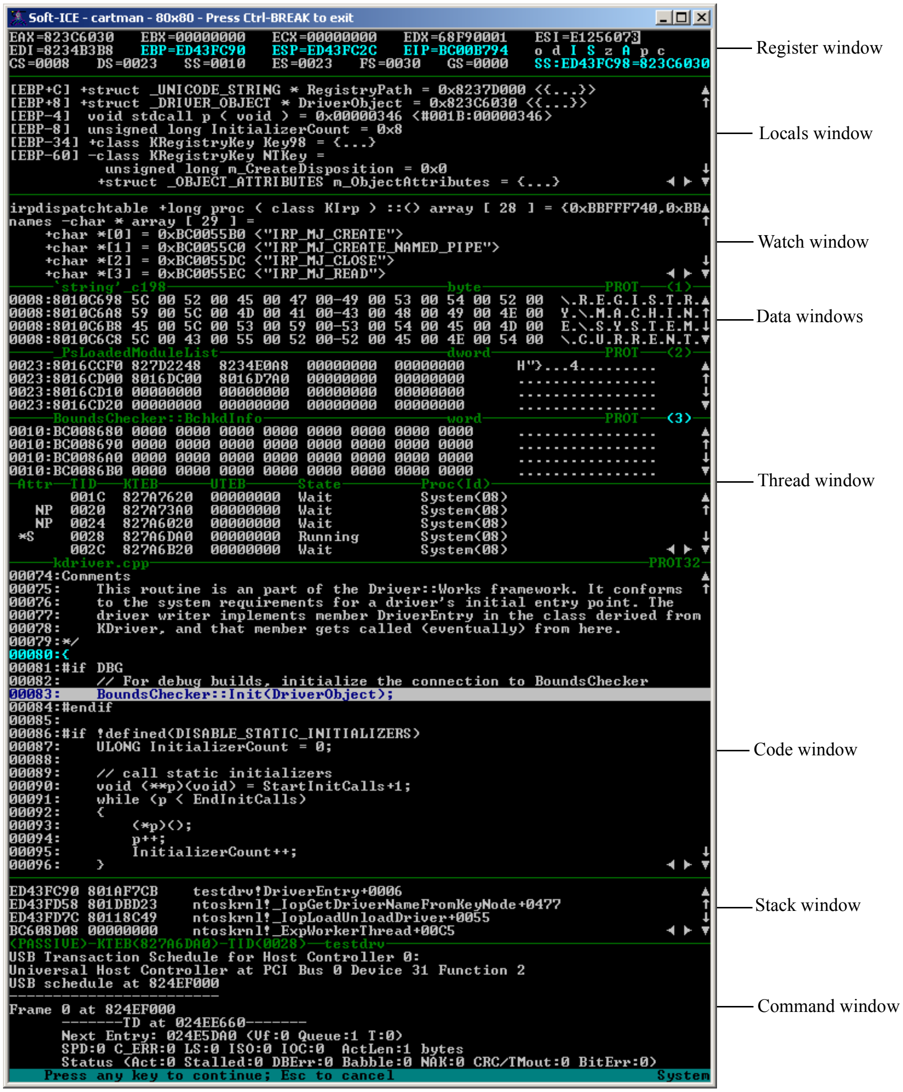
Using the SoftICE Screen

The SoftICE screen serves as the central location for debugging your code. It provides several windows and a Help line to view and control various aspects of your debugging session. These windows are listed below:

Table 5-2. SoftICE Windows

SoftICE Windows	Use
Command window	Enter user commands and display information.
Code window	Display unassembled instructions and/or source code.
Locals window	Display locals for the current stack frame.
Watch window	Display the value of the variables watched with the WATCH command.
Register window	Display and edit the current state of the registers and flags.
Data window	Display and edit memory.
Stack Window	Display call stack for DOS programs, Windows tasks, and 32-bit code
Thread Window	Display information on threads for a given process
PIII Register Window	Display Pentium III registers
FPU Stack window	Display the current state of the FPU (Floating Point Unit) stack /MMX registers.
Help line	Provide information about SoftICE commands.

By default, SoftICE displays the Help line and the Command, Code, and Locals windows. You can open and close the remaining windows as necessary. Figure 5-1 illustrates a typical SoftICE window.



```
EAX=823C6030   EBX=00000000   ECX=00000000   EDX=68F90001   ESI=E1256073
EDI=8234B3B8   EBP=ED43FC90   ESP=ED43FC2C   EIP=BC00B794   o d I S z A p c
CS=0008   DS=0023   SS=0010   ES=0023   FS=0030   GS=0000   SS:ED43FC98=823C6030

[EBP+C] +struct _UNICODE_STRING * RegistryPath = 0x8237D000 <{...}>
[EBP+8] +struct _DRIVER_OBJECT * DriverObject = 0x823C6030 <{...}>
[EBP-4]  void stdcall p ( void ) = 0x00000346 <#001B:00000346>
[EBP-8]  unsigned long InitializerCount = 0x8
[EBP-34] +class KRegistryKey Key98 = {...}
[EBP-60] -class KRegistryKey NTKey =
           unsigned long m_CreateDisposition = 0x0
          +struct _OBJECT_ATTRIBUTES m_ObjectAttributes = {...}

irpdispatchtable +long proc ( class KIrp ) ::() array [ 28 ] = {0xBBFFF740,0xBB
names -char * array [ 29 ] =
    +char *[0] = 0xBC0055B0 <"IRP_MJ_CREATE">
    +char *[1] = 0xBC0055C0 <"IRP_MJ_CREATE_NAMED_PIPE">
    +char *[2] = 0xBC0055DC <"IRP_MJ_CLOSE">
    +char *[3] = 0xBC0055EC <"IRP_MJ_READ">
-----`string'_c198--------------------------------byte-----------PROT---(1)--
0008:8010C698 5C 00 52 00 45 00 47 00-49 00 53 00 54 00 52 00  \.R.E.G.I.S.T.R.
0008:8010C6A8 59 00 5C 00 4D 00 41 00-43 00 48 00 49 00 4E 00  Y.\.M.A.C.H.I.N.
0008:8010C6B8 45 00 5C 00 53 00 59 00-53 00 54 00 45 00 4D 00  E.\.S.Y.S.T.E.M.
0008:8010C6C8 5C 00 43 00 55 00 52 00-52 00 45 00 4E 00 54 00  \.C.U.R.R.E.N.T.
-----_PsLoadedModuleList--------------------------dword----------PROT---(2)--
0023:8016CCF0 827D2248  8234E0A8  00000000  00000000     H"}...4.........
0023:8016CD00 8016DC00  8016D7A0  00000000  00000000     ................
0023:8016CD10 00000000  00000000  00000000  00000000     ................
0023:8016CD20 00000000  00000000  00000000  00000000     ................
-----BoundsChecker::BchkdInfo---------------------word-----------PROT---(3)--
0010:BC008680 0000 0000 0000 0000 0000 0000 0000 0000     ................
0010:BC008690 0000 0000 0000 0000 0000 0000 0000 0000     ................
0010:BC0086A0 0000 0000 0000 0000 0000 0000 0000 0000     ................
0010:BC0086B0 0000 0000 0000 0000 0000 0000 0000 0000     ................
-Attr--TID---KTEB-------UTEB-------State-------------Proc(Id)----------------
       001C  827A7620  00000000  Wait              System(08)
   NP  0020  827A73A0  00000000  Wait              System(08)
   NP  0024  827A6020  00000000  Wait              System(08)
 *S    0028  827A6DA0  00000000  Running           System(08)
       002C  827A6B20  00000000  Wait              System(08)
-----kdriver.cpp------------------------------------------------------PROT32-
00074:Comments
00075:    This routine is an part of the Driver::Works framework. It conforms
00076:    to the system requirements for a driver's initial entry point. The
00077:    driver writer implements member DriverEntry in the class derived from
00078:    KDriver, and that member gets called (eventually) from here.
00079:*/
00080:{
00081:#if DBG
00082:    // For debug builds, initialize the connection to BoundsChecker
00083:    BoundsChecker::Init(DriverObject);
00084:#endif
00085:
00086:#if !defined(DISABLE_STATIC_INITIALIZERS)
00087:    ULONG InitializerCount = 0;
00088:
00089:    // call static initializers
00090:    void (**p)(void) = StartInitCalls+1;
00091:    while (p < EndInitCalls)
00092:    {
00093:        (*p)();
00094:        p++;
00095:        InitializerCount++;
00096:    }

ED43FC90 801AF7CB    testdrv!DriverEntry+0006
ED43FD58 801DBD23    ntoskrnl!_IopGetDriverNameFromKeyNode+0477
ED43FD7C 80118C49    ntoskrnl!_IopLoadUnloadDriver+0055
BC608D08 00000000    ntoskrnl!_ExpWorkerThread+00C5
(PASSIVE)-KTEB(827A6DA0)-TID(0028)--testdrv----------------------------------
USB Transaction Schedule for Host Controller 0:
Universal Host Controller at PCI Bus 0 Device 31 Function 2
USB schedule at 824EF000
------------------------
Frame 0 at 824EF000
      -------TD at 024EE660-------
      Next Entry: 024E5DA0 (Vf:0 Queue:1 T:0)
      SPD:0 C_ERR:0 LS:0 ISO:0 IOC:0  ActLen:1 bytes
      Status (Act:0 Stalled:0 DBErr:0 Babble:0 NAK:0 CRC/TMout:0 BitErr:0)
    Press any key to continue; Esc to cancel                           System
```

Figure 5-1. Typical SoftICE Window

Resizing the SoftICE Screen

By default, the SoftICE screen uses a total of 25 lines to display information in the various windows. If you are using VGA or Text Mode, you can use the LINES command to switch the total lines for the SoftICE screen to 43, 50, or 60 lines instead of the standard 25 lines. If you are using UVD you can set the total lines to any value from 25 to 100. Monochrome screens limit you to 25 lines. The WIDTH command allows you to set the number of display columns between 80 and 160.

```
LINES 60
WIDTH 80
```

The SoftICE display can also be moved on the Windows desktop. Use the Ctrl-Alt and cursor keys to move the SoftICE display. Use the Ctrl-Alt-Home keys to return the display to the 0,0 position, or the Ctrl-Alt-C keys to center the display.

Controlling SoftICE Windows

You can do the following to the SoftICE windows:

- Open and close all the windows except the Command window.
- Resize the Code, Data, Locals, Stack, Thread, and Watch windows.
- Scroll the Code, Command, Data, Locals, Stack, Thread, and Watch windows.

SoftICE provides two methods for controlling these windows: mouse and keyboard input.

Opening and Closing Windows

To open a SoftICE window, use the appropriate command listed in the following table. If you specify -o, the window is forced open. If you specify -c, and the window is already open, it is forced closed. These options are useful for macros, as they allow you to set the screen layout to a known state.

To use your mouse to close a window, select the line below the window you want to close and drag it up past the top line of the window.

Table 5-3. SoftICE Window Commands

Command	Window
WC	Code
WD.#	Data Where # is a number 0 through 3 to open that specified data window. Use without 0-3 extension to switch to or open the next sequential Data window.
WF	FPU Stack
WL	Locals
WR	Register
WW	Watch
WS	Stack
WT	Thread
WX	Pentium III Register

Resizing Windows

To resize a window, drag the line at the bottom of the window you want to resize either up or down. You can also use the same commands that you use for opening and closing windows to resize the windows. Simply type the command followed by a decimal number that represents the number of lines you want to display in the window.

```
WD 7
```

Note that the number of lines in the Command window automatically increases or decreases when you resize a window. Although you cannot explicitly resize the Command widow, changing the size of other windows in your display automatically resizes the Command window.

You can also resize by relative amounts by using a "+" or "-" sign. For example, `WD +7` will enlarge the data window by 7 lines.

Forcing Windows Open and Closed

To enhance SoftICE Macro writing, you can set the state of a window to a known value, open or closed, via the `-o` and `-c` flags.

Moving the Cursor Among Windows

The cursor is located in the Command window by default. To move the cursor to another window, click the mouse in the window where you want to place the cursor. If the cursor is in the Command or Code windows, you can use one of the Alt key combinations in the following table to move the cursor. Repeat the same Alt key combination to return the cursor to the Command or Code window.

Table 5-4. SoftICE Window Alt Key Combinations

Window	Alt Key Combination
Code	Alt-C
Data	Alt-D
FPU Stack	Cannot move the cursor to the FPU Stack window.
Locals	Alt-L
Register	Alt-R
Stack	Alt-S
Thread	Alt-T
Watch	Alt-W

Scrolling Windows

You can scroll the Code, Command, Data, Locals, Stack, Thread, and Watch windows. The FPU Stack and Register windows are not scrollable, because they are limited to four and three lines respectively.

SoftICE provides for three window scrolling methods: key sequences, mouse scroll arrows, and use of the “wheel” mouse. The following table describes how to use key sequences and scroll arrows to scroll windows.

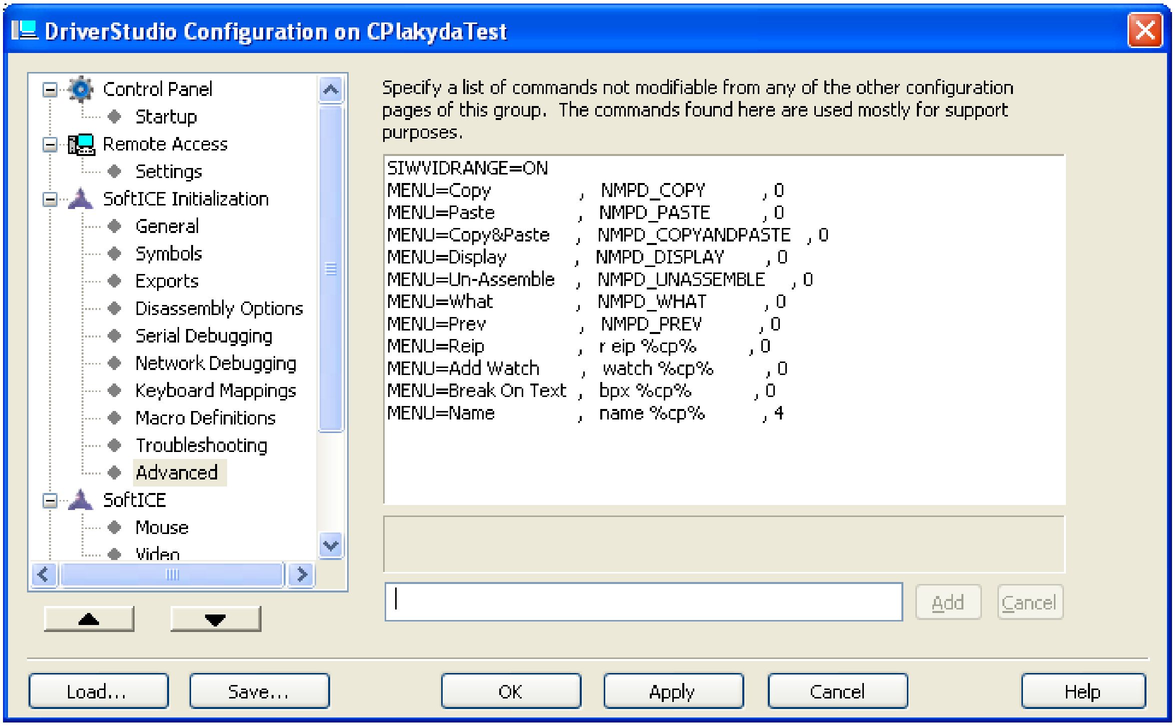
Note: The key sequences for some windows vary. For example, some windows do not let you jump to the first or last lines of the file. See the sections that describe the individual windows for specific information about scrolling particular windows.

Table 5-5. SoftICE Window Scrolling Methods

Scroll Direction and Distance	Key Sequence	Mouse Action
Scroll the window to the previous page.	PageUp	Click the innermost up scroll arrow
Scroll the window to the next page.	PageDown	Click the innermost down scroll arrow
Scroll the window to the previous line.	UpArrow	Click the outermost up scroll arrow
Scroll the window to the next line.	DownArrow	Click the outermost down scroll arrow
Jump to the first line of the source file.	Home	Not supported.
Jump to the last line of the source file.	End	Not supported.
Scroll the window left one character.	LeftArrow	Click the left scroll arrow.
Scroll the window right one character.	RightArrow	Click the right scroll arrow.

User-definable Pop-up Menus

SoftICE allows you to customize the content of the pop-up menus that appear when you right-click with the mouse. The menu entries are defined in `winice.dat`. To access the editor and customize the pop-up menus, select **Advanced** from the SoftICE Initialization menu on the Configuration screen.



Specify a list of commands not modifiable from any of the other configuration pages of this group. The commands found here are used mostly for support purposes.

```
SIWVIDRANGE=ON
MENU=Copy           ,  NMPD_COPY        , 0
MENU=Paste          ,  NMPD_PASTE       , 0
MENU=Copy&Paste     ,  NMPD_COPYANDPASTE  , 0
MENU=Display        ,  NMPD_DISPLAY     , 0
MENU=Un-Assemble    ,  NMPD_UNASSEMBLE    , 0
MENU=What           ,  NMPD_WHAT        , 0
MENU=Prev           ,  NMPD_PREV        , 0
MENU=Reip           ,  r eip %cp%       , 0
MENU=Add Watch      ,  watch %cp%       , 0
MENU=Break On Text  ,  bpx %cp%         , 0
MENU=Name           ,  name %cp%        , 4
```

Figure 5-2. Pop-up Menu Editor

The format of entries in `winice.dat` is as follows:

```
MENU=Description, Command Field, [Modifier]
```

- *Description* is the text that will appear on the menu. It can contain any valid character, can have spaces, and must have a maximum length of 13 characters. All trailing spaces are removed.
- *Command Field* is the SoftICE command, macro, expression evaluator command, or predefined command to be executed upon selection of that menu item. You must use full command names and may not use shortcuts. In addition you can add a special *Modifier* flag, %cp%, which will copy the data or text that is underneath the cursor and paste it into the string at that position.

If you have a line of the screen that reads `80001000 ntoskrnl!kitrap0E` and you have defined a menu item as `what %cp%`, you can place the mouse on `80001000` and select that menu item to submit the command **`what 80001000`** to SoftICE.

In addition, several predefined commands have been provided for backwards compatibility with the menus in earlier versions of SoftICE.

The predefined commands are as follows:

- NMPD_COPY — Copies the text under the cursor into a paste buffer.
- NMPD_PASTE — Pastes the text from the paste buffer into the active location.
- NMPD_COPYANDPASTE — Copies the text from under the cursor and pastes it into the active location.
- NMPD_DISPLAY — Displays the address under the cursor in the Data window.
- NMPD_UNASSEMBLE — Unassembles the address under the cursor.
- NMPD_WHAT — Issues the WHAT command for the item under the cursor.
- NMPD_PREV — Causes the Data window to step back to the previous address it displayed. This is useful for walking pointer chains.

The following predefined commands are not on the default menu set, but can be used in your own custom menus.

- NMPD_DATAREALIGN — Causes the Data window to be realigned such that the address currently under the mouse (in the Data window) becomes the new base address of the Data window.

Inline Editing

SoftICE is able to do inline editing of variables displayed in either the Locals Window (WL) or the Watch Window (WW).

Usage

- Navigate to the variable you wish to edit in either the Locals Window or the Watch Window.
- Use the hotkey sequence, **Alt-E**, to launch Inline Editing.
- Edit your data.
- Press either **Enter** to store your data, or **Esc** to abort your changes.

Navigation Keys

The following keys are available for the Inline Editing feature:.

Table 5-6. Inline Editing Commands

Command	Action
Enter	Stores your modifications.
Esc	Aborts any changes.
Left/Right Arrow	Changes your position within the edit field; additionally, pressing either of these keys puts you into Overtype Mode.
Home	Moves to start of field; additionally puts you into Overtype Mode.

Notes

All input is done in hex.

When you enter Inline Editing, the information to the right of the field being edited will be overwritten until you complete your edit. This is the intended functionality.

If you start typing in the edit field, the entire entry will be erased.

You will enter Overtype Mode if you press the left/right arrow, Home, or End keys .

Copying and Pasting Data

If you have a mouse, you can copy and paste data among windows. This is useful for copying addresses and data into expressions. To copy and paste data, do the following:

1 Select the data you want to copy.

2 Press the right mouse button to display the following list of available commands.

3 Click the left mouse button to select the command (Copy, Copy and Paste, or Paste) you want to use. The following table describes these commands.

Table 5-7. Copy and Paste Commands

Command	Description
Copy	Copies the selected item to the Copy-and-Paste buffer.
Copy and Paste	Copies the selected item and pastes it to the location of the cursor.
Paste	Pastes the contents of the Copy-and-Paste buffer to the location of the cursor.

Entering Commands from the Mouse

SoftICE provides shortcuts for entering the D, U, and WHAT commands with your mouse. (Refer to the *SoftICE Command Reference* for more information about these commands.)

To use your mouse to enter one of these commands, do the following:

1. Select the data you want the command to act upon.

   For example, select an expression to identify.

2. Click the right mouse button to display the list of available commands.

3. Click the left mouse button to select the command you want to use. The following table describes these commands.

Table 5-8. SoftICE Mouse Commands

Mouse Command	SoftICE Command Equivalent	Description
Display	D	Displays the memory contents at the specified address.
Un-Assemble	U	Displays either source code or unassembled code at the specified address.
What	WHAT	Determines if a name or expression is a known type.
Previous	N/A	Undoes the previous mouse command.

Obtaining Help

SoftICE provides you with two methods for obtaining help online while debugging your module: the Help line and H command.

Using the Help Line

The bottom line of the screen always contains the Help line. This line updates as you type characters on the command line. The Help line provides several different types of information, as follows:

- When the characters you type do not specify a complete command, the Help line displays all the valid commands that start with the characters you typed.
- When the characters you type match a command, the Help line displays a description of the command.
- If you enter a space after a command, the Help line displays the syntax for that command.
- If you are editing in the Register or Data windows, the Help line contains the valid editing keys for that window.

Using the H Command

Use the H command to provide general help on all the SoftICE commands or detailed help on a specific command. To display a brief description of all the SoftICE commands by function, enter the H command with no parameters.

To display detailed help on a specific command, type the H command and specify the command on which you want to receive help as the parameter. SoftICE displays a description of the command, the command syntax, and an example.

The following example displays help for the BPINT command:

```
:H BPINT
Breakpoint on interrupt
BPINT interrupt-number {IF expression] [DO bp-action]
ex: BPINT 50
```

Using the Command Window

The Command window lets you enter commands and displays information about your debugging session. The contents of the Command window are saved in the SoftICE history buffer.

The Command window is always open and is at least two lines long. Although you cannot explicitly resize the Command widow, changing the size of other windows in your display automatically resizes the Command window.

Scrolling the Command Window

To scroll the Command window, either use the scroll arrows or the keys listed in the following table.

Table 5-9. Command Window Scrolling Keys

Function	Key
Scroll the history buffer to the previous page.	PageUp
Scroll the history buffer to the next page.	PageDown
Scroll the history buffer to the previous line.	UpArrow
Scroll the history buffer to the next line.	DownArrow

Entering Commands

Tip As you type characters, the Help line displays the list of valid commands that start with those characters. When only one command displays, you can press the space bar to complete the command automatically. SoftICE fills in the remaining characters of the command followed by a trailing space.

You can enter commands whenever the cursor is in the Command window or the Code window.

To enter a command, type the command and press the Enter key to execute it.

When you type most SoftICE commands in the Command window, related information about the command automatically displays on the line beneath the command. If information displays on the last line of the window, the window scrolls. If all the information cannot fit in the window, the following prompt appears on the help line:

```
Any Key To Continue, ESC To Cancel
```

To disable this prompt, use the following command:

```
SET PAUSE OFF
```

Command Syntax

SoftICE commands share the following syntax and rules:

- All commands are text strings of one to six characters in length and are not case sensitive.
- All parameters are either ASCII strings or expressions.
- An address in SoftICE can be a selector:offset, a segment:offset, or just an offset.
- Expressions in SoftICE are comprised of the following:
  - Grouping symbols
  - Numbers in hexadecimal or decimal format
  - Addresses
  - Line numbers
  - String literals
  - Symbols
  - Operators
  - Built-in functions
  - Registers.

  Example: `(1+2)*3 is an expression.`

  Any command that accepts a number or an address can accept an arbitrarily complex expression. Use the `?` command to display the value of an expression. In addition, breakpoints can be conditionally based on the result of an expression; that is, the breakpoint only triggers when the expression evaluates to non-zero (TRUE).

TAB AutoCompletion

When entering commands, frequently you have to enter a parameter from a limited list of possibilities. For example, symbol names are often used with commands, and are also often quite long. SoftICE can complete some parameters for you using its TAB AutoCompletion feature. This feature functions whenever you press the TAB key while typing a parameter for a command.

If SoftICE finds more than one match for the partially-typed name, it will complete as many characters in the name as are common to all the possible matches, and then display the list of matches in the help text area below the command prompt. If you press TAB again, the list of matches will be displayed in the command window itself, which is helpful when the list of matches is too long for the help text bar.

Some SoftICE commands imply a certain list of parameters. The FILE command, for instance, is used to select and display one of the currently loaded source files. In these cases, the TAB AutoCompletion feature searches the list of possibilities implied by the command. For all other commands, SoftICE will use the symbols from the current symbol table, and all of the loaded export tables.

Using Function Keys

SoftICE provides several function key assignments to save you time when entering commonly-used SoftICE commands. These assignments are shown in the following table.

Table 5-10. SoftICE Function Key Assignments

Function Key	Command	Function
F1	H	Display Help
F2	WR	Display or hide the register window
F3	SRC	Switch among source code, mixed code, and disassembled code
F4	RS	Show program screen
F5	X	Go
F6	EC	Move the cursor to or from the Code window
F7	HERE	Execute to the cursor
F8	T	Single step
F9	BPX	Set an execution breakpoint on the current line
F10	P	Step over
F11	G @SS:EIP	Go to
F12	P RET	Return from the procedure call
Shift-F3	FORMAT	Change the format for the active Data window
Alt-F1	WR	Open or close the Register window
Alt-F2	WD	Open or close the Data window
Alt-F3	WC	Open or close the Code window
Alt-F4	WW	Open or close the Watch window

Table 5-10. SoftICE Function Key Assignments (Continued)

Function Key	Command	Function
Alt-F5	CLS	Clear the Command window
Alt-F11	dd dataaddr->0	Indirect first dword in the Data window.
Alt-F12	dd dataaddr->4	Indirect second dword in the Data window.

You can modify the commands assigned to these keys or assign commands to additional function keys. Refer to *Modifying Keyboard Mappings* on page 203.

Editing Commands

Use the following keys to edit the command line.

Table 5-11. SoftICE Command Line Edit Commands

Editing Function	Key
Move the cursor to column 0 of the command line.	Home
Move the cursor past the last character of the command line.	End
Toggle insert mode. When in insert mode, the cursor displays as a block cursor and the characters entered are inserted at the current cursor position, shifting the text to the right by one space. When not in insert mode, a character entered overwrites the character at the cursor position.	Insert
Delete the character at the current cursor position and shift text to the left by one space.	Delete
Delete the previous character.	Bksp
Cancel command line.	Esc
Move the cursor horizontally within the command line.	Arrow Keys

Recalling Commands

SoftICE remembers the last thirty-two commands you typed in the Command window. You can recall these commands for editing and execution from within either the Command or Code windows.

Use the following keys to recall a command from within the Command window.

Table 5-12. SoftICE Command Window Recall Commands

Function	Key
Get the previous command from the command history buffer.	UpArrow
Get the next command from the command history buffer.	DownArrow

Note: Prefixes are supported. For example, if you type the letter A, the UpArrow only cycles through commands that start with the letter A.

Use the following keys to recall a command from within the Code window.

Table 5-13. SoftICE Code Window Recall Commands

Function	Key
Get the previous command from the command history buffer.	Shift-UpArrow
Get the next command from the command history buffer.	Shift-DownArrow

Regular Expressions in SoftICE

A few of the SoftICE commands (SYM, EXP, FILTER, HS, and TYPES) support using regular expressions to search for text. Regular expressions are a complex subject, and a full definition is beyond the scope of this manual. This section, however, will probably be more than sufficient to get you comfortable using regular expressions in SoftICE. If you want more information on regular expressions, a number of references are available both in print and on the Web.

SoftICE uses a consistent syntax for all commands that accept regular expressions. In order to differentiate between a regular expression and a plain text search pattern, regular expressions must always be enclosed in slash (/) characters. If a regular expression contains spaces anywhere, the whole regular expression, slashes and all, must also be enclosed in double-quotes so that the SoftICE command parser will tokenize it properly.

Regular expressions are a combination of ordinary and special characters. The ordinary characters match themselves, and are defined simply as any character that is not a special character. Table 5-14 on page 95 lists the supported special characters.

Note: Regular expression support is case-insensitive by default. Use the SET CASESENSITIVE command to toggle this property on and off.

Generally any special character can be escaped with a backslash (\) character, and it will be treated as an ordinary character instead.

Table 5-14. Special Characters

Character	Matches	Example
.	Any single character	/Foo./ would match "FooA", "FooB", etc.
*	Zero or more of the previous character	/Fo*/ would match "F", "Fo", "Foooooooo".
+	One or more of the previous character	/Fo+/ would match "Fo", "Foooo", but not "F".
?	Zero or one of the previous character	/Fo?/ would match "F" or "Fo", but not "Foo".
()	Grouping	/(Foo|bar)/ would match "Foo" or "bar".
[]	Any character inside the brackets	/[fo]*/ would match "foo". /[0..9]/ would match any digit from 0 to 9.
[^]	Any character not inside the brackets	/[^0..9abcdef]/ would match any character which is not a valid hex digit.
[x-y]	Any character within the range from x to y (inclusive)	/[a-z]/ would match any character from a to z, inclusive.
{min,max}	Range of occurrences	/(foo){3}/ would match "foofoofoo". /(bar){1,2}/ would match "bar" or "barbar". /(foo){1,}/ would match one or more "foo"s together.
^	Beginning of data string (must be first character in pattern)	/^foo/ would match "foo" only at the beginning of a string.
$	End of data string (must be last character in pattern)	/foo$/ would match "foo" only at the end of a string.

A number of “character classes” are also supported. These are used as shorthand for sets of characters. Character classes are only valid when used within separate enclosing brackets. Table 5-15 on page 96 lists the suppoted character classes.

Table 5-15. Character Classes

Class	Matches	Example
[:alnum:]	Any alphanumeric character	Equivalent to [A-Za-z0-9] /CR[[:alnum:]]/ would match “HalpCr4Exists”.
[:alpha:]	Any alphabetic character	Equivalent to [A-Za-z] /Config[[:alpha:]]/ would match “ConfigA” but not “Config1”.
[:blank:]	Tab and space	/Test[[:blank:]]+String/ would match “Test String”, with any number of spaces or tabs between the two words.
[:cntrl:]	Any control character	
[:digit:]	Any decimal digit	
[:graph:]	Any printable character that is not a space	
[:lower:]	Any lowercase alphabetic character	
[:print:]	Any printable character (including spaces)	
[:punct:]	Any punctuation character	
[:space:]	Any whitespace (newline, linefeed, carriage return, formfeed, tab, and space)	
[:upper:]	Any uppercase alphabetic character	
[:xdigit:]	Any valid hex digit	

Using Run-time Macros

Macros are user-defined commands that you use in the same way as built-in commands. The definition, or body, of a macro consists of a sequence of command invocations. The allowable set of commands includes other user-defined macros and command-line arguments.

There are two ways to create macros. You can create run-time macros that exist until you restart SoftICE or persistent macros that are saved and automatically loaded with SoftICE. This section describes how to use run-time macros. Refer to *Working with Persistent Macros* on page 205 for more information about creating and using persistent macros.

Tip You can use the MACRO command with persistent macros to temporarily modify them during run time. When you reload SoftICE, your persistent macros revert to their original state.

The following table shows how to create, delete, edit, and list run-time macros.

Table 5-16. SoftICE Run-time Macros

Action	Command
Create or modify a macro	MACRO *macro-name* = "*command1;command2;...*"
Delete a macro	MACRO *macro-name* *
Delete all macros	MACRO *
Edit a macro	MACRO *macro-name*
List all macros	MACRO

The body of a macro is a sequence of SoftICE commands or other macros separated by semicolons. You are not required to terminate the final command with a semicolon. Command-line arguments to the macro can be referenced anywhere in the macro body with the syntax `%<parameter#>`, where *parameter#* is a number between one and eight.

The command `MACRO asm = "a %1"` defines an alias for the A (ASSEMBLE) command. The `%1` is replaced with the first argument following `asm` or simply removed if no argument is supplied.

If you need to embed a literal quote character (") or a percent sign (%) within the macro body, precede the character with a backslash character (\). To specify a literal backslash character, use two consecutive backslashes (\\).

Note: Although it is possible for a macro to call itself recursively, it is not particularly useful, because there is no programmatic way to terminate the macro. If the macro calls itself as the last command of the macro (tail recursion), the macro executes until you use the ESC key to terminate it. If the recursive call is not the last command in the macro, the macro executes 32 times (the nesting limit).

The following table shows some examples of run-time macros.

Table 5-17. Run-time Macro Examples

Run-time Macro Commands	Examples
MACRO Qexp = “addr explorer; Query %1”	Qexp
	Qexp 140000
MACRO 1shot = “bpx %1 do \”bc bpindex\””	1shot eip
	1shot @esp
MACRO ddt = “dd thread”	ddt
MACRO ddp = “dd process”	ddp
MACRO thr = “thread %1 tid”	thr
	thr -x
MACRO dmyfile = “macro myfile = \”TABLE %1;file \%1\””	dmyfile mytable myfile myfile.c

Special Macros: POPUP and POPDOWN

There are two special macros that can be defined in SoftICE: the POPUP and POPDOWN macros. If defined, these macros are executed automatically whenever SoftICE pops up or down. These macros are defined and deleted like any other macro.

Saving the Command Window History Buffer to a File

The SoftICE history buffer contains all the information displayed in the Command window. Saving the SoftICE history buffer to a file is useful for doing the following:

- Dumping large amounts of data or register values
- Disassembling code
- Listing breakpoints logged by the BPLOG expression
- Showing Windows messages logged by the BMSG command
- Saving debugging messages sent from user programs that call OutputDebugString and kernel-mode programs that call KdPrint

Refer to *History Buffer Size* on page 194 for more information about changing the size of the SoftICE history buffer.

To save the contents of the SoftICE history buffer to a file, do the following:

1. Make sure the information you want to save is displaying to the Command window, so that it is saved in the History Buffer.

   For example, before dumping data, remove the Data window to force the data to display in the Command window.Run-time

2. Open Symbol Loader.

3. Either choose SAVE SOFTICE HISTORY AS... from the File menu or click the SAVE SOFTICE HISTORY button.

4. Use the Save SoftICE History dialog box to determine the file name and location where you want to save the file.

Associated Commands

The following command is associated with the Command window. Refer to the *SoftICE Command Reference* for more information about using this command.

Table 5-18. Command Window SET Command

Command	Function
SET [set variable] [ON | OFF] [value]	Displays or sets user preferences.

Using the Code Window

The Code window displays source code, disassembled code, or both source and disassembled code (mixed). It also lets you set breakpoints. (Refer to *Chapter 7:* on page 131 for an explanation of how to set breakpoints.)

Controlling the Code Window

Use the following commands to control the Code window.

Table 5-19. SoftICE Code Window Control Commands

Command	Action
WC	Opens and closes the Code window.
WC *[+ | -] [num lines]*	Resizes the Code window.
Alt-C	Moves the cursor into or out of the Code window.

Scrolling the Code Window

To scroll the Code window, either use the scroll arrows or the following keys when the cursor is in the Code window.

Table 5-20. Cursor-in-Code Window Functions

Function (from within the Code window)	Key Sequence
Scroll Code window to the previous page.	PageUp
Scroll Code window to the next page.	PageDown
Scroll Code window to the previous line.	UpArrow
Scroll Code window to the next line.	DownArrow
Jump to the first line of the source file.	Ctrl-Home
Jump to the last line of the source file.	Ctrl-End
Scroll Code window left one character (source mode only).	Ctrl-LeftArrow
Scroll Code window right one character (source mode only).	Ctrl-RightArrow

You can also scroll the Code window when the cursor is in the Command window, as follows.

Table 5-21. Cursor-in-Command Window Functions

Function (from within the Command window)	Key
Scroll the Code window to the previous page.	Ctrl-PageUp
Scroll the Code window to the next page.	Ctrl-PageDn
Scroll the Code window to the previous line.	Ctrl-UpArrow
Scroll the Code window to the next line.	Ctrl-DownArrow
Jump to the first line of the source file.	Ctrl-Home
Jump to the last line of the source file.	Ctrl-End
Scroll the Code window left one character (in source mode only).	Ctrl-LeftArrow
Scroll the Code window right one character (in source mode only).	Ctrl-RightArrow

The Code Window display has a few display controls which can be used to change its behavior at any time. These are accessed through the SET command. Some of these controls can also be set in the Disassembly Options page of the Settings application

Table 5-22. Code Window Controls

Control	Description
CheckStrings	If enabled, the disassembler will examine operands. If an operand appears to point to an ASCII or Unicode string, the disassembler will display the string as a comment.
Code	Controls the display of the actual code bytes for each instruction.
DisassemblyHints	When enabled, the disassembler will display directional hints for branch instructions. If a disassembled instruction is a branch (conditional or unconditional), the disassembler will display a directional arrow next to the address operand, pointing towards the destination address.
Lowercase	Controls the display of disassembled instructions. If set, all instructions will be displayed in lower case.
Selectors	Controls the display of selectors in the Code Window. If set, the selector value is shown with each disassembly address. If clear, the selector value is shown in the Code Window’s title bar only.
Symbols	Controls the resolution of addresses to symbols in the Code Window. If set, addresses are resolved to symbols where possible. If clear, the numeric values are shown instead.

Viewing Information

The Code window provides three modes to display source code, disassembled code, or both. The following table defines these modes.

Table 5-23. Code Window Modes

Code Mode	Description
Source	If source code is available, the source file displays in the Code window.
Mixed	In mixed mode, both source lines and disassembled instructions display in the Code window. Each source line is followed by its assembler instructions.
Code	In code mode, only disassembled instructions display in the Code window.

To switch among the Code window modes, use the SRC command (F3).

Using Code and Mixed Modes

Each disassembled instruction in code or mixed mode contains the following fields.

Table 5-24. Code and Mixed Mode Fields

Field	Description
Location	Hexadecimal address of the instruction. If there is a public code symbol or a user-defined name for the location, it displays on the line above the instruction.
Code bytes	Actual hexadecimal bytes of the instruction. The default is to suppress the code bytes because they are usually not needed. Use the SET CODE ON command to display the code bytes.
Instruction	Disassembled mnemonics of the instruction. This is the current assembly language instruction. If any of the memory address references of the instruction match a symbol, the symbol displays instead of the hexadecimal address. Use SET SYMBOLS OFF to display hexadecimal addresses instead.
Comment	Helpful comment from the disassembler.

The following output shows a disassembled instruction:

```
00008:F1A19104  56       PUSH    ESI
```

Additionally, the SoftICE disassembler automatically provides these comments:

- INT 2E calls are commented with the kernel routine that will be called and the number of parameters it takes. If you have loaded the symbols for NTOSKRNL and that is the current symbol table, you will see the name of the OS routine rather than an address.
- If an instruction uses an immediate operand that matches a Windows NT family status code, the name of the status code displays as a comment.
- INT 21 calls are commented with their DOS function names.
- INT 31 calls are commented with their DPMI function names.
- VxD service names are shown as code labels where appropriate.

Viewing Additional Information

In addition to source and disassembled code, the Code window displays the following information:

- When SoftICE pops up, the instruction located at the current EIP is highlighted in bold. If the instruction is a relative jump, the disassembler's comment field contains either the string JUMP or NO JUMP, indicating whether or not the jump will be taken. For the JUMP string, an up or down arrow indicates where the jump is going: backwards (JUMP ↑) or forwards (JUMP ↓). Use the arrow to determine which way to scroll the Code window to view the target of the JUMP.
- The target of a JUMP instruction is always marked with a highlighted arrow indicator (=>) beside the destination address.
- If the instruction references a memory location, the effective address and the value at the effective address display on the end of the code line. If the Register window is visible, however, the effective address and the value at the effective address display in that window beneath the flags field.
- If a breakpoint exists at any instruction in the Code window, the corresponding line displays in bold text.
- The lines above and below the Code window show more information about the code.
  - Information above the Code window includes one of the following:
    - Symbolname + Offset
    - Source file name, if viewing source
    - One of the following segment types:
      V86 Code from a real-mode segment:offset address.
      PROT16 Code from a 16-bit protected mode selector:offset address
      PROT32 Code from a 32-bit protected mode selector:offset address
  - Information below the Code window includes one of the following:
    - Windows module name, section name, and OFFSET if it is a 32-bit Windows module. For example, `KERNEL32!.Text + 002f`
    - Windows module name and segment number in parentheses if it is a 16-bit Windows module. For example, `Display (01)`

– Owner name of the code segment if it is in V86 mode. For example, `DOS`.

Entering Commands From the Code Window

You can still enter commands when the cursor is in the Code window. After you type the first letter of a command, the cursor moves down to the Command window. After you press Enter and the command completes, the cursor moves back to the Code window. You can also use function key commands while the cursor is in the Code window. Refer to *Using the Command Window* on page 90 for more information about entering commands.

The following commands are particularly useful.

Table 5-25. Code Windows Commands

Command	Function
. (Dot)	View the instruction at the current EIP.
A *address*	Assemble instructions directly into memory.
BPX (F9)	Set point-and-shoot breakpoints.
FILE *file-name*	Select the source file to view. The filename can be a partial name. If you do not know the name of the filename, enter FILE * to display all the files loaded for the symbol table.
HERE (F7)	Set breakpoints that execute one time.
SET	Display or set user preferences.
SRC	Switch among the Code window modes: source, mixed, and code.
SS *string*	Move the source display to the next occurrence of the specified string.
TABS *tab-setting*	**Note:** TABS is now part of the SET command. See the SET command entry in the *SoftICE Command Reference* for details.
U *address*	Unassemble any code address. If you specify a function name for the address parameter, SoftICE scrolls the Code window to the function you specify.

Refer to the *SoftICE Command Reference* for more information about these commands.

Using the Locals Window

The Locals window displays the current stack. You can view the contents of structures, arrays, and character strings within the stack by expanding them.

Controlling the Locals Window

Use the following commands to control the Locals window.

Table 5-26. Locals Windows Commands

Command	Action
WL	Opens and closes the Locals window.
WL *[num lines]*	Resizes the Locals window.
Alt-L	Moves the cursor into or out of the Locals window.
Alt-E	Invoke inline editing.

Scrolling the Locals Window

To scroll the Locals window, either use the scroll arrows or use Alt-L to move the cursor into the Locals window, then use the following keys.

Table 5-27. Locals Window Scrolling Functions

Function	Key Sequence
Scroll the Locals window to the previous page.	PageUp
Scroll the Locals window to the next page.	PageDn
Scroll the Locals window to the previous line.	UpArrow
Scroll the Locals window to the next line.	DownArrow
Jump to first item.	Home
Jump to last item.	End
Scroll the Locals window left one character.	LeftArrow
Scroll the Locals window right one character.	RightArrow

Expanding and Collapsing Stacks

You can expand structures, arrays, and character strings to display their contents. These items are delineated with a plus sign (+) to indicate that you can expand them. To expand or collapse an item, do the following:

- Pentium PCs only—Double-click the item.
- All PCs—Use Alt-L to enter the Locals window, scroll to the item, and press Enter.

Associated Commands

The following commands are associated with the Locals window. Refer to the *SoftICE Command Reference* for more information about using these commands.

Table 5-28. Locals Window Commands

Command	Function
LOCALS	Lists local variables from the current stack frame.
TYPES *[type-name]*	Lists all types in the current context or lists all type information for the type-name specified.

Using the Watch Window

The Watch window lets you monitor the values of expressions that you set with the WATCH command. Refer to the *SoftICE Command Reference* for more information about the WATCH command.

Controlling the Watch Window

Use the following commands to control the Watch window.

Table 5-29. Watch Window Commands

Command	Action
WW	Opens and closes the Watch window.
WW *[num lines]*	Resizes the Watch window.
Alt-W	Moves the cursor into or out of the Watch window.
Alt-E	Invoke inline editing.

Scrolling the Watch Window

To scroll the Watch window, either use the scroll arrows or use Alt-W to move the cursor into the Watch window and use the following keys.

Table 5-30. Watch Window Scrolling Functions

Function	Key Sequence
Scroll the Watch window to the previous page.	PageUp
Scroll the Watch window to the next page.	PageDown
Scroll the Watch window to the previous line.	Arrow
Scroll the Watch window to the next line.	DownArrow
Jump to first item.	Home
Jump to last item.	End
Scroll the Watch window left one character.	LeftArrow
Scroll the Watch window right one character.	RightArrow

Setting an Expression to Watch

Use the WATCH command to set an expression to watch. The expression can use global and local symbols, registers, and addresses.

Note: To set a watch on a local variable, the variable must be in scope.

The following examples illustrate how to use the WATCH command.

- Monitors the value of ds:esi:

```
WATCH ds:esi
```

- Monitors the value ds:esi *points to*:

```
WATCH *ds:esi
```

Deleting a Watch

You can use either the mouse or keyboard to delete a watch. To use your mouse to delete a watch, click on the watch and press Delete. To use your keyboard to delete a watch, use Alt-W to enter the Watch window, use the arrow keys to select the watch, and press Delete.

Viewing Information

The Watch window contains the following fields in the order shown.

Table 5-31. Watch Window Fields

Watch Line Field	Description
Expression	Actual expression that was typed on the WATCH command. This expression is re-evaluated every time the Watch window displays.
Type definition	Type definition of the expression.
Value	Current value of the expression being watched.

Expanding and Collapsing Typed Expressions

You can expand typed expressions to display their contents. Typed expressions are delineated with a plus sign (+) to indicate that you can expand them. To expand or collapse a typed expression, do the following:

- Pentium PCs only — Double-click the item.
- All PCs — Use Alt-W to enter the Watch window, scroll to the item, then press Enter.

Associated Commands

The following command is associated with the Watch window. Refer to the SoftICE Command Reference for more information about using this command.

Table 5-32. Watch Window Command

Command	Function
WATCH expression	Adds a watch expression.

Using the Register Window

The Register window displays the current value of the system registers, flags, and the effective address if applicable. Use this window to determine which registers are altered by a procedure call or to edit the registers and flags.

Controlling the Register Window

Use the following commands to control the Register window.

Table 5-33. Register Window Commands

Command	Action
WR	Opens and closes the Register window.
Alt-R	Moves the cursor into or out of the Register window.

If you are not using the Register window, close it to free up screen space for other windows.

Viewing Information

The first three lines in the Register window show the following registers, flags, and address if available:

```
EAX, EBX, ECX, EDX, ESI
EDI, EBP, ESP, EIP, o d i s z a p c
CS, DS, SS, ES, FS, GSeffective address=value
```

When you use the T (trace), P (step over), and G (go to) commands, SoftICE highlights the registers that change. This feature is useful for seeing which registers were altered by a procedure call.

In the second line of the Register window, the CPU flags are defined as follows.

Table 5-34. Register Window CPU Flag Definitions

Flag	Description	Flag	Description
o	Overflow flag	z	Zero flag
d	Direction flag	a	Auxiliary carry flag
i	Interrupt flag	p	Parity flag
s	Sign flag	c	Carry flag

Note: A lowercase letter that is not highlighted indicates a flag value of 0. A highlighted uppercase letter indicates a flag value of 1, for example, o d I s Z a p c.

If the current instruction references a memory location, the effective address and the value at the effective address display in the third line of the Register window. You can use the effective address and value in expressions with the Eaddr and Evalue functions; refer to *Built-in Functions* on page 164.

Editing Registers and Flags

You can use the Register window to edit the registers and flags. Move the cursor into the Register window, then edit the registers and flags in place. To move the mouse into the Register window, either click the mouse in the Register window or press Alt-R. The following keys are available for editing within the Register window. You can also toggle the flags register by clicking on the appropriate flag.

Table 5-35. Register Window Editing Functions

Editing Function	Active Keys
Position cursor at the beginning of the next register field.	Tab or Shift-RightArrow
Position cursor at the beginning of the previous register field.	Shift-Tab or Shift-LeftArrow
Accept changes and exit edit register mode.	Enter
Exit edit register mode. The register that the cursor is currently on will not change, but other previously-modified registers change.	Esc
Toggle the value of a flag when the cursor is positioned in the flags field.	Insert or Mouse Click
Move the cursor left, right, up, and down in the Register window.	Arrow keys

Associated Commands

Table 5-36 provides commands associated with the Register window. Refer to the *SoftICE Command Reference* for more information about using these commands.

Table 5-36. Associated Register Window Commands

Command	Function
CPU	Displays CPU register information.
G *[=start-address] [break-address]*	Goes to an address.
P	Executes one program step.
T *[=start-address] [count]*	Traces one instruction.

Using the Data Window

The Data window lets you view and edit the contents of memory. You can use up to four different Data windows at any given time. Each Data window can view different memory locations and display information in its own unique format, as well as display an address that is independent of the other Data windows.

Controlling the Data Window

Use the following commands to control the Data window.

Table 5-37. Data Window Commands

Command	Action
WD.*n*	Opens and closes the Data window, where *n* is a number from 0 through 3 specifying the Data window. If you do not specify a value for *n*, 0 is assumed.
WD.*n [#-lines]*	Resizes the Data window, or open the specified Data window to the specified size.
Alt-D	Moves the cursor into or out of the current Data window.
DATA *n*	Opens the next sequential Data window, or switches to the next sequential Data window once all four are open. Specifying a value for *n* will set the specified window as the active Data window.
D *[address]*	Select an address to view in the current Data window.
FORMAT (Shift-F3)	Selects a format to display in the current Data window.

There can only be one active Data window at a time. SoftICE signifies the active window by displaying the Data window number, on the right edge of the title bar, in bold type. To make a specific Data window the active window, either select it with the mouse, or use the `DATA n` command.

Scrolling the Data Window

To scroll the Data window, either click the scroll arrows or press Alt-D to move the cursor into the Data window and use the following keys.

Table 5-38. Data Window Scroll Functions

Function	Key Sequence
Scroll the window to the previous page.	PageUp
Scroll the window to the next page.	PageDown
Scroll the window to the previous line.	UpArrow
Scroll the window to the next line.	DownArrow

Viewing Information

The line above the Data window displays the following four fields in the order shown.

Table 5-39. Data Window Description Fields

Field	Description
A String	If the window was assigned an expression with the DEX command, the ASCII expression displays on this line. Otherwise, the nearest symbol preceding the data location displays. This can be one of the following strings: • Symbol name followed by the hexadecimal offset from the symbol name, for example, MySYMBOL+00010 • Windows module name followed by a type, if the data segment is part of the Windows heap, for example, mouse.moduleDB • Owner name of the data segment if it is part of a virtual DOS machine. • Windows module name, section name, and hexadecimal offset from the name, for example, KERNEL32!.text+001F If the location does not have an associated symbol, this field is blank.
Data format type	Displays either byte, word, dword, short real, long real, or 10-byte real.

Table 5-39. Data Window Description Fields (Continued)

Field	Description
Segment type	Either V86 or PROT displays. V86 indicates data from a real-mode segment:offset address and PROT indicates data from a protected-mode selector:offset address.
Window number	Data window number from 0 to 3.

Each line in a Data window shows 16 bytes of data in the current format of either byte, word, dword, short real, long real, or 10-byte real. If the current format is 10-byte real, each line shows 20 bytes of data. The data bytes also display in ASCII on the right side of the window if the current format is hexadecimal (byte, word, or dword).

Changing the Memory Address and Format

Tip You can also use the D command to specify the format for the address you display. Refer to the SoftICE Command Reference for more information about the D command.

Either click on the format name listed in the top line of the Data window or use the FORMAT command (Shift-F3) to change the format of the current Data window. The format cycles among the following: byte, word, dword, short real, long real, and 10-byte real.

To change the memory address displayed in the current Data window, enter the D command and specify an address. The following example displays the memory starting at address ES:1000h:

```
: D es:1000
```

Editing Memory

To edit memory, move the cursor into the Data window and use either hexadecimal or ASCII characters.

Tip You can also use the E command to edit data.

Use the following keys for editing within the Data window.

Table 5-40. Data Window Editing Functions

Editing Function	Active Keys
Toggle between numeric and ASCII areas.	Tab
Position cursor at the beginning of the previous data field (previous byte, word, or dword in hexadecimal mode, or previous character in ASCII mode).	Shift-Tab
Accept changes and exit edit data mode.	Enter
Exit edit data mode. The data field the cursor is currently on will not change, but other previously-modified data fields change.	Esc

Assigning Expressions

Use the DEX command to assign an expression to any of the Data windows. When SoftICE pops up, the expressions are evaluated and the resulting locations display in their assigned Data windows. This is useful for setting up a window that always displays the contents of the stack. For example, the following command displays the current contents of the stack in Data window 0, each time SoftICE pops up:

```
DEX 0 SS:ESP
```

Associated Commands

The following commands are associated with the Data window. Refer to the *SoftICE Command Reference* for more information about using these commands.

Table 5-41. Associated Data Window Commands

Command	Function
D [size] [address]	Displays memory.
DEX *[data-window-number [expression]]*	Displays or assigns an expression to the Data window.
E *[size] [address [data-list]]*	Edits memory.
S *[-cu] [address L length data list]*	Searches memory for data.

Using the Stack Window

The Stack Window displays the call stacks for 32-bit code. The Stack window has three columns: Frame pointer, return address, and instruction pointer (EIP):

```
0012FFC0   77F1B304      WINMAIN
0012FFF0   00000000      KERNEL32!GetProcessPriorityBoost+0117
```

Use the following commands to control the Stack window.

Table 5-42. Stack Window Commands

Command/Keys	Function
WS	Opens and closes the Stack window
ALT-S	Gives Stack window focus
Arrow Keys	Select a particular call stack element
Enter	Updates Locals and Code windows when a call stack item is selected

You can also click the mouse in the Stack window to set focus, single click an item to select it, and double click an item to update the Locals, Code, and Thread windows.

Using the Thread Window

The Thread Window displays information for threads within a given process. The data displayed in the Thread window depends on whether you are running Windows 9x or the Windows NT family. Refer to the SoftICE online help for details (the information can be found under the WT command).

Controlling the Thread Window

Use the following commands to control the Thread window:

Table 5-43. Thread Window Commands

Command	Action
WT	Opens and closes the Thread Window
WT *[num lines]*	Resizes the Thread Window
Alt-T	Moves the cursor into or out of the thread window

To scroll the Thread window, either click the scroll arrows or press Alt-T to move the cursor into the Thread window and use the following keys

Table 5-44. Thread Window Scrolling Key Sequences

Function	Key Sequence
Scroll the window to the previous page.	PageUp
Scroll the window to the next page.	PageDown
Scroll the window to the previous line.	UpArrow
Scroll the window to the next line.	DownArrow

Using the Pentium III/IV Register Window

The Intel Pentium III/IV instruction set is supported, including disassembly and assembly of new opcodes. Pentium III/IV registers can be viewed using the WX command.

Table 5-45. Pentium III/IV Register Commands

CPU	Command	Function
P-III	f	Display as short real values
P-III	d	Display as dword values
P-III	*	Toggle between dword and real
P-IV+	-dq	Double quad-word
P-IV+	-sf	Single float
P-IV+	-df	Double float
P-IV+	-q	Quad word

Using the FPU Stack Window

The FPU Stack window displays the current state of the floating point unit (FPU) stack and MMX/MMX2 registers.

Use the WF command to open or close the FPU Stack window.

Viewing Information

If the values of the FPU registers display as a question mark (?), the FPU is disabled or not present. The Windows NT family enables the FPU for a thread after it executes one FPU-related instruction.

The Intel architecture aliases the 64-bit MMX/MMX2 registers upon the FPU stack.

Note: MMX refers to the multimedia extensions to the Intel Pentium and Pentium-Pro processors.

To display registers in the FPU Stack window, select one of the data formats listed in Table 5-46 on page 117.

Table 5-46. FPU Stack Window Register Data Formats

Data Format	Description	Use
WF F	Floating point	Floating point only
WF B	Byte packed	MMX only
WF W	Word packed	
WF D	Dword packed	

Tip Use the WF -D command to display the contents of the registers, the status, and the control words in the Command window.

When they are viewed as floating points, the registers are labeled ST0 through ST7. When they are viewed packed, as byte/word/dword, the registers are labelled MM0 through MM7. (See the *SoftICE Command Reference* for more information about the WF command.)

Chapter 6
Using SoftICE

- Debugging Multiple Programs at Once
- Trapping Faults
- About Address Contexts
- Using INT 0x41 .DOT Commands
- Understanding Transitions From Ring 3 to Ring 0

Debugging Multiple Programs at Once

Symbol Loader lets you load several symbol tables at the same time. Thus, you can debug complex sets of system software that may contain several different components, including applications, DLLS, and drivers.

Use the TABLE command to view a list of all the symbol tables currently loaded and to select a different symbol table. When you reach a breakpoint in a program that has a corresponding symbol table, enter the TABLE command followed by the first few characters of the symbol table name to change the current symbol table to the one that matches your program.

If you are not sure which table is the current table, enter the TABLE command with no parameters to list all the loaded tables. The current table is highlighted.

You can also switch tables to a symbol table that does not match the code you are currently executing. This is useful for setting a breakpoint in a program other than the one you are currently executing.

Note: Symbol tables now do time and date checking to ensure that the symbol files are up to date with the binary in use. Correct any discrepancies by retranslating and reloading the symbol table.

Trapping Faults

SoftICE provides fault trapping support for the following types of code:

- Ring 0 driver code (kernel mode device drivers)
- Ring 3 (32-bit) protected mode (Win32 programs)
- Ring 3 (16-bit) protected mode (16-bit Windows programs)

SoftICE does not provide fault trapping for DOS machines. This includes both straight V86 programs and DOS extender applications.

The following sections describe fault trapping support.

Ring 0 Driver Code (Kernel Mode Device Drivers)

SoftICE handles all ring 0 exceptions that result in a call to KeBugCheckEX. KeBugCheckEX is the routine that displays the “blue screen” in the Windows NT family.

If the KeBugCheckEX bug code is the result of a page fault, GP fault, stack fault, or invalid opcode, SoftICE attempts to restart the faulting instruction. Control stops on the actual faulting instruction with all the registers in their original state. If the code continues to fault on the same instruction, either reboot or attempt to skip the fault by altering the EIP or fixing the fault condition.

If the KeBugCheckEx bug code is not the result of a page fault, GP fault, stack fault, or invalid opcode, the instruction cannot be restarted. SoftICE pops up and displays the first instruction in KeBugCheckEX and a message similar to the following:

```
Break Due to KeBugCheckEx (Unhandled kernel mode exception)
Error=1E (KMODE_EXCEPTION_NOT_HANDLED) P1=8000003 P2=804042B1
P3=0 P4=FFFFFFFF
```

The error field is the hexadecimal bug code followed by a description of the error. Bug code definitions are contained in the Windows NT family DDK in the include file bugcodes.h.

The P1 through P4 fields are the parameters passed to the KeBugCheckEX routine. These fields do not have a standard defined meaning.

If you attempt to continue from this point, the Windows NT family platforms display a blue screen and hang. If you want to gain control after the blue screen, turn on I3HERE (SET I3HERE ON); a Windows NT family machine will execute an INT 3 instruction after it displays the blue screen.

Ring 3 (32-bit) Protected Mode (Win32 Programs)

SoftICE traps all unhandled exceptions that normally cause an error dialog box. SoftICE automatically restarts the instruction that caused the fault, pops up the SoftICE window, and displays the instruction and a message similar to the following:

```
Break due to Unhandled Exception
NTSTATUS=STATUS_ACCESS_VIOLATION
```

The NTSTATUS field contains the appropriate error message corresponding to the status code. (Refer to the include file NTSTATUS.H in the Windows 2000/XP DDK for a complete list of status codes.)

If execution continues after SoftICE traps the fault, SoftICE ignores the fault and lets the system do its normal exception processing. For example, it could present an application failure dialog box.

Ring 3 (16-bit) Protected Mode (16-bit Windows Programs)

SoftICE handles 16-bit fault trapping somewhat differently than 32-bit fault trapping. When a 16-bit fault occurs, the machine eventually displays a dialog box that describes the fault and gives you the choice of CANCEL or CLOSE.

If you click CANCEL, the faulting instruction is restarted and Windows issues a debugger notification for trapping the faulting instruction. SoftICE uses this debugger hook to pop up and display the faulting instruction. In other words, SoftICE pops up ***after*** you receive the crash dialog box and select CANCEL, not before.

If you click CLOSE, Windows does not restart the instruction and SoftICE does not pop up. Thus, if you want to debug the fault, make sure you click CANCEL.

Some Windows faults display more than one dialog box. If this happens, the first dialog box provides a choice of CLOSE or IGNORE. Choose IGNORE to instruct Windows to skip the faulting instruction and to continue to execute the program. Choose CLOSE to instruct Windows to display the second dialog box, as previously described.

SoftICE Crash Dump Utility

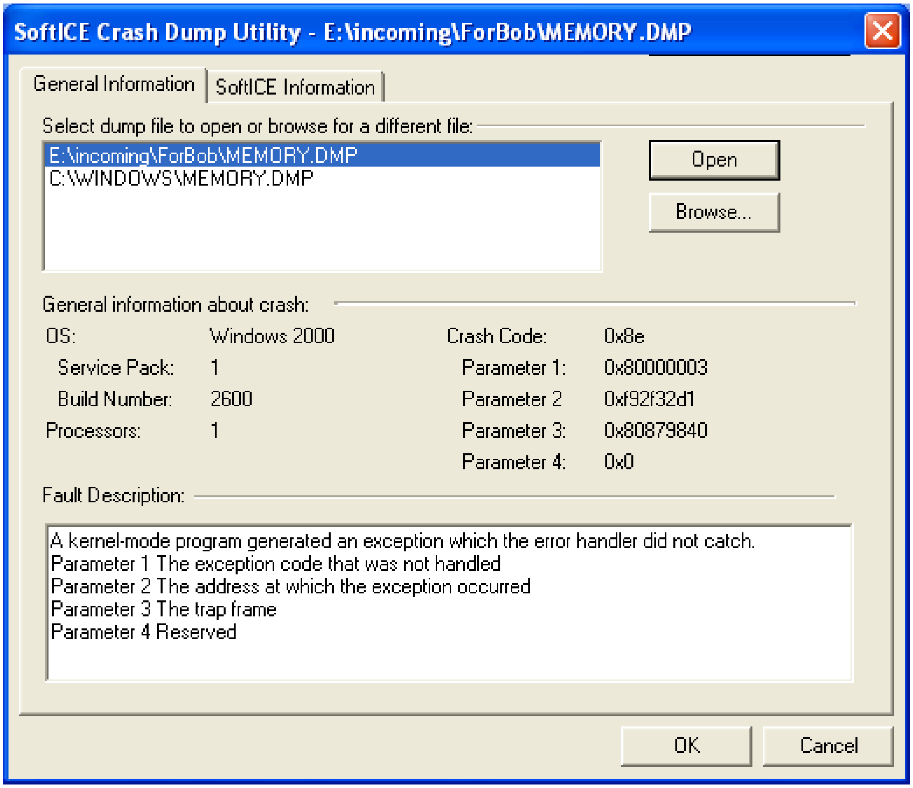
SoftICE ships with a crash dump utility that will allow you to:

- View the SoftICE history at the point in time the crash occurred
- View and merge the following SoftICE information back into SoftICE
  - Breakpoints
  - Named Memory
  - Macros
- View a brief description of the crash code and its parameters as well as a short description of what the bug code is
- View a list of details on the machine that crashed such as OS Build, Service Pack, and Number of Processors
- Email the entire SoftICE state, including history, in a single file for remote analysis

The design of SoftICE and its interaction with the operating system does not allow for safe access to the file system at the point in time when a blue screen crash occurs. This can be an annoyance, as there is no easy way to rebuild the state of SoftICE at the time of the crash. We now provide a utility that will allow post mortem analysis locally or remotely.

To run the utility, select **SoftICE Crash Dump Utility** from the `Compuware\DriverStudio\Debug` menu or run the executable from:

`\program files\compuware\driverstudio\softice\sicrashutil.exe.`



Select dump file to open or browse for a different file:

E:\incoming\ForBob\MEMORY.DMP
C:\WINDOWS\MEMORY.DMP

General information about crash:

OS: Windows 2000
Service Pack: 1
Build Number: 2600
Processors: 1

Crash Code: 0x8e
Parameter 1: 0x80000003
Parameter 2 0xf92f32d1
Parameter 3: 0x80879840
Parameter 4: 0x0

Fault Description:

A kernel-mode program generated an exception which the error handler did not catch.
Parameter 1 The exception code that was not handled
Parameter 2 The address at which the exception occurred
Parameter 3 The trap frame
Parameter 4 Reserved

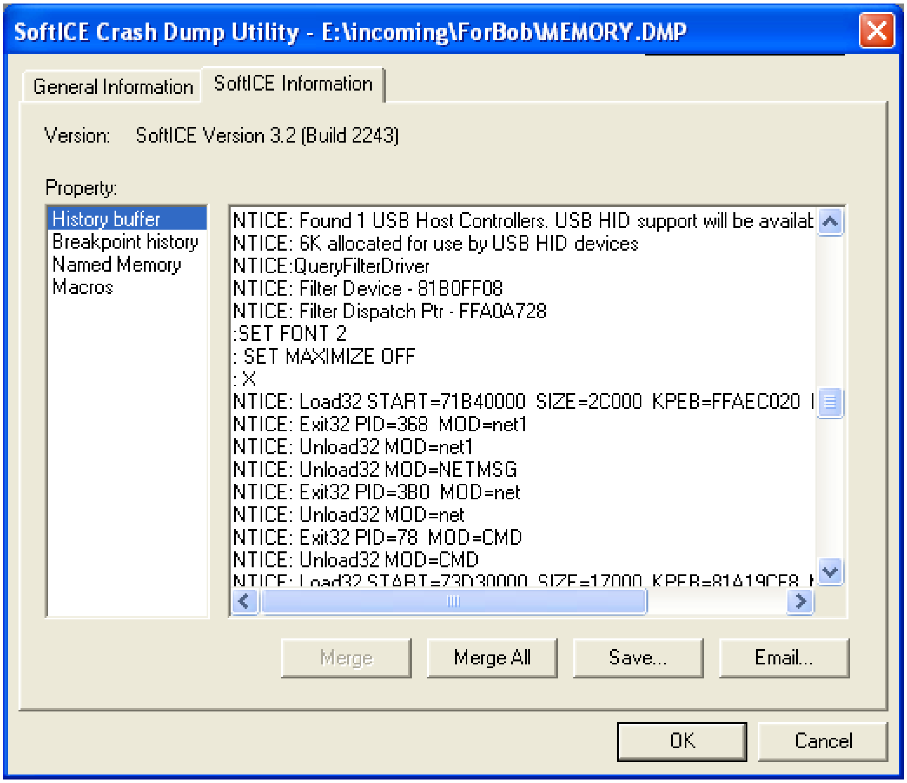
Figure 6-3. Crash Dump Utility Main Screen

You will be presented with the general information screen. The information on this screen is not meant to be a full featured debugger, but is meant to give you an overview of why the crash occurred. For full featured debugging of a crash dump file we suggest using the DriverWorkbench application that ships in the DriverStudio kit.

The fault description field displays a brief overview of the definition of the crash code, as well as a brief description of the parameter description for that particular crash.

Viewing and Merging SoftICE Information

The SoftICE information that is within the crash dump is viewable from the SoftICE Information tab. This information can be saved to a file, merged back into SoftICE, or emailed to a remote recipient for analysis. In addition, statistics are given for your breakpoints.



Version: SoftICE Version 3.2 (Build 2243)

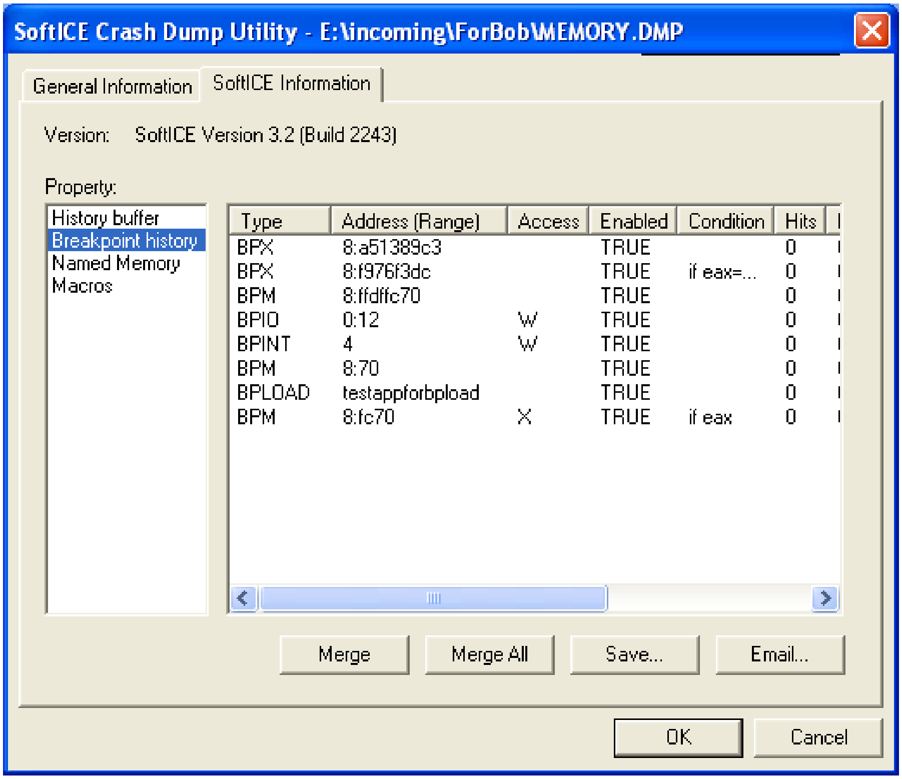
Property:

```
NTICE: Found 1 USB Host Controllers. USB HID support will be availat
NTICE: 6K allocated for use by USB HID devices
NTICE:QueryFilterDriver
NTICE: Filter Device - 81B0FF08
NTICE: Filter Dispatch Ptr - FFA0A728
:SET FONT 2
: SET MAXIMIZE OFF
: X
NTICE: Load32 START=71B40000  SIZE=2C000  KPEB=FFAEC020  I
NTICE: Exit32 PID=368  MOD=net1
NTICE: Unload32 MOD=net1
NTICE: Unload32 MOD=NETMSG
NTICE: Exit32 PID=3B0  MOD=net
NTICE: Unload32 MOD=net
NTICE: Exit32 PID=78  MOD=CMD
NTICE: Unload32 MOD=CMD
NTICE: Load32 START=73D30000  SIZE=17000  KPEB=81A19CE8  K
```

Figure 6-4. History Window

Merging data back into the appropriate SoftICE configuration files can be done either on an individual basis, or with all of the items that can be merged. For example, you may choose to merge back in only the Named Memory. To do this, select the **Named Memory** property and then click the **Merge** button. Note that the Merge button is only available when an item can be merged back into SoftICE.

The breakpoint page is extremely helpful to see what code has, and has not, been hit post mortem. With well placed breakpoints and the BPLOG modifier, it is possible to effectively add debug prints without the need for modifying your code. Within SoftICE you always have access to breakpoint statistics. With the crash dump utility you have access to that same information. You can view the number of times the breakpoint was hit, the number of times the conditional expression evaluated to true or false, as well as the condition in use.



Version: SoftICE Version 3.2 (Build 2243)

Property:

Type	Address (Range)	Access	Enabled	Condition	Hits
BPX	8:a51389c3		TRUE		0
BPX	8:f976f3dc		TRUE	if eax=...	0
BPM	8:ffdffc70		TRUE		0
BPIO	0:12	W	TRUE		0
BPINT	4	W	TRUE		0
BPM	8:70		TRUE		0
BPLOAD	testappforbpload		TRUE		0
BPM	8:fc70	X	TRUE	if eax	0

Figure 6-5. Breakpoint Page

Accessing merged data

If SoftICE is already running when the merge is run, you will need to reboot your machine, as the configuration information is read only upon startup of SoftICE. If SoftICE is not running when the Merge or Merge All option was selected, this information will be available after starting SoftICE. Named memory and macros will be automatically instantiated and can be viewed with the **NAME** and **MACRO** commands.

The breakpoints will need to be reinstantiated. To do this:

1. Pop into SoftICE and issue the **BH** command.
2. Use the cursor keys to move to the breakpoints you want to reenable.
3. Use the **<Insert>** key to enable each selected breakpoint.

Saving and emailing crash dump information

All information can be saved to disk for further review, or emailed to a recipient, by clicking on the appropriate tabs. As the size of the email attachment can be quite large (this varies with the size of your history buffer) it is suggested that you save the file to disk, use some sort of file compression utility, and then email the compressed file.

About Address Contexts

Windows 9x and the Windows NT family machines give each process its own address space from 0 GB to 2GB. In addition, Windows ME reserves the first 4 MB for each virtual machine (where DOS and its drivers reside). Memory from 2GB to 4GB is shared between all processes.

The process-specific virtual address space is known as the _address context_ (or _process_). SoftICE displays the name of the current process on the far right side of the status bar at the bottom of the screen. Be aware that the current context is not always your application's context, particularly if you hotkey into SoftICE. If you are not in the context of your application, use the ADDR command to switch to your application before examining or modifying your application's data or setting breakpoints in your application's code.

SoftICE automatically switches address contexts for your convenience under the following circumstances:

- If you use the TABLE command to switch to a 32-bit table, SoftICE automatically sets the current address context to the address context for that module.
- If you use the FILE command to display a source file from a 32-bit table, SoftICE sets the current address context to the address context for that module.
- If you use a symbol name in an expression, SoftICE changes the address context to the appropriate context. This includes export symbols loaded through Symbol Loader.

When you change address contexts, confusion might arise if you are viewing code or data located in the application's private address space (a linear address between 0x400000 to 0x7FFFFFFF for Windows 9x, and 0 to 0x7FFFFFFF for the Windows NT family). This occurs because the data or code that is displayed changes even though the selector:offset address do not. This is normal. The linear addresses remain the same, but the underlying system page tables now reflect the physical memory for the specified address context.

SoftICE does not allow you to specify an address context as part of an expression. If you are using bare addresses in an expression, be sure that the current address context is set appropriately. For example, D 137:401000 displays memory at 401000 in the current address context.

Caution: **Before you use bare addresses to set breakpoints, be sure you are in the correct address context. SoftICE uses the current context to translate addresses.**

Using INT 0x41 .DOT Commands

Under Windows 9x, Microsoft provides a set of extensions that allow a VxD or 32-bit DLL to communicate with a kernel-level debugger. (See the DEBUGSYS.INC file distributed with the Windows 9x DDK.) The .DOT API allows a VxD to provide VxD-specific debug information or command extensions interactively through the standard user interface of the kernel-level debugger. Although the API was originally designed for Microsoft's WDEB386, SoftICE supports a rich subset of the .DOT API. Thus, you can use SoftICE to access VMM and VxD .DOT commands, as well as any .DOT commands you might implement for your own VxD.

Caution: **The debug functionality for all .DOT extensions is built into VMM or another VxD. It is not part of SoftICE. SoftICE cannot guarantee that these extensions work correctly. Also, .DOT extensions might not perform error checking, which can lead to a system crash if invalid input is entered. Finally, SoftICE cannot determine whether or not a .DOT extension requires the system to be in a specific state. Using the .DOT extension at an inappropriate time might result in a system crash.**

SoftICE supports the following .DOT commands in Windows 9x:

- Registered .DOT extensions

  To get a list of registered dot commands, use the following command:

  ◇ **.?**

- Debug_Query .DOT extensions

  To invoke these .DOT handlers, type the VxD name after the dot. Most of these commands, if implemented, display menus. For example, the following VxDs have .DOT handlers in both the retail and debug versions of Windows 9x:

  - ◇ .VMM
  - ◇ .VPICD
  - ◇ .VXDLDR

  To determine if a VxD has a .DOT handler, try it. The .DOT handlers in the debug version of the DDK sometimes provide more functionality than the .DOT handlers in the retail version.

- VMM-embedded .DOT extensions

  VMM provides a variety of .DOT extensions that are available in both the debug and retail versions. To get a list of .DOT extensions supported by VMM, use the following command:

  ```
  ..?
  ```

  In the Windows 9x retail build, the ..? command yields the .DOT extensions shown in Table 6-1 on page 128.

Table 6-1. Win9x .DOT Extensions

.DOT Extension	Description
.R[#]	Displays the registers of the current thread.
.VM[#]	Displays the complete VM status.
.VC[#]	Displays the current VMs control block.
.VH[#]	Displays a VMM linked list, given list handle.
.VR[#]	Displays the registers of the current VM.
.VS[#]	Displays the current VMs virtual mode stack.
.VL	Displays a list of all VM handles.
.DS	Dumps protected mode stack with labels.
.VMM	Menu VMM state information.
.<dev-name>	Display device-specific information.

Understanding Transitions From Ring 3 to Ring 0

Many times when tracing into code using Windows 9x, you arrive at either an INT 0x30 or an ARPL. Both are methods for making a transition from Ring-3 to Ring-0. When you wish to follow the ring transition, you can save yourself the time and effort of stepping through a large amount of VMM code by using the G(o) command to execute up to the address shown in the disassembly.

Windows 9x uses the following methods to transition Ring-3 code to Ring-0 code:

- For V86 code, Windows 9x uses the ARPL instruction, which causes an invalid opcode fault. The invalid opcode handler then passes control to the appropriate VxD. The ARPL instruction is usually in ROM. Windows 9x uses only one ARPL and it varies the V86 segment:offset to indicate different VxD addresses. For example, if the ARPL is at FFFF:0, Windows 9x uses the addresses FFFF:0, FFFE:10, FFFD:20, FFFC:30 and so on.

  The following example shows sample output for disassembling an ARPL:

```
FDD2:220D     ARPL     DI,BP     ;     #0028:C0078CC9     IFSMgr(01)+0511
```

- For PM code, Windows 9x uses interrupt 0x30h. Segment 0x3B contains nothing but interrupt 0x30 instructions, each of which transfers control to a VxD.

  The following example shows sample output for disassembling segment:offset 3B:31A:

```
003B:031A     INT30     ;     #0028:C008D4F4     VPICD(01)+0A98
003B:031C     INT30     ;     #0028:C007F120     IOS(01)+0648
003B:031E     INT30     ;     #0028:C02C37FC     VMOUSE(03))00F0
003B:0320     INT30     ;     #0028:C02C37FC     VMOUSE(03))00F0
003B:0322     INT30     ;     #0028:C023B022     BIOSXLAT(05)=0022
003B:0324     INT30     ;     #0028:C230F98      BIOSXLAT(04)=0008
003B:0326     INT30     ;     #0028:C023127C     BIOSXLAT(04)=02EC
```

Chapter 7
Using Breakpoints

Introduction

You can use SoftICE to set breakpoints on program execution, memory location or I/O port reads and writes, interrupts, and module loads and unloads. SoftICE assigns a breakpoint index, starting from 0, to each breakpoint. You can use this breakpoint index to identify breakpoints when you set, delete, disable, enable, or edit them.

All SoftICE breakpoints are *sticky*, which means that SoftICE tracks and maintains a breakpoint until you intentionally clear or disable it using the BC or the BD command. After you clear breakpoints, you can recall them with the BH command, which displays a breakpoint history.

By default, the maximum number of breakpoints you can set at one time in SoftICE is limited to 32. The limit can be changed using the SoftICE configuration utility, or by placing an entry in `WINICE.DAT`, `BREAKPOINTS=xx`. Once SoftICE is started there is no way to increase the number of available breakpoints on the fly.

Note: The maximum number of memory location (BPMs) and I/O breakpoints (BPIOs) is a total of four, due to the number of available debug registers on x86 processors.

Where symbol information is available, you can set breakpoints using function names. When in source or mixed mode, you can set point-and-shoot style breakpoints on any source code line. A valuable feature is that you can set point-and-shoot breakpoints in a module before it is loaded.

Types of Breakpoints Supported by SoftICE

SoftICE provides a powerful array of breakpoint capabilities that take full advantage of the x86 architecture, as follows:

- **Execution Breakpoints:** SoftICE replaces an existing instruction with INT 3. You can use the BPX command to set execution breakpoints.
- **Memory Breakpoints:** SoftICE uses the x86 debug registers to break when a certain byte/word/dword of memory is read, written, or executed. You can use the BPM command to set memory breakpoints.
- **Interrupt Breakpoints:** SoftICE intercepts interrupts by modifying the IDT (Interrupt Descriptor Table) vectors. You can use the BPINT command to set interrupt breakpoints.
- **I/O Breakpoints:** SoftICE uses a debug register extension available on Pentium and Pentium-Pro CPUs to watch for an IN or OUT instruction going to a particular port address. You can use the BPIO command to set I/O breakpoints.
- **Window Message Breakpoints:** SoftICE traps 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 can use the BMSG command to set window message breakpoints.
- **Module Load/Unload Breakpoints:** SoftICE traps when it detects that a given module (specified by name) is loading or unloading from memory.

Breakpoint Options

SoftICE can accept command modifiers to limit the scope of a breakpoint for all breakpoint commands, including bpx, bpm, bpio, and bpint. Depending on the OS, the modifiers differ.

- Windows 9x allows modifiers of *.t*, *.p*, *.a*, and *.v*
- Windows NT family allow modifiers of *.t* and *.p*

If the currently executing process ID (PID) is 0x200 and you issue a bpint.p 2e within SoftICE, future int 2e breakpoints will get hit only if the executing process is 0x200. By contrast, issuing a command of bpint 2e will cause every single int 2e to pop-up SoftICE.

Table 7-1. SoftICE Command Modifiers

Command Modifier	Description
.t	Conditionally set the breakpoint to trigger in the active thread.
.p	Conditionally set the breakpoint to trigger in the active Process ID.
.a	Conditionally set the breakpoint to trigger in the active address context.
.v	Conditionally set the breakpoint to trigger in the active VMM ID.

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

- A conditional expression [IF `expression`]: The expression must evaluate to non-zero (TRUE) for the breakpoint to trigger. Refer to *Conditional Breakpoints* on page 141.
- A breakpoint action [DO “`command1;command2;`...”]: A series of 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 140.

Note: For complete information on each breakpoint command, refer to the *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 INT 3 instruction, SoftICE takes control when execution reaches the INT 3 breakpoint.

SoftICE provides two ways for setting execution breakpoints: using a mouse and using the BPX command. The following sections describe how to use these methods for setting breakpoints.

Using a Mouse to Set Breakpoints

If you are using a Pentium processor and a mouse, you can use the mouse to set or clear point-and-shoot (sticky) and one-shot breakpoints. To set a sticky breakpoint, double-click the line on which you want to set the breakpoint. SoftICE highlights the line to indicate that you set a breakpoint. Double-click the line again to clear the breakpoint. To set a one-shot breakpoint, click the line on which you want to set the breakpoint and use the HERE command (F7) to execute to that line.

Using the BPX Command to Set Breakpoints

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

```
BPX [address] [IF expression] [DO "command1;command2;..."]
```

IF expression	Refer to *Conditional Breakpoints* on page 141.
DO "command1;command2;..."	Refer to *Setting a Breakpoint Action* on page 140.

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

```
BPX WinMain
```

Use the BPX command without specifying any parameter to set a point-and-shoot execution breakpoint in the source code. Use Alt-C to move the cursor into the Code window. Then use the arrow keys to position the cursor on the line on which you want to set the breakpoint. Finally, use the BPX command (F9). If you prefer to use your mouse to set the breakpoint, click the scroll arrows to scroll the Code window, then double-click the line on which you want to set the breakpoint.

Memory Breakpoints

A memory breakpoint uses the debug registers found on the 386 CPUs and later models 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 can only set four memory breakpoints at one time, because the CPU contains only four debug registers.

Use the BPM command to set memory breakpoints:

BPM[B|W|D] *address* [R|W|RW|X] [*debug register*] [IF *expression*] [DO "*command1;command2;…*"]

BPM and BPMB	Set a byte-size breakpoint.
BPMW	Sets a word (2-byte) size breakpoint.
BPMD	Sets a dword (4-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.
debug register	Specifies which debug register to use. SoftICE normally manages the debug register for you, unless you need to specify it in an unusual situation.
IF expression	Refer to *Conditional Breakpoints* on page 141.
DO "command1;command2; …"	Refer to *Setting a Breakpoint Action* on page 140.

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

Use an interrupt breakpoint to trap an interrupt through the IDT. The breakpoint only triggers when a specified interrupt is dispatched through the IDT.

Use the BPINT command to set interrupt breakpoints:

```
BPINT interrupt-number [IF expression] [DO
"command1;command2;…"]
```

interrupt-number	Number ranging from 0 to 255 (0 to FF hex).
IF expression	See *Conditional Breakpoints* on page 141.
DO "command1;command2;…"	See *Setting a Breakpoint Action* on page 140.

If an interrupt is caused by a software INT instruction, the instruction displayed will be the INT instruction. (SoftICE pops up when execution reaches the INT instruction responsible for the breakpoint, but before the instruction actually executes.) Otherwise, the current instruction will be the first instruction of an interrupt handler. You can list all interrupts and their handlers by using the IDT command.

Use the following command to set a breakpoint to trigger when a call to the kernel-mode routine NtCreateProcess is made from user mode:

```
BPINT 2E IF EAX==1E
```

Note: The NtCreateProcess is normally called from ZwCreateProcess in the NTDLL.DLL, which is in turn called from CreateProcessW in the KERNEL32.DLL. In the conditional expression, 1E is the service number for NtCreateProcess. Use the NTCALL command to find this value.

You can use the BPINT command to trap software interrupts, for example INT 21, made by 16-bit Windows programs. Note that software interrupts issued from V86 mode do not pass through the IDT vector that they specify. INT instructions executed in V86 generate processor general protection faults (GPF), which are handled by vector 0xD in the IDT. The Windows GPF handler realizes the cause of the fault and passes control to a handler dedicated to specific V86 interrupt types. The types may end up reflecting the interrupt down to V86 mode by calling the interrupt handler entered in the V86 mode Interrupt Vector Table (IVT). In some cases, a real-mode interrupt is reflected (simulated) by calling the real-mode interrupt vector.

In the case where the interrupt is reflected, you can trap it by placing a BPX breakpoint at the beginning of the real-mode interrupt handler.

To set a breakpoint on the real-mode INT 21 handler, use the following command:

```
BPX *($0:(21*4))
```

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. SoftICE implements I/O breakpoints by using the debug register extensions introduced with the Pentium. As a result, I/O breakpoints require a Pentium or Pentium-Pro CPU. A maximum of four I/O breakpoints can be set at one time. The I/O breakpoint is effective in kernel-level (ring 0) code as well as user (ring 3) code.

Notes:

With Windows 9x, SoftICE relies on the I/O permission bitmap, which restricts I/O trapping to ring 3 code.

You cannot use I/O breakpoints to trap IN/OUT instructions executed by MS-DOS programs. The IN/OUT instructions are trapped and emulated by the operating system, and therefore do not generate real port I/O, at least not in a 1:1 mapping.

Use the BPIO command to set I/O breakpoints:

```
BPIO port-number [R|W|RW] [IF expression]
[DO "command1;command2;…"]
```

R, W, and RW	Break on reads (IN instructions), writes (OUT instructions), or both, respectively.
IF expression	See *Conditional Breakpoints* on page 141.
DO "command1;command2;…"	See *Setting a Breakpoint Action* on page 140.

When an I/O breakpoint triggers and SoftICE pops up, 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, or dword, 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.

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 window-handle [L] [begin-message [end-message]]
[IF  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.

L
Signifies that the window message should be printed to the Command window without popping into SoftICE.

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
See *Conditional Breakpoints* on page 141.

DO "command1;command2;…"
See *Setting a Breakpoint Action* on page 140.

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 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
```

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.

You can construct an equivalent BPX-style breakpoint using a conditional expression. Use the HWND command to get the address of the window procedure, then use the following BPX command (Win32 only):

```
BPX 5FEBDD12 IF (esp->8)==WM_NCPAINT
```

Caution: When setting a breakpoint using a raw address (not a symbol), it is vital to be in the correct address context.

Module Load/Unload Breakpoints

A module load breakpoint will cause SoftICE to pop up whenever it detects that a named module is about to load or unload. This command can be used with any module type: applications, DLLs, or drivers.

```
BPLOAD mod-name [L|U|B] [IF expression] [DO
“command1;command2;”]
```

mod-name	The name of the module to watch for. This should be an exact match.
L|U|B	Load, Unload, or Both. This parameter determines when SoftICE will pop up. The default value if none is specified is Load.
IF expression	See *Conditional Breakpoints* on page 141.
DO “command1;command2;…”	See *Setting a Breakpoint Action* on page 140.

When SoftICE pops up on a module load or unload, it does so in the midst of the kernel code that is loading or unloading the module. It is therefore possible to use module load breakpoints to watch the process being created by the kernel.

Understanding Breakpoint Contexts

A breakpoint context consists of the address context in which the breakpoint was set and in what code module the breakpoint is in, if any. Breakpoint contexts apply to the BPX and BPM commands, and breakpoint types based on those commands such as BMSG.

For Win32 applications, breakpoints set in the upper 2GB of address space are global; they break in any context. Breakpoints set in the lower 2GB are *context-sensitive*; they trigger according to the following criteria and SoftICE pops up:

- SoftICE only pops up if the address context matches the context in which the breakpoint was set.
- If the breakpoint triggers in the same code module in which the breakpoint was set, then SoftICE disregards the address context and pops up. This means that a breakpoint set in a shared module like KERNEL32.DLL breaks in every address context that has the module loaded, regardless of what address context was selected when the breakpoint was set.

Breakpoints set on MS-DOS and 16-bit Windows programs are context-sensitive in the sense that the breakpoint only affects the NTVDM process in which the breakpoint was set. The breakpoint never crosses NTVDMs, even if the same program is run multiple times.

Breakpoint contexts are more important for BPM-type breakpoints than for BPX. BPM sets an x86 hardware breakpoint that triggers on a certain virtual address. Because the CPU breakpoint hardware knows nothing of address spaces, it could potentially trigger on an unrelated piece of code or data. Breakpoint contexts give SoftICE the ability to discriminate between false traps and real ones.

Virtual Breakpoints

In SoftICE, you can set breakpoints in Windows modules before they load, and it is not necessary for a page to be present in physical memory for a BPX (INT 3) breakpoint to be set. In such cases, the breakpoint is *virtual*; it will be automatically armed when the module loads or the page becomes present. Virtual breakpoints can only be set on either symbols or source lines.

Setting a Breakpoint Action

You can set a breakpoint to execute a series of 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 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. Refer to *Working with Persistent Macros* on page 205 for more information about 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 you need to embed a literal quote character (") or a percent sign (%) within the macro (breakpoint) body, precede the character with a backslash character (\). To specify a literal backslash character, use two consecutive backslashes (\\).

If a breakpoint is being logged (refer to the built-in function *BPLOG* on page 145), 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, SoftICE will only trigger if the breakpoint evaluates to non-zero (TRUE). Because the SoftICE expression evaluator handles complex expressions easily, conditional expressions take you right to the problem or situation you want to debug with ease.

All SoftICE breakpoint commands (BPX, BPM, BPIO, BMSG, and BPINT) 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), SoftICE pop ups and displays the reason for the break, which includes the conditional expression.

The following examples show conditional expressions used during the development of SoftICE.

Note: Most of these examples contain system-specific values that vary depending on the exact version of the Windows NT family you are running.

- Watch a thread being activated:

  ```
  bpx ntoskrnl!SwapContext IF (edi==0xFF8B4020)
  ```

- Watch a thread being deactivated:

  ```
  bpx ntoskrnl!SwapContext IF (esi==0xFF8B4020)
  ```

- 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 thread state become terminated (enum == 4):

  ```
  bpmb _thread->29 IF byte(_thread->29) == 4)
  ```

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

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

- Watch a specific process make a system call:

  ```
  bpint 2E if (process == _process)
  ```

Many of the previous examples use the *thread* and *process* intrinsic functions provided by SoftICE. These functions refer to the active thread or process in the operating system. In some cases, the examples precede the function name with an underscore “_”. This is a special feature that makes it easier to refer to a dynamic value such as a register’s contents or the currently running thread or process as a constant. The following examples should help to clarify this concept:

- This example sets a conditional breakpoint that will be triggered if the dynamic (run-time) value of the EAX register equals its current value.

  ```
  bpx eip IF (eax == _eax)
  ```

  This is equivalent to:

  ```
  ? EAX
  00010022
  bpx eip IF (eax == 10022)
  ```

- This example sets a conditional breakpoint that will be triggered if the value of an executing thread's thread-id matches the thread-id of the currently executing thread.

  ```
  bpx eip IF (tid == _tid)
  This is equivalent to:
  ? tid
  8
  bpx eip IF (tid == 8)
  ```

When you precede a function name or register with an underscore in an expression, the function is evaluated immediately and remains constant throughout the use of that expression.

Conditional Breakpoint Count Functions

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
- BPLOG
- 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 5, so the conditional expression evaluates to TRUE and SoftICE pops up.

```
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 5, the full expression evaluates to TRUE and SoftICE pops up. If BPCOUNT != 5, the expression fails, BPMISS is incremented and SoftICE will not pop up (although BPCOUNT is now 1 greater).

Once the full expression returns TRUE, SoftICE pops up, 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 SoftICE pop up 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 SoftICE pops up. Otherwise, BPMISS is updated each time the conditional evaluates to FALSE. After 5 consecutive failures, the expression evaluates to TRUE and SoftICE pops up.

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 popup 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 SoftICE will not pop up. Every time after 50, the condition evaluates to TRUE, and SoftICE pops up 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 4, SoftICE pops up every fourth time the breakpoint triggers.

BPLOG

Use the BPLOG expression function to log the breakpoint to the history buffer. SoftICE does not pop up when logged breakpoints trigger.

Note: Actions only execute when SoftICE pops up, so using actions with the BPLOG function is pointless.

The BPLOG expression function always returns TRUE. It causes SoftICE to log the breakpoint and relevant information about the breakpoint to the SoftICE history buffer.

Any time the breakpoint triggers and the value of EAX equals 1, SoftICE logs the breakpoint in the history buffer. SoftICE will not popup.

```
bpx myaddr if ((eax==1) && bplog)
```

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 SoftICE to pop up. 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”
```

Note: 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

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). 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’ll 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 and initialized, as follows:

```
:BPX .3 if(foo==0)
```

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. To set a conditional at the end of the foobar function, use source line 12, as follows:

```
:BPX.12 if(fooTmp==1)
```

Note: 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 146. If, however, you are debugging without full symbol information, you need to reference function parameters and local variables on the stack. For example, if you translated a module with publics only or you want to debug a function for an operating system, reference function parameters and local variables on the stack.

Note: The following section is specific to 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:

```
PUSHEBP
MOVEBP,ESP
SUBESP,size (locals)
```

Which sets up a stack frame as follows:

	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	

- 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
  ```

  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: 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 or SwapContext), 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

SoftICE supports using the time stamp counter (RDTSC instruction) on all Pentium and Pentium-Pro machines. When SoftICE first starts, it displays the clock speed of the machine on which it is running. Every time SoftICE pops up due to a breakpoint, the elapsed time displays since the last time SoftICE popped up. The time displays after the break reason in seconds, milliseconds, or microseconds:

```
Break due to G (ET=23.99 microseconds)
```

The Pentium cycle counter is highly accurate, but you must keep the following two issues in mind:

1. There is overhead involved in popping SoftICE up and down. On a 100MHz machine, this takes approximately 5 microseconds. This number varies slightly due to caching and privilege level changes.
2. If a hardware interrupt occurs before the breakpoint goes off, all the interrupt processing time is included. Interrupts are off when SoftICE pops up, so a hardware interrupt almost always goes off as soon as the Windows NT family resumes.

Breakpoint Statistics

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 *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. 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

SoftICE provides a variety of commands for manipulating breakpoints such as listing, modifying, deleting, enabling, disabling, and recalling breakpoints. Breakpoints are identified by breakpoint index numbers, which are numbers ranging from 0 to FF (hex). Breakpoint index numbers are assigned sequentially as breakpoints are added. The breakpoint manipulation commands are described in Table 7-2 on page 151.

Table 7-2. SoftICE Breakpoint Manipulation Commands

Command	Description
BD	Disable a breakpoint.
BE	Enable a breakpoint.
BL	List current breakpoints.
BPE	Edit a breakpoint.
BPT	Use breakpoint as a template.
BC	Clear (remove) a breakpoint.
BH	Display breakpoint history.

Note: Refer to the *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 SoftICE. To embed a breakpoint in your program, do the following:

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

Chapter 8

Using Expressions

Expressions

The SoftICE expression evaluator (EE) determines the values of expressions used with SoftICE commands and conditional breakpoints. The expression evaluator uses a C-like syntax, with full operator precedence, arithmatic and logical operators, pointer operations, and a data type system. If you are comfortable with the syntax of C or C++, the SoftICE expression evaluator will feel quite natural and familiar.

The expression evaluator can operate on symbols defined in loaded symbol files. It also understands literal values and register names in an expression.

Other than the maximum length of a SoftICE command line (which is equal to the width of the command window), 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.

The SoftICE expression evaluator uses type information loaded from symbol files (NMS files) for all of its operations. Type information can come from any loaded symbol table, so the expression evaluator can display values with user-defined types correctly. If no symbol tables are loaded, the expression evaluator uses a small set of built-in basic types.

Typecasting is allowed by the expression evaluator using the standard C syntax, and any value may be cast to any known type within an expression.

Using the Expression Evaluator

Most of the SoftICE commands that take parameters use the expression evaluator to resolve them. This means that commands can operate on the results of complex expressions, which makes the command syntax quite powerful. For example, you can use the **DD** command to display the memory where a variable is located simply by entering `DD &foo` on the command line. The expression evaluator will evaluate `&foo`, and return an address, which the **DD** command will then display.

You can also access the expression evaluator directly by using the **?** command. When the expression evaluator is used in this way, the results are simply displayed in the command window.

Supported Operators

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

Table 8-1. SoftICE Pointer Operators

Pointer Operators	Name	Example
->	Pointer Operator	pstruct->element ebp->8 (equivalent to *((*ebp)+8))
.	Member Operator	struct.element eax.1c (equivalent to *(ebp+8))
*	Dereference	*pFoo *esi (gets the value pointed to by esi)
&	Indirection	&foo (gets the address of foo)
[ ]	Array Subscript	foo[2] (gets the second element of the array Foo)

Table 8-2. SoftICE Math Operators

Math Operators	Name	Example
+	Unary +	+42 (changes radix to decimal)
-	Unary -	-42 (negation, also changes radix to decimal)

Table 8-2. SoftICE Math Operators (Continued)

Math Operators	Name	Example
+	Addition	foo + 1 eax + 0x040
-	Subtraction	foo – bar eax – 40
*	Multiplication	foo * ecx al * 100
/	Division	foo / 5 bar / (eax << 4)
%	Modulo	ecx % 4 foo % bar
<<	Logical Left Shift	bl << 1
>>	Logical Right Shift	0x80000 >> 2

Table 8-3. SoftICE Bitwise Operators

Bitwise Operators	Name	Example
&	Bitwise AND	foo & ff esi & 2020
|	Bitwise OR	f000 | 10 eax | 80400000
^	Bitwise XOR	ebx ^ 0xFF
~	Bitwise NOT	~al ~(foo & 0xf)

Table 8-4. SoftICE Logical Operators

Logical Operators	Name	Example
!	Logical NOT	!eax !(foo == 1)
&&	Logical AND	foo && bar eax && (ebx > 5)
||	Logical OR	foo || (ebp.8)
==	Compare Equality	ecx == 4
!=	Compare Inequality	foo != 0
<	Less Than	ecx < A00
>	Greater Than	foo > bar

Table 8-4. SoftICE Logical Operators (Continued)

Logical Operators	Name	Example
<=	Less or Equal	esi <= &foo
>=	Greater or Equal	foo >= (bar % 1000)

Table 8-5. SoftICE Special Operators

Special Operators	Name	Example
.	Line Number	.123 (value is address of line 123 in the current source file)
( )	Grouping Symbols	(eax+4) * 8
(typename)	Typecast	(uchar)eax
:	Segment Override	fs:50 1b:40100
function()	Macro Function	WSTR(eax)

Pointer Operations

Pointer operations on symbols in the SoftICE expression evaluator function in exactly the same way as they do in C. The expression evaluator also allows registers to be used as pointers. Table 8-6 provides some additional descriptions for these operations.

Table 8-6. Pointer Operations

Operator	Example	Equivalent To...	Details
->	esp->4	*((*esp)+4)	The first operand is treated as a pointer-to-struct
	foo->bar	*((*foo)+offset of bar)	This is the familiar form, where foo is a pointer-to-struct (or class) and bar is an element of the structure. If foois not a pointer-to-struct, an error will be produced.
.	esp.4	*(esp+4)	The first operand is treated as a structure, and the offset is added to it before dereferencing. This is the correct form for retrieving parameters passed on the stack.
	foo.bar	*(foo+offset of bar)	In this case, foo must be a structure (or class), and bar an element of that structure.
*	*esp	*DS:esp	Returns the value pointed to by the stack pointer. Since no selector was specified, the current value of DS is used.
	*foo	*DS:foo	If foo is a pointer, this returns the value it points to. If foo isan integer, the expression evaluator will attempt to use the value of foo as a pointer.
	*fs:0	*fs:0	Returns the value at the specified selector:offset address.

Table 8-6. Pointer Operations (Continued)

Operator	Example	Equivalent To...	Details
&	&foo	&foo	Returns the address of foo.
	&foo.bar	&(*(foo+offset of bar))	Returns the address of element bar in structure foo.
	&esp	Illegal	Registers do not have addresses, so this is an error.
	&0xff	Illegal	Literals do not have addresses, so this is also illegal.
[]	foo[4]	foo[4]	Returns the value of the fifth element in array foo. If foo is not an array type, an error is generated.
	foo[4][2]	foo[4][2]	This will work, as long as foo was declared as a multi-dimensional array.
	esp[2]	Illegal	The array operator cannot be used on registers.

Operator Precedence

Operator precedence in the SoftICE expression evaluator is the same as that of C, with the addition of the special SoftICE operators. As with C, operator precedence can be overridden by grouping operations within parentheses; the expressions within parentheses are always evaluated before the expressions outside.

Table 8-7 lists all the operators in order of precedence, with highest precedence at the top of the table. Operators with the same precedence are evaluated according to associativity.

Table 8-7. SoftICE Operator Precedence

Operator	Associates	Comment
( ), function(), [ ] ->,.,#	left-to-right	grouping, macro function, array subscript pointer operator, member operator, symbol name override
:	left-to-right	selector override
*, & unary + unary - !, ~ .	right-to-left	dereference, indirection default radix = decimal negation, default radix = decimal logigal and binary negation line number
(typename)	right-to-left	typecast
*, /, %	left-to-right	multiplication, division, modulo
+, -	left-to-right	addition, subtraction
<<, >>	left-to-right	shift

Table 8-7. SoftICE Operator Precedence (Continued)

Operator	Associates	Comment
<, >, <=, >=	left-to-right	comparisons
==, !=	left-to-right	equality comparisons
&	left-to-right	bitwise AND
^	left-to-right	bitwise XOR
|	left-to-right	bitwise OR
&&	left-to-right	logical AND
||	left-to-right	logical OR

Forming Expressions

The SoftICE expression evaluator accepts three basic types of operands: registers, literals, and symbols. In general, these types can be used interchangeably in expressions; anywhere a register can be used, a symbol or literal is also valid. There are some exceptions: for example, taking the address of a register or literal is meaningless and will result in an error.

Registers

The expression evaluator understands all of the registers in the standard x86 register set, as well as a few of the x86 control registers. Where registers have multiple aliases (such as al, ah, ax, eax), the expression evaluator will return the correct data from the correct position in the register. Register names in expressions are not case sensitive. Table 8-8 displays all of the register names understood by the expression evaluator.

Table 8-8. Register Names Understood by the Expression Evaluator

Form 1	Form 2	Form 3	Form 4
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL
ESI	SI		
EDI	DI		

Table 8-8. Register Names Understood by the Expression Evaluator (Continued)

Form 1	Form 2	Form 3	Form 4
EBP	BP		
ESP	SP		
EIP	IP		
CS			
DS			
SS			
ES			
FS			
GS			
EFL (EFLAGS)	FL (FLAGS)		
CR2			
CR3			

The EFL and FL register names can be used to retrieve the values of the EFLAGS register, but the expression evaluator also includes some built-in aliases for individual flags which can be used in expressions. For more information, refer to the section on Built-in Functions on page 164.

Register names take precedence over any symbols which may have the same names. An override character, ‘#’, can be pre-pended to a symbol name to force the expression evaluator to treat the name as a symbol, rather than a register.

Literals

The SoftICE expression evaluator has two basic types of literals: numeric values and character constants. The two are interchangeable within expressions, but the syntax for entering them is different.

Internally, the expression evaluator uses a 64-bit representation for storing literals, so the maximum value that can be entered as a literal is 2^{64}-1, or 0xFFFFFFFF FFFFFFFF.

Numeric literals are entered as simple integers in an expression. The default radix used by the expression evaluator is hexadecimal, so hex numbers can be entered in expressions without preceding them with ‘0x’. This differs from standard C syntax, where numbers are assumed to be decimal unless preceded by ‘0x’.

Decimal numbers can be entered using the unary '+' and '-' operators, which change the radix of the number that follows to decimal (unless the number that follows includes hex characters or is preceded by '0x'.)

Table 8-9 illustrates how the expression evaluator will interpret numeric literal inputs.

Table 8-9. Numeric Literal Interpretation

Input	Interpretation	Notes
FF	Hex	Default interpreation for numbers is hexadecimal.
123	Hex	
0x123	Hex	
+42	Decimal	Unary+ overrides the default radix.
-42	Decimal	Unary- overrides the default radix.
+1A	Hex	Hexadecimal digits cancel the unary+ override.
-0x12	Hex	0x prefix cancels the unary- override.

Character constants are entered using single-quotes. The internal representation of a character constant is also 64-bits wide, so a single character constant may contain up to 8 bytes of data, all of which will be converted into a single integer value. Character values can be entered as characters, which are translated into their ASCII representations; decimal values up to three digits long preceded by '\', or hex values up to two digits long preceded by '\x'. Character constants with multiple bytes are translated to integers, with the first byte in the constant becoming the least-significant byte of the integer.

Character constants can also contain most of the common character escape sequences used in C. These all consist of a backslash followed by a single character, as shown in Table 8-10.

Table 8-10. Common Character Escape Sequences

Sequence	ASCII Value (Hex)
\a	Bell (0x7)
\b	Backspace (0x8)
\f	Formfeed (0xC)
\n	Newline (0xA)
\r	Carriage Return (0xD)

Table 8-10. Common Character Escape Sequences (Continued)

Sequence	ASCII Value (Hex)
\t	Tab (0x9)
\v	Vertical Tab (0xB)
\?	Question Mark (0x3F)
\’	Single Quote (0x27)
\”	Double Quote (0x22)
\\	Backslash (0x5C)

Table 8-11 shows some valid character constants.

Table 8-11. Valid Character Constants

Input	Notes
‘T’	Result is the ASCII value of the character ‘T’, or 0x54.
‘\84’	Result is the ASCII character with the decimal value of 84, or ‘T’.
‘\x54’	Result is the ASCII character with the hex value of 0x54, or ‘T’.
‘ABCD’	Result is the DWORD value of the four bytes, with ‘A’ as the least-significant byte, or 0x44434241.
‘ABCDEFGH’	Result is the QWORD value of the 8 bytes, with ‘A’ as the least-significant byte, or 0x4847464544434241.
‘\85\x54’	Result is ‘TU’.
‘\n’	Result is newline (0xA).

Symbols

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

Symbol names in 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 and C++ symbol names. SoftICE therefore accepts a wider range of characters in symbol names than a compiler would. Table 8-12 shows the characters which may be found in a legal symbol name. Symbols must begin with one of the characters marked as valid first characters in a symbol.

Table 8-12. Characters Contained in a Legal Symbol Name

Characters	Valid as First Character of Name?
A through Z a through z	Yes
0 through 9	No
@	Yes
$	Yes
_ (underscore)	Yes
` (single back-quote)	Yes
´ (single quote)	No
?	No
!	No
~	No
<...> (template syntax)	No
:: (scope operator)	No
operator (operator override syntax)	No

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 can be evaluated properly.

Template syntax is also allowed to appear in symbols. The type names contained within the template delimiters '<' and '>' are considered part of the symbol name. Commas are allowed to appear in templates as well.

Operator overrides are also allowed; in this case the last character of the symbol is allowed to be any of the valid C++ operators.

If a symbol has the same name as one of the register names, the expression evaluator will always return the value in the register when the symbol name is given. The command, `? ds`, for example, will return the contents of the DS register, not the value of any symbol of that name. In this case, a special override, '#', must be pre-pended to the symbol name to retrieve the symbol value. The command, `? #ds`, will return the value of a symbol named 'ds'.

Each symbol file loaded into 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 SoftICE Command Reference for more information on changing the active table.)

To specify a symbol from an inactive symbol table in an expression, 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 loaded from export tables or defined by the NAME command are always active, because SoftICE treats these symbol sources as a homogenous unit.

Note: Symbol tables now do time and date checking to ensure that the symbol files are up to date with the binary in use. Correct any discrepancies by retranslating and reloading the symbol table.

Symbol Sources and Search Order

When searching for a symbol used in an expression, SoftICE follows the same scope convention used in C/C++: locals are searched first, then globals. If both of these searches fail, SoftICE will search in any loaded export tables, then the user-defined names, and finally the list of built-in functions (see the next section). Locals which are not currently in-scope cannot be used in expressions, except when setting a conditional breakpoint. SoftICE will stop searching as soon as it finds a match, so local symbols will override globals with the same name, just as they do in C. The following shows the symbol search order used by the expression evaluator:

- Local Symbols in the current Scope
- Global Symbols
- Exports from any Loaded Export Table
- User-Defined Names (from the NAME command)
- Built-in Functions (refer to Built-in Functions on page 164.)

If the symbol is not found in any of these places, and it contains only valid hex digits, SoftICE will interpret it as a literal; otherwise, the expression evaluator will return an error.

Symbols which are not yet instantiated, meaning that the module that contains them is not currently loaded into memory, cannot be used in expressions (except when setting a conditional breakpoint). Uninstantiated symbols can be viewed using the SYM or EXP commands.

When SoftICE searches any of the symbol sources shown above, it uses a case-insensitive search routine by default. Case sensitivity for symbol searching can be turned on or off using the SET CASESENSITIVE [on | off] command. In addition, if a symbol search in the locals, globals, or exports fails, SoftICE will add a leading underscore to the symbol and repeat the search. This will find user symbols that have had standard C name decoration added, where `foo` becomes `_foo` after compilation.

Built-in Functions

The SoftICE expression evaluator has a number of built-in functions which can be used in expressions. Most of the built-in functions are simply aliases for useful values within SoftICE or the operating system; some take the form of function-style macros that translate data or return information about a parameter.

The simple aliases behave exactly like symbols: when used in an expression, the expression evaluator will translate the symbol into a value of the correct type. The function-style macros use the C-style function syntax: the name of the macro is followed by a parameter enclosed in parentheses. All of the function-style macros take exactly one parameter.

Table 8-13. SoftICE Predefined Functions

Name	Description	Example
FALSE	0	? eax == FALSE
TRUE	1	? foo == TRUE
NULL	0	? pFoo != NULL
CFL	Carry FLag	
PFL	Parity Flaf	
AFL	Auxiliary Flag	
ZFL	Zero Flag	
SFL	Sign Flag	
OFL	Overflow Flag	
RFL	Resume Flag	
TFL	Trap Flag	
DFL	Direction Flag	

Table 8-13. SoftICE Predefined Functions (Continued)

Name	Description	Example
IFL	Interrupt Flag	
NTFL	Nested Task Flag	
IOPL	Current I/O Privilege Flag	
VMFL	Virtual Machine Flag	
IRQL	Windows OS IRQ Level	
DataAddr	Returns the address of the first data item displayed in the active Data window	
CodeAddr	Returns the address of the first instruction displayed in the Code window	
EAddr	Effective address, if any, of the current instruction	
EValue	Current value of the effective address	
Process	Kernel Process Environment Block (KPEB) of the active OS process	
Thread	Kernel Thread Environment Block (KTEB) of the active OS thread	
PID	Active process ID	
TID	Active thread ID	
BPCount	Breakpoint Instance Count. For these BP functions, refer to *Conditional Breakpoint Count Functions* on page 143	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
BPLog	Breakpoint silent log	bp <bp params> IF bplog
BPIndex	Current Breakpoint Index #	bp <bp params> DO “bd bpindex”
BPx	Address of the specified breakpoint, if set	? BP2
BYTE()	Get low-order byte	? Byte(0x1234) = 0x34
WORD()	Get low-order word	? Word(0x12345678) = 0x5678
DWORD()	Get low-order dword	? Dword(10000000B) =0xB
HIBYTE()	Get high-order byte	? Hibyte(0x1234) = 0x12

Table 8-13. SoftICE Predefined Functions (Continued)

Name	Description	Example
HIWORD()	Get high-order word	? Hiword(0x12345678) = 0x1234
HIDWORD()	Get high-order dword	? Hidword(10000000B) = 0x1
SWORD()	Convert signed byte to signed word	? Sword(0x80) = 0xFF80
LONG()	Convert signed byte or word to signed long	? Long(0xFF) = 0xFFFFFFFF
WSTR()	Display as Unicode string	? WSTR(eax)
FLAT()	Convert a selector-relative address to a linear (flat) address	? Flat(fs:0) = 0xFFDFF000
SIZEOF()	Returns the size of the type of the specified parameter	? Sizeof(foo)

Eaddr Function

The EAddr function returns the effective address, if any, that the instruction at the current EIP uses. (The effective address of the current instruction, if any, and the value at that address are also displayed in the Register Window directly beneath the flags).

The x86 processor supports a variety of addressing modes such as register+offset and register+register. The result of computing the memory address for an instruction is called the effective address. An instruction that uses a memory addressing mode is said to have an effective address as its source or destination.

Some instructions do not involve an effective address, either because only registers are used or because the memory addressing is done in a way specific to the instruction type, such as with PUSH and POP instructions. An x86 instruction can never have an effective address as both source and destination.

For example, if the current instruction is:

```
MOV ECX,[ESP+4]
```

The EAddr function returns a value equal to `ESP+4`, that is, the current stack pointer plus 4.

If the current instruction is:

```
ADD BYTE PTR [ESI+EBX+2],55
```

EAddr returns the result of `ESI+EBX+2`.

Evalue Function

EValue returns the value at the effective address, if any, of the current instruction. This is not necessarily the same as `EAddr.0`, because EValue is sensitive to the operand size. EValue returns a byte, word, or dword as appropriate.

Expression Evaluator Type System

The SoftICE expression evaluator uses type information loaded from symbol files (NMS files) for all of its operations. Type information is used in both the evaluation and display of expression results.

While SoftICE understands all the defined types for a module, the goal of the expression evaluator is to be flexible rather than strict. Wherever possible, the expression evaluator makes reasonable assumptions to allow expressions to produce a result. For example, basic arithmetic operations on void pointers are permitted by the expression evaluator. Member operations on registers, for example `? ebp.4`, are also allowed to succeed – this is a convenient syntax for retrieving parameters to functions when source is not available.

Register values and literals used in expressions will not have an associated type. In this case the expression evaluator will use a basic unsigned integer type of the appropriate size. For registers, a BYTE, WORD, or DWORD type is used; for literals DWORD is the default type, unless the value is larger than the maximum value of a DWORD, in which case a 64-bit QUAD type is used.

Results Types

The SoftICE expression evaluator will determine the type of the result from the types of the operands. For example, given an array of integers declared like this:

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

The expression:

```
? TinyArray[ 1 ]
```

will cause 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
```

There are some cases where the expression evaluator will produce a different result type than a compiler would.

Taking the address of a symbol in the expression evaluator (&foo), will produce a result with a type of PVOID, because the expression evaluator does not have the ability to synthesize a pointer type to a user-defined type. You can overcome this limitation by casting the result to the desired type, if one is defined in your symbol table.

Where arithmetic is used on symbols of differing types, the expression evaluator does not do type promotion as a compiler would. Instead it uses the type of the left-hand operand as the result type. For example, in this expression:

```
? ’t’ + 1
```

The type of the first operand (t) is BYTE (an unsigned, 8-bit value), and the type of the second operand is DWORD (32-bit unsigned). The resulting type will be BYTE, not DWORD as you might expect. If a DWORD result is required, you can either reverse the order of the operands, or cast the result to a DWORD. Internally, the expression evaluator evaluates all arithmetic operations on 64-bit values, so typecasting the result will produce the correct answer even if the result overflows a BYTE-sized value.

In practice these differences between the expression evaluator and a compiler are generally unimportant.

Typecasting

The expression evaluator supports C-style typecasting, in the form of ‘(type-name)foo’. Unlike C, the expression evaluator does not alter the binary representation of the value when doing typecasting, so it does not handle sign extension (see the section “Built-in Functions” on page 164 for some routines that do).

Typecasting can be used to cast any value to any type in the active symbol table. Type names used in typecasting are always case-sensitive, so be sure you have entered the type name verbatim. There are also some default type names which are available even if no symbol table is loaded. These are shown in Table 8-14.

Note: A user-defined type or symbol of the same name in the active symbol table will override these default typecasts.

Table 8-14. Default Typecasts

Type-name	Result
BYTE	unsigned 8-bit
UCHAR	unsigned 8-bit
CHAR	signed 8-bit
WORD	unsigned 16-bit
USHORT	unsigned 16-bit
SHORT	signed 16-bit
DWORD	unsigned 32-bit
ULONG	unsigned 32-bit
LONG	signed 32-bit
UQUAD	unsigned 64-bit
ULONGLONG	unsigned 64-bit
QUAD	signed 64-bit
LONGLONG	signed 64-bit
PVOID	pointer to 32-bit value

The pointer and member operators, '->' and '.', can operate on expressions after typecasting. So `? ((MYSTRUCT)fedc0000).member` is a valid expression if 'member' is found in the 'MYSTRUCT' type.

Result Formats

When the ? command is used to evaluate an expression, the expression evaluator displays the results differently depending on the result type. Structures and classes are expanded, and all their data elements are displayed. Pointers to values or to complex types are dereferenced and the value or type pointed to is displayed along with the pointer value. Basic integer types are displayed in three different formats: hex, decimal, and a character representation. Strings of characters or wide characters are displayed up to their trailing NUL characters. Arrays are displayed up to their maximum dimensions.

The format used to display typed values can be changed using the SET TYPEFORMAT command. The options represent different arrangements of the type information, the symbol name, and the value. See the SET command documentation in the SoftICE Command Reference for more information.

Chapter 9

Loading Symbols for System Components

Loading Export Symbols for DLLs and EXEs

Exports are an aspect of the 16-bit and 32-bit Windows executable formats that enable dynamic (run-time) linking, usually between an executable that imports the functions and a .DLL that exports the functions.

The information in the executable file format associates an ASCII name and an ordinal number, or sometimes just an ordinal number, to an entry point in the module. It is advantageous to load the export information as symbols into the debugger, particularly when debugging information is not available. Exports are ordinarily used only by DLLs, but occasionally an .EXE may have exports as well; NTOSKRNL.EXE is such a case.

You can set the SoftICE initialization settings to load export symbols for any 16-bit or 32-bit .DLL or .EXE. When SoftICE loads, it loads the export files and makes their symbols available for use in any SoftICE expression. They are also automatically displayed when disassembling code. To see a list of all exported symbols that SoftICE knows about, use the EXP command. Refer to *Modifying SoftICE Initialization Settings* on page 191 for more information about pre-loading exports.

When displaying 32-bit exports in SoftICE, if the module is not yet loaded, the ordinal segment displays as FE: and the offset is the offset from the 32-bit image base. Once the module is mapped into any process, selector:offset appears. The offset now contains the image base address added in.

When a 32-bit module is unloaded from all processes that might have opened it, all addresses return to the ordinal FE:offset address.

Note: When a .DLL is mapped into two processes at different base virtual addresses, the export table uses the base address of the first process to open the .DLL, but the addresses will be wrong for the other. You can normally avoid this by choosing an appropriate preferred load address for the .DLL or by rebasing the .DLL.

The only 16-bit exports loaded are those from the non-resident export section; this is usually most or all of the exports for the module.

Using Unnamed Entry Points

For 32-bit exports, SoftICE shows all exported entry points even if they do not have names associated with them. For 16-bit exports, SoftICE only shows names. For exported entry points without names, SoftICE forms a name in the following format:

ORD_*xxxx*

where *xxxx* is the ordinal number.

Names of this form can overlap, because multiple DLLs can have unnamed ordinals. To be sure you are using the correct symbol, precede the symbol with the module name followed by an exclamation point.

To refer to KERNEL32 export ordinal number one, use the following expression:

```
KERNEL32!ORD_0001
```

The number following the ORD_ prefix does not require the correct number of leading zeroes; either ORD_0001 or ORD_1 is acceptable. The following expression is equivalent to the preceding example:

```
KERNEL32!ORD_1
```

Using Export Names in Expressions

SoftICE searches all 32-bit export tables prior to searching 16-bit export tables. This means that if the same name exists in more than one type of table, SoftICE uses the 32-bit export table. If you need to override this behavior, precede the export symbol with the module name followed by an exclamation point.

When specifying the symbol GlobalAlloc, SoftICE uses the 32-bit export symbol from KERNEL32.DLL rather than the 16-bit export symbol of the same name in KRNL386.EXE. You can access the 16-bit version of GlobalAlloc by specifying the complete export symbol name:

```
KERNEL!GlobalAlloc
```

Also, for each type of export (32-bit and 16-bit), the search order is controlled by the order in which the exports are loaded.

Loading Exports Dynamically

To load 32-bit exports dynamically, do the following:

1. Start Symbol Loader.
2. Either choose LOAD EXPORTS from the File menu or click the LOAD EXPORTS button.
3. The Load Exports window appears.
4. Select the files you want to load and click OPEN.

Using the Windows NT family Symbol Files with SoftICE

Microsoft supplies debugging information for most Windows NT family components. You can find the debug information on the Windows CD-ROM, or as a download from Microsoft. The Symbol Retriever tool, included with SoftICE, is a convenient way of retrieving symbol information directly from Microsoft's public symbol server for any given system component.

In older versions of the operating system, Microsoft supplied debug information in the form of .DBG files, which contained COFF debug data for the corresponding component. Since Windows 2000, debug information has been available in the form of .PDB files, which are in Microsoft's Program Database format. The procedure for loading symbol information from these two file formats is slightly different.

To load .DBG files into SoftICE, use Symbol Loader to translate the file into an .NMS file and load it. To load a .PDB file into SoftICE, open the *module itself* with Symbol Loader, then translate to an .NMS file and load. If the symbol file path is set up correctly, Symbol Loader will find the correct .PDB file automatically and translate it.

Symbol files need to be translated to .NMS files only once, unless the module in question changes. Once translated, .NMS files can be loaded quickly and simply by double-clicking on them in an Explorer window. SoftICE can also load .NMS files automatically on startup; you can add files to this list using the Settings application.

Using Windows 9x Symbol (.SYM) Files with SoftICE

The Windows 9x DDK includes symbol information for some system modules in the form of .SYM files. Use either Symbol Loader or NMSYM to translate the .SYM files into NMS format and load them into SoftICE

Chapter 10

Remote Debugging with SoftICE

Introduction

There may be times during the development process when you need SoftICE to do more than single-machine debugging, and remote debugging is required. For example, you may want to debug OpenGL/Direct 3D programming, Video playback, or a Video Display Driver, and the machine being debugged is located in another office, at a customers site, or on the other side of the world. For this type of debugging situation, SoftICE provides an extensive array of remote debugging options.

This chapter describes the types of remote connections available and how to configure SoftICE for each connection type.

Types of Remote Connections

SoftICE offers remote debugging through the following methods:

- Direct Null Modem connection.
- Dial-up Modem.
- Network Interface Card (NIC) interface. With the NIC option, you have the ability to debug any machine that has an IP address with the proper configuration and connection.

Through all types of remote connection, the SoftICE screen remains visible on the target computer, unless one additional step is taken. (For definition purposes, the **target computer** is the computer that has the SoftICE debugger running on it. This is the machine that is being debugged. The **host computer** is the machine that runs the SoftICE front end, siremote.exe.)

To prevent the SoftICE screen from being visible on the target computer, change the SoftICE configuration option to "Headless Mode" using the DriverStudio Configuration dialog SoftICE Initialization General Settings page. Remember that setting this option to "Headless Mode" will prevent the input devices on the target from functioning.

Alternatively, you could go to the registry and change the entry at `HKLM\System\CurrentControlSet\Services\Ntice` titled "NullVGA." Set the value of NullVGA to 1, and reboot. This will allow input on the target computer while preventing the display of the SoftICE screen.

Which type of remote connection is right for me?

This depends upon many factors, the first of which is *location*. If the target computer is at a **remote location**, your options are either debugging over a network, or debugging over a dial-up modem. If the target machine is a local machine (i.e., located in the same office), then serial debugging or local network (LAN) debugging is most appropriate.

What are the advantages/disadvantages for each type of connection?

The following table lists the connection advantages and disadvantages .

Table 10-1. Connection Advantages/Disadvantages

Connection Type	Advantage	Disadvantage
Serial Connection	No additional hardware required other than null modem cable. Decent performance.	Machine must be located within reach of null modem cable. Performance at slower connection speeds. Not supported in the DriverStudio Remote Data extension.

Table 10-1. Connection Advantages/Disadvantages (Continued)

Connection Type	Advantage	Disadvantage
NIC – Universal Network Driver	Performance close to that of single machine debugging. Ability to debug any machine at one location. Ability to debug over the internet through tcp/ip protocol (firewall restrictions and ip limitations apply). Uses any PCI based NIC card. Can be used for boot time debugging. Full support of the DriverStudio RemoteData NameSpace Extension.	Firewall's get in the way (can be circumvented with VPN, SSH). Machines may need to be on same subnet. Network performance can decrease if using the SIVNIC (SoftICE Virtual NIC) (additional details below).
NIC – Specialized Network Drivers	Performance close to that of single machine debugging. Does not interfere with normal network traffic. Ability to debug any machine connected to your local subnet, as well as machines directly connected to the Internet. Full support of the DriverStudio RemoteData NameSpace Extension.	Cannot be used to debug early boot time drivers. Requires one of 3 classes of network cards.
Modem	Can connect to any machine that has a modem. Firewalls are not a concern.	Slow. Modem hardware must be present in both machines. Phone line is tied up.

DSR Namespace Extension

Both DriverStudio and the SoftICE Driver Suite have a desktop feature called the DriverStudio Remote Data (DSR) namespace extension.

Figure 10-1. DSR Namespace Extension

Clicking this feature displays a Remote Data environment similar to that shown in Figure 10-2.

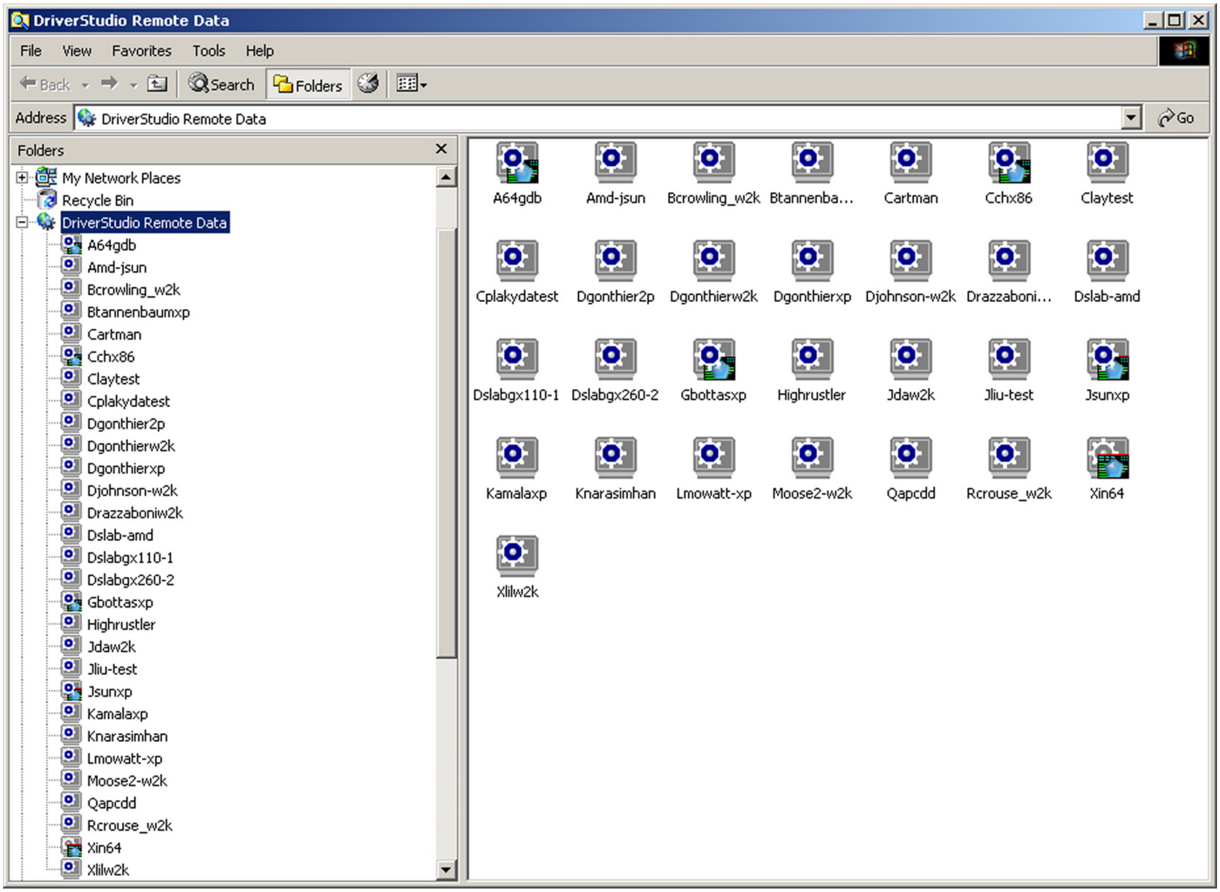


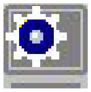
Figure 10-2. Typical Remote Data Environment for Debugging

This environment allows you to monitor the status of your entire network from one location. From this one location you can start SoftICE, change configuration parameters for all tools in the suite, connect to a remote machine, collect BoundsChecker, TrueCoverage™, TrueTime®, and Crash Dump files, and finally debug that machine with SoftICE.

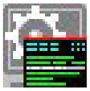
Note: In order to debug a remote machine through the DriverStudio Remote Data environment, you will need to have either the UND or the Specialized network drivers installed.

Remote Target State Icons

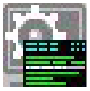
In the Remote Data environment, the following icons represent all the possible remote target states that would be encountered during normal use. These state icons appear in the leftmost column of a detail view in the DriverStudio Remote Data folder. They also appear in list, small icon, and large icon folder views.



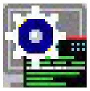
DriverStudio Tools:	Available
Debugger State:	Not available
Operating System State:	Running



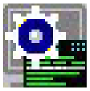
DriverStudio Tools:	Not available
Debugger State:	Busy
Debugger Type:	SoftICE
Operating System State:	Stopped



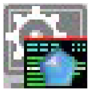
DriverStudio Tools:	Not available
Debugger State:	Ready
Debugger Type:	SoftICE
Operating System State:	Stopped



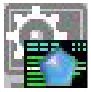
DriverStudio Tools:	Available
Debugger State:	Busy
Debugger Type:	SoftICE
Operating System State:	Running



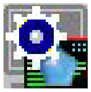
DriverStudio Tools:	Available
Debugger State:	Ready
Debugger Type:	SoftICE
Operating System State:	Running



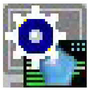
DriverStudio Tools:	Not available
Debugger State:	Busy
Debugger Type:	Visual SoftICE
Operating System State:	Stopped



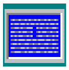
DriverStudio Tools:	Not available
Debugger State:	Ready
Debugger Type:	Visual SoftICE
Operating System State:	Stopped



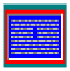
DriverStudio Tools:	Available
Debugger State:	Busy
Debugger Type:	Visual SoftICE
Operating System State:	Running



DriverStudio Tools:	Available
Debugger State:	Ready
Debugger Type:	Visual SoftICE
Operating System State:	Running



DriverStudio Tools:	Not available
Debugger State:	Ready
Operating System State:	Stopped



DriverStudio Tools:	Not available
Debugger State:	Busy
Operating System State:	Stopped

By right-clicking on an icon, you can choose to change the options, start SoftICE, or reboot the machine. By default, the folder view contains static information from a snapshot at a given point in time.

It is possible to refresh the display manually by choosing "View Refresh" or by specifying an interval of time. To set the time interval, first right-click on the desktop DSR feature. Then, choose Properties, and select the Refresh Rate.

Remote Debugging Details

Each type of networking has certain requirements and may require preparation steps. Please be sure to follow all directions closely.

Specialized Network Drivers

Description

The specialized network drivers offer the best in all-around performance with minimal intrusion upon the system and network stacks. However, their limitations may preclude you from using them. The two main limitations are:

1. They cannot be used for early boot-mode debugging, and
2. You must use one of the three supported classes of network cards.

The specialized network drivers will run on all Windows NT based operating systems as well as the Win9x based operating systems.

Hardware Requirements

A network card based on any of the three classes of network cards:

- Novell NE2000 series of cards
- 3com 3c90x series of cards, including the 3C905, 3C900, 3C920, 3C921, and all variants of those cards
- Intel E100 series of cards.

Installation

Installation and removal is straight forward.

To install the specialized network drivers:

1. Go to **Control Panel**.
2. Choose **Networking and Dial-up Connections**.
3. Right click on **Local Area Connection**.

4 Choose **properties**.

5 Click on **Configure**.

6 Click on **Driver**.

7 Click on **Update Driver**.

8 Click on **Next**.

9 Choose **Specify a location**

10 Browse to your \program files\compuware\driverstudio\softice\network\ folder, and choose the appropriate subfolder. From here, choose the appropriate.inf file: i.e., nt4, win9x (oemxxxx.inf) or filename.inf (for Win2K and later platforms).

If any messages appear regarding "Driver Signing," these messages can be safely ignored.

11 After installation is complete, reboot your computer.

Establishing a Connection

Establishing a connection for the specialized network drivers is identical to that for the Universal Network Driver. (See "Universal Network Driver" on page 183.)

Removal

Use the following procedure to uninstall the specialized network drivers.

1 Go to **Control Panel**.

2 Choose **Networking and Dial-up Connections**.

3 Right-click on **Local Area Connection**.

4 Choose **properties**.

5 Click on **Configure**.

6 Click on **Driver**.

7 Click on **Update Driver**.

8 Click on **Next**.

9 Choose **Search for a suitable driver for my device**. Follow the prompts from there.

Universal Network Driver

Description

The Universal Network Driver (UND) works on all PCI based network cards for the Windows 2000, Windows XP (and later) Operating Systems. Two drivers are supplied with the UND. The first driver allows SoftICE to interact with the networking card. This driver prevents normal network traffic, e-mail, web browsing, or file sharing to occur on that NIC card. To get around this limitation we suggest using a second network card which is dedicated to SoftICE. If this is impractical, we provide an additional driver called the SoftICE Virtual NIC (SIVNIC). This driver allows the NIC to be shared between SoftICE and normal Windows networking.

Note: You will notice a decrease in Windows networking performance when using the SIVNIC. As such, it is suggested that you install a second network card that is for the exclusive use of SoftICE.

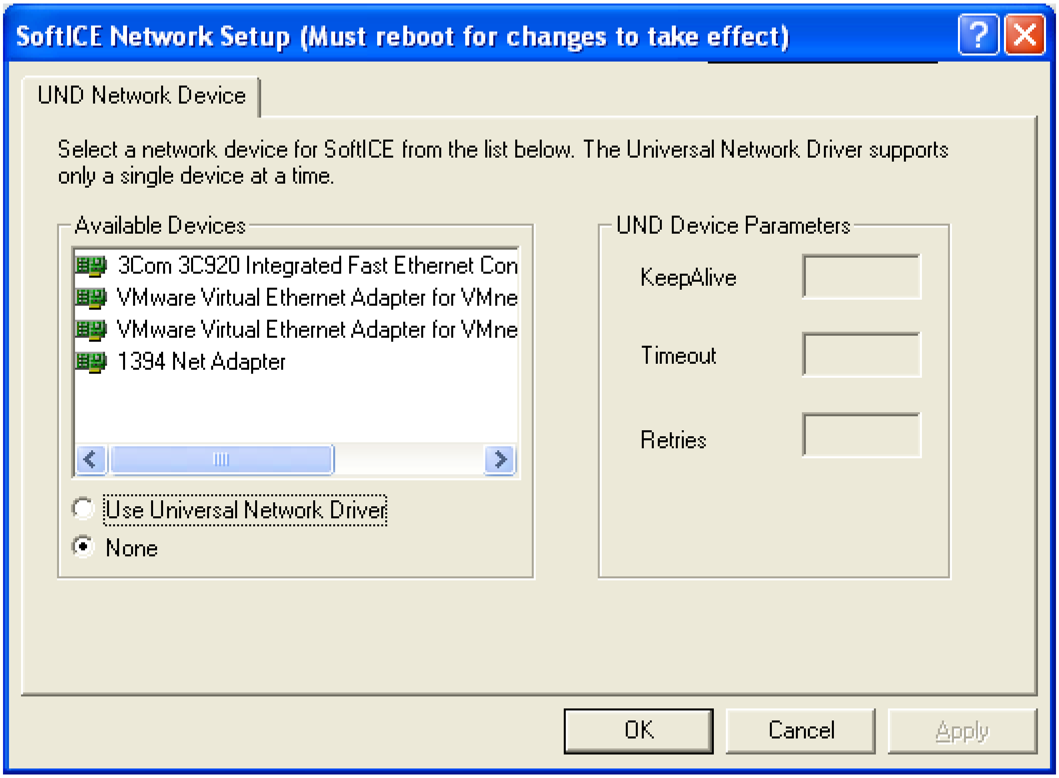
Hardware Requirements

The only hardware requirement is a PCI-based Network Card on the **target** machine. The host can have any type of network card (i.e., most built-in laptop NIC cards are PCI based).

Note: At this time there is no support for PCMCIA or USB network cards.

Installation

SIDN Installation. Installing the SIDN driver (the base driver used by SoftICE for debugging) is done through the supplied UNDSETUP.EXE application which is located in `c:\program files\compuware\driverstudio\softice\network\und`. Run this application and choose the network card that you wish to attach to the UND. Follow the prompts and reboot your machine.



SoftICE Network Setup (Must reboot for changes to take effect)

UND Network Device

Select a network device for SoftICE from the list below. The Universal Network Driver supports only a single device at a time.

Available Devices

- 3Com 3C920 Integrated Fast Ethernet Con
- VMware Virtual Ethernet Adapter for VMne
- VMware Virtual Ethernet Adapter for VMne
- 1394 Net Adapter

○ Use Universal Network Driver

● None

UND Device Parameters

KeepAlive

Timeout

Retries

Figure 10-3. SoftICE Network Setup Dialog

SIVNIC Installation. If Windows networking is required on the target computer (and it is not practical to install a second network card), you will need to install the SIVNIC.

1 Open the Control Panel and select **Add/Remove Hardware**.

2 When the wizard opens, select **Add/Troubleshoot**, click **Next**, select **Add a new device**, then specify that you want to select the device from a list.

3 When the list of hardware types appears, select **Network adapter**, click **Have disk**, and browse to:

```
Program Files\Compuware\DriverStudio\SoftICE\Network\UND\VNIC
```

4 Select **sivnic.inf** from the list, and continue through the remaining prompts.

Note: If you run into problems with the VNIC, press Esc during the boot process when the UND driver prompts you. This will abort the loading of the UND, as well as the VNIC.

5 Once the SIVNIC is installed, reboot your computer.

Removal

To uninstall the SIVNIC, simply delete it from the device list, or use the 'Remove' option in the Hardware Wizard.

To uninstall the UND, rerun the UNDSETUP.EXE program and choose the 'Uninstall' Option.

Establishing a Network Connection

Note: Presented here are the easiest methods of setting up a connection between the host and target computers. There are additional options such as password protecting, IP limiting, gateway and subnet masks that can be specified. Please refer to the *SoftICE Command Reference* for full details. Also, at the end of this chapter are additional details on the networking commands used with SoftICE.

TARGET SIDE: On the target computer, you have several options for starting SoftICE networking. You can:

1 Choose **Enable Network Support** from the SoftICE Settings-Network Debugging dialog. The easiest setup option is to accept all the defaults. When SoftICE is restarted, networking will be enabled with the options on this screen.

2 From the command line – You can start and stop networking from the command line within SoftICE. The easiest way to start networking is "net setup dhcp". To stop networking, use 'net stop' and to restart it 'net setup dhcp' or 'net start'.

3 From the init string – You can specify the same command lines as in Step 2 above.

HOST SIDE: On the host side, you have two ways to connect.

To start networking on the target computer with the **default options**:

1 Click on the DriverStudio Remote Data Namespace.

2 Right-click on the computer you wish to debug.

3 Choose **Connect to SoftICE**.

OR

1 Go to a command prompt.

2 Run the command line equivalent for connecting to a SoftICE target.

3 Change to the SoftICE directory.

4 If you started SoftICE debugging on the target with the default options, you can connect to the machine by typing in the following command:

```
siremote [machinename]
```

Note: If you don't know the machine name, you can supply the IP address of the machine, instead. To get the IP address from the machine with SoftICE, type 'net status' from the SoftICE command line and note the IP address.

If you started network debugging on the SoftICE target with additional options such as password, or if you need to specify a default gateway or subnet mask, you will need to use the SIREMOTE command line utility with the appropriate options. (See *The SIREMOTE Utility (Host Computer)* page 189, or type `siremote /help` on the command line.)

Serial Connection

Description

Serial connection offers the easiest of the remote connection options. Its performance is quite good at a baud rate of 57600 and near single-machine performance rate of 115200 baud.

Hardware Requirements

There are two Serial Connection hardware requirements:

1 A serial port dedicated to SoftICE use on both the host and target computers.

2 A null modem cable.

Note: These cables a readily available at your local computer store. If you wish to make one yourself, see the appendix for specifics on creating a null modem cable.

Installation

To install a serial connection, perform the following two steps:

1 Connect the cable between the two machines. You may want to confirm that the connection between the two machines is valid by using any 'dumb terminal' program. (HyperTerm ships with Windows.)

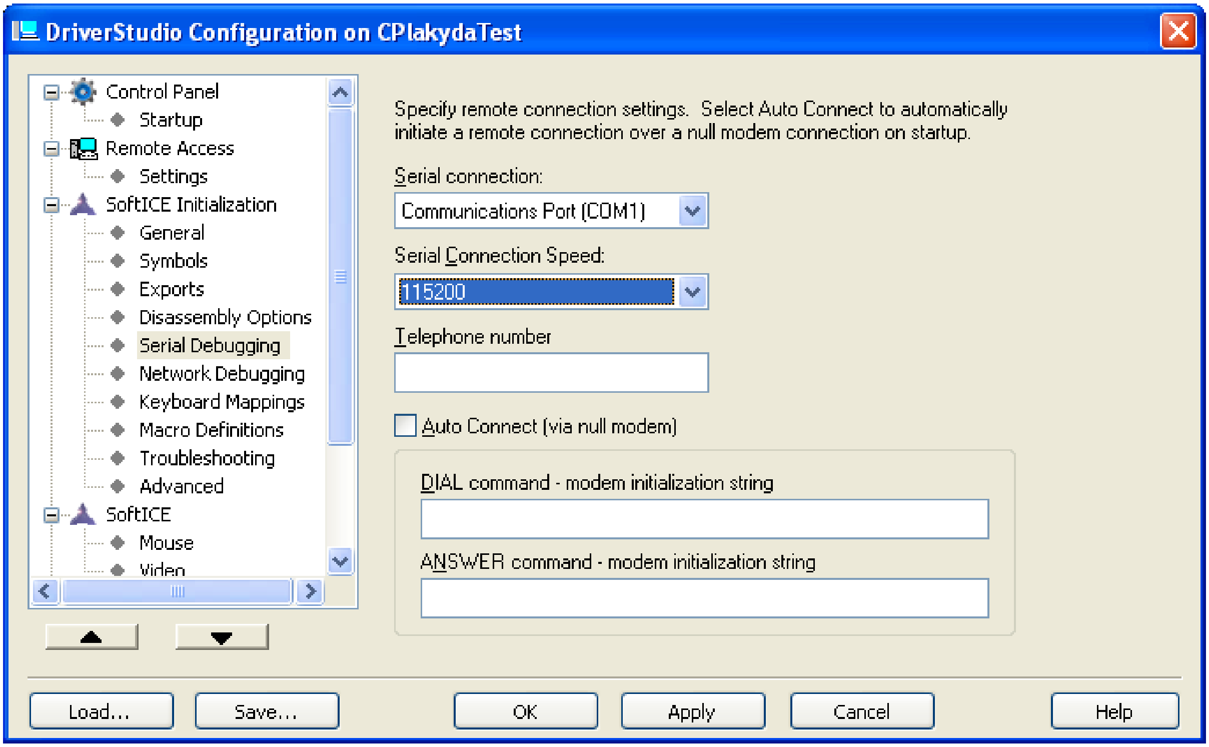
2 Make sure that your connection options are set to the appropriate settings. If you are running Win2K or WinXP, you will need to use the SoftICE Settings utility to choose which comport you will be using for debugging. For the following example, we will be remote debugging on COM1 at a speed of 115200 baud.

Removal

There are no special requirements to uninstall other than removing the cable, if so desired. If you are running Win2K or WinXP (and later), you will want to change Serial Connection in the SoftICE Settings dialog back to **None**.

Establishing a Connection

To establish a connection you must first turn on the serial debugging option within SoftICE on the target computer (as shown in the following figure).



DriverStudio Configuration on CPlakydaTest

Specify remote connection settings. Select Auto Connect to automatically initiate a remote connection over a null modem connection on startup.

Serial connection: Communications Port (COM1)

Serial Connection Speed: 115200

Telephone number

□ Auto Connect (via null modem)

DIAL command - modem initialization string

ANSWER command - modem initialization string

Figure 10-4. Establishing a Connection

Now, connect to the target from the host computer.

TARGET SIDE: Enable serial debugging using one of the following methods:

- Click on the “Auto Connect (via null modem)” option on the Serial Debugging page of SoftICE settings. (You will need to reboot your machine for the changes to take effect.)

OR

- From the SoftICE command line type in “NET COMx baudrate” (where *COMx* is one of four possible ports – COM1, COM2, COM3, or COM4 – and *baudrate* is one of four speeds – 19200, 38400, 57600, or 115200).

OR

- Add the “`NET COMx baudrate`” to the init line on the General tab.

HOST SIDE: Enable serial debugging as follows:

1. From the target side, you will need to open up a command prompt and navigate to the SoftICE directory.
2. Execute the SIREMOTE COMx baudrate (where *COMx* is the comport to which the cable is connected and *baudrate* is your connect speed.)

Modem

Description

You can operate SoftICE remotely over a modem. This is particularly useful for debugging program faults that occur at an end-user site that you cannot reproduce locally.

When you operate SoftICE over a modem, the local PC runs both SoftICE and the application you are debugging. The remote PC behaves as a ‘dumb terminal’ that serves to display the output for your SoftICE session and to accept keyboard input. SoftICE does not provide mouse support for the remote computer.

Hardware Requirements

SoftICE has the following hardware requirements for the modems you use to connect the local and remote systems:

- The modem must accept the industry-standard AT commands such as ATZ and ATDT, and returns standard result codes such as RING and CONNECT.

- The modem must execute a reliable error detecting and correcting protocol such as V.42 or MNP5. This is important because the communication protocol used by SoftICE **does not include error detection**.

Establishing a Connection

When using SoftICE over a modem, either the local or remote party can dial to initiate a connection.

Do the following to establish a connection where the local SoftICE user (you) dials the remote user:

1. Have the remote user run SIREMOTE.EXE.
2. Invoke the DIAL command on your machine.

   A connection is established and the remote user is in control of SoftICE.

Do the following to establish a connection where the remote user dials the local SoftICE user:

1. Local SoftICE user invokes the ANSWER command to prepare to answer a call.
2. Remote user dials out using SIREMOTE.EXE..

   A connection is established and the remote user is in control of SoftICE.

Removal

There are no special requirements to uninstall the modem connection.

SIREMOTE Utility (Host Computer)

The support application, **siremote.exe**, is the front end for all of SoftICE remote debugging options. When using the DriverStudio Remote Data namespace extension to connect to SoftICE on a remote target, you are essentially issuing a blind command of ‘siremote ipaddressoftarget.’

The command line options for siremote.exe vary based upon what type of connection you are using.

Serial Connection – The only options are COMport and Baudrate. For example:

- **Siremote COM1 115200 –** This will connect to a remote target with the hosts com port of COM1 at a speed of 115200.

For network connections, the commands are similar. For example:

- **Siremote cartman –** This will connect to the remote target named cartman.
- **Siremote 192.168.0.10 secret** – This will connect to the target machine with an IP address of 192.168.0.10 and a password of ‘secret.’

NET Command (Target Computer)

On the target computer, as specified earlier, you can enable remote debugging either through the user interface, or from the command line within SoftICE. The easiest method is to use the SoftICE Settings configuration utility.

Note: Any changes made here will take effect the next time SoftICE starts. This most often means on the next reboot.

Online Help can be viewed by issuing the ‘NET HELP’ command from within SoftICE.

```
    :net help
```

```
NET SETUP <IP address|DHCP> [MASK=<subnet mask>] [GATEWAY=<IP
address>]                              [ALLOW=<IP address|ANY>]
[PASSWORD=<password>]

NET START <IP address|DHCP> [MASK=<subnet mask>] [GATEWAY=<IP
address>]

NET COMx [baud-rate]

NET ALLOW <IP address|ANY> [AUTO] [PASSWORD=<password>]

NET PING <IP address>

NET RESET - Reset the current connection

NET DISCONNECT - Reset the current connection

NET STOP - Close connection and disable networking

NET HELP

NET STATUS
```

Chapter 11

Customizing SoftICE

- Modifying SoftICE Initialization Settings
- Modifying General Settings
- Pre-Loading Symbols and Source Code
- Pre-Loading Exports
- Serial Debugging
- Configuring Network Debugging
- Modifying Keyboard Mappings
- Working with Persistent Macros
- Setting Troubleshooting Options
- Specifying Advanced Options

Modifying SoftICE Initialization Settings

The SoftICE Configuration settings provides a variety of choices that determine your debugging environment at initialization. These settings are categorized as follows:

- **General** — Provides a variety of useful SoftICE settings, including an initialization string of commands that automatically executes when you start SoftICE.
- **Symbols** — Specifies .NMS symbol files to load at initialization for debugging device drivers.
- **Exports** — Specifies DLLs and EXEs from which to load export symbols at initialization.
- **Serial Debugging** — Specifies remote connection settings. Can select AutoConnect to automatically initiate a remote connection over a null modem connection on startup.

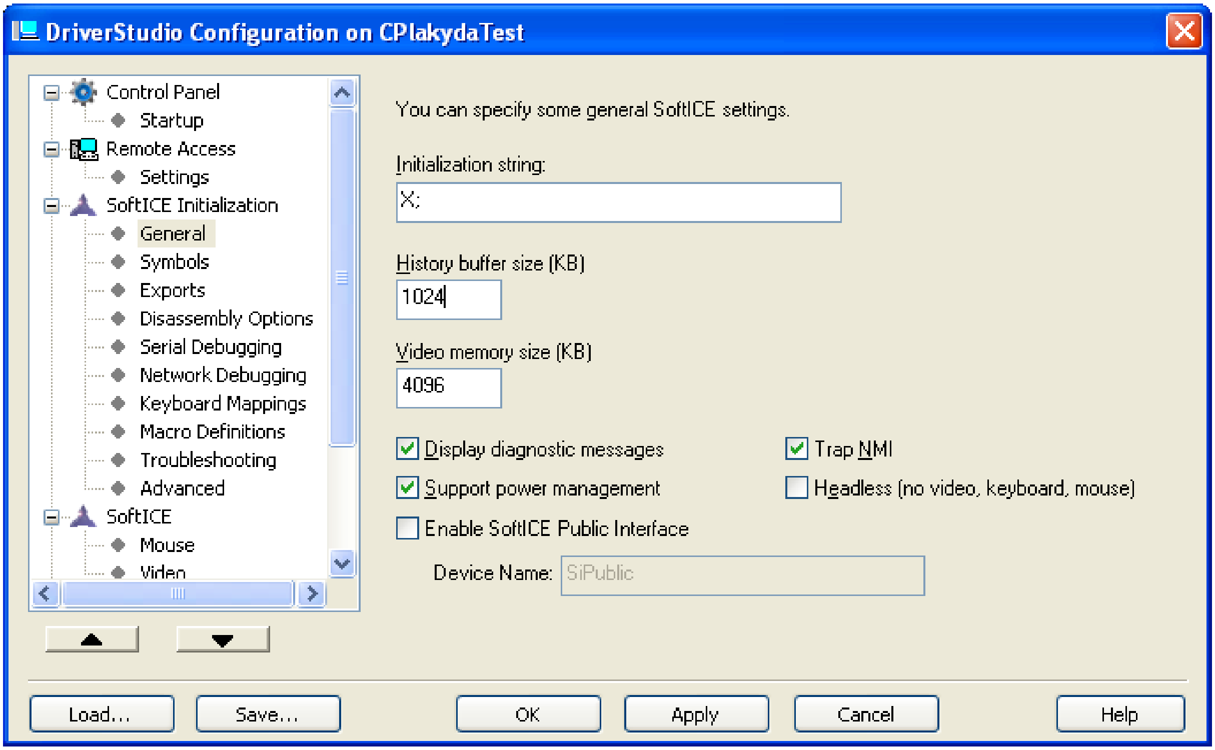
- **Network Debugging** — Use this page to determine how SoftICE should resolve machine IP addresses on the network. You can also set which network machines are allowed to connect to your computer.
- **Keyboard Mappings** — Assigns SoftICE commands to function keys.
- **Macro Definitions** — Defines your own commands to use within SoftICE.
- **Troubleshooting** — Provides solutions to potential problems.
- **Advanced** — Specifies a command list that cannot be changed from any other configuration page in this group. These commands are used mostly for support purposes.

To modify the SoftICE initialization settings, do the following:

1 Start Symbol Loader.

2 From within Symbol Loader, choose SOFTICE INITIALIZATION SETTINGS... from the Edit menu.

SoftICE displays the following SoftICE Initialization Settings window.



You can specify some general SoftICE settings.

Initialization string: X:

History buffer size (KB): 1024

Video memory size (KB): 4096

☑ Display diagnostic messages ☑ Trap NMI

☑ Support power management ☐ Headless (no video, keyboard, mouse)

☐ Enable SoftICE Public Interface

Device Name: SiPublic

Figure 11-1. SoftICE Initialization Settings

3 Click on the settings you want to modify.

4 Modify the settings and click **OK**.

5 Reboot your computer and run SoftICE to apply your changes.

Note: The following sections describe these settings.

Modifying General Settings

Modify the General SoftICE initialization settings as follows.

Initialization

Initialization executes a series of commands when SoftICE initializes. By default, the Initialization string contains the X (exit) command delimited with a semi-colon, as follows:

```
X;
```

You might want to add additional commands to the initialization string to change the Ctrl-D hot key sequence that pops up the SoftICE window, to change SoftICE window sizes, to increase the number of lines displayed by SoftICE, or to use the Serial command for remote debugging. If you are debugging a device driver, you might want to remove the X command (or the semicolon that follows it) to prevent SoftICE from automatically exiting upon initialization.

To add commands to the initialization string, type one or more semicolon delimited commands before the X (exit) command. Commands are processed in the order in which you place them. Thus, placing a command after the X command, means the command does not execute until you pop up the SoftICE window. If you type a command without a semicolon, SoftICE loads the command into the Command window, but does not execute it.

The following initialization string switches SoftICE to 50-line mode, changes the hot key sequence to Alt-Z, toggles the Register window on, and exits from SoftICE:

```
LINES 50;ALTKEY ALT Z;WR;X;
```

Note: If you type a string that exceeds the width of the Initialization field, the field automatically scrolls horizontally to allow you to view the information as you enter it.

History Buffer Size

History buffer size determines the size of the SoftICE history buffer. By default, the History buffer size is 256KB.

The SoftICE history buffer contains all the information displayed in the Command window. Thus, saving the SoftICE history buffer to a file is useful for dumping large amounts of data, disassembling code, logging breakpoints with the BPLOG command, and listing Windows messages logged by the BMSG command. Refer to *Saving the Command Window History Buffer to a File* on page 98.

Trace BufferSize (Windows 9x Only)

This setting determines the size of the trace buffer. The trace buffer can maintain back trace for the BPR and BPRW commands. By default, **Trace buffer size** is set to 8 KB.

Total RAM (Windows 9x Only)

This setting indicates the amount of physical memory installed in your system. Set **Total RAM** to a value equal to or greater than to the amount of memory on your system.

Due to subtle architectural differences between systems, SoftICE cannot detect the amount of physical memory installed in your computer under Windows 9x. To map the relationship between linear and physical memory, SoftICE uses a default value of 128 MB. While this value is reasonable for most current development systems with 128 MB or less of physical memory, this does not work correctly on systems with larger physical address spaces. This is due to the fact that appropriate data structures for memory pages above 128 MB are not created.

If your system contains less than 128 MB of physical memory, you can save a small amount of memory by setting this field to the right value. The memory savings result because fewer data structures are needed to map physical memory.

Display Diagnostic Messages

Display diagnostic messages determines whether or not SoftICE turns on verbose mode to display additional information, such as module loading and unloading, in the Command window. By default, **Display diagnostic messages** is turned on.

Trap NMI

Trap NMI determines whether Non-maskable interrupt (NMI) trapping is turned on or off. By default, **Trap NMI** is turned on. NMI trapping is useful if you have a means of generating an NMI, such as a breakout switch. Generating an NMI allows you to enter SoftICE even when all interrupts are disabled. Simple ISA-based breakout switches are available. Contact Compuware for more information.

Lowercase Disassembly

Lowercase disassembly determines whether or not SoftICE uses lowercase letters for disassembling instructions. By default, **Lowercase disassembly** is turned off.

Support Power Management

SoftICE supports Power Management on the Windows NT family platforms. (**Support power management** is the default selection.) This allows SoftICE to run on a Windows family system that will go into Standby or Hibernate mode without interfering with hardware management.

To disable power management support, follow these steps:

1. Run Symbol Loader by selecting **Start** > **Programs** > **Compuware DriverStudio** > **Debug** > **Symbol Loader** on the Windows Start menu.
2. On the Symbol Loader menu bar, select **Edit** > **SoftICE Initialization Settings**.
3. On the SoftICE Initialization/General page of the Configuration (Settings) dialog, uncheck the **Support power management** check box, and click **OK**.

Debugging Driver Power Management Code

To debug driver power management code, follow these steps.

1. Configure SoftICE for remote debugging.
2. Configure it to automatically connect for remote debugging.
3. Make sure that the com port used for the connection is configured properly.
4. Test that the connection can be established.
5. Configure SoftICE to run in Headless Mode.

6 Reboot and establish the remote connection.

Now SoftICE will be able to popup remotely during power management cycle.

Headless

SoftICE can be configured so it does not program the video, keyboard and mouse hardware. This is a useful option when debugging remotely.

To activate this option, go to the Configuration (Settings) dialog and select SoftICE Initialization. On the General page, select the **Headless (no debugger video, keyboard or mouse)** check box.

Individual hardware component programming can be disabled via keys in the NTICE service key in the registry. Setting the NullKeyboard REG_DWORD value will configure SoftICE to not program the mouse and keyboard. Setting the NullVGA REG_DWORD value will configure SoftICE to not program video.

Enable SoftICE Public Interface

SoftICE can expose a public interface to allow for querying of its presence from a driver or application. We suggest using a different name, rather than accepting the default. For additional information, see *SoftICE API Specification* on page 269.

Pre-Loading Symbols and Source Code

Use the Symbols initialization settings in conjunction with the Module Translation settings to pre-load symbols and source code when you start SoftICE. Pre-loading symbols and source code is useful for debugging device drivers.

To pre-load symbols or source code, do the following:

1 In the Module Translation settings, select **Symbols and source code** if you want your source code loaded in addition to the symbols.

2 Select **Package source with symbol table**.

3. In Symbol Loader, choose Translate from the Module menu to translate the module to a .NMS symbol file.

Tip You can use the Symbol Loader command-line utility, NMSYM, to specify the output file name.

4. Use the Symbols SoftICE Initialization settings to add your .NMS symbol file to the Symbols list. The following section describes how to do this.

Note: Normally, your .NMS symbol file has the same base name as the file you translated. With Windows 9x, SoftICE can not pre-load files with long file names, because SoftICE is in real-mode DOS when it initializes. If your module is a long file name, create the .NMS file, rename the .NMS file to an eight-character name with the extension .NMS, and select the renamed .NMS file when you add it to the symbols list.

Adding Symbol Files to the Symbols List

Tip When you select PACKAGE SOURCE WITH SYMBOL TABLE, source files are part of the .NMS symbol file. Thus, there are no restrictions on source file name lengths even within Windows 9x.

From the Symbols selection in the SoftICE Initialization settings, do the following:

1. Click **Add**.

   SoftICE displays a browse window for you to locate the .NMS files that contain the symbols and source code you want to pre-load.

2. Select one or more .NMS symbol files and click **OK**.
3. Every time you modify your source code, retranslate your module to create a new version of the .NMS symbol file.

Removing Symbols and Source Code Pre-Loading

To prevent SoftICE from pre-loading the symbols or source code associated with a particular file, select the file in the symbols list and click **Remove**.

Reserving Symbol Memory

Symbol buffer size specifies, in kilobytes, the amount of memory to reserve for storing certain types of debug information (for example, line number information). With SoftICE for Windows 9x, this memory region also serves as a buffer for holding .NMS images at boot time. By default, SoftICE reserves 1024KB for Windows 9x and 512KB for the Windows NT family.

Typically 512KB is adequate. However, you may need to increase the Symbol buffer size under the following circumstances:

- If you are debugging large programs, use 1024KB or more.

- If you are using Windows 9x, and you are loading symbols at boot time, determine the total size of all the .NMS files that are loaded at boot time and set the Symbol buffer size to this number.

To determine how much symbol memory is available, use the TABLE command. Note that most symbol information is stored in dynamically-allocated memory.

Pre-Loading Exports

Use the Export initialization settings to select files from which SoftICE can extract export information upon SoftICE initialization. Extracting export information is useful for debugging DLLs when no debugging information is available.

Extracting Export Information

To select one or more files from which to extract export information, do the following:

1. Click **Add**. SoftICE displays a browse window for you to locate the files.
2. Select one or more files from which to extract the information and click **OK**.
3. SoftICE places the files you selected in the Exports list.

Removing Files from the Exports List

To remove a file from the Exports list, select the file and click **Remove**.

Serial Debugging

The Serial Debugging page allows you to specify remote connection settings. You can select **AutoConnect** to automatically initiate a remote conection over a null modem on startup.

Note: Information for configuring SoftICE for remote debugging over a serial cable can be found in the *DriverStudio and SoftICE Driver Suite Installation Guide.*

Configuring Remote Debugging with a Modem

The Remote Debugging settings allow you to define the type of serial connection, and preset a modem initialization string and phone number for the DIAL and ANSWER commands. Alternately, you can specify these parameters directly when using the commands. Refer to your modem documentation for the exact commands for your particular modem.

Serial Connection (Windows 9x Only)

If you are using SoftICE for Windows 9x, and are debugging a remote system, choose the communications port on the local system (COM1, COM2, COM3, or COM4) that you are using for serial communication. When you are through debugging the remote system, change this setting to **None**. By default, **Serial connection** is set to **None**.

Note: If you are using SoftICE for the Windows NT family, SoftICE automatically determines your serial connection.

Telephone Number

Telephone number presets a phone number for the DIAL command, for example, 717-555-1212.

DIAL Initialization String

DIAL initialization string presets the modem initialization string for the DIAL ccommand, for example, ATX0.

ANSWER Initialization String

ANSWER initialization string presets the modem initialization string for the ANSWER command, for example, ATX0.

Configuring Network Debugging

Remote SoftICE allows you to use a standard internet connection to remotely control SoftICE. This allows greater flexibility and easier access for debugging functions. Remote SoftICE is supported by Windows 9x and the Windows NT family.

Requirements for Remote SoftICE Support

The machine that runs SoftICE is referred to as the **target** machine.

- The target machine requires a supported ethernet adapter that is connected to the local IP network.
- Currently supported Ethernet adapters are:
  - NE2000 and compatibles (use NE2000.SYS)
  - 3Com 3C90X (use EL90X.SYS)
  - Intel E100 Series Network Adapter

    The machine that controls the target machine is called the **host** machine.
- The host must be connected to an IP network that is directly or indirectly connected to the IP network of the target machine. The host must also be running Windows 9x or the Windows NT family.

Setting Up SoftICE for Remote Debugging

Verify the target system is operating properly using a supported adapter and driver. Replace the adapter driver file (for the Windows NT family, it's in the \WINNT\SYSTEM32\DRIVERS directory; for Windows 9x, it's in the \WINDOWS\SYSTEM directory) with the file of the same name from the distribution. Rename the original driver file in case you need it again.

After replacing the driver file, you will need to reboot the system in order to use Remote SoftICE.

Enabling Remote Debugging from the Target Side

Once the correct adapter and driver is installed, SoftICE will not allow remote debugging until it is enabled using the NET commands. The following commands are available:

- NET START
- NET ALLOW
- NET PING
- NET RESET
- NET STOP
- NET HELP
- NET STATUS

NET START Command

The NET START command (`NET START <IP address|DHCP> [MASK=<subnet mask>] [GATEWAY=<IP address>]`) enables the IP stack within SoftICE. This command identifies your IP parameters to SoftICE (IP address, subnet mask, and gateway address). If your local network supports DHCP (Dynamic Host Configuration Protocol), you can tell SoftICE to obtain the IP parameters from your network DHCP server. At this point, the IP stack is running but SoftICE does not allow remote debugging until you get an IP address.

NET ALLOW Command

The NET ALLOW command (NET ALLOW <IP address|ANY> [AUTO] [PASSWORD=<password>]) defines which machines can be used to remotely control SoftICE.

- A remote machine can be defined as a specific IP address, or ANY IP address.
- If the AUTO option was specified on the NET ALLOW command, then it is not necessary to issue the NET ALLOW command to enable a new session after closing the current session.
- Access to SoftICE control can also be qualified with a case-sensitive password.

When you begin a remote debugging session, SoftICE will pop up on the target machine, no matter what the current state of the machine.

NET PING Command

The NET PING command (NET PING <IP address>) allows you to do a basic network connectivity test by sending an ICMP Echo Request (PING) packet to an IP address. SoftICE sends the request and indicates if it receives a response within four seconds.

NET RESET Command

The NET RESET command terminates any active remote debugging session, or cancels the effect of the previous NET ALLOW command. Use the NET ALLOW command to re-enable remote debugging.

NET STOP Command

The NET STOP command terminates any active remote debugging session, or cancels the effect of the previous NET ALLOW command. It also disables the IP stack and the network adapter.

NET HELP Command

The NET HELP command shows a list of the available network commands with their respec-tive syntax.

NET STATUS Command

The NET STATUS command shows the current status of the network adapter (if the NET START command has been issued, this includes the node address). It also displays the cur-rent IP parameters (IP address, subnet mask, and gateway) and the status of the remote debugging connection.

Starting the Remote Debugging Session

Once the target is set up for remote debugging, the remote machine can issue the SIREMOTE command. Following is the syntax for the SIREMOTE command.

```
SIREMOTE <target IP address> [<password>]
```

The **target IP address** is the IP address assigned to the ethernet adapter in the target machine. If the target machine uses a password, specify the case-sensitive password on the command line.

SIREMOTE tries to create a connection to the target machine. If the target machine responds, SIREMOTE authenticates the remote machine with the specified password (blank if no password is being used). If the target accepts the authentication of the remote machine, Soft-ICE makes the connection and SIREMOTE obtains the current screen parameters of the target machine. A console window emulates the SoftICE display, which is visible on both the target and remote machines.

All standard SoftICE keys react whether they are entered from the remote or target keyboard. The only exception is that the pop-up key on the remote machine is always Ctrl-D, even if it is redefined on the target machine.

To terminate the remote SoftICE session, press **Ctrl-Break** on the remote keyboard, or use the NET RESET command from the target machine.

Modifying Keyboard Mappings

Use Keyboard Mappings to reassign commands to SoftICE function keys or to specify new ones. You can assign SoftICE commands to any of the twelve function keys or key combinations involving Shift, Ctrl, or Alt and a function key.

Note: Keyboard mappings assumes that you are using a 'QWERTY' keyboard layout. If you happen to be using a non-QWERTY layout keyboard, you will need to copy the included **keymap.exe** utility program into your \winnt\system32\drivers directory and execute **keymap**. If SoftICE is currently running, reboot your system so the changes can take effect. Running keymap will remap all the keyboard scan codes to the keyboard layout that is currently being used by Windows. The one key combination that cannot be remapped is the popup hotkey. The popup hotkey will always be the third character from the left on the second row above the space bar.

To patch SoftICE to match the current Windows keymap, issue one of the commands listed in Table 11-1.

Table 11-1. Match Current Windows Keymap

Platform	Command
Windows NT family	`KEYMAP.EXE NTICE.SYS`
Windows 9x	`KEYMAP.EXE WINICE.EXE`

To restore the keyboard mappings to the default USA keymap, issue one of the commands listed in Table 11-2.

Table 11-2. Restore Keyboard Mappings to Default USA Keymap

Platform	Command
Windows NT family	`KEYMAP.EXE NTICE.SYS /USA`
Windows 9x	`KEYMAP.EXE WINICE.EXE /USA`

Command Syntax

When modifying and creating function keys, you can use any valid SoftICE command and the characters; caret(^) and semicolon (;). Place a caret (^) at the beginning of a command to instruct SoftICE to execute the command without placing it in the command line. The semicolon behaves like the Enter key and instructs SoftICE to execute the command. You can place one or more semicolons in the same string.

Modifying Function Keys

SoftICE uses the following abbreviations for the Function, Alt, Ctrl, and Shift keys:

Table 11-3. Function Key Abbreviations

Key	Abbreviation	Example
Function	F	F1
Alt	A	AF1
Ctrl	C	CF1
Shift	S	SF1

To modify the SoftICE command assigned to a function key, do the following:

1. Select the function key you want to modify from the list of keyboard mappings and click **Add**.
2. Change the command in the Command field and click **OK**.

Creating Function Keys

To assign a command to a new function key or function key combination, do the following:

1. Determine a function key or function key combination to which no commands are assigned.
2. Click **Add**.
3. Select the function key you want to use from the Key list.
4. Select a modifier. To assign a command to a function key, click **None**. To assign a command to a function key combination, select **Shift**, **Ctrl**, or **Alt**.
5. Type a command in the Command field and click **OK**.

Deleting Function Keys

To delete a function key assignment, choose the function key and click **Remove**.

Restoring Function Keys

The following table lists the default function key assignments.

Table 11-4. Default Function Key Assignments

Key	Assignment	Key	Assignment
F1	H;	F12	^P RET;
F2	^WR;	SF3	^FORMAT;
F3	^SRC;	AF1	^WR;
F4	^RS;	AF2	^WD;
F5	^X;	AF3	^WC;
F6	^EC;	AF4	^WW;
F7	^HERE;	AF5	CLS;
F8	^T;	AF11	dd dataaddr->0;
F9	^BPX;	AF12	dd dataaddr->4;
F10	^P;	F12	^P RET;
F11	^G @SS:ESP;	SF3	^FORMAT;

You can modify individual function key assignments or click **Restore defaults** to restore all the keys you edited or removed to their original settings. **Restore defaults** does not remove any function keys you create.

Working with Persistent Macros

Macros are user-defined commands that you can use in the same way as built-in commands. The definition, or body, of a macro consists of a sequence of command invocations. The allowable set of commands includes other user-defined macros and command-line arguments.

There are two ways to create macros. You can create run-time macros that exist until you restart SoftICE or persistent macros that are saved in the initialization file and automatically loaded with SoftICE. This section describes how to create persistent macros. Refer to *Using Run-time Macros* on page 96 for more information about creating run-time Macros.

Creating Persistent Macros

To create a persistent macro, do the following:

1 Click **Add**.

The Add Macro definition window appears.

2 Type the name of the macro in the Name field.

The macro name may be from three to eight characters long and may contain any alpha-numeric character or underscore (_). It must include at least one alphabetic character. A macro-name cannot duplicate an existing SoftICE command.

3 Type the macro definition in the Definition field.

The definition of a macro is a sequence of SoftICE commands or other macros separated by semicolons. You are not required to terminate the final command with a semicolon. Command-line arguments to the macro can be referenced anywhere in the macro body with the syntax %<*parameter#*>, where *parameter#* is a number between one and eight.

Note: Although it is possible for a macro to call itself recursively, it is not particularly useful, because there is no programmatic way to terminate the macro. If the macro calls itself as the last command of the macro (tail recursion), the macro executes until you use the ESC key to terminate it. If the recursive call is not the last command in the macro, the macro executes 32 times (the nesting limit).

4 Click **OK**. SoftICE places your persistent macro in the Macro Definitions list.

Macro Definition Examples

The following table provides examples of legal macro commands.

Table 11-5. Legal Macro Commands

Legal Name	Legal Definition	Example
Qexp	addr explorer; Query %1	Qexp
		Qexp 140000

Table 11-5. Legal Macro Commands (Continued)

Legal Name	Legal Definition	Example
1shot	bpx %1 do \”bc bpindex\”	1shot eip or 1shot @esp
ddt	dd thread	ddt
ddp	dd process	ddp
thr	thread %1 tid	thr or thr -x
dmyfile	macro myfile = \”TABLE %1;file \%1\”	dmyfile mytable myfile myfile.c

The following table provides examples of illegal macro commands:

Table 11-6. Illegal Macro Commands

Illegal Name or Definition	Explanation
~~Name: DD~~ ~~Definition: dd dataaddr~~	This macro uses the name of a SoftICE command. SoftICE commands cannot be redefined.
~~Name: AA~~ ~~Definition: addr %1~~	The macro command name is too short. A macro name must be between 3 and 8 characters long.
~~Name: tag~~ ~~Definition: ? *(%2-4)~~	The macro body references parameter %2 without referencing parameter %1. You cannot reference parameter %n+1 without referencing parameter %n.

Starting and Stopping Persistent Macros

Type the name of the persistent macro to execute it. To stop the execution of a persistent macro, press **Esc**.

Setting the Macro Limit

Use **Macro limit** to specify the maximum number of macros and breakpoint actions you can define during a SoftICE session. This number includes both run-time macros and persistent macros. The default value of 32 is the minimum value. The maximum value is 256.

Modifying Persistent Macros

To modify a persistent macro, do the following:

1. Select the persistent macro you want to modify and click **Add**.
2. In the Add macro definitions window, modify the Name and Definition fields as appropriate, then click **OK**.

Deleting Persistent Macros

To delete a persistent macro, select the macro you want to delete and click **Remove**.

Setting Troubleshooting Options

Tip If you want to return all the troubleshooting settings to their original states, click RESTORE DEFAULTS.

The following settings let you troubleshoot SoftICE. Modify these settings only when directed to do so by Compuware Technical Support or to remedy the specific situations described within this documentation. By default, the Troubleshooting settings are all turned off.

Disable Mouse Support

If you are having problems using your mouse in SoftICE, select **Disable mouse support**.

Disable Num Lock and Caps Lock Programming

Tip If you've turned on more than one troubleshooting setting and you want to turn all the settings off, use Restore Defaults instead of clicking each individual check box.

If your keyboard locks or behaves erratically when you load SoftICE, select **Disable Num Lock and Caps Lock programming.** If this does not solve the problem and you are using the Windows NT family, try the **Do not patch keyboard driver** setting.

Do Not Patch Keyboard Driver (Windows NT family Only)

If your keyboard locks or behaves erratically when you load SoftICE, select this setting to prevent SoftICE from patching the keyboard driver. When you select this option, SoftICE uses an alternate, typically less robust, method for keyboard handling. If this does not solve the problem, try the **Disable Num Lock and Caps Lock programming** setting.

Disable Mapping of Non-Present Pages

SoftICE attempts to find a page in physical memory even if the page table entry is marked as not present. Select **Disable mapping of non-present pages** to turn off this feature.

Disable Pentium Support

SoftICE automatically detects whether or not you are using a Pentium processor. If you are using a new CPU with which SoftICE is unfamiliar and SoftICE mistakenly determines that you are using a Pentium processor, select this setting to turn off Pentium support.

Disable Thread-Specific Stepping

The P (step over) command is thread sensitive. The return breakpoint set by the P command triggers only for the thread that was active when the P command was issued. Note that you would normally want to be in the same thread you are debugging. To turn off this feature, select **Disable thread-specific stepping**.

Specifying Advanced Options

Use the Advanced Options page to specify a list of commands that cannot be modified from any of the other configuration pages of this group. The commands found on this page are used mostly for support purposes.

Chapter 12

Exploring Windows NT

- Overview
- Inside the Windows NT Kernel
- Win32 Subsystem

Overview

Without qualification, the Windows NT operating system family (Windows NT, Windows 2000, and Windows XP) represents an incredible feat of software engineering and system design. It is hard to imagine a design of such complexity reaching all of its goals, including three of the most difficult: portability, reliability, and extensibility, without compromising either interfaces or implementation. Yet, somehow the system engineers at Microsoft who design and develop the Windows NT operating system family have managed to keep each and every component of these systems smoothly interlocked, not unlike the precision gears of a finely-made watch. 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 SoftICE, you should be able to gain a basic understanding of how the components of Windows NT fit together.

For the purposes of this chapter, the use of the term “Windows NT” refers to all of the 32 bit operating systems, released by Microsoft, where the kernel is the Windows NT kernel. This includes: Windows NT 4, Windows NT 4 Embedded, Windows 2000, Windows XP, Windows XP Embedded, Windows Server 2003, and early pre-releases of Windows Code-named Longhorn.

Also, for the purposes of this chapter, the use of the term “SoftICE” applies to the entire family of SoftICE debuggers for the Windows NT family. The output and Command Line may be slightly different depending on which version of SoftICE you are using.

Resources for Advanced Debugging

Microsoft provides several resources for advanced Windows NT debugging including: checked build, the Windows NT DDK, symbol files, driver verifier, and kernel debugger extensions.

Checked Build

If you are not currently using the checked build (that is, the debug version) of Windows NT, you are missing a lot of valuable information and debugging support that the operating system provides. The checked build contains a wealth of information that is absent from the free build (retail version). This includes basic debug messages, special flags used by the kernel components that allow you to trace the system’s operation, and relatively strict sanity checking of most system API calls. The size and layout of system data structures as well as the implementation of system APIs in the checked build are nearly identical to that of the free build. This allows you to learn and explore using the more verbose checked build, but still feel completely comfortable if you end up debugging under the free build.

It is also possible to use individual components from the checked build on a free build installation. This is often helpful when trying to pin down a crash or other problem that is happening inside an OS component. Using checked build components in this way is as simple as copying the checked build module and its associated debug information file onto the target system, and loading that debug information into SoftICE.

All in all, if you want to write more robust applications and drivers, use the checked build.

Windows NT DDK

The Windows NT 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 also 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 WDM.H, NTDEF.H, BUGCODES.H, and NTSTATUS.H are generally useful.

The Windows NT 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. SoftICE for Windows NT provides similar functionality with the OBJDIR, DEVICE, and DRIVER commands. The utility DRIVERS.EXE, like the SoftICE MOD command, lists all drivers within the system, including basic information about the driver. Some versions of the Windows NT 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 NT 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 NT device drivers. Simply find something in your area of interest, build that sample, and step through it with SoftICE.

Symbol Files

Tip Using symbol files is probably the most important aspect of setting up your development and debugging environment. Select those components that are most relevant to your development needs, find the corresponding .DBG or .PDB file and use Symbol Loader to create a .NMS file that SoftICE can load.

Debug files come in one of two formats. Depending on which version of the operating system you are using, you may either have .DBG debug files or .PDB debug files. For operating systems prior to Win2k all OS symbols were released in the .DBG format. Prior to Win2k-sp3 most OS files were released in .PDB form with the exception of ntoskrnl and a few other key system files. For all operating systems after Win2k-sp3, the debug format has been exclusively .PDB files.

Microsoft provides a separate debug file for every distributed executable file with both the checked and free builds of the Windows NT operating system. This includes the systems components that make up the kernel executive, device drivers, Win32 system DLLs, sub-system processes, control panel applets, and even accessories and games. The .DBG 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). The .PDB-format symbol files include most information on most structures; the older .DBG files do not.

Information on locals is not provided in Microsoft's public symbol files, but having access to a public definition for each API makes debugging through system calls a lot easier.

To examine the symbols in the symbol files, issue the SYM command. To get known types, issue the TYPES command. To get a breakout of a structure, issue the TYPES structname command. Within SoftICE you can also cast a block of memory to a structure type by casting an address to a type, for example:

```
? (_KTEB)address
```

or

```
WATCH (unicode_string)address
```

Microsoft has introduced a technology called "Symbol Server" that makes it very easy to get the matching symbols for a given binary. Symbol Server was introduced with the release of Windows XP because of the large scale use of Windows Update. The basic mechanism behind symbol server is that it maintains a collection of debug files that are stored on a server and are uniquely identifiable to a given binary through time/date stamps, GUUIDS, files size, and age.

SoftICE provides a stand alone utility called “Symbol Retriever” that will get the correct symbols for a binary and optionally translate and load these symbol files into SoftICE. Microsoft also supplies the server creation software so that you can setup your own local symbol server for your own binaries. Symbol Retriever will be able to get symbols from any symbol server.

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

Table 12-1. Key System Component Symbols

Component	Description
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.
WIN32K.SYS	A system device driver that minimizes inter-process communication between applications and CSRSS. Provides kernel mode equivalents for many of the Win32 APIs.
USER32.DLL	Basic implementation of USER functionality. Mostly stubs to WIN32K.SYS (via LPC to CSRSS).
KERNEL32.DLL.	Some basic implementation of traditional KERNEL functionality, but mostly stubs to NTDLL.DLL.

Driver Verifier

Microsoft has started adding large numbers of runtime validation checks to the operating system. The types of checks that it provides varies based upon the OS, but in general items such as pool corruption, IRQL violation errors, and low resource simulation. Each release of Windows NT adds additional options and checks. By default these checks are not enabled and need to be enabled by running the `verifier.exe` utility. From this utility you choose which drivers to analyze and what items to validate. When Driver Verifier finds an exception case, (for example, overrunning a buffer) it generates a blue screen with a stop code of (usually) `0xc4` or `0xc9` and its bugcheck parameters provide additional information. If you have SoftICE loaded you can debug the case that the verifier flagged. BoundsChecker, part of the DriverStudio product, also supplies most of the same features and functionality as driver verifier but is directly integrated with SoftICE, has a full user mode UI, and will not cause a crash.

Resources

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

- *Microsoft Developers Network* (MSDN)

  MSDN, published quarterly on CD-ROM, 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 Microsoft Windows 2000* - David A. Solomon, Mark E. Russinovich, Microsoft Press

  Inside Microsoft Windows 2000 provides a high-level view of the design for the Windows 2000 operating system. Each major sub-system is thoroughly discussed, and many block diagrams illuminate internal data structures, policies, and algorithms. Currently, this is the most definitive work on Windows 2000 operating system internals. You will gain the most benefit from the information in this book if you use SoftICE to explore the actual implementation of the system design, for when you step into OS code with SoftICE, many of the higher-level relationships become clear.

- *Advanced Windows* - Jeffrey 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.

- *Programming the Windows Driver Model* – Walter Oney, Microsoft Press

  Programming the Windows Driver Model is an excellent resource and the definitive resource for the device driver programmer.

- *Undocumented Windows 2000 Secrets* – Sven B. Schreiber, Addison Wesley

  Undocumented Windows 2000 Secrets focuses on undocumented interfaces and APIs, and is a good introduction to exploratory debugging on Windows NT.

Inside the Windows NT Kernel

To gain a basic understanding of Windows NT, look at the platform from many different perspectives. A general knowledge of how Windows NT 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 NT Kernel. It describes how Windows NT configures the core operating system data structures, such as the IDT and TSS, and how to use corresponding SoftICE commands to illustrate the Windows NT configuration of the CPU. It also examines a general map of the Windows NT system memory area, 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 NT hardware abstraction layer. Its purpose is to isolate as many hardware platform dependencies as possible into one module. This makes the Windows NT 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 NT operating system resides in the Windows NT 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 Management
  - 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
  - Power Management
  - Multi Processor Synchronization

- I/O subsystems

Managing the Intel 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 SoftICE commands to obtain a general idea of how Windows NT uses the Intel architecture to provide a secure and robust environment.

Table 12-2. SoftICE Architecture Commands

Command	Description
IDT	Display information on the Interrupt Descriptor Table
TSS	Display information about the Task State Segment
GDT	Display information on the Global Descriptor Table
LDT	Display information on the Local Descriptor Table (16-bit code only)
MSR	Displays information on the Model Specific Registers

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

IDT (Interrupt Descriptor Table)

Windows NT 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 SoftICE IDT command to view the Windows NT Interrupt Descriptor Table.

Table 12-3. 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 (older versions of Windows NT).
2D	Debug service.

Table 12-3. Interrupt Descriptor Table (Continued)

Interrupt #	Purpose
2E	Execute System Service. Prior to Windows XP, Windows used INT 2E to transition from user to system mode. Since Windows XP, this mechanism has been replaced on newer processors with the faster SYSENTER/SYSEXIT instructions, but the old mechanism is still in place. For more information, refer to the NTCALL command in the *SoftICE Command Reference*.
30-37	Primary Interrupt Controller (IRQ0-IRQ7) on older PIC-based machines 30 - HAL clock interrupt (IRQ0) on older PIC-based machines.
38-3F	Secondary Interrupt Controller (IRQ8-IRQ15) on older PIC-based machines.

On older machines using the 8259 PIC for controlling interrupts, 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. Most machines produced today use interrupt controllers based on Intel's Advanced Programmable Interrupt Controller (APIC) specification. Such systems are not limited to 15 hardware interrupts. While the IDT itself is the same on these systems, the mapping of hardware interrupts to interrupt numbers in the IDT is not. To determine which interrupt number in the IDT the OS has assigned to a given hardware interrupt, you can use the SoftICE IDT command, which will read this information out of the APIC. The vector column of the IDT output will tell you which IDT entry to look at.

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 NT does not use this architectural feature and maintains only one TSS per processor. As noted in the previous section on the Windows NT IDT, other TSS data types exist, but are only used during exceptional conditions to ensure that the system will not spontaneously reboot before Windows NT can properly crash itself. Use the SoftICE TSS command to view the current TSS.

The TSS contains the offset from the base of the TSS to the start of the I/O bitmap. The I/O bitmap determines which ports, if any, the code executing at Ring 3 can access directly. When executing a Win32 application, the TSS contains an *invalid* offset (it points beyond the segment limit of the TSS). This forces the operating system to trap all direct I/O.

Inside the actual TSS data structure, the only field of real interest is the address of the Level 0 stack. This is the stack that is used when the CPU transitions from user mode to system mode.

GDT (Global Descriptor Table)

Windows NT 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 SoftICE GDT command that shows the selectors in the Global Descriptor Table.

```
GDTbase=80036000  Limit=03FF
0008   Code32   Base=00000000   Lim=FFFFFFFF   DPL=0   P   RE
0010   Data32   Base=00000000   Lim=FFFFFFFF   DPL=0   P   RW
001B   Code32   Base=00000000   Lim=FFFFFFFF   DPL=3   P   RE
0023   Data32   Base=00000000   Lim=FFFFFFFF   DPL=3   P   RW
0028   TSS32    Base=8000B000   Lim=000020AB   DPL=0   P   B
0030   Data32   Base=FFDFF000   Lim=00001FFF   DPL=0   P   RW
003B   Data32   Base=7FFDE000   Lim=00000FFF   DPL=3   P   RW
0043   Data16   Base=00000400   Lim=0000FFFF   DPL=3   P   RW
0048   LDT      Base=E156C000   Lim=0000FFEF   DPL=0   P
0050   TSS32    Base=80143FE0   Lim=00000068   DPL=0   P
0058   TSS32    Base=80144048   Lim=00000068   DPL=0   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 usually 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.

Selector value 48h is an LDT type selector and is only used for VDM processes. Win32 applications and drivers do not use LDT selectors. When a Win32 process is active, the Intel CPU’s LDT register is NULL. In this case, the SoftICE LDT command gives you a No LDT error message. When a VDM or 16-bit WOW process is active, a valid LDT selector is set, and it comes from this GDT selector. During a process context switch, LDT selector information within the kernel process environment block (KPEB) is poked into this selector to set the appropriate base address and limit.

LDT (Local Descriptor Table)

Under Windows NT, Local Descriptor Tables are per process data structures and are only used for Virtual DOS Machines (VDM). The 16-bit WOW box (Windows On Windows) is executed within a NTVDM process and has an LDT. Like Windows 3.1, the LDT for a WOW contains the selectors for every 16-bit protected mode code and data segment for each 16-bit application or DLL that is loaded. It also contains the selectors for each task database, module database, local heaps, global allocations, and all USER and GDI objects that require the creation of a selector.

Under a WOW, because the number of selectors needed can be quite large, a full LDT is created with a majority of the entries initially reserved. These reserved selectors are allocated as needed. Under a non-WOW VDM, the size of the LDT is significantly smaller.

Windows NT System Memory Map

Windows NT 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. (It is also possible to change this behavior by adding the /3gb switch to your `boot.ini` and have 3 gigs available to user mode and 1GB available to the kernel.) While you cannot create an exact map of the Windows NT 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 SoftICE MOD and 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.

Note: LDT data structures are created from the Paged Pool area.

- System View area

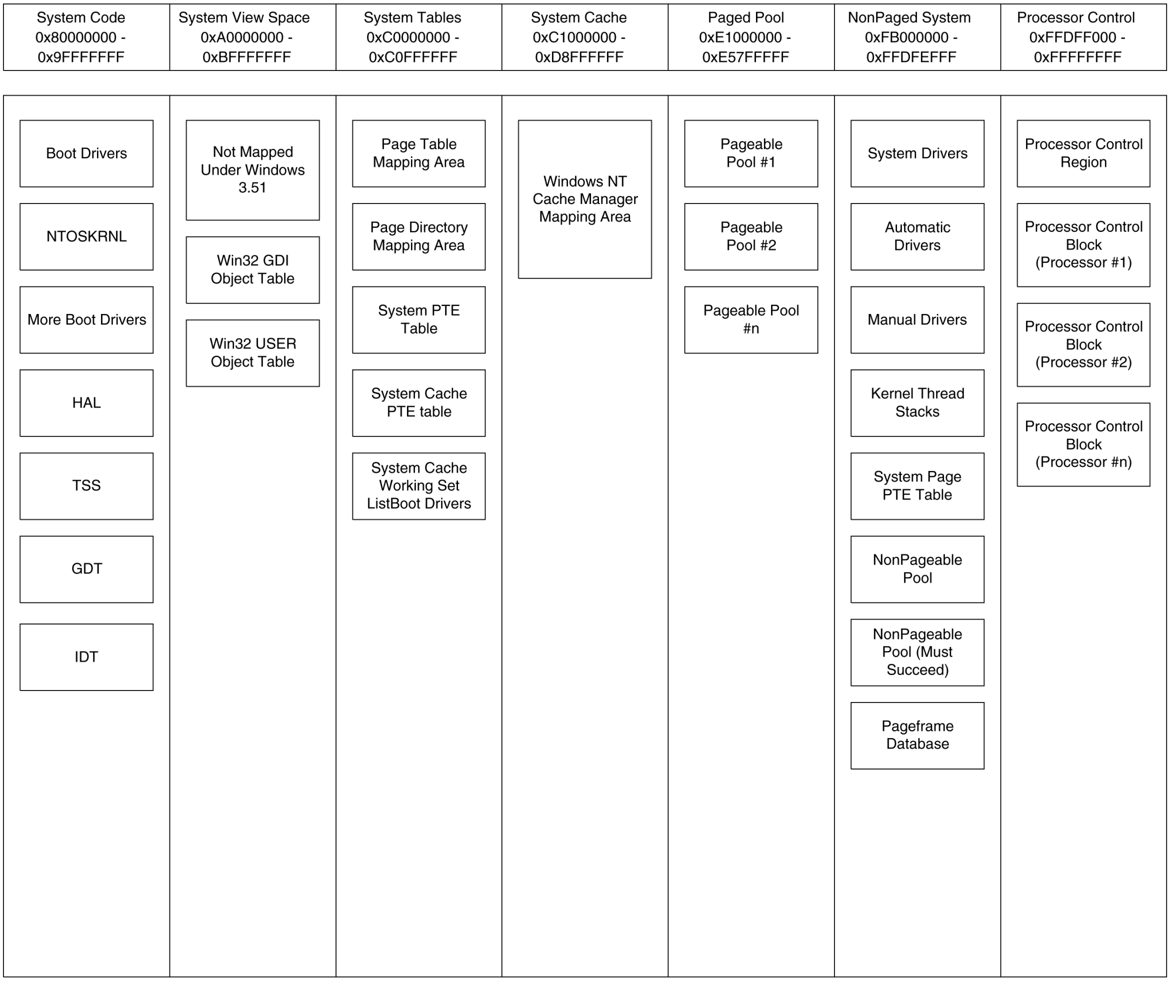
  The System View address space is symbolically referenced, but does not ever seem to be mapped under Windows NT 3.51. Under newer versions of Windows NT, the System View address space maps the global tables for GDI and USER objects. You can use the SoftICE OBJTAB command to view information about the 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 NT 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. In SoftICE terminology, the Page Directory is essentially an Address Context. When you use the SoftICE ADDR command to change to a specific process context, you are *loading the Page Directory information for this process*.

  To manage the mapping of linear memory to physical memory, Windows NT reserves a 4MB region of the system linear address space for Page Tables. This 4MB 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 (which is 4KB under Windows NT), so multiplying 1024 by 4096 (the page size) yields the expected 4MB value. Thus an operating system that uses paging and a 4KB page size requires 4MB of memory to map the entire address space. Windows NT, Windows 95 and Windows 98 take the simple and efficient approach of using a contiguous region of linear memory for this purpose.

  The diagram on the next page shows the system memory map for Windows NT.



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	Not Mapped Under Windows 3.51	Page Table Mapping Area	Windows NT Cache Manager Mapping Area	Pageable Pool #1	System Drivers	Processor Control Region
NTOSKRNL	Win32 GDI Object Table	Page Directory Mapping Area		Pageable Pool #2	Automatic Drivers	Processor Control Block (Processor #1)
More Boot Drivers	Win32 USER Object Table	System PTE Table		Pageable Pool #n	Manual Drivers	Processor Control Block (Processor #2)
HAL		System Cache PTE table			Kernel Thread Stacks	Processor Control Block (Processor #n)
TSS		System Cache Working Set ListBoot Drivers			System Page PTE Table	
GDT					NonPageable Pool	
IDT					NonPageable Pool (Must Succeed)	
					Pageframe Database	

Figure 12-1. Windows NT System Memory Map

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 of 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 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 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 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.

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

- Use the ADDR command to obtain the *physical* address of the Page Directory (CR3).

```
:addr
CR3           LDT Base:Limit      KPEB Addr     PID      Name
00030000                          FF116020      0002     System
0115A000                          FF0AAA80      0051     RpcSs
0073B000                          FF083020      004E     nddeagnt
00653000      E13BB000:0C3F       FF080020      0061     ntvdm
00AEE000                          FF07A600      0069     Explorer
01084000                          FF06ECA0      0077     FINDFAST
010E9000                          FF06CDE0      007B     MSOFFICE
*01F6E000                         FF088C60      006A     WINWORD
01E0A000                          FF09CCA0      008B     4NT
017D3000      E1541000:018F       FF09C560      006D     ntvdm
00030000                          80140BA0      0000     Idle
```

- 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
```

- 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
```

- Use the PAGE command without any parameters to view the mapping of the entire linear address range. This is useful for obtaining the physical address of the Page Directory and verifying that the operating system page tables are mapped at linear address 0xC0000000. The output for this command is abbreviated.

```
:page

Page Directory    Physical=01F6E000

Physical          Attributes          Linear Address Range

01358000          P     A S RW        A0000000 - A03FFFFF

017F0000          P     A S RW        A0400000 - A07FFFFF

01727000          P     A S RW        A0800000 - A0BFFFFF

01F6E000          P     A S RW        C0000000 - C03FFFFF

0066F000          P     A S RW        C0400000 - C07FFFFF

00041000          P     A S RW        C0C00000 - C0FFFFFF

00042000          P     A S RW        C1000000 - C13FFFFF
```

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 NT 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 NT 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 NT 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 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. An important design note, in case you plan on using similar techniques in your driver or application, is that the overhead for a Paged Pool (or Non-Paged Pool) descriptor is very minimal. Thus its 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 NT maintains Processor Control Block (PRCB) 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 use the SoftICE intrinsic function, *IRQL*, as follows:

  ```
  ? IRQL
  <uchar> = 0x2, 2
  ```

  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. SoftICE makes it much easier to get the current thread pointer or thread id by using the intrinsic functions *thread or tid*:

```
? thread
<void *> = 0xFF088E90

? tid
<ushort> = 0x0071
```

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 NT 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)
```

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, most of the core functionality is actually implemented in WINSRV.DLL and WIN32K.SYS 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 have their functionality implemented in 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. The WIN32K.SYS module allows USER and GDI services to execute more efficiently through a simple transition from user to system mode. Depending on which processor and OS you are using, this can occur through a SYSENTER instruction or through an int 2e. 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 SoftICE HEAP32 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 SoftICE QUERY command, but do not appear in the output of the HEAP32 command. The reason for this is quite simple: for each user-mode process, a list of process heaps is maintained and the SoftICE HEAP32 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 SoftICE QUERY command traverses the user-mode address space for your application, using the SoftICE WHAT engine to identify regions of memory that are mapped.

When the WHAT engine encounters a region whose base address is equivalent to a heap that is listed as part of the process heap list, it is identified as a heap. If the WHAT engine cannot identify a region as a heap in this manner, it probes the data area looking for key signatures that identify the area as heap or heap segment.

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 NT, 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 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 NT 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

The protected Win32 subsystem process, CSRSS, provides a majority of the traditional USER functionality. APIs and data structures provided by the WINSRV.DLL and WIN32K.SYS modules manage window classes and window data structures, as well as many other USER data types.

The following USER object types exist. Object type IDs are listed in parentheses.

FREE (0)	Object Entry is unused/invalid.
HWND (1)	Window Objects.
MENU (2)	Windows MENU object.
ICON/CURSOR (3)	Windows ICON or CURSOR object.
DEFERWINDOWPOS (4)	Object returned by the BeginDeferWindowPosition API.
HOOK (5)	Windows Hook thunk.

THREADINFO (6)	CSRSS Client Thread Instance Data.
CLIPBOARD FORMAT (7)	Registered Clipboard Formats.
CPD (8)	Call Procedure Data thunk.
ACCELERATOR (9)	Accelerator Table Object.
WINDOW STATION (0xD)	
KEYBOARD LAYOUT (0xE)	Object to describe a keyboard layout.
DDEOBJECT (0xA)	DDE Objects such as strings.

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 1. 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 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 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 SoftICE WHAT command as follows:

```
:what ff0e45d8
The value FF0E45D8 is (a) Kernel Desktop object (handle=0068) for
winlogon(21)
```

The value at offset 0x14, of 0xE12E7008, is the same value that was in the object entry owner field.

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 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 syntax “esp->c” dereferences the requested object type, and is equivalent to *(esp+c). 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 12-4. 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 NT, 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 NT is 0x10000. Processes often change their load address to a different base address. Applications that were designed to run on Windows 95 and Windows 98 have a default load address of 0x400000. Use the linker or the REBASE utility to set the default load address of a DLL or EXE.

The linear range at 0x70000000 is an approximation of the area where Win32 subsystem modules load. Use the SoftICE MOD, MAP32, or QUERY 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 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 SoftICE QUERY command. The QUERY command provides a high-level view of the linear regions that were reserved and/or committed. It uses the 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 8 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.

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.

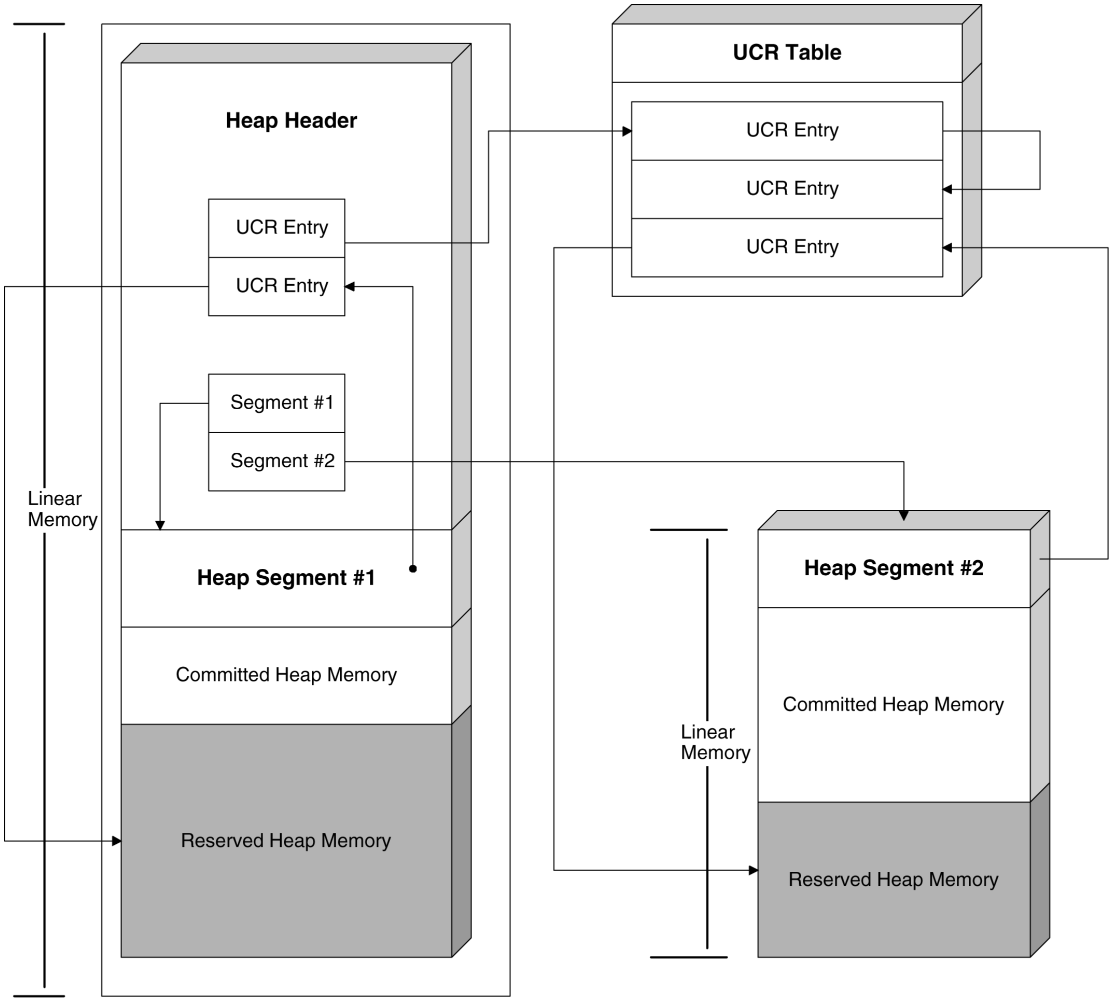


Figure 12-2. Typical Heap Construction

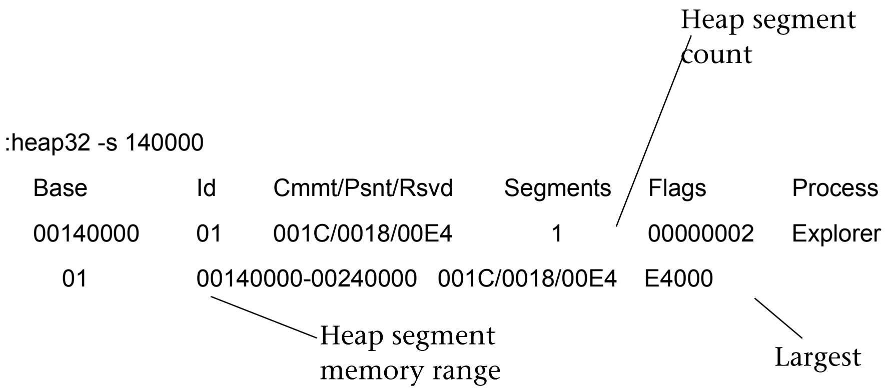
On the right side of the diagram, a second area of virtual memory was allocated and is managed by Heap Segment #2. Additional heap segments are created when an allocation request exceeds the size of the largest uncommitted range within the existing segment. This is only true if the size of the requested 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.

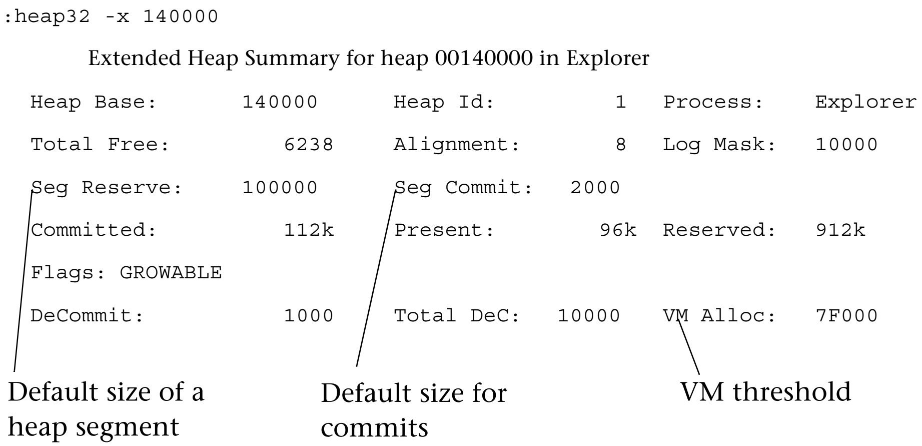
The following examples use the SoftICE HEAP32 command to examine the default heap for the Explorer process.

1 Use the -S option of the HEAP32 command to display segment information for the default heap:



```
:heap32 -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 HEAP32 command to display extended information about the default heap:



```
:heap32 -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
```

3 Use the -B option of the HEAP32 command to display the base addresses of heap blocks within the default heap:

```
:heap32 -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 Figure 12-3, 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. 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).

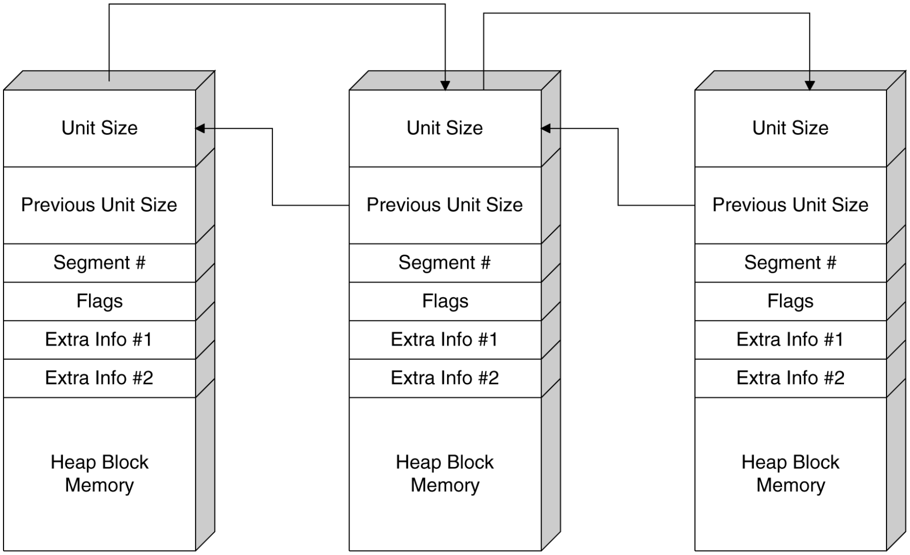


Figure 12-3. Tracking Blocks of Contiguous Memory

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.

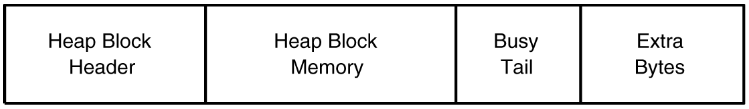


Figure 12-4. Basic Architecture of an Allocated Heap Block

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.

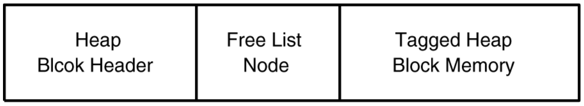


Figure 12-5. Basic Architecture of a Free Heap Block

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 the 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.

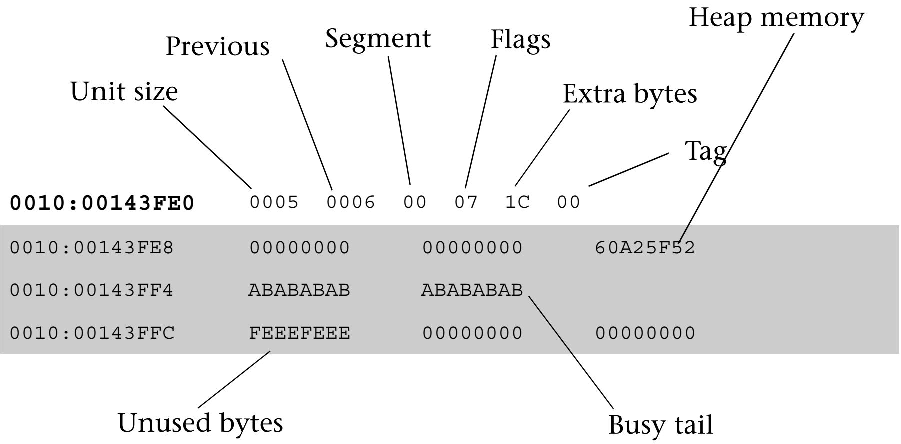
The following two examples show how to use the SoftICE HEAP32 command to aid in monitoring and debugging Win32 heap issues.

The first example uses the HEAP32 command to walk all the entries for the heap based at 0x140000. The -B option of the HEAP32 command causes the base address and size information to display as the heap manager would view the information. Without the -B option, the HEAP32 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.

```
:HEAP32 -b 140000

Base         Type        Size        Seg#        Flags

00140000     HEAP        580         01

00140580     SEGMENT     38          01          TAGGED | BUSYTAIL

001405B8     ALLOC       40          01

. . .

00143FE0     ALLOC       28          01          TAGGED | BUSYTAIL

00144008     FREE        FF8         01          FREECHECK | SENTINEL
```

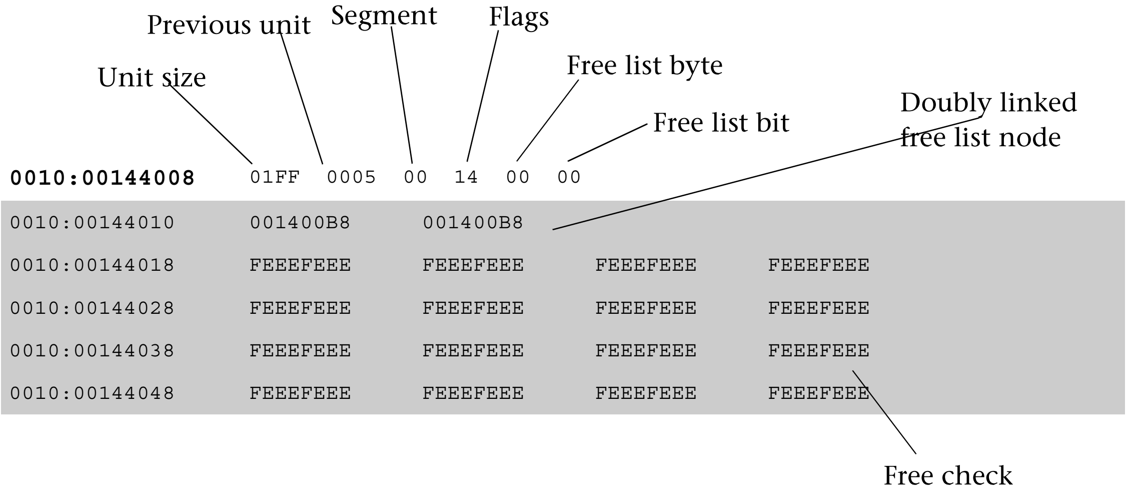
To examine the contents of an allocated heap block and a free block, the second example dumps memory at the base address of the heap block at 0x143FE0. Enough memory is dumped to show the subsequent block, which is a free block at address 0x144008.

- The heap block header fields from the memory dump at address 0x143FE0 are identified with call-outs. This heap block is 5 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 5 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).



```
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
```

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.

Appendix A

Error Messages

All break registers used, use in RAM only

You were trying to set a BPX breakpoint in ROM and all the debug registers were already used. BPX will still work in RAM, because it uses the INT 3 method. You must clear one of the BPM-style breakpoints before this will work.

Attach to serial device has FAILED

The initial serial handshaking sequence failed. This might happen if the wrong serial port is selected, the target machine is not running SERIAL.EXE, or the serial cable is faulty.

BPM breakpoint limit exceeded

Only four BPM-style breakpoints are allowed due to restrictions of x86 processors. You must clear one of the BPM-style breakpoints before this will work.

BPMD address must be on DWord boundary

The address specified in BPMD did not start on a Dword boundary. A Dword boundary must have the two least significant bits of the address equal 0.

BPMW address must be on Word boundary

The address specified in BPMW did not start on a Word boundary. A Word boundary must have the least significant bit of the address equal 0.

Breakpoints not allowed within SoftICE

You cannot set breakpoints in SoftICE code.

Cannot interrupt to a less privileged level

You cannot use the GENINT command to go from a lower level to a higher privilege level. This is a restriction of the x86 processor.

Debug register is already being used

Debug-register specified in BPM command was already used in a previous

BPM command.

Duplicate breakpoint

The specified breakpoint already exists.

Expecting value, not address

The expression evaluator broadly classifies operands as addresses and values. Addresses have a selector/segment and offset component even if the address is flat. Certain operators such as * and / expect only plain values, not addresses, and an attempt to use them on addresses produces this message. In some cases using the indirection operators produces an address; refer to *Supported Operators* on page 154 for details.

Expression?? What expression?

The expression evaluator did not find anything to evaluate. Note that in some older versions of SoftICE the ? command could be used to get help. This is no longer the case; use the H command (F1).

Int0D fault in SoftICE at address XXXXX offset XXXXX Fault Code=XXXX

(or the following message)

Int0E Fault in SoftICE at address XXXXX offset XXXXX Fault Code=XXXX

These two messages are internal SoftICE errors. The code within SoftICE caused either a general protection fault (0D) or a page fault (0E). The offset is the offset within the code that caused the fault. Please write down the information contained in the message and e-mail or call us. These messages also display the values in the registers. Be sure to write down these values also.

Invalid Debug register

A BPM debug-register greater than 3 was specified. Valid debug registers are DR0, DR1, DR2, and DR3.

No code at this line number

The line number specified in the command has no code associated with it.

No current source file

You entered the SS command and there was no source file currently on the screen.

No embedded INT 1 or INT 3

The ZAP command did not find an embedded interrupt 1 or interrupt 3 in the code. The ZAP command only works if the INT 1 or INT 3 instruction is the one before the current CS:EIP.

No files found

The current symbol table does not have any source files loaded for it.

No LDT

This message displays when you use certain 16-bit Windows information commands (HEAP, LHEAP, LDT, and TASK) and the current context is not set to the proper NTVDM process.

No Local Heap

The LHEAP command specified a selector that has no local heap.

No more Watch variables allowed

A maximum of eight watch variables are allowed.

No search in progress

You specified the S command without parameters and no search was in progress. You must first specify S with an address and a data-list for parameters. To search for subsequent occurrences of the data-list, use the S command with no parameters.

NO_SIZE

During an A command, the assembler cannot determine whether you wanted to use byte, word, or double word.

No symbol table

You entered the SYM, SS, or FILE command and there are no symbols currently present.

No TSS

You entered the TSS command while there was no valid task state segment in the system.

Only valid in source mode

You cannot use the SS command in mixed mode or code mode.

Page not present

The specified address was marked not present in the page tables. When SoftICE was trying to access information, it accessed memory that was in a page marked not present.

Parameter is wrong size

One of the parameters you entered in the command was the wrong size. For example, if you use the EB or BPMB commands with a word value instead of a byte value.

Pattern not found

The S command did not find a match in its search for the data-list.

Press ‘C’ to continue, and ‘R’ to return to SoftICE

SoftICE popped up due to a fault (06, 0C, 0D, 0E). Press R to return control to SoftICE. Press C to pass the fault on to the Windows fault handler.

SoftICE is not active

This message displays on the help line on monochrome and serial displays when SoftICE is no longer active.

Specified name not found

You typed TABLE with an invalid table-name. Type TABLE with no parameters to see a list of valid table names.

Symbol not defined (mysymbol)

You referred to a non-existent symbol. Use the SYM command to get a list of symbols for the current symbol table.

Appendix B

Supported Display Adapters

The following table lists the display adaptors SoftICE supported when the product most recently shipped. However, Compuware regularly adds new display adaptor support to enhance SoftICE. You can download the latest support files from the Compuware FTP or BBS sites. Refer to *Installing SoftICE* in *Getting Stared with DriverStudio* for more information about downloading support files.

Supported Display Adaptors

Standard Display Adapter (VGA)	Actix GraphicsEngine 32I VL	Actix GraphicsEngine 32VL Plus
Actix GraphicsEngine 64	Actix GraphicsEngine Ultra 64	Actix GraphicsEngine Ultra Plus
Actix GraphicsEngine Ultra VL Plus	Actix ProSTAR	Actix ProSTAR 64
ATI 8514-Ultra	ATI Graphics Pro Turbo	ATI Graphics Pro Turbo PCI
ATI Graphics Ultra	ATI Graphics Ultra Pro	ATI Graphics Ultra Pro EISA
ATI Graphics Ultra Pro PCI	ATI Graphics Vantage	ATI Graphics Wonder
ATI Graphics Xpression	ATI 3d Xpression PCI	ATI VGA Wonder
ATI Video Xpression PCI	ATI WinTurbo	Boca SuperVGA
Boca SuperX	Boca Voyager	Cardinal VIDEOcolor
Cardinal VIDEOspectrum	Chips & Technologies 64310 PCI	Chips & Technologies 65545 PCI
Chips & Technologies 65548 PCI	Chips & Technologies Accelerator	Chips & Technologies Super VGA
Cirrus Logic	Cirrus Logic 5420	Cirrus Logic 5430 PCI
Cirrus Logic New	Cirrus Logic PCI	Cirrus Logic RevC
Cirrus Logic 7542 PCI	Cirrus Logic 7543 PCI	Compaq Qvision 2000
DEC PC76H-EA	DEC PC76H-EB	DEC PC76H-EC
DEC PCXAG-AJ	DEC PCXAG-AK	DEC PCXAG-AN

Supported Display Adaptors

DFI WG-1000	DFI WG-1000VL Plus	DFI WG-1000VL/4 Plus
DFI WG-3000P	DFI WG-5000	DFI WG-6000VL
Diamond Edge 3D 2200XL	Diamond Edge 3D 3200XL	Diamond Edge 3D 3400XL
Diamond SpeedStar	Diamond SpeedStar 24	Diamond SpeedStar 24X
Diamond SpeedStar 64	Diamond SpeedStar Pro	Diamond SpeedStar Pro SE
Diamond Stealth 3D 2000	Diamond Stealth 24	Diamond Stealth 32
Diamond Stealth 64 2001	Diamond Stealth 64 (S3 964)	Diamond Stealth 64 (S3 968)
Diamond Stealth 64 Video	Diamond Stealth Pro	Diamond Stealth SE
Diamond Viper OAK	Diamond Viper PCI	Diamond Viper VLB
Diamond Stealth VRAM	ELSA WINNER 1000AVI	ELSA WINNER 1000PRO
ELSA WINNER 1000Trio	ELSA WINNER 1000 VL	ELSA WINNER 1280
ELSA WINNER 2000PRO	ELSA WINNER 2000 VL	ELSA WINNER/2-1280
Genoa Digital Video Wizard 1000	Genoa Phantom 32I	Genoa Phantom 64
Genoa WindowsVGA 24 Turbo	Genoa WindowsVGA 64 Turbo	Hercules Dynamite
Hercules Dynamite Pro	Hercules Graphite 64	Hercules Graphite Terminator 64
Hercules Graphite Terminator Pro	IBM 8514	IBM ThinkPad 755CX
IBM Think Pad 365XD	Matrox MGA Impression Lite	Matrox MGA Impression Plus
Matrox MGA Impression Plus 220	Matrox MGA Ultima Plus	Matrox MGA Ultima Plus 200
Matrox MGA Millennium	Number Nine GXE	Number Nine GXE64
Number Nine GXE64 Pro	Number Nine 9FX Vision 330	Number Nine 9FX Motion 531
Number Nine 9FX Motion 771	Number Nine FlashPoint 32	Number Nine FlashPoint 64
Number Nine Imagine 128	Number Nine Reality 332	Nvidia NVI Media Controller
Oak Technology 087	Oak Technology Super VGA	Orchid Fahrenheit 1280 Plus
Orchid Fahrenheit Pro 64	Orchid Fahrenheit VA	Orchid Kelvin 64
Orchid Kelvin EZ	Orchid ProDesigner II	Paradise Accelerator Ports O'Call
Paradise Accelerator VL Plus	Paradise Bahamas	Paradise Barbados 64

Supported Display Adaptors		
Paradise Super VGA	S3 805	S3 911/924
S3 928 PCI	S3 Trio32/64 PCI	S3 ViRGE PCI
S3 Vision864/964 PCI	S3 Vision868/968 PCI	Spider 32 VLB
Spider 32Plus VLB	Spider 64	Spider Tarantula 64
STB Ergo MCX	STB Horizon	STB Horizon Plus
STB LightSpeed	STB MVP-2X	STB MVP-4X
STB Nitro	STB Pegasus	STB PowerGraph Pro
STB PowerGraph VL-24	Trident 9420 PCI	Trident Cyber 93XX
Trident Super VGA	Tseng Labs	Tseng Labs ET4000
Tseng Labs ET4000/W32	Tseng Labs ET6000	Video Logic 928Movie
Video Seven VRAM/VRAM II/1024i	Western Digital	Western Digital (512K)
Weitek Power 9000	Weitek Power 9100	

Appendix C
Troubleshooting SoftICE

If you encounter any of the following problems, try the corresponding solution. If you encounter further difficulties, 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.

Problem	Solution
The SoftICE screen is black or unreadable.	Either your display adaptor does not match the display adaptor set at installation or SoftICE does not support your display adaptor. Refer to *Appendix B:* on page 255.
The PC crashes when you run SoftICE and you are not using a Pentium or Pentium-Pro processor.	SoftICE incorrectly determined that your system is using a Pentium processor. Modify the SoftICE Initialization Settings to disable Pentium support. Refer to *Setting Troubleshooting Options* on page 208.
The PC crashes when you run SoftICE for Windows 9x.	SoftICE does not support the shutdown option RESTART THE COMPUTER IN MS-DOS MODE?. If you reload SoftICE after choosing this option, SoftICE eventually crashes. Instead, change the statement BootGUI=1 to BootGUI=0 within the Windows 95 and Windows 98 hidden file MSDOS.SYS. Then, choose SHUT DOWN THE COMPUTER? to exit to DOS.
You have difficulty establishing a modem connection.	The modem is returning result codes SoftICE does not expect. SoftICE looks for the codes OK, COMNECT, and RING. Place ATXO in the initialization string.

Problem	Solution
The mouse behaves erratically within SoftICE.	Press Ctrl-M.
Windows NT only: the mouse pointer behaves erratically in the SoftICE screen.	Moving the mouse while the SoftICE screen pops up, can cause Windows NT and the mouse hardware to become out of synchronization. Switch to a full screen DOS box.
Your keyboard locks or behaves erratically when you load SoftICE.	Modify the SoftICE Initialization Settings to disable num lock and caps lock programming. If this does not work and you are using Windows NT, instruct SoftICE not to patch the keyboard driver. Refer to *Setting Troubleshooting Options* on page 208.
Windows 9x crashes when attempting to scan for serial ports.	If you placed the SERIAL command in the Initialization string, SoftICE establishes a connection to the port before Windows 9x initializes. When Windows 9x initialize, it might scramble the connection. Disable the port selected in the Device Manager. The Device Manager is located within the System Properties in your Control Panel.

Appendix D
Kernel Debugger Extensions

SoftICE for the Windows NT family supports Kernel Debugger (KD) Extensions written for WinDBG. SoftICE will take a WinDBG extension, convert it to a Kernel mode driver, and allow the user to execute informational commands. Users can also write their own extensions following the WinDBG interface (as found in Wdbgexts.h), and convert them for use in SoftICE.

To prepare a KD Extension for use with SoftICE:

1 Use the KD2SYS or KD2SYSXLAT program to convert the DLL to a system driver. This program:

   a Copies the DLL to the \SYSTEMROOT\SYSTEM32\DRIVERS directory and gives it an extension of .SYS

   b Modifies the file to tell the system that the file can be loaded as a system driver and redirect many API calls to SoftICE

   c Creates the necessary keys in the system registry to identify the new file as a system driver

2 Reboot the system. When any system drivers (services) are added or removed from your system, it must be rebooted. This allows the service control manager to refresh the list of services in the system.

3 If you are starting SoftICE manually, you will need to start the extension, in this case by using the “NET START <KDExtension name>” command from the command prompt to load the extension into SoftICE.

   If you are using other start modes, the extension will be started automatically at the appropriate time. Further, when you change the start mode of SoftICE using the ‘Startup Mode Setup’ shortcut, all extensions will be changed to start with SoftICE.

4 After the service is started, press Ctrl-D to open the SoftICE window. Type ‘!?’ or ‘!help’ to get a list of the commands and a short explanation of each one.

The requirements for using Kernel Debugger Extensions are listed below:

1. You must have the current NTOSKRNL.nms loaded. Translate the .dbg file and use Loader32 to automatically load the file when SoftICE starts.
2. No file IO is allowed in a KD Extension. The DLL will be converted, but any attempt to call a file IO function will result in the command that issued the request being terminated.
3. Do not use exception handling in a KD Extension. Again, the extension will convert, but any command that attempts to execute an exception handler will be terminated.
4. A default stack of 32k and a default heap of 8k are allocated when SoftICE starts. These values can be increased or decreased via the registry keys: KDHeapSize and KDStackSize (HKey_LocalMachine\CurrentControlSet\Services\NTICE).

   If you change the values using the registry keys, a reboot will be necessary to refresh the values.

Appendix E

SoftICE and VMware

Beginning with SoftICE 3.1 and VMware 4.0, SoftICE can be used as a debugger within a Windows based “virtual machine.” The host operating system can be any OS that VMware supports. There are certain restrictions, limitations, and differences between SoftICE running on a “virtual machine” and SoftICE running on a real machine. SoftICE can be used as a single machine debugger with the UVD. It can also be used with the standard VGA driver (without the VMware Tools installed). Remote debugging can be accomplished on the same machine between the physical host machine and the virtual machine with no cables involved, or you can perform remote debugging using the serial port or named pipes.

OS Support

The operating systems supported by SoftICE for virtual machines are the same as those on physical machine. SoftICE 3.1 supports Win9x, Win31, DOS, and NT4 through frozen versions. Win2k and later are in active development.

Hardware Support

This is where the differences between SoftICE on a physical machine and SoftICE on a virtual machine come into play. Because all hardware within VMware is virtualized, a few oddities seem to occur. These are detailed below in the limitations and setup section.

Setup/Installation

For installation, do the normal DriverStudio/SoftICE installation. Be certain that your VMware virtual OS is completely configured with VMware Tools, if you choose to have the tools installed prior to installing SoftICE. If you install the VMware Tools afterwards, you will need to reconfigure your SoftICE settings by bringing up the DriverStudio/SoftICE Configuration dialog in the virtual OS and reselecting your video and possibly mouse settings.

Limitations and Restrictions

Due to the nature of the virtualized hardware there are several features that do not work within “virtual” SoftICE.

By default, the UVD will not draw properly in SoftICE. You will need to set the “svga.maxFullscreenRefreshTick” in your VMware configuration file as specified in the “Universal Video Driver” section below.

Remote debugging via TCP/IP is not operational at this time. For remote debugging, use the serial port either with a physical cable or with a named pipe. (See the “Remote Debugging” section below.)

Remote Debugging

There are several options available when performing remote debugging. For remote debugging, you can connect machines by one of several methods. All remote debugging is limited to either the serial port or a named pipe. The preferred method of remote debugging is over named pipes. TCP/IP based network debugging is not operational in this release.

The methods of serial connection that SoftICE support are:

- Between virtual machine and VMware host over physical serial port
- Between virtual machine and VMware host over named pipes for serial ports
- Between two virtual machines over physical serial port
- Between two virtual machines over virtual serial ports

Configuration

The following procedures are needed for all serial connections types.

1. Within the virtual OS, run DriverStudio/SoftICE Settings and choose the "Serial Debugging" tab. Choose the serial port in the "Serial Connection" drop list. Note that serial port X in this dialog box should match the VMware Virtual Machine Setting's "Serial Y" which may use any arbitrary physical serial port.

2. Within SoftICE, start your remote connection exactly as if you had real hardware. This means that you should use the 'net comX baudrate' or check the Auto Connect option in the SoftICE Serial Debugging page.

> Note The remaining configuration steps will differ based upon the method of serial connection chosen.

3. If you choose '(1) between virtual machine and VMware host over physical serial port', you will need to:

   ◇ Have two unused serial ports on your machine.

   ◇ Connect a null modem cable between these two ports.

   ◇ From within VMware, edit the Virtual Machine Settings. Add a serial port to the virtual machine if it is not already in the setting. Choose 'Serial N', making sure that the following items are checked:

     – *Connect at power on.*

     – *Use physical serial port.* Choose the proper serial port and be certain that this serial port is also chosen in the DriverStudio/SoftICE Settings in the virtual OS and is used on the SoftICE command line.

     – *Yield CPU on poll.* On the host machine run the 'siremote comY baudrate'. The comY needs to be the comm port that is not used by the VMware session.

   At this point you should have a connection. If not, go back and verify each step above.

4. If you choose '(2) between virtual machine and VMware host over named pipes for serial ports', you will need to:

   ◇ From within the VMware, edit the Virtual Machine Settings. Choose 'serial N', making sure the following items are checked:

     – *Connect at power on.*

    - *Use Named Pipe*. For the name, choose whatever come to mind - a good name might be `\\.pipe\sipipe`

  - Choose “This end is the server”, “The other end is an application”

  - Yield CPU on poll.

  - On the host machine, run `siremote PIPE sipipe`. The name part of the pipe will match whatever is set in the VMware Virtual Machine Settings.

  At this point you should have a connection. If not, go back and verify each step above.

5. If you choose ‘(3) between two virtual machines over physical serial port’, the setup and settings are identical to the first entry, depending upon your chosen connection type.

6. If you choose ‘(4) between two virtual machines over virtual serial ports’, you will need to configure both VMware sessions and SoftICE.

   - On both virtual machines, you will need to setup the comm ports to use a named pipe.

   - On the virtual machine running the SoftICE debugger, you will need to configure the following:
     - In the VMware configuration settings, choose the serial port on which to use a named pipe, “This end is the server”, and “The other end is a virtual machine”
     - Within the VMware session, enable SoftICE serial debugging on ComX, where X is the value that is in the VMware configuration of “Serial X”
     - Within SoftICE, choose `net comX baudrate`

   - On the other virtual machine that will be running siremote, you will need to configure it as follows:
     - In the VMware configuration settings, choose the serial port that will use a named pipe, “This end is the client”, and “The other end is a virtual machine”
     - Open to a Command Prompt, change to the SoftICE directory, and issue the command `siremote comX baudrate`, where X is the value that is in the VMware configuration of “Serial X”

Mouse

If you encounter lockups, you may need to disable the mouse within the configuration file for the VM in question. You can find the VM by going to the directory where the VM is located and edit the VMware configuration text file that ends in .vmx. Add the following entry to the file:

```
vmmouse.present = "FALSE"
```

Universal Video Driver

In UVD mode, SoftICE does not correctly redraw inside VMware. This is due to the virtual machine not recognizing direct writes into the VM’s frame buffer. To work around this, you will need to add an entry to your VMware configuration file (located in the directory of the VM with an extension of .vmx):

```
svga.maxFullscreenRefreshTick = "2"
```

The lower you set this value, the more responsive the SoftICE screen will be. Setting it to 1 will cause SoftICE to redraw on par with a physical single machine. Higher values will delay the redraws. The downside to setting a lower value is that mouse flickering increases within the VM.

Appendix F

SoftICE API Specification

This appendix provides an overview of:

- The Purpose of Having a Public Interface and API
- Setting Up SoftICE for API Access
- Running the Sample Driver
- Setting Up Your Driver
- Checking for the Existence of SoftICE
- Using the SoftICE API
- API Definition

The API feature of SoftICE will only apply to specific users in specific cases. Unless you have a need for accessing SoftICE externally, we suggest that you not take advantage of the public interface.

By default SoftICE does not expose a public device interface. This is done by design, as there have been past incompatibilities between SoftICE and third-party software. We are now providing a means to create a user-defined public device name.

The Purpose of Having a Public Interface and API

There are several purposes for defining a public interface for SoftICE. The main reason for defining a public interface is that you can add conditional code to your driver that will execute if SoftICE is present. For example, you may want to add an embedded int3 into your DriverEntry, but only if SoftICE is running.

Note: Having an int3 in your code, without a debugger present, would cause a blue screen.

Note: The SoftICE API is subject to change at any point in time. We will do our best retain any functionality once it exists, and maintain backwards compatibility. However, in some extreme cases we may not be able to do so. There are no guarantees with the API, other than that we will fix discovered bugs, and listen to your suggestions for enhancements.

Setting Up SoftICE for API Access

1. Edit your winice.dat file by whatever means you are comfortable with. We suggest you use the **Advanced** page of the **SoftICE Initialization** tab in the **Settings** utility (from the **Start** menu under **Compuware|Driver Studio**).

2. Make a new entry entitled PUBLICDEVICENAME=nameofyourchoice The name that you enter must be the pure device name without the starting \\device\\ and it must adhere to all valid naming conventions as defined by the DDK. No validation is done on this name. Internally, when the name is created, a \\device\\ is pre-pended for the device name and \\DosDevices\\ is pre-pended for the symbolic link.

3. Reboot your machine if SoftICE is already running.

Note: If there is a problem creating the device when SoftICE starts, an entry will be added to **Event Viewer** under the **System** tab. Within SoftICE this event information is also available via the ver –x command.

Running the Sample Driver

We have provided sample code that shows how to open up and query for the existence of SoftICE. Note that this sample will exit after running and is not indicative of a real driver. We have only provided the necessary code to access SoftICE.

1. Search for SI_PUBLIC_NAME and change this to the name that you defined in your winice.dat for PUBLICDEVICENAME.

2. Compile the sample driver located at
   c:\program files\Compuware\driverstudio\SoftICE\API\Sample\QuerySI
   A .reg file has been included for installation. Any means of driver installation can be used.

3. Optionally, start up SoftICE.

4. Issue net start querysi from a dos command box, or by whatever means you like to start a driver.

Inside of SoftICE (in the command window) you should see several DbgPrint messages detailing each step of the process (driver startup, query for existence, version number querying, and exit).

Setting Up Your Driver

We suggest you copy or implement your code based on the sample code in querysi.cpp.

Note: Pay particular attention to the Si_XXXXX calls.

Checking for the Existence of SoftICE

For a driver, get a device object pointer to SoftICE using the name that you defined in your winice.dat for PUBLICDEVICENAME.

This can be done through, IoGetDeviceObjectPointer. If you are able to get this object pointer, it means that SoftICE is loaded and running with a public interface. Not being able to get an object pointer means either that SoftICE is not running, or it has not had a public interface defined.

Note: We suggest that you store the existence of SoftICE in a global variable and avoid constantly trying to get a device object pointer for each conditional piece of code.

Note: Once you have retrieved the object pointer, be certain to delete it when you don't need it and/or when your driver is shutting down.

For an application, you will want to make a call to:

```
handleTest =
CreateFile("\\\\.\\whatever_your_public_devicename_is",
GENERIC_READ|GENERIC_WRITE, 0, NULL, OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL, NULL);
```

If you are able to get the handle, SoftICE is loaded and has a publicly defined interface. Once you have the handle, make a call to:

```
Close(handleTest);
```

Note: We suggest that you store the existence of SoftICE in a global variable and avoid constantly trying to get a device object pointer for each conditional piece of code.

Using the SoftICE API

The SoftICE API will be an ever expanding set of functionality to meet the needs of our users. In its initial incarnation it is very small and limited. It will grow to expand the requests of our customers. Please feel free to submit any requests for new API functionality through our support group or by emailing to:

driverstudioeng@compuware.com

To make API calls from a driver, you will need to use the normal Windows DDK interdriver communication mechanism of IOCTLs. The defined IOCTLs are located in SoftICE\API\Include\si_api.h. This file should be included in your code.

Depending upon what features of the API you are using, you should make your first IOCTL call getting the SoftICE version information, and then modify your code according to the capabilities of the particular version of SoftICE in use. For example, in SoftICE 3.2 we may only define one API IOCTL called GetVersion, while in SoftICE 3.3 we may define a 1000 IOCTLs. If your code assumed that all users will be running SoftICE version 3.3 with its 1000 APIs you could run into problems. Basically, write your code to take advantage of the capabilities for current version and newer. In pseudo code, “if softiceversion is >= 432 then set_a_breakpoint_API”. This way, for users of SoftICE version 4.3.1, your code will still function properly by not attempting to set a breakpoint, whereas users of SoftICE versions later than 4.3.2 will have the enhanced functionality.

API Calls from a Driver

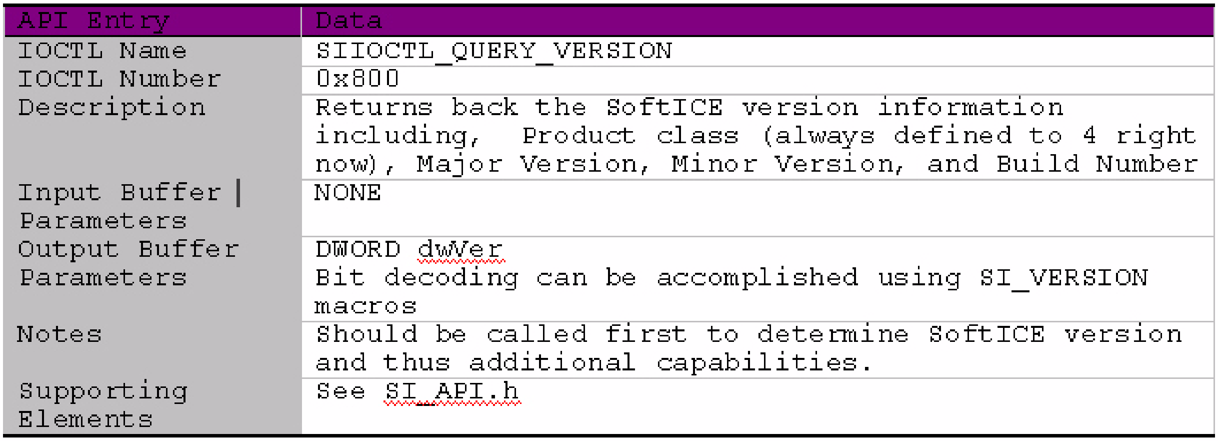
1. Open up a connection to SoftICE when your driver starts
2. For each ioctl call, initialize an event with KeInitializeEvent
3. Build up an ioctl with IoBuildDeviceIoControlRequest
4. Call SoftICE with an IoCallDriver
5. Wait on the event with a KeWaitForSingleObject call
6. Release the connection to SoftICE with ObDereferenceObject once your driver exits, or the connection to SoftICE is no longer needed

API Calls from a Ring3 Application

1. Open up a file handle connection to SoftICE with CreateFile when your application starts
2. For each call into SoftICE, call SoftICE with DeviceIoControl
3. Release the connection with a CloseHandle call once your application exits, or the connection to SoftICE is no longer needed

API Definition

This information is also in the APIs public header file located in your installation directory underneath SoftICE\API\Include\si_api.h.



API Entry	Data
IOCTL Name	SIIOCTL_QUERY_VERSION
IOCTL Number	0x800
Description	Returns back the SoftICE version information including, Product class (always defined to 4 right now), Major Version, Minor Version, and Build Number
Input Buffer Parameters	NONE
Output Buffer Parameters	DWORD dwVer Bit decoding can be accomplished using SI_VERSION macros
Notes	Should be called first to determine SoftICE version and thus additional capabilities.
Supporting Elements	See SI_API.h

Glossary

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 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).

Point-and-Shoot Breakpoint

Breakpoint you set by moving the cursor into the code window using the BPX or HERE command.

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

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