

Using SoftICE™

Windows NT™
Windows® 98
Windows® 95
Windows® 3.1
DOS

Compuware®
NUMEGA™

April 1999

Information in this document is subject to change without notice and does not represent a commitment on the part of Compuware Corporation. The software described in this document may be used or copied only in accordance with the terms of the license. The purchaser may make one copy of the software for a backup, but no part of this user manual may be reproduced, stored in a retrieval system, or transmitted in any form or by any means electronic or mechanical, including photocopying and recording for any purpose other than the purchaser's personal use, without prior written permission from Compuware Corporation.

NOTICE: The accompanying software is confidential and proprietary to Compuware Corporation. No use or disclosure is permitted other than as expressly set forth by written license with Compuware Corporation.

Copyright © 1996, 1998, 1999 Compuware Corporation.

All Rights Reserved.

Compuware, the Compuware logo, NuMega, the NuMega logo, BoundsChecker, SoftICE, and On-Demand Debugging are trademarks or registered trademarks of Compuware Corporation.

Microsoft, Windows, Win32, Windows NT, Visual Basic, and ActiveX are trademarks or registered trademarks of Microsoft Corporation.

Borland and Delphi are trademarks or registered trademarks of Borland International, Incorporated.

Watcom is a trademark of Sybase, Incorporated or its subsidiaries.

Other brand and product names are trademarks or registered trademarks of their respective holders.

This Software License Agreement is not applicable if Licensee has a valid Compuware License Agreement and has licensed this Software under a Compuware Product Schedule.

Software License Agreement

Please Read This License Carefully

You, either an individual or entity ("Licensee"), are purchasing a license ("Agreement") to use this Compuware Corporation software ("Software") and related user manuals ("Documentation"). The Software is the property of Compuware Corporation ("Compuware") and/or its licensors, is protected by intellectual property laws, and is provided to Licensee only under the license terms set forth below. This Agreement does not transfer title to the intellectual property contained in the Software. Compuware reserves all rights not expressly granted herein. Opening the package and/or using the Software indicates your acceptance of these terms. If you do not agree to all of the terms and conditions, or if after using the Software you are dissatisfied, you may return the Software, Documentation and any copies within thirty (30) days of purchase to the party from whom you received it for a refund, subject in certain cases to a restocking fee.

Title and Proprietary Rights: Licensee acknowledges and agrees that the Software is proprietary to Compuware and/or its licensors, and is protected under the laws of the United States and other countries. Licensee further acknowledges and agrees that all rights, title and interest in and to the Software, including intellectual property rights, are and shall remain with Compuware and/or its licensors. Unauthorized reproduction or distribution is subject to civil and criminal penalties.

Use Of The Software: Compuware grants Licensee the limited right to use the Software and Documentation included in the package with this license, subject to the terms and conditions of this Agreement. Licensee agrees that the Software will be used solely for internal purposes. Only one copy of the Software may be installed on a single computer at any one time unless:

- (i) The Software is designed and intended by Compuware for use in a shared network client server environment, as set forth in the Documentation; and
- (ii) Licensee agrees to provide technical or procedural methods to prevent use of the Software, even at different times, by anyone other than Licensee; and
- (iii) Licensee has purchased a license for each individual user of the Software and/or for each computer that will have access to the Software. Any unauthorized use of this Software may cause termination of this Agreement.

Licensee may make one machine-readable copy of the Software for BACK UP PURPOSES ONLY. This copy shall display all proprietary notices, be labeled externally to show that it is the property of Compuware, and that its use is subject to this Agreement. Documentation may not be copied in whole or part. Licensee agrees to provide technical or procedural methods to prevent use of the Software by anyone other than Licensee, even at different times.

Licensee may not use, transfer, assign, export or in any way permit the Software to be used outside the country of purchase, unless authorized in writing by Compuware. Except as expressly provided in this Agreement, Licensee may not modify, reverse engineer, decompile, disassemble, distribute, sub-license, sell, rent, lease, give or in any way transfer the Software, by any means or in any medium, including telecommunications. Licensee will use its best efforts and take all reasonable steps to protect the Software from unauthorized use, copying or dissemination, and will retain all proprietary notices intact.

Redistribution Rights of Device Driver Development Software: This section shall only apply if the Software is device driver development software, used by Licensee to develop application or device driver programs ("User Software"), as specified in the Documentation. The User Software may include run-time components ("RTCs") that have been extracted by the Software from the library files of the Software, programs to remotely test the User Software, and compiled code examples. These RTCs, examples, and programs are specifically designated as redistributable in the Documentation. Licensee has a non-exclusive, royalty-free, restricted license to:

- (i) modify, compile, and distribute the driver code examples;
- (ii) distribute the remote testing program for testing purposes only;
- (iii) embed the RTCs and driver code examples in its User Software, in object code form only; and
- (iv) reproduce and distribute the RTCs and driver code examples embedded in its User Software, in object code form only, provided that:
 - (a) Licensee distributes the RTCs and driver code examples only in conjunction with and as a part of its User Software;
 - (b) Licensee will be solely responsible to anyone receiving its User Software for any updates, technical and other support obligations, and any other liability which may arise from the distribution of its User Software;
 - (c) Licensee does not use Compuware's or its licensors' names, logos, or trademarks to market or distribute its User Software;
 - (d) Licensee includes Compuware's and its licensors' copyright and/or proprietary notices and legends within the executable images of its User Software and on Licensee's software media and documentation; and
 - (e) Licensee agrees to indemnify, hold harmless and defend Compuware and its licensors from and against any claims or lawsuits, including attorney's fees, that arise or result from the use or distribution of its User Software.

Government Users: With respect to any acquisition of the Software by or for any unit or agency of the United States Government, the Software shall be classified as "commercial computer software," as that term is defined in the applicable provisions of the Federal Acquisition Regulation (the "FAR") and supplements thereto, including the Department of Defense (DoD) FAR Supplement (the "DFARS"). If the Software is supplied for use by DoD, the Software is delivered subject to the terms of this Agreement and either (i) in accordance with DFARS 227.7202-1(a) and 227.7202-3(a), or (ii) with restricted rights in accordance with DFARS 252.227-7013(c)(1)(ii) (OCT 1988), as applicable. If the Software is supplied for use by a Federal agency other than DoD, the Software is restricted computer software delivered subject to the terms of this Agreement and (i) FAR 12.212(a); (ii) FAR 52.227-19; or (iii) FAR 52.227-14(ALT III), as applicable. Licensor: Compuware Corporation, 31440 Northwestern Highway, Farmington Hills, Michigan 48334.

Limited Warranty and Remedy: Compuware warrants the Software media to be free of defects in workmanship for a period of ninety (90) days from purchase. During this period, Compuware will replace at no cost any such media returned to Compuware, postage prepaid. This service is Compuware's sole liability under this warranty. **COMPUWARE DISCLAIMS ALL EXPRESS AND IMPLIED WARRANTIES, INCLUDING THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. SOME STATES DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO THE ABOVE EXCLUSION MAY NOT APPLY TO LICENSEE. IN THAT EVENT, ANY IMPLIED WARRANTIES ARE LIMITED IN DURATION TO THIRTY (30) DAYS FROM THE DELIVERY OF THE SOFTWARE. LICENSEE MAY HAVE OTHER RIGHTS, WHICH VARY FROM STATE TO STATE.**

Infringement of Intellectual Property Rights: In the event of an intellectual property right claim, Compuware agrees to indemnify and hold Licensee harmless, provided Licensee gives Compuware prompt written notice of such claim, permits Compuware to defend or settle the claim, and provides all reasonable assistance to Compuware in defending or settling the claim. In the defense or settlement of such claim, Compuware may obtain for Licensee the right to continue using the Software or replace or modify the Software so that it avoids such claim, or if such remedies are not reasonably available, accept the return of the infringing Software and provide Licensee with a pro-rata refund of the license fees paid for such Software based on a three (3) year use period.

Limitation of Liability: LICENSEE ASSUMES THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE. IN NO EVENT WILL COMPUWARE BE LIABLE TO LICENSEE OR TO ANY THIRD PARTY FOR ANY SPECIAL, INDIRECT, INCIDENTAL, OR CONSEQUENTIAL DAMAGES, INCLUDING BUT NOT LIMITED TO, LOSS OF USE, DATA, REVENUES OR PROFITS, ARISING OUT OF OR IN CONNECTION WITH THIS AGREEMENT OR THE USE, OPERATION OR PERFORMANCE OF THE SOFTWARE, WHETHER SUCH LIABILITY ARISES FROM ANY CLAIM BASED UPON CONTRACT, WARRANTY, TORT (INCLUDING NEGLIGENCE), PRODUCT LIABILITY OR OTHERWISE, AND WHETHER OR NOT COMPUWARE OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH LOSS OR DAMAGE. SOME STATES DO NOT ALLOW THE LIMITATION OR EXCLUSION OF LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THE ABOVE LIMITATION OR EXCLUSION MAY NOT APPLY TO LICENSEE. IN NO EVENT SHALL COMPUWARE BE LIABLE TO LICENSEE FOR AMOUNTS IN EXCESS OF PURCHASE PRICE PAID FOR THE SOFTWARE.

Term and Termination: This License Agreement shall be effective upon Licensee's acceptance of this Agreement and shall continue until terminated by mutual consent, or by election of either Licensee or Compuware in case of the other's unremediated material breach. In case of any termination of this Agreement, Licensee will immediately return to Compuware the Software that Licensee has obtained under this Agreement and will certify in writing that all copies of the Software have been returned or erased from the memory of its computer or made non-readable.

General: This Agreement is the complete and exclusive statement of the parties' agreement. Should any provision of this Agreement be held to be invalid by any court of competent jurisdiction, that provision will be enforced to the maximum extent permissible and the remainder of the Agreement shall nonetheless remain in full force and effect. This Agreement shall be governed by the laws of the State of Michigan and the United States of America.

Contents

| | | |
|---|-----------|--|
| Purpose of This Manual | xi | |
| Audience | xi | |
| Organization of This Manual | xi | |
| Typographical Conventions | xiii | |
| How to Use This Manual | xiii | |
| Other Useful Documentation | xiii | |
| Customer Assistance | xiv | |
| For Non-technical Issues | xiv | |
| For Technical Issues | xiv | |
| Chapter 1: Welcome to SoftICE | 1 | |
| Product Overview | 3 | |
| About SoftICE | 3 | |
| About Symbol Loader | 5 | |
| Chapter 2: SoftICE Tutorial | 7 | |
| Introduction | 9 | |
| Loading SoftICE | 9 | |
| Building the GDIDEMO Sample Application | 10 | |
| Loading the GDIDEMO Sample Application | 11 | |
| Controlling the SoftICE Screen | 12 | |
| Tracing and Stepping through Source Code | 13 | |
| Viewing Local Data | 14 | |
| Setting Point-and-Shoot Breakpoints | 15 | |
| Setting a One-Shot Breakpoint | 15 | |
| Setting a Sticky Breakpoint | 15 | |
| Using SoftICE Informational Commands | 17 | |
| Using Symbols and Symbol Tables | 18 | |
| Setting a Conditional Breakpoint | 19 | |
| Setting a BPX Breakpoint | 19 | |
| Editing a Breakpoint | 20 | |
| Setting a Read-Write Memory Breakpoint | 21 | |
| Chapter 3: Loading Code into SoftICE | 25 | |
| Debugging Concepts | 27 | |
| Preparing to Debug Applications | 27 | |
| Preparing to Debug Device Drivers and VxDs | 28 | |
| Loading SoftICE Manually | 28 | |
| Loading SoftICE for Windows 95 and Windows 98 | 28 | |
| Loading SoftICE for Windows NT | 29 | |
| Building Applications with Debug Information | 30 | |
| Using Symbol Loader to Translate and Load Files | 31 | |
| Modifying Module Settings | 33 | |
| Modifying General Settings | 34 | |
| Modifying Translation Settings | 35 | |
| Modifying Debugging Settings | 36 | |
| Specifying Program Source Files | 37 | |
| Deleting Symbol Tables | 38 | |
| Using Symbol Loader From a DOS Prompt | 38 | |
| Using the Symbol Loader Command-Line Utility | 40 | |
| NMSYM Command Syntax | 40 | |
| Using NMSYM to Translate Symbol Information | 41 | |
| Using NMSYM to Load a Module and Symbol Information | 45 | |
| Using NMSYM to Load Symbol Tables or Exports | 47 | |
| Using NMSYM to Unload Symbol Information | 48 | |
| Using NMSYM to Save History Logs | 49 | |

| | | | |
|--|-----------|--|-----------|
| Getting Information about NMSYM | 50 | Using the Register Window | 74 |
| Chapter 4: Navigating Through SoftICE | 51 | Controlling the Register window | 74 |
| Introduction | 53 | Viewing Information | 74 |
| Universal Video Driver | 53 | Editing Registers and Flags | 75 |
| Setting the Video Memory Size | 54 | Associated Commands | 76 |
| Popping Up the SoftICE Screen | 54 | Using the Data Window | 76 |
| Disabling SoftICE at Startup | 54 | Controlling the Data Window | 76 |
| Using the SoftICE Screen | 55 | Viewing Information | 77 |
| Resizing the SoftICE Screen | 56 | Changing the Memory Address and Format | 78 |
| Controlling SoftICE Windows | 56 | Editing Memory | 78 |
| Copying and Pasting Data | 58 | Assigning Expressions | 78 |
| Entering Commands From the Mouse | 58 | Associated Commands | 79 |
| Obtaining Help | 59 | Using the Stack Window | 79 |
| Using the Command Window | 60 | Using the Pentium III Register Window | 80 |
| Scrolling the Command Window | 60 | Using the FPU Stack Window | 80 |
| Entering Commands | 60 | Viewing Information | 80 |
| Recalling Commands | 63 | Chapter 5: Using SoftICE | 81 |
| Using Run-time Macros | 64 | Debugging Multiple Programs at Once | 83 |
| Saving the Command Window History Buffer to a File | 65 | Trapping Faults | 83 |
| Associated Commands | 66 | Ring 3 32-bit protected mode (Win32 programs) | 83 |
| Using the Code Window | 66 | Ring 0 driver code (Kernel-mode device drivers) | 84 |
| Controlling the Code Window | 66 | Ring 3 16-bit protected mode (16-bit Windows programs) | 84 |
| Viewing Information | 68 | About Address Contexts | 85 |
| Entering Commands From the Code Window | 69 | Using INT 0x41 .DOT Commands | 86 |
| Using the Locals Window | 70 | Understanding Transitions From Ring-3 to Ring-0 | 87 |
| Controlling the Locals Window | 70 | Chapter 6: Using Breakpoints | 89 |
| Expanding and Collapsing Stacks | 71 | Introduction | 91 |
| Associated Commands | 71 | Types of Breakpoints Supported by SoftICE | 91 |
| Using the Watch Window | 72 | Breakpoint Options | 92 |
| Controlling the Watch Window | 72 | Execution Breakpoints | 92 |
| Setting an Expression to Watch | 73 | Memory Breakpoints | 93 |
| Viewing Information | 73 | Interrupt Breakpoints | 94 |
| Expanding and Collapsing Typed Expressions | 73 | | |
| Associated Commands | 74 | | |

| | | | |
|--|-----|--|--|
| I/O Breakpoints | 95 | | |
| Window Message Breakpoints | 96 | | |
| Understanding Breakpoint Contexts | 97 | | |
| Virtual Breakpoints | 97 | | |
| Setting a Breakpoint Action | 98 | | |
| Conditional Breakpoints | 98 | | |
| Conditional Breakpoint Count Functions | 100 | | |
| Using Local Variables in Conditional Expressions | 103 | | |
| Referencing the Stack in Conditional Breakpoints | 104 | | |
| Performance | 106 | | |
| Duplicate Breakpoints | 106 | | |
| Elapsed Time | 106 | | |
| Breakpoint Statistics | 107 | | |
| Referring to Breakpoints in Expressions | 107 | | |
| Manipulating Breakpoints | 108 | | |
| Using Embedded Breakpoints | 108 | | |
| Chapter 7: Using Expressions | 109 | | |
| Expressions | 111 | | |
| Operators | 111 | | |
| Operator Precedence | 113 | | |
| Forming Expressions | 114 | | |
| Expression Types | 118 | | |
| Type Casting | 121 | | |
| Evaluating Symbols | 123 | | |
| Using Indirection With Symbols | 123 | | |
| Chapter 8: Loading Symbols for System Components | 125 | | |
| Loading Export Symbols for DLLs and EXEs | 127 | | |
| Using Unnamed Entry Points | 127 | | |
| Using Export Names in Expressions | 128 | | |
| Loading 32-bit DLL Exports Dynamically | 128 | | |
| Using Windows NT Symbol (DBG) Files with SoftICE | 129 | | |
| Using Windows 95 and Windows 98 Symbol (.SYM) Files with SoftICE | 129 | | |
| Chapter 9: Using SoftICE with a Modem | 131 | | |
| Introduction | 133 | | |
| Hardware Requirements | 133 | | |
| Establishing a Connection | 133 | | |
| Using SERIAL32.EXE by Modem | 134 | | |
| DIAL Command | 135 | | |
| ANSWER Command | 135 | | |
| Chapter 10: Customizing SoftICE | 137 | | |
| Modifying SoftICE Initialization Settings | 139 | | |
| Modifying General Settings | 139 | | |
| Pre-loading Symbols and Source Code | 141 | | |
| Pre-loading Exports | 143 | | |
| Configuring Remote Debugging - Internet Control | 143 | | |
| Configuring Remote Debugging - Dial up Control | 146 | | |
| Modifying Keyboard Mappings | 147 | | |
| Working with Persistent Macros | 148 | | |
| Setting Troubleshooting Options | 151 | | |
| Chapter 11: Exploring Windows NT | 153 | | |
| Overview | 155 | | |
| Resources for Advanced Debugging | 155 | | |
| Inside the Windows NT Kernel | 158 | | |
| Managing the Intel Architecture | 159 | | |
| Windows NT System Memory Map | 163 | | |
| Win32 Subsystem | 170 | | |
| Inside CSRSS | 170 | | |
| USER and GDI Objects | 172 | | |
| Process Address Space | 177 | | |
| Heap API | 178 | | |

Chapter 12: Using BoundsChecker Driver Edition 189

- Configuring BoundsChecker 191
- Using the EVENT Command 192
 - Viewing Events in the Event Window 193
 - Viewing Events in the Command Window 194
- Reviewing Event Data 195
 - Understanding Special Display Characteristics 195
 - Finding Events 195
 - Reading Summary Information 196
 - Reading Detail Information 197
- Filtering Events 198
 - Filtering by Event Type 199
 - Filtering by Parameters 199

Appendix A: Error Messages 201

Appendix B: Supported Display Adapters 205

Appendix C: Troubleshooting SoftICE 209

Appendix D: Kernel Debugger Extensions 211

Appendix E: Events Monitored by BoundsChecker 213

- Windows NT Kernel-mode API Calls 213
- Standard Driver Routines and Callbacks 214
- System-wide Events 215
- Errors 217

Glossary 219

Index 221

Some books are to be tasted, others to be swallowed, and some few to be chewed and digested.

◇ Francis Bacon

Preface

| | |
|------------------------------------|-------------|
| Purpose of This Manual | xi |
| Audience | xi |
| Organization of This Manual | xi |
| Typographical Conventions | xiii |
| How to Use This Manual | xiii |
| Other Useful Documentation | xiii |
| Customer Assistance | xiv |

Purpose of This Manual

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. This manual describes how to install and use SoftICE to load and debug code for Windows 95, Windows 98, and Windows NT. Since many programmers prefer to learn through hands on experience, this manual includes a tutorial that leads you through the basics of debugging code.

Audience

This manual is for programmers who want to use SoftICE to debug code for Windows 95, Windows 98, and Windows NT.

Organization of This Manual

The *Using SoftICE* manual is organized as follows:

- Chapter 1, “Welcome to SoftICE”
Briefly describes SoftICE and its components and features. Chapter 1 also explains how to contact the NuMega Technical Support Center.
- Chapter 2, “Installing SoftICE”
Lists the hardware and software requirements for SoftICE, then explains how to install SoftICE for Windows 95, Windows 98, and Windows NT.
- 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 SoftICE provides for debugging code.
- 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 Windows NT symbol files with SoftICE.
 - Chapter 10, “Using SoftICE with a Modem”
Explains how to establish a modem 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 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.
 - Glossary
 - Index

Typographical Conventions

The following conventions are used consistently throughout this manual to identify certain types of information:

| Convention | Description |
|-----------------|--|
| Enter | Indicates that you should type text, then press RETURN or click OK. |
| <i>italics</i> | Indicates variable information. For example: <i>library-name</i> . |
| 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. |

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 earlier versions of SoftICE. | Read Chapter 4, "Loading Code into SoftICE." Then skim Chapter 5, "Navigating Through SoftICE" to learn about using the mouse and the Locals and Watch windows. |

Other Useful Documentation

In addition to this manual, NuMega 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 SI30UG.PDF (*Using SoftICE*) or SI30CR.PDF (*SoftICE Command Reference*) files.

Customer Assistance

For Non-technical Issues

NuMega 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: 888-283-9896
- International: +1 603 578 8103

For Technical Issues

NuMega Technical Support can assist you with all your technical problems, from installation to troubleshooting.

Before contacting technical support please read the relevant sections of the product documentation and the ReadMe files.

You can contact Technical Support by:

| | |
|----------------|--|
| E-Mail | Include your serial number and send as many details as possible to Tech@numega.com |
| World Wide Web | Submit issues and access our support knowledge base at www.numega.com. Go to Support. |
| 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: 888 NUMEGA-S International customers, call: +1 603 578 8100 |
| | * Technical Support handles installation and setup issues free of charge. |
| Fax | Include your serial number and send as many details as possible to 603 578 8401 |

Before contacting Technical Support, please obtain and record the following information:

- Product/service pack name and version
- Product serial number
- System configuration: operating system, network configuration, amount of RAM, environment variables, and paths
- Name and version of your compiler and linker and the options you used in compiling and linking
- Problem details; settings, error messages, stack dumps, and the contents of any diagnostic windows
- If the problem is repeatable, the details of how to create the problem

*To a great experience one thing is essential,
an experiencing nature.*

◇ Walter Bagehot

1 Welcome to SoftICE

| | |
|----------------------------|----------|
| Product Overview | 3 |
| About SoftICE | 3 |
| About Symbol Loader | 5 |

Product Overview

SoftICE is available for Windows 95, Windows 98, and Windows NT. 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 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.

About 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, Windows NT device drivers (both kernel and user mode), Windows 95 and Windows 98 drivers, VxDs, 16-bit windows programs, and DOS programs.
- Debugging virtually any code, including interrupt routines and the Windows 95, Windows 98, and Windows NT kernels.
- Setting real-time breakpoints on memory reads/writes, port reads/writes, and interrupts.
- Setting breakpoints on Windows messages.
- Setting conditional breakpoints and breakpoint actions.
- Displaying elapsed time to the breakpoint trigger using the Pentium clock counter.
- Kernel-level debugging on one machine.
- Displaying internal Windows 95, Windows 98, and Windows NT 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 to a remote user. This enables you to diagnose a remote user's problem, such as a system crash.
- Supporting the MMX instruction set extensions.
- Creating user-defined macros.

How the SoftICE Debugger is Implemented

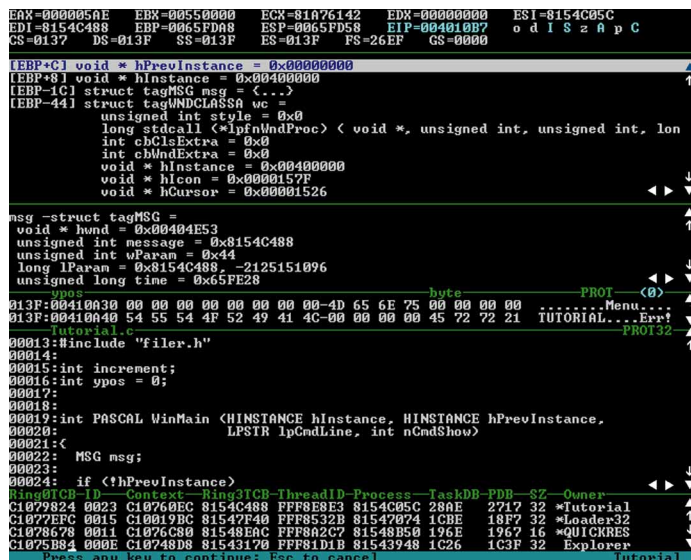
SoftICE for Windows 95, Windows 98, and Windows NT are implemented in slightly different ways. SoftICE for Windows 95 and Windows 98 comprises two VxDs, and SoftICE for Windows NT comprises two NT kernel device drivers, as follows:

| Windows 95, Windows 98 (VxD) | Windows NT (NT Kernel Device Driver) | Description |
|---|---|-------------------------------------|
| WINICE.EXE | NTICE.SYS | Provides the debugger. |
| SIWVID.386 | SIWVID.SYS | Provides video support for your PC. |

Note: SoftICE for Windows NT must be loaded by the Windows NT operating system because it is implemented as a device driver. Therefore, you cannot debug the following for Windows NT: any boot driver's DriverEntry routine, the HAL and NTOSKRNL initialization code, and any Windows NT loader or NTDETECT code.

The 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. Hence, SoftICE uses a full-screen character-oriented display, as follows:



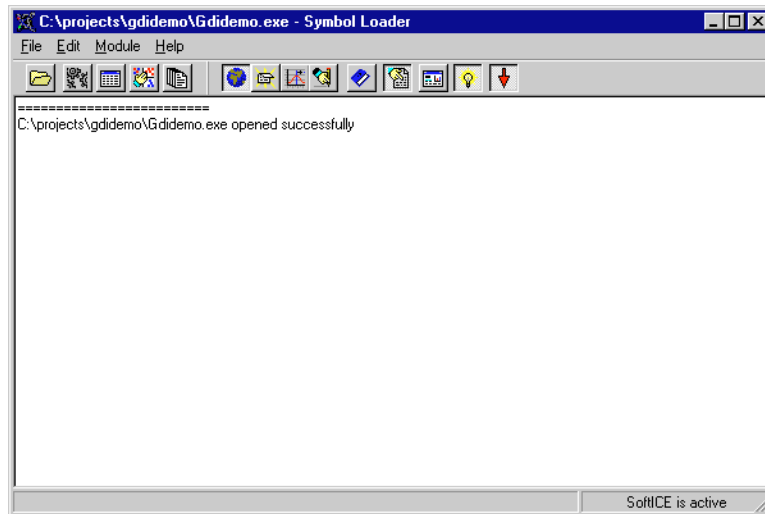
Refer to *Chapter 4: Navigating Through SoftICE* on page 51 for more information about using the SoftICE screen.

About Symbol Loader

Symbol Loader is a graphical utility that extracts debug symbol information from your device drivers, EXEs, DLLs, ODXs, 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.
- Automatically start your application and set a breakpoint at its entry point.
- Save your debugging session to a file.

The following figure illustrates 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.

*We will now discuss in a little more detail
the struggle for existence.*

◇ Charles Darwin

2 **SoftICE Tutorial**

| | |
|---|-----------|
| Introduction | 9 |
| Loading SoftICE | 9 |
| Building the GDIDEMO Sample Application | 10 |
| Loading the GDIDEMO Sample Application | 11 |
| Controlling the SoftICE Screen | 12 |
| Tracing and Stepping through Source Code | 13 |
| Viewing Local Data | 14 |
| Setting Point-and-Shoot Breakpoints | 15 |
| Setting a One-Shot Breakpoint | 15 |
| Setting a Sticky Breakpoint | 15 |
| Using SoftICE Informational Commands | 17 |
| Using Symbols and Symbol Tables | 18 |
| Setting a Conditional Breakpoint | 19 |
| Setting a BPX Breakpoint | 19 |
| Editing a Breakpoint | 20 |
| Setting a Read-Write Memory Breakpoint | 21 |

Introduction

This tutorial gives you hands-on experience debugging a Windows application, teaching you the fundamental steps for debugging applications and drivers. During this debugging session, you will learn how to do the following:

- Load SoftICE
- Build an application
- Load the application source and symbol files
- Trace and step through source code and assembly language
- View local data and structures
- Set point-and-shoot breakpoints
- Use SoftICE informational commands to explore the state of the application
- 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 GDIDEMO application as its basis. GDIDEMO provides a demonstration of GDI functionality. GDIDEMO is located in the \EXAMPLES\GDIDEMO directory on your CD-ROM. If you use the GDIDEMO on the CDROM, copy it to your hard drive.

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.

Note: The examples in this tutorial are based on Windows NT. If you are using Windows 95 or Windows 98, your output may vary slightly.

Loading SoftICE

If you are running SoftICE with Windows 95, Windows 98, or under Windows NT in Boot, System, or Automatic mode, SoftICE automatically loads when you start or reboot your PC. If you are running SoftICE in Manual Startup mode with Windows NT, SoftICE does not load automatically.

To load SoftICE for Windows 95 or Windows 98, enter the command WINICE.

To load SoftICE for Windows NT, do one of the following:

- Select START SOFTICE.



- Enter the command: **NET START NTICE**

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

To verify that SoftICE is loaded, press the SoftICE hot key sequence Ctrl-D. The SoftICE screen should appear. To return to the Windows operating system, use the X (exit) or G (go to) command (F5).

Building the GDIDEMO Sample Application

The first step in preparing to debug a Windows application is to build it with debug information. The makefile for the sample application GDIDEMO is already set up for this purpose.

To build the sample program, perform the following steps:

- 1 Open a DOS shell.
- 2 Change to the directory that contains the sample code.
- 3 Execute the NMAKE command:

```
C:\MSTOOLS\SAMPLES\WIN32\GDIDEMO>NMAKE
```

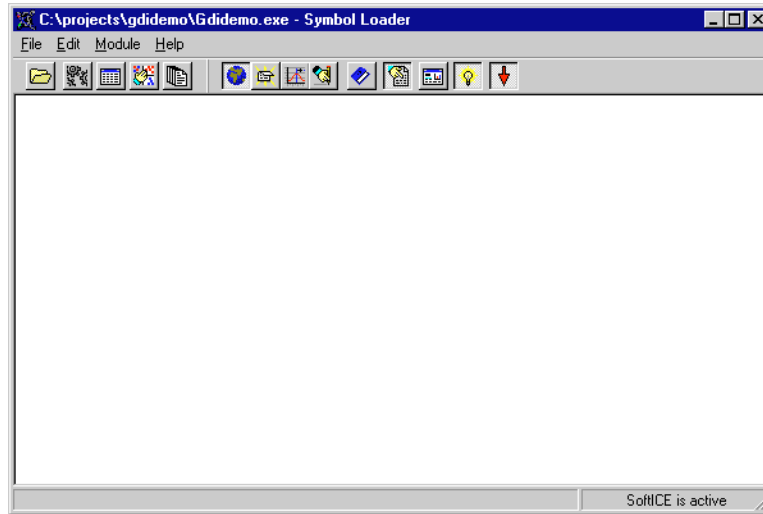
If GDIDEMO is located in another directory, change the path as appropriate.

Loading the GDIDEMO Sample Application

Loading an application entails creating a symbol file from the application's debug information and loading the symbol and source files into SoftICE. To Load the GDIDEMO application, perform the following steps:

- 1 Start Symbol Loader.

The Symbol Loader window appears.



- 2 Either choose OPEN MODULE from the File menu or click the OPEN button.

The Open window appears.

- 3 Locate GDIDEMO.EXE and click Open.

- 4 Either choose LOAD from the Module menu or click the LOAD button to load GDIDEMO.

Symbol Loader translates the debug information into a .NMS symbol file, loads the symbol and source files, starts GDIDEMO, pops up the SoftICE screen, and displays the source code for the file GDIDEMO.C.

Controlling the SoftICE Screen

The SoftICE screen is your central location for viewing and debugging code. It provides up to seven windows and one help line to let you view and control various aspects of your debugging session. By default, it displays the following:

- Locals window—Displays and expands variables allocated on the stack
- Code window—Displays source code or unassembled instructions
- Command window—Enters user commands and display information
- Help line—Provides information about SoftICE commands and shows the active address context

The screenshot shows the SoftICE interface with the following windows and content:

- Register window:** EAX=000005AE EBX=00550000 ECX=81A76142 EDX=00000000 ESI=8154C05C EDI=8154C488 EBP=0065FD48 ESP=0065FD58 EIP=004010B7 CS=0137 DS=013F SS=013F ES=013F FS=26EF GS=0000
- Locals window:**

```

[EBP+C] void * hPrevInstance = 0x00000000
[EBP+8] void * hInstance = 0x00400000
[EBP+1C] struct tagMSG msg = <...>
[EBP-44] struct tagWNDCLASS wc =
    unsigned int style = 0x0
    long stdcall (&lpfnWndProc) < void *, unsigned int, unsigned int, lon
    int cbClsExtra = 0x0
    int cbWndExtra = 0x0
    void * hInstance = 0x00400000
    void * hIcon = 0x0000157F
    void * hCursor = 0x00001526

```
- Watch window:**

```

msg - struct tagMSG =
void * hwnd = 0x00404E53
unsigned int message = 0x8154C488
unsigned int wParam = 0x44
long lParam = 0x8154C488, -2125151096
unsigned long time = 0x65FE28

```
- Data window:**

```

013F:00410A30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
013F:00410A40 54 55 54 4F 52 49 41 4C 00 00 00 00 45 72 21 21
Tutorial.c
00013:#include "filer.h"
00014:
00015:int increment;
00016:int ypos = 0;
00017:
00018:
00019:int PASCAL WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance,
00020: LPSTR lpCmdLine, int nCmdShow)
00021:C
00022: MSG msg;
00023:
00024: if (!hPrevInstance)

```
- Code window:**

```

Ring3TCB-ID-Context-Ring3TCB-ThreadID-Process-TaskDB-PDB-SZ-Owner
C1079824 0023 C10760EC 8154C488 FFF8B8E3 8154C05C 288E 2717 32 *Tutorial
C1077E9C 0015 C10019DC 81547740 FFF8522B 81547874 1CDE 18F7 32 *Loader32
C1078678 0011 C1076C80 81548E0C FFF8A2C7 81548B50 196E 1967 16 *QUICKRES
C1075B84 000E C10748D8 81543170 FFF81D1B 81543948 1C26 1C3F 32 Explorer

```
- Command window:** Press any key to continue; Esc to cancel
- Help line:** Tutorial

- 1 Look at the contents of the Code window. Note that SoftICE is displaying the WinMain routine at line 34. By default, SoftICE creates a breakpoint and stops at the first main module it encounters when loading your application.
- 2 To see all the source files that SoftICE loaded, enter the FILE command with the wild card character:

```
.FILE *
```

SoftICE displays the source files for GDIDEMO: draw.c, maze.c, xform.c, poly.c, wininfo.c, dialog.c, init.c, bounce.c, and gidemo.c. 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 4: Navigating Through SoftICE* on page 51 for information about resizing windows.)

- 3 Many SoftICE windows can be scrolled. If you have a mouse, you can click on the scroll arrows. If not, SoftICE provides key sequences that let you scroll specific windows. Try these methods for scrolling the Code window:

| Scroll the 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 |

- 4 Enter the U command followed by EIP to disassemble the instructions for the current instruction pointer.

```
:U EIP
```

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

```
::
```

Tracing and Stepping through 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
```

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

```
if(!hPrevInst)
```

- 2 The Code window is currently displaying source code. However, it can also display disassembled code or mixed (both source and disassembled) code. To view mixed code, use the SRC command (F3).

```
:SRC
```

Note that 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.

Execution proceeds until it reaches the line that executes the RegisterAppClass function.

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

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

Hint: 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.

Viewing Local Data

The Locals window displays the current stack frame. In this case, it contains the local data for the WinMain function.

The following steps illustrate how to use the Locals window:

- 1 Enter the T command to enter the RegisterAppClass function. The Locals window is now empty because local data is not yet allocated for the function.

The RegisterAppClass function is implemented in the source file INIT.C. SoftICE displays the current source file in the upper left corner of the Code window.

- 2 Enter the T command again.

The Locals window contains the parameter passed to the RegisterAppClass (hInstance) and a local structure wndClass. The structure tag wndClass is marked with a plus sign (+). This plus sign indicates that you can expand the structure to view its contents.

Note: You can also expand character strings and arrays.

- 3 If you have a Pentium-class processor and a mouse, double-click the structure WNDCLASSA to expand it. To collapse the structure wndClass, double-click its contents.
- 4 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. Double-click the minus sign (-) to collapse it.

Setting Point-and-Shoot Breakpoints

This section shows you how to set two handy 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 To shift focus to the Code window, either use your mouse to click in the window or press Alt-C.

If you wanted to shift focus back to the Command window you could press Alt-C again.

- 2 Either use the Down arrow key, the down scroll arrow, or the U command to place the cursor on line 61, the first call to the Win32 API function RegisterClass. If you use the U command, specify the source line 61 as follows:

```
:U .61
```

SoftICE places source line 61 at the top of the Code window.

- 3 Use the HERE command (F7) to execute to line 61.

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(!RegisterClass(&wndClass))
```

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 .61
```

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.

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

- 1 Find the next call to RegisterClass that appears on source line 74. With the cursor on line 74, enter the BPX command (F9) to set an execution breakpoint. The BPX command sets an execution breakpoint by inserting an INT3 instruction into the code. Note that the line is highlighted when you set a breakpoint.

- 2 Press the F9 key to clear the breakpoint.

If you are using a Pentium-class processor and you have a mouse, you can double-click on a line in the Code window to set or clear a breakpoint.

- 3 Set a breakpoint on line 74, then use the G or X command (F5) to execute the instructions until the breakpoint triggers:

```
:G
```

When the INT3 instruction is executed, 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.

- 4 To view information about breakpoints that are currently set, use the BL command:

```
:BL  
00) BPX #0137:00402442
```

Note: The address you see might be different.

From the output of the BL command, one breakpoint is set on code address 0x402442. This address equates to source line 74 in the current file INIT.C.

- 5 You can use the SoftICE expression evaluator to translate a line number into an address. To find the address for line 74, use the ? command:

```
?: .74  
void * = 0x00402442
```

- 6 The RegisterAppClass 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
```

The RET parameter to the P command causes SoftICE to execute instructions until the function call returns. Because RegisterAppClass was called from within WinMain, SoftICE pops up in WinMain on the statement after the RegisterAppClass function call. The following source line in WinMain should be highlighted:

```
msg.wParam = 1;
```

- 7 Enter the BC command with the wild card parameter to clear all the breakpoints:

```
BC *
```


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.

- 1 The H and Class commands work best when you have more room to display information, so use the WL command to close the Locals window. Closing this window automatically increases the size of the Command window.
- 2 The H command provides general help on all the SoftICE commands or detailed help on a specific command. 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 detailed use, including any options and/or parameters the command accepts. The third line is an example of the command.

- 3 The purpose of the RegisterAppClass function is to register window class templates that are used by the GDIDEMO application to create windows. Use the CLASS command to examine the classes registered by GDIDEMO.

```
:CLASS GDIDEMO
```

| 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

Now that you are familiar with using SoftICE to step, trace, and create point-and-shoot style breakpoints, it is time to explore symbols and tables. When you load symbols for an application, SoftICE creates a symbol table that contains all the symbols defined for that module.

- 1 Use the TABLE command to see all the symbol tables that are loaded:

```
:TABLE
GDIDEMO [NM32]
964657 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:

```
:TABLE GDIDEMO
```

- 2 Use the SYM command to display the symbols from the current symbol table. With the current table set to GDIDEMO, the SYM command produces output similar to the following abbreviated output:

```
:SYM
.text(001B)
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.

Setting a Conditional Breakpoint

One of the symbols defined for the GDIDEMO application is the LockWindowInfo function. The purpose of this routine is to retrieve a pointer value that is specific to a particular instance of a window.

To learn about conditional and memory breakpoints, you will perform the following steps:

- Set a BPX breakpoint on the LockWindowInfo function.
- Edit the breakpoint to use a conditional expression, thus setting a conditional breakpoint.
- Set a memory breakpoint to monitor access to a key piece of information, as described in *Setting a Read-Write Memory Breakpoint* on page 21.

Setting a BPX Breakpoint

Before setting the conditional breakpoint, you need to set a BPX-style breakpoint on LockWindowInfo.

- 1 Set a BPX-style breakpoint on the LockWindowInfo function:

```
:BPX LockWindowInfo
```

When one of the GDIDEMO windows needs to draw information in its client area, it calls the LockWindowInfo function. Every time the LockWindowInfo function is called, SoftICE pops up to let you debug the function. The GDIDEMO windows continually updates, so this breakpoint goes off quite frequently.

- 2 Use the BL command to verify that the breakpoint is set.
- 3 Use either the X or G command to exit SoftICE.

SoftICE should pop up almost immediately on the LockWindowInfo function.

Editing a Breakpoint

From the LockWindowInfo function prototype on source line 47, you can see that the function accepts one parameter of type HWND and returns a void pointer type. The HWND parameter is the handle to the window that is attempting to draw information within its client area. At this point, you want to modify the existing breakpoint, adding a conditional breakpoint to isolate a specific HWND value.

- 1 Before you can set the conditional expression, you need to obtain the HWND value for the POLYDEMO window. The HWND command provides information about application windows. Use the HWND command and specify the GDIDEMO process:

```
:HWND GDIDEMO
```

The following example illustrates what you should see if you are using Windows NT. If you are using Windows 95 or Windows 98, your output will vary.

| Handle | Class | WinProc | TID | Module |
|----------------------|------------|----------|-----|---------|
| 07019C | GDIDEMO | 004010B5 | 2D | GDIDEMO |
| 100160 | MDIClient | 77E7F2F5 | 2D | GDIDEMO |
| 09017E | BOUNCEDEMO | 004015A4 | 2D | GDIDEMO |
| <u>100172</u> | POLYDEMO | 00402F34 | 2D | GDIDEMO |
| 11015C | DRAWDEMO | 00403CE4 | 2D | GDIDEMO |

The POLYDEMO window handle is bold and underlined. This is the window handle you want to use to form a conditional expression. If the POLYDEMO window does not appear in the HWND output, exit SoftICE using the G or X commands (F5) and repeat Step 1 until the window is created.

The value used in this example is probably not the same value that appears in your output. For the exercise to work correctly, you must use the HWND command to obtain the actual HWND value on your system.

Using the POLYDEMO window handle, you can set a conditional expression to monitor calls to LockWindowInfo looking for a matching handle value. When the LockWindowInfo function is called with the POLYDEMO window handle, SoftICE pops up.

- 2 Because you already have a breakpoint set on `LockWindowInfo`, use the `BPE` command (Breakpoint Edit) to modify the existing breakpoint:

```
:BPE 0
```

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 LockWindowInfo
```

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

- 3 Remember to substitute the `POLYDEMO` window handle value that you found using the `HWND` command instead of the value (100172) used in this example. Your conditional expression should appear similar to the following example. The conditional expression appears in bold type.

```
:BPX LockWindowInfo IF ESP->4 == 100172
```

Note: Win32 applications pass parameters on the stack and at the entry point of a function; the first parameter has a positive offset of 4 from the ESP register. Using the SoftICE expression evaluator, this is expressed in the following form: `ESP->4`. ESP is the CPU stack pointer register and the “->” operator causes the lefthand side of the expression (ESP) to be indirected at the offset specified on the righthand side of the expression (4). For more information on the SoftICE expression evaluator refer to *Chapter 7: Using Expressions* on page 109 and for referencing the stack in conditional expressions refer to *Conditional Breakpoints* on page 98.

- 4 Verify that the breakpoint and conditional expression are correctly set by using the `BL` command.
- 5 Exit SoftICE using the `G` or `X` command (F5).

When SoftICE pops up, the conditional expression will be `TRUE`.

Setting a Read-Write Memory Breakpoint

We set the original breakpoint and subsequently the conditional expression so that we could obtain the address of a data structure specific to this instance of the `POLYDEMO` window. This value is stored in the window’s extra data and is a global handle. The `LockWindowInfo` function retrieves this global handle and uses the Win32 API `LocalLock` to translate it into a pointer that can be used to access the window’s instance data.

- 1 Obtain the pointer value for the windows instance data by executing up to the return statement on source line 57:

```
:G .57
```

- 2 Win32 API functions return 32-bit values in the EAX register, so you can use the BPMD command and specify the EAX register to set a memory breakpoint on the instance data pointer.

```
:BPMD EAX
```

The BPMD command uses the hardware debug registers provided by Intel CPUs to monitor reads and writes to the Dword value at a linear address. In this case, you are using BPMD to trap read and write accesses to the first Dword of the window instance data.

- 3 Use the BL command to verify that the memory breakpoint is set. Your output should look similar to the following:

```
:BL
00) BPX LockWindowInfo IF ((ESP->4)==0x100172)
01) BPMD #0023:001421F8 RW DR3
```

Breakpoint index 0 is the execution breakpoint on LockWindowInfo and breakpoint index 1 is the BPMD on the window instance data.

- 4 Use the BD command to disable the breakpoint on the LockWindowInfo.

```
:BD 0
```

SoftICE provides the BC (breakpoint clear) and BD (breakpoint disable) commands to clear or disable a breakpoint. Disabling a breakpoint is useful if you want to re-enable the breakpoint later in your debugging session. If you are not interested in using the breakpoint again, then it makes more sense to clear it.

- 5 Use the BL command to verify that the breakpoint on LockWindowInfo is disabled. SoftICE indicates that a breakpoint is disabled by placing an asterisk (*) after the breakpoint index. Your output should appear similar to the following:

```
:BL
00) * BPX _LockWindowInfo IF ((ESP->4)==0x100172)
01) BPMD #0023:001421F8 RW DR3
```

Note: You can use the BE command to re-enable a breakpoint:

```
:BE breakpoint-index-number
```

6 Exit SoftICE using the G or X command.

When the POLYDEMO window accesses the first Dword of its window instance data, the breakpoint triggers and SoftICE pops up.

When SoftICE pops up due to the memory breakpoint, you are in the PolyRedraw or PolyDrawBez function. Both functions access the nBezTotal field at offset 0 of the POLYDRAW window instance data.

Note: The Intel CPU architecture defines memory breakpoints as traps, which means that the breakpoint triggers after the memory has been accessed. In SoftICE, the instruction or source line that is highlighted is the one after the instruction or source line that accessed the memory.

7 Clear the breakpoints you set in this section by using the BC command:

```
:BC *
```

Note: You can use the wildcard character (*) with the BC, BD, and BE commands to clear, disable, and enable all breakpoints.

8 Exit SoftICE using the G or X command.

The operating system terminates the demo.

Congratulations on completing your first SoftICE debugging session. 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 the many SoftICE commands that help you debug smarter and faster. Refer to the *SoftICE Command Reference* for a complete explanation of all the SoftICE commands.

3 Loading Code into SoftICE

| | |
|--|-----------|
| Debugging Concepts | 27 |
| Preparing to Debug Applications | 27 |
| Preparing to Debug Device Drivers and VxDs | 28 |
| Loading SoftICE Manually | 28 |
| Loading SoftICE for Windows 95 and Windows 98 | 28 |
| Loading SoftICE for Windows NT | 29 |
| Building Applications with Debug Information | 30 |
| Using Symbol Loader to Translate and Load Files | 31 |
| Modifying Module Settings | 33 |
| Modifying General Settings | 34 |
| Modifying Translation Settings | 35 |
| Modifying Debugging Settings | 36 |
| Specifying Program Source Files | 37 |
| Deleting Symbol Tables | 38 |
| Using Symbol Loader From a DOS Prompt | 38 |

| | |
|--|-----------|
| Using the Symbol Loader Command-Line Utility | 40 |
| NMSYM Command Syntax | 40 |
| Using NMSYM to Translate Symbol Information | 41 |
| Using NMSYM to Load a Module and Symbol Information | 45 |
| Using NMSYM to Load Symbol Tables or Exports | 47 |
| Using NMSYM to Unload Symbol Information | 48 |
| Using NMSYM to Save History Logs | 49 |
| Getting Information about NMSYM | 50 |

Debugging Concepts

SoftICE allows you to debug Windows applications and device drivers at the source level. To accomplish this, SoftICE uses a utility, called Symbol Loader, to translate the debug information from your compiled module into an .NMS symbol file. Then Symbol Loader can load the .NMS file and, optionally, source 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 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 OPEN MODULE from the File Menu and open the module 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 33.
- 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 141.
- 8 Reboot your PC.

Loading SoftICE Manually

SoftICE does not load automatically under the following configurations:

- If you did not run WINICE.EXE from the AUTOEXEC.BAT before starting Windows 95 and Windows 98.
- When you set SoftICE for Windows NT to Manual Startup mode.

If you are using these configurations, you need to load SoftICE manually. The following sections describe how to load SoftICE manually for Windows 95, Windows 98 and Windows NT.

Loading SoftICE for Windows 95 and Windows 98

Load SoftICE for Windows 95 and Windows 98 from the DOS command line. SoftICE will automatically run Windows 95 and Windows 98 after SoftICE initializes. Use the following command syntax.

Command Syntax

```
WINICE [/HST n] [/TRA n] [/SYM n] [/M]
        [/LOAD[x] name]
        [/EXP name][drive:\path\WIN.COM
        [Windows-command-line]]
```

Where the following are optional switches.

| Optional Switch | Definition |
|-----------------|--|
| /EXP name | Adds exports from the DLL or Windows application specified by name to the SoftICE export list. This lets you symbolically access these exported symbols. |
| /HST n | Increases the size of the command recall buffer, where n is a decimal number that represents the number of kilobytes. The default is 8KB. |
| /LOAD name[x] | Loads symbol and source, where name is the complete path and file name for a VxD, DOS T8SR, DOS loadable device driver, DOS program, Windows driver, Windows DLL, or Windows program that was built with symbols. If x is present, source is not loaded. |
| /M | Directs SoftICE output to the secondary monochrome monitor, bypassing any initial VGA programming. You can also use this optional switch for serial debugging by specifying /M on the command line and including a serial command in the Initialization string. |
| /SYM n | Allocates a symbol table, where n is a decimal number that represents the number of kilobytes. The default is 0KB. |
| /TRA n | Increases the size of the back trace history buffer, where n is a decimal number that represents the number of kilobytes. The default is 8KB. |

Hint: You can specify these switches in the Initialization string. Refer to *Modifying SoftICE Initialization Settings* on page 139.

Loading SoftICE for Windows NT



To load SoftICE for Windows NT, do one of the following:

- Select **START SOFTICE**.
- 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.

| Compiler | Generating Debugging Information |
|-------------------------|--|
| Borland C++ 4.5 and 5.0 | To generate Borland's standard debug information: <ul style="list-style-type: none">• Compile with /v• Link with /v |
| Delphi 2.0 | To generate Delphi's standard debug information: Compile with the following: <ul style="list-style-type: none">• -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 |

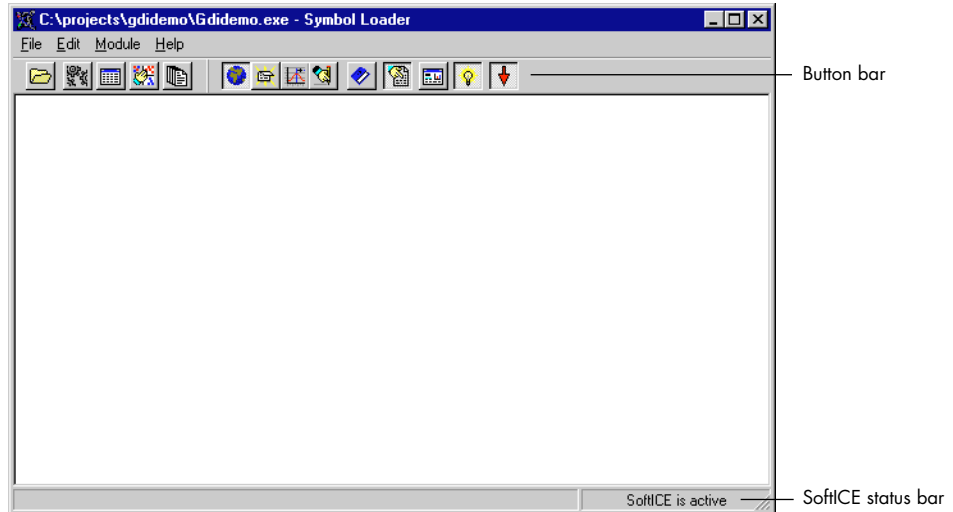
| | |
|--|---|
| MASM 6.11 | <p>To generate Codeview debug information:</p> <ul style="list-style-type: none"> • Assemble with <code>/Zi /COFF</code> • Use Microsoft's 32-bit LINK.EXE to link with <code>/DEBUG /DEBUGTYPE:CV /PDB:NONE</code> |
| Microsoft Visual C++ 2.x, 4.0, 4.1, 4.2, and 5.0 | <p>To generate Program Database (PDB) debug information:</p> <ul style="list-style-type: none"> • Compile with Program Database debug information, using the command-line option <code>/Zi</code> • Use Microsoft's linker to link with <code>/DEBUG /DEBUGTYPE:CV</code> <p><i>Note:</i> VxDs require you to generate PDB debug information.</p> <p>To generate Codeview debug information:</p> <ul style="list-style-type: none"> • Compile with C7-compatible debug information, using the command-line option <code>/Z7</code> • Use Microsoft's linker to link with <code>/DEBUG /DEBUGTYPE:CV /PDB:NONE</code> <p><i>Note:</i> If you are using the standard Windows NT DDK make procedure, use the following environment variables: <code>NTDEBUG=ntsd</code> and <code>NTDEBUGTYPE=windbg</code>.</p> |
| Symantec C++ 7.2 | <p>The Symantec compiler produces Codeview debug information by default. All that is required is to enable the output of debug info:</p> <ul style="list-style-type: none"> • Compile with <code>-g</code> Other qualifiers are available, for example, <code>-gh</code>, <code>-gE</code>, and <code>-gg</code> • Link with <code>/CO</code> |
| Watcom C++ 10.5 | <p>To generate Codeview full debug information:</p> <ul style="list-style-type: none"> • Compile with <code>-hc -d3</code> • Link with <code>DEBUG CODEVIEW OPTION CVPACK</code> |

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 Program Source Files* on page 37.

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

1 Start Symbol Loader.



2 Choose OPEN MODULE from the File menu, or click the OPEN button.

3 Select the file you want to load from the Open dialog box and click OPEN.

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.

Due to a file format restriction in .SYM files, SoftICE cannot determine whether .SYM files are 16-bit or 32-bit.

5 Choose LOAD from the Module menu, or click the LOAD button to load the open file.

Symbol Loader translates your application's debug information to an .NMS symbol file. Then Symbol Loader loads the symbol and source files into SoftICE. 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* on page 33 for more information about modifying Translation and Debugging settings.

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 source file search paths.
- 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.

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.

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

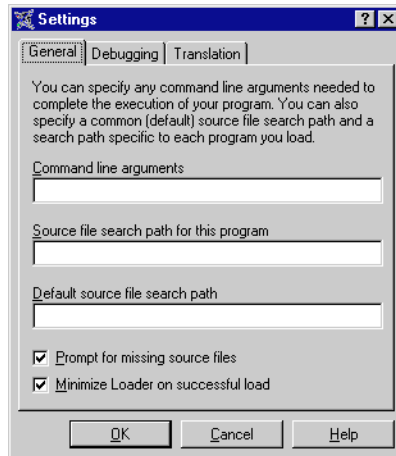
- 1 Open the file if it is not already open.

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

- 2 From the Module menu, select SETTINGS.
- 3 Click the tab that represents the settings you want to modify.
See the sections that follow for more information about specific settings for each tab.
- 4 When you are done modifying the settings, click OK.
- 5 Load the file to apply your changes.

Modifying General Settings

The General settings allow you to set command-line arguments and specify source file search paths.



The following sections describe the General settings.

Command line arguments

Use `COMMAND LINE ARGUMENTS` to specify command-line arguments to pass to your program.

Source file search path for this program

Use `SOURCE FILE SEARCH PATH FOR THIS PROGRAM` 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 `DEFAULT SOURCE FILE SEARCH PATH` to expand its search.

Default source file search path

Use `DEFAULT SOURCE FILE SEARCH PATH` to determine the search path SoftICE uses to locate files in general. This setting is a global setting.

Note that if you use the `SOURCE FILE SEARCH PATH FOR THIS PROGRAM` setting to specify the search path for a specific program, Symbol Loader uses the search path you specified for the application before looking in this global search path.

Prompt for missing source files

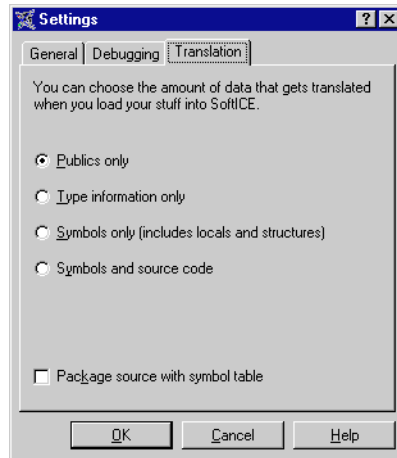
Use `PROMPT FOR MISSING SOURCE FILES` to determine if Symbol Loader prompts you when it cannot find a source file. This setting is global and is turned on by default.

Minimize loader on successful load

Use MINIMIZE LOADER ON SUCCESSFUL LOAD to minimize Symbol Loader automatically after you load an .EXE file. This setting is a global setting and it is turned on by default.

Modifying Translation Settings

Translation settings 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 sections describe the Translation settings.

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 141.

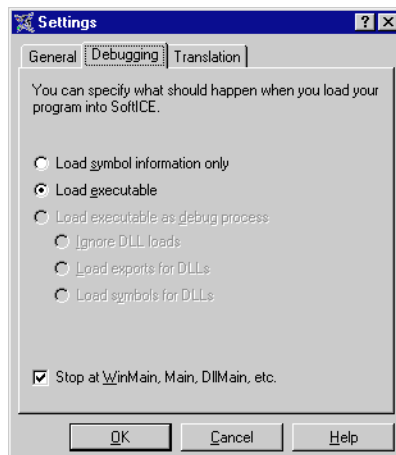
- Debugging on a system that does not have access to your source files

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.

Warning: 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 settings determine what type of information to load and whether or not to stop at the module entry point.



The following sections describe the Debugging settings.

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.

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 selects this setting for .EXE files.

Stop at WinMain, Main, DIIMain, etc.

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

Specifying Program Source 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.

To avoid loading unnecessary source files, SoftICE 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.

Example: 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
```

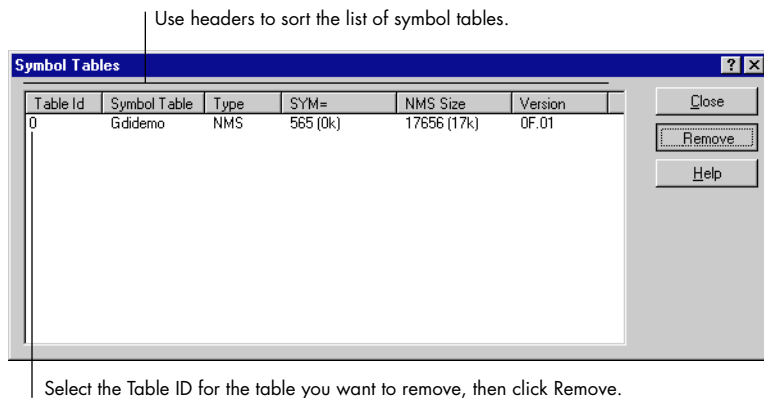
Assuming that FILE2.C was a valid program source file, it would not be loaded because it does not appear in the .SRC file. FILE1.C, FILE3.CPP, and FILE4.C would be loaded.

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 PC.

To delete a symbol table, do the following:

- 1 Choose Symbol Tables from the Edit menu.



- 2 Click the Table ID for the table you want to delete and click Remove.

Hint: Use the headers to sort the symbol tables by Table ID, Symbol Table, Type, SYM=, NMS Size, or Version. The list of symbol tables is sorted in ascending order.

Using Symbol Loader From a 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 33.

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 follows.

| 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:

| 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 sections describe the translation options. Use these options to translate symbol information for an individual module.

/TRANSLATE:<translation-specifier-list>

The /TRANSLATE 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.

| 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: 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:

| 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:<path-list>

Use the /SOURCE 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:<filename>

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 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.exe
```

/PROMPT

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.

An example of using NMSYM to translate, load, and execute a module follows:

```
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:<load-specifier-list>

The /LOAD 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.

| 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.

| 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:<program-arguments>

The /ARGS 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 program-arguments is a string that 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:<module-list>

The /SYMLOAD 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 module, 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:<module-list>

The /EXPORTS 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 an 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:<module-list>

The module-list specifier 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:<filename>[,logfile-specifier-list]

The filename specifier 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
```

Warning: 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

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

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

4 Navigating Through SoftICE

| | |
|---|-----------|
| Introduction | 53 |
| Universal Video Driver | 53 |
| Setting the Video Memory Size | 54 |
| Popping Up the SoftICE Screen | 54 |
| Disabling SoftICE at Startup | 54 |
| Using the SoftICE Screen | 55 |
| Resizing the SoftICE Screen | 56 |
| Controlling SoftICE Windows | 56 |
| Copying and Pasting Data | 58 |
| Entering Commands From the Mouse | 58 |
| Obtaining Help | 59 |
| Using the Command Window | 60 |
| Scrolling the Command Window | 60 |
| Entering Commands | 60 |
| Recalling Commands | 63 |
| Using Run-time Macros | 64 |
| Saving the Command Window History Buffer to a File | 65 |
| Associated Commands | 66 |

- Using the Code Window 66**
 - Controlling the Code Window 66**
 - Viewing Information 68**
 - Entering Commands From the Code Window 69**

- Using the Locals Window 70**
 - Controlling the Locals Window 70**
 - Expanding and Collapsing Stacks 71**
 - Associated Commands 71**

- Using the Watch Window 72**
 - Controlling the Watch Window 72**
 - Setting an Expression to Watch 73**
 - Viewing Information 73**
 - Expanding and Collapsing Typed Expressions 73**
 - Associated Commands 74**

- Using the Register Window 74**
 - Controlling the Register Window 74**
 - Viewing Information 74**
 - Editing Registers and Flags 75**
 - Associated Commands 76**

- Using the Data Window 76**
 - Controlling the Data Window 76**
 - Viewing Information 77**
 - Changing the Memory Address and Format 78**
 - Editing Memory 78**
 - Assigning Expressions 78**
 - Associated Commands 79**

- Using the Stack Window 79**

- Using the Pentium III Register Window 80**

- Using the FPU Stack Window 80**
 - Viewing Information 80**

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. If you are familiar with SoftICE, read the section that describe the Stack window.

Universal Video Driver

SoftICE now uses a Universal Video Driver (UVD) to display on the users 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. You can use the following commands and key sequences to move, size, and customize the SoftICE display window:

| 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. |
| Control-Alt- cursor key | Moves the SoftICE window by a character increment. |
| Control-Alt-Home | Resets the SoftICE window position to (0, 0) |
| 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. |

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, and (for DirectDraw applications) how many flipping surfaces are in use. 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 139.
- 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 Windows NT results in "Blue Screen" Mode.

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.

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

Disabling SoftICE at Startup

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

```
Press Esc to cancel loading SoftICE
```

If you installed SoftICE as an automatic driver with Windows NT, you cannot disable it unless you change your startup mode and reboot your PC. In the unlikely event that SoftICE causes difficulties during booting, select the following option from the Windows NT boot menu:

```
Last known good configuration
```

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:

| SoftICE Windows | Use |
|----------------------|--|
| Command window | Enter user commands and display information. |
| Code window | Display unassembled instructions and/or source code. |
| Locals window | Display 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 |
| 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. The following figure illustrates a typical SoftICE screen:

The screenshot shows the SoftICE interface with several windows open. On the right side, labels with arrows point to specific windows:

- Register window:** Shows register values like EAX=0000050E, EBX=00550000, ECX=81076142, EDX=00000000, ESI=8154C05C, etc.
- Locals window:** Shows local variables for a function, such as `void * hPrevInstance = 0x00000000`.
- Watch window:** Shows variables being watched, like `void * hwnd = 0x00404E53`.
- Data window:** Shows memory addresses and their contents, such as `013F:00410A30 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00`.
- Code window:** Shows the source code being debugged, including `#include "filer.h"` and `int PASCAL WinMain`.
- Command window:** Shows the current command prompt, `tutorial`.
- Help line:** Shows the prompt `Press any key to continue; Esc to cancel`.

Vertical scroll arrows are visible on the right side of the Locals, Watch, and Code windows.

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 any display option other than a monochrome screen, 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. Monochrome screens limit you to 25 lines. The `WIDTH` command allows you to set the number of display columns between 80 and 160.

Example: `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, and Watch windows.
- Scroll the Code, Command, Data, Locals, 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. To close a window, either repeat the command or use your mouse, if you have one available. 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.

| Command | Window |
|---------|----------------------|
| WC | Code |
| WD | Data |
| WF | FPU Stack |
| WL | Locals |
| WR | Register |
| WW | Watch |
| WS | Stack |
| 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.

Example: `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 window, changing the size of other windows in your display automatically resizes the Command window.

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.

| 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 |
| Watch | Alt-W |

Scrolling Windows

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

SoftICE provides two methods for scrolling windows: key sequences and mouse scroll arrows. The following table describes how to use scroll arrows and key sequences 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.

| 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. |

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:.

| 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

The mouse provides shortcuts for entering the D, U, and WHAT commands. (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.

| 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 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 window, 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 following 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

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.

Hint: 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.

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).

Using Function Keys

SoftICE provides several function key assignments to save you time when entering commonly-used SoftICE commands. The following table lists these 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 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 |
| 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 147.

Editing Commands

Use the following keys to edit the command line.

| 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 32 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.

| 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.

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

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 148 for more information about creating and using persistent macros.

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

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

Hint: 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 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.

Example: 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.

| Run-time Macro Commands | Examples |
|--|------------------------------------|
| MACRO Qexp = "addr explorer; Query %1" | Qexp Qexp 140000 |
| MACRO lshot = "bpx %1 do \"bc bpindex\"" | lshot eip lshot @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 |

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 140 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.

| 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 6: Using Breakpoints* on page 89 for an explanation of how to set breakpoints.)

Controlling the Code Window

Use the following commands to control the Code window.

| 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.

| Function (from within the Code window) | Key Sequence |
|---|---------------------|
| Scroll the Code window to the previous page. | PageUp |
| Scroll the Code window to the next page. | PageDown |
| Scroll the Code window to the previous line. | UpArrow |
| Scroll the 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 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 |

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

| 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 |

Viewing Information

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

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

| Field | Description |
|-------------|--|
| Location | Hexadecimal address of the instruction. If there is a public code symbol 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. |

Example: The following output shows a disassembled instruction:

```
00FD:00001DA1 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 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 the JUMP instruction is always marked with a highlighted arrow indicator (⇒) overlaying the selector portion of the 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 60 for more information about entering commands.

The following commands are particularly useful.

| Command | Function |
|-------------------------|--|
| . | View the instruction at the current EIP. |
| <i>A address</i> | Assemble instructions directly into memory. |
| BPX (F9) | Set point-and-shoot breakpoints. |
| FILE <i>file-name</i> | 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 <i>string</i> | Move the source display to the next occurrence of the specified string. |
| TABS <i>tab-setting</i> | Set tabs for source file display. |
| U <i>address</i> | 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.

| Command | Action |
|-------------------------|--|
| WL | Opens and closes the Locals window. |
| WL [<i>num lines</i>] | Resizes the Locals window. |
| Alt-L | Moves the cursor into or out of the Locals window. |

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.

| 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.

| Command | Function |
|----------------------------|---|
| LOCALS | Lists local variables from the current stack frame. |
| TYPES [<i>type-name</i>] | 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.

| 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. |

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.

| 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.

Example: Monitors the value of ds:esi:

```
WATCH ds:esi
```

Example: 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.

| 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.

| Command | Function |
|-------------------------|--------------------------|
| WATCH <i>expression</i> | 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.

| 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, GS           effective 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.

| 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 116.

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.

| 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 |
| Move the cursor left, right, up, and down in the Register window. | Arrow keys |

Associated Commands

The following commands are associated with the Register window. Refer to the *SoftICE Command Reference* for more information about using these 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 to view different memory locations in a specified format. However, you can view only one Data window at a time.

Controlling the Data Window

Use the following commands to control the Data window.

| Command | Action |
|-------------------|---|
| WD | Opens and closes the Data window. |
| WD [num-lines] | Resizes the Data window. |
| Alt-D | Moves the cursor into or out of the Data window. |
| DATA | Cycles through Data windows. |
| 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. |

Cycling Through Data Windows

To cycle through the Data windows, either enter the DATA command or click the line at the top of the Data window. The Data windows cycle in order from 0 to 3.

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.

| 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.

| Field | Description |
|------------------|---|
| A String | <p>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:</p> <ul style="list-style-type: none"> • 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. |
| 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. Only one Data window displays at a time. |

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

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
```

Hint: 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.

Editing Memory

To edit memory, move the cursor into the Data window and use either hexadecimal or ASCII characters. Use the following keys for editing within the Data window.

| 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 |

Hint: You can also use the **E** command to edit data.

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.

| 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 WS command to open and close the Stack window.

| Command/Keys | Function |
|--------------|--|
| 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 and Code windows.

Using the Pentium III Register Window

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

| Command | Function |
|---------|-------------------------------|
| f | Display as short real values |
| d | Display as dword values |
| * | Toggle between dword and real |

Using the FPU Stack Window

The FPU Stack window displays the current state of the floating point unit (FPU) stack and MMX 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. Windows NT enables the FPU for a thread after it executes one FPU-related instruction.

The Intel architecture aliases the 64-bit MMX registers upon the FPU stack. To display registers in the FPU Stack window, select one of the following data formats.

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

| 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 | |

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.)

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

5 Using SoftICE

Debugging Multiple Programs at Once 83

Trapping Faults 83

Ring 3 32-bit protected mode (Win32 programs) 83

Ring 0 driver code (Kernel-mode device drivers) 84

Ring 3 16-bit protected mode (16-bit Windows programs) 84

About Address Contexts 85

Using INT 0x41 .DOT Commands 86

Understanding Transitions From Ring-3 to Ring-0 87

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.

Trapping Faults

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

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

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

The following sections describe fault trapping support.

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 NT 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 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 Windows NT.

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 NT 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, Windows NT displays a blue screen and then hangs. If you want to gain control after the blue screen, turn on I3HERE (SET I3HERE ON); Windows NT executes an INT 3 instruction after it displays the blue screen.

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, Windows NT 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 NT 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 NT 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 faults in Windows NT 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 NT to skip the faulting instruction and to continue to execute the program. Choose CLOSE to instruct Windows NT to display the second dialog box, as previously described.

About Address Contexts

Windows 95, Windows 98, and Windows NT give each process its own address space from 0 GB to 2GB. Additionally, Windows 95 and Windows 98 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 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 95 and Windows 98 and 0 to 0x7FFFFFFF for Windows NT). 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.

Warning: 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 95 and Windows 98, 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 95 and Windows 98 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.

Warning: The debug functionality for all .DOT extensions is built into VMM or another VxD. It is not part of SoftICE. Thus, 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. Thus, using the .DOT extension at an inappropriate time might result in a system crash.

SoftICE supports the following .DOT commands in Windows 95 and Windows 98:

- 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 95 and Windows 98:

```
◇ .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 95 and Windows 98 retail build, the ..? command yields the following .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 95 and Windows 98, 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 95 and Windows 98 uses the following methods to transition Ring-3 code to Ring-0 code:

- For V86 code, Windows 95 and Windows 98 use 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 95 and Windows 98 use 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 95 and Windows 98 use 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 95 and Windows 98 use 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
```


6 Using Breakpoints

| | |
|---|------------|
| Introduction | 91 |
| Types of Breakpoints Supported by SoftICE | 91 |
| Breakpoint Options | 92 |
| Execution Breakpoints | 92 |
| Memory Breakpoints | 93 |
| Interrupt Breakpoints | 94 |
| I/O Breakpoints | 95 |
| Window Message Breakpoints | 96 |
| Understanding Breakpoint Contexts | 97 |
| Virtual Breakpoints | 97 |
| Setting a Breakpoint Action | 98 |
| Conditional Breakpoints | 98 |
| Conditional Breakpoint Count Functions | 100 |
| Using Local Variables in Conditional Expressions | 103 |
| Referencing the Stack in Conditional Breakpoints | 104 |
| Performance | 106 |
| Duplicate Breakpoints | 106 |
| Elapsed Time | 106 |
| Breakpoint Statistics | 107 |
| Referring to Breakpoints in Expressions | 107 |
| Manipulating Breakpoints | 108 |
| Using Embedded Breakpoints | 108 |

Introduction

You can use SoftICE to set breakpoints on program execution, memory location reads and writes, interrupts, and reads and writes to I/O ports. SoftICE assigns a breakpoint index, from 0 to FF, 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.

You can set up to 256 breakpoints at one time in SoftICE. However, the number of breakpoints you can set on memory location (BPMs) and I/O ports (BPIOs) is a total of four, due to restrictions of the 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 even 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.

Breakpoint Options

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 98.
- 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 98.

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 98.

DO "command1;command2;..."

Refer to *Setting a Breakpoint Action* on page 98.

Example: 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;..."]
```

| | |
|-----------------------------------|---|
| <i>BPM and BPMB</i> | Set a byte-size breakpoint. |
| <i>BPMW</i> | Sets a word (2-byte) size breakpoint. |
| <i>BPM D</i> | Sets a dword (4-byte) size breakpoint. |
| <i>R, W, and RW</i> | Break on reads, writes, or both. |
| <i>X</i> | 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. |
| <i>debug register</i> | Specifies which debug register to use. SoftICE normally manages the debug register for you, unless you need to specify it in an unusual situation. |
| <i>IF expression</i> | Refer to <i>Conditional Breakpoints</i> on page 98. |
| <i>DO "command1;command2;..."</i> | Refer to <i>Setting a Breakpoint Action</i> on page 98. |

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

```
BPM D 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;..."]
```

| | |
|-----------------------------------|---|
| <i>interrupt-number</i> | Number ranging from 0 to 255 (0 to FF hex). |
| <i>IF expression</i> | Refer to <i>Conditional Breakpoints</i> on page 98. |
| <i>DO "command1;command2;..."</i> | Refer to <i>Setting a Breakpoint Action</i> on page 98. |

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.

Example: 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.

Example: 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 95 and Windows 98, 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;..."]
```

| | |
|-----------------------------------|---|
| <i>R, W, and RW</i> | Break on reads (IN instructions), writes (OUT instructions), or both, respectively. |
| <i>IF expression</i> | Refer to <i>Conditional Breakpoints</i> on page 98. |
| <i>DO "command1;command2;..."</i> | Refer to <i>Setting a Breakpoint Action</i> on page 98. |

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 2FE, a word I/O to port 2FE would trigger the breakpoint.

Example: 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;..."]
```

| | |
|-----------------------------------|--|
| <i>window-handle</i> | Value returned when the window was created; you can use the HWND command to get a list of windows with their handles. |
| <i>L</i> | Signifies that the window message should be printed to the Command window without popping into SoftICE. |
| <i>begin-message</i> | 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. |
| <i>end-message</i> | Higher message number in a range of Windows messages. |
| <i>IF expression</i> | Refer to <i>Conditional Breakpoints</i> on page 98. |
| <i>DO "command1;command2;..."</i> | Refer to <i>Setting a Breakpoint Action</i> on page 98. |

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.

Example: 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 5FE8DD12 IF (esp->8)==WM_NCPAINT
```

Warning: When setting a breakpoint using a raw address (not a symbol), it is vital to be in the correct address context.

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.

The exception is if another process mapped the module at a different base address than the one in which the breakpoint is set. In this case, the breakpoint does not trigger. Avoid this situation by basing your DLLs at non-conflicting addresses.

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 148 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 102), 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 pops up 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 Windows NT 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!HMAAllocObject 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 (runtime) 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.

Example: 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.

Example: `bpx myaddr if (eax==43) || (bpmisss>=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.

Example: 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.

Example: 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.

Example: This 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:

```
bpv myaddr do "bstat bpindex"
```

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

```
bpv myaddr do "t;bpv @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. The following concepts use the foobar function to explain this concept.

FooBar Function

```

1:DWORD foobar ( DWORD foo )
2:{
3:    DWORD  fooTmp=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 103. 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.

This 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:

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

Which sets up a stack frame as follows:

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

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

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

- 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 $EBP+(\text{param\#} * 4)+4$ to address parameters, where param\# is $1 \dots n$. 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 Windows NT 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.

Example: 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 following table describes the 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 the following command:
 - **SET I1HERE ON** for INT 1 breakpoints
 - **SET I3HERE ON** for INT 3 breakpoints

*It has long been an axiom of mine that the little things are
infinitely the most important.*

◇ Sir Arthur Conan Doyle

7 Using Expressions

| | |
|---------------------------------------|------------|
| Expressions | 111 |
| Operators | 111 |
| Operator Precedence | 113 |
| Forming Expressions | 114 |
| Expression Types | 118 |
| Type Casting | 121 |
| Evaluating Symbols | 123 |
| Using Indirection With Symbols | 123 |

Expressions

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

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

Other than the maximum length of a SoftICE command line (80 characters), 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.

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

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

Note: The expression esp->4 is shorthand notation for *(esp+4).

Operators

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

| Indirection Operators | Example |
|-----------------------|---|
| -> | ebp->8 (gets Dword pointed to by ebp+8) |
| . | eax.1C (gets Dword pointed to by eax+1c) |
| * | *eax (gets Dword value pointed to by eax) |
| @ | @eax (gets Dword value pointed to by eax) |

| Math Operators | Example |
|----------------|---------------|
| unary + | +42 (decimal) |
| unary - | -42 (decimal) |
| + | eax + 1 |
| - | ebp - 4 |
| * | ebx * 4 |

| Math Operators | Example |
|--------------------------|---|
| / | Symbol / 2 |
| % (modulo) | eax % 3 |
| << (logical shift left) | bl << 1 (result is bl shifted left by 1) |
| >> (logical shift right) | eax >> 2 (result is eax shifted right by 2) |

| Bitwise Operators | Example |
|--------------------------|----------------|
| & (bitwise AND) | eax & F7 |
| (bitwise OR) | Symbol 4 |
| ^ (bitwise XOR) | ebx ^ 0xFF |
| ~ (bitwise NOT) | ~dx |

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

| Special Operators | Example |
|--------------------------|--|
| . (line number) | .123 (value is address of line 123 in the current source file) |
| (,) (grouping symbols) | (eax+3) * 4 |
| , (arglist) | function(eax,ebx) |
| : (segment operator) | es:ebx |
| function | word(Symbol) |
| # (prot-mode selector) | #es:ebx (address is protected-mode selector:offset) |
| \$ (real-mode segment) | \$es:di (address is real-mode segment:offset) |

Operator Precedence

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

Example: In previous versions of SoftICE, the addition operator (+) and the multiplication operator (*) had the same precedence, so the expression $3+4*5$ evaluated to 35. The new expression evaluator gives multiplication higher precedence, so the result of $3+4*5$, which is equivalent to $3+(4*5)$, is 23. To achieve the result of 35, use parentheses to force the addition to be evaluated first: $(3+4)*5$.

The following table lists all the operators in order of precedence. Operators of equivalent precedence are evaluated according to their associativity.

| Operator | Associates | Comment |
|----------------|---------------|--------------------------|
| (,), function | | scopes, function |
| ->, . | left-to-right | indirection |
| : | left-to-right | selector : offset |
| #, \$ | right-to-left | selector overrides |
| *, @, | right-to-left | indirection |
| unary + | | default radix == decimal |
| unary - | | default radix == decimal |
| !, ~ | | Line Number |
| . | | |
| *, /, % | left-to-right | |
| +, - | left-to-right | |
| <<, >> | left-to-right | |
| <, <=, >, >= | left-to-right | |
| ==, != | left-to-right | |
| & | left-to-right | |
| ^ | left-to-right | |
| | left-to-right | |
| && | left-to-right | |
| | left-to-right | |
| comma | left-to-right | arglist |

Forming Expressions

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

Hint: Use the ? (evaluate expression) command to display the result of any expression.

Numbers

The SoftICE expression evaluator accepts numeric input in the following forms:

| Input | Description |
|-------------|--|
| Hexadecimal | <p>Hexadecimal is the default radix for all numeric input and output. The valid character set for hexadecimal numbers is [0-9, A-F]. Hexadecimal input can be optionally preceded by the standard C language radix identifier: 0x. Examples of valid hexadecimal numbers include:</p> <p style="text-align: center;">FF, ABC, 0x123, 0xFFFF0000</p> <p>The symbolic form of a valid hexadecimal number could conflict with a symbol name. For example, ABC. Use the 0x form to ensure that the number is not misinterpreted as a symbol name.</p> |
| Decimal | <p>SoftICE uses the implied semantics of the unary + and unary - operators to force the default radix to temporarily become decimal. This is based on the fact that +FF and -ABC are relatively unnatural, but still legal, forms of saying decimal 255 and -2748. If you directly precede a number with a unary + or unary -, SoftICE attempts to evaluate that number as decimal and, if that fails, as hexadecimal.</p> <p>The following examples use the unary + and unary - operators to affect how the radix of a number is interpreted:</p> <ul style="list-style-type: none"> • ? +42 0000002A 0000000042 "" • ? -42 FFFFFFD6 4294967254 (-42) "ÿÿÿö" • ? -1a FFFFFFE6 4294967270 (-26) "ÿÿÿæ" • ? +ff 000000FF 0000000255 "ÿ" • ? +(12) 00000012 0000000018 "" <p><i>Note:</i> The SoftICE line number operator (.) also changes the default radix to decimal. The unary + operator is a NOP for expression evaluation, and other than changing the default radix, it has no effect.</p> |

Character Constants

SoftICE supports the use of standard C language character constants such as 'b', 'ABCD', or '\x23'. The default radix for character constants that begin with a backslash '\' is decimal. To specify a hex character constant, use an x prefix such as in '\x23'.

Registers

SoftICE supports the standard names for the Intel register set:

| | | | |
|----|-----|-----|----|
| AH | CS | EBX | FL |
| AL | CX | ECX | FS |
| AX | DH | EDI | GS |
| BH | DI | EDX | IP |
| BL | DL | EFL | SI |
| BP | DS | EIP | SP |
| BX | DX | ES | SS |
| CH | EAX | ESI | |
| CL | EBP | ESP | |

Hint: You can use built-in functions to access individual flags within the EFL and FL flags register. Refer to *Built-in Functions* on page 116.

Symbols

Symbol names are the symbolic representation of an address or value. They can be present in a variety of forms, including symbol tables, exports, or built-in functions.

The form of symbol names closely follows the C language definition, but extensions have been made for other languages, including assembler and C++. Legal symbol names start with an at sign (@), underscore (_), or a letter from A through Z and are composed of an unbroken string of letters (A through Z), numbers (0 through 9), or other legal symbol name characters such as an underscore.

Note: C++ symbols may include the scope (::) operator, where the symbol name form is CLASS::MEMBER.

Each time symbols are loaded into SoftICE, they are placed in a separate table. Symbols used for each executable are placed in a separate table. SoftICE does not find a symbol unless it is in the currently active table, because each table is treated as a separate entity. Refer to the TABLE command in the *SoftICE Command Reference*.

Note: This does not apply to exports, because SoftICE treats all exports as one homogeneous unit whereas symbol tables are discrete entities.

A symbol table specifier can precede a symbol name; this enables symbols from different tables to be used in a single expression. The table name and the symbol name are delimited by an exclamation point (!), for example:

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

Built-in Functions

SoftICE predefines a number of functions for use in expressions. They take a variety of forms and represent static values, dynamic values within the operating system or SoftICE, or functions that can be used within expressions to modify values or translate data types.

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

FUNC (arg-list)

Note: Function names are superseded by a symbol of the same name within a symbol table or export table.

The following functions are defined for SoftICE:

| Function Name | Description | Example |
|---------------|--|--|
| Byte | Get low-order byte | ? Byte(0x1234) = 0x34 |
| Word | Get low-order word | ? Word(0x12345678) = 0x5678 |
| Dword | Get low-order dword | ? Dword(0xFF) = 0x000000FF |
| HiByte | Get high-order byte | ? HiByte(0x1234) = 0x12 |
| HiWord | Get high-order word | ? HiWord(0x12345678) = 0x1234 |
| Sword | Convert byte to signed word | ? Sword(0x80) = 0xFF80 |
| Long | Convert byte or word to signed long | ? Long(0xFF) = 0xFFFFFFFF ? Long(0xFFFF) = 0xFFFFFFFF |
| WSTR | Display as Unicode string | ? WSTR(eax) |
| Flat | Convert a selector-relative address to a linear (flat) address | ? Flat(fs:0) = 0xFFDFF000 |
| CFL | Carry Flag | ? CFL = bool-type |
| PFL | Parity Flag | ? PFL = bool-type |
| AFL | Auxiliary Flag | ? AFL = bool-type |
| ZFL | Zero Flag | ? ZFL = bool-type |
| SFL | Sign Flag | ? SFL = bool-type |
| OFL | Overflow Flag | ? OFL = bool-type |
| RFL | Resume Flag | ? RFL = bool-type |
| TFL | Trap Flag | ? TFL = bool-type |
| DFL | Direction Flag | ? DFL = bool-type |
| IFL | Interrupt Flag | ? IFL = bool-type |
| NTFL | Nested Task Flag | ? NTFL = bool-type |

| Function Name | Description | Example |
|---------------|---|-------------------------------------|
| IOPL | IOPL level | ? IOPL = current IO privilege level |
| VMFL | Virtual Machine Flag | ? VMFL = bool-type |
| IRQL | Windows NT OS IRQ Level | ? IRQL = unsigned-char |
| DataAddr | Returns the address of the first data item displayed in the Data window | dd @dataaddr |
| CodeAddr | Returns the address of the first instruction displayed in the Code window | ? codeaddr |
| Eaddr | Effective address, if any, of the current instruction. Refer to <i>Eaddr Function</i> on page 117 | |
| Evalue | Current value at the effective address. Refer to <i>Evalue Function</i> on page 118 | |
| Process | KPEB (Kernel Process Environment Block) of the Active OS process | ? process |
| Thread | KTEB (Kernel Thread Environment Block) of the Active OS thread | ? thread |
| PID | Active process Id | ? pid == Test32Pid |
| TID | Active thread Id | ? tid == Test32MainTid |
| BPCount | Breakpoint instance count. For these BP functions, refer to <i>Conditional Breakpoint Count Functions</i> on page 100 | 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" |

Eaddr Function

The Eaddr function returns the effective address, if any, that the instruction at the current EIP uses. The EIP register points to that instruction.

Note: The effective address of the current instruction, if any, and the value at that address also display in the Register window directly beneath the flag settings.

The x86 processor supplies a variety of memory addressing modes such as register+offset and register+register. The result of computing the memory address 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. An x86 instruction never has an effective address as both source and destination.

Some instructions may 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 the PUSH and POP instructions.

Example: The current instruction is:

```
MOV ECX, [ESP+4]
```

The Eaddr function returns a value equal to ESP+4, that is, the current value of ESP plus 4.

Example: The current instruction is:

```
ADD BYTE PTR [ESI+EBX+2], 55
```

The 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.

Note: The effective address of the current instruction, if any, and the value at that address display in the Register window directly beneath the flag settings.

Expression Types

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

| Type | Example |
|---------------|-----------------------|
| Literal-type | 1, 0x80000000, 'ABCD' |
| Register-type | EAX, DS, ESP |
| Symbol-type | PoolHitTag, IsWindow |
| Address-type | 40:17, FS:18, 8Symbol |

Note: As a class, functions do not have a type, but they resolve into one of the types previously listed.

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

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

SoftICE treats a symbol as an address-type if you use it in an expression where an address-type is legal and it makes sense to use an address. Otherwise, SoftICE automatically indirections the symbol, taking the contents of the memory the symbol represents. There are many operations that are illegal or do not make sense for address-types such as multiplication and division, so a majority of the operators used with the symbol-type act like a C compiler and automatically take the contents-of at the address for the symbol.

The following summary shows how SoftICE interprets symbols within expressions:

| Example | Equivalent Expression | Result Type (for Symbol) |
|--|--|--------------------------|
| <code>u Symbol</code> | <code>u &Symbol</code> | address-of |
| <code>db Symbol + 1</code> | <code>db &Symbol + 1</code> | address-of |
| <code>db Symbol + ds:8000</code> | <code>db *Symbol + ds:8000</code> | contents-of |
| <code>db Symbol + Symbol2</code> | <code>db &Symbol + *Symbol2</code> | address-of |
| <code>? Symbol - 1</code> | <code>? &Symbol - 1</code> | address-of |
| <code>? Symbol - ds:8000</code> | <code>? &Symbol - ds:8000</code> | address-of |
| <code>? Symbol - Symbol2</code> | <code>? *Symbol - *Symbol2</code> | contents-of |
| <code>? Symbol && 1</code> | <code>? *Symbol && 1</code> | contents-of |
| <code>? Symbol && ds:8000</code> | <code>? *Symbol && ds:8000</code> | contents-of |
| <code>? Symbol && Symbol2</code> | <code>? *Symbol && *Symbol2</code> | contents-of |
| <code>? Symbol <= 8000</code> | <code>? *Symbol <= 8000</code> | contents-of |
| <code>? Symbol != &Symbol2</code> | <code>? &Symbol != &Symbol2</code> | address-of |
| <code>? Symbol == Symbol2</code> | <code>? *Symbol == *Symbol2</code> | contents-of |

| Example | Equivalent Expression | Result Type (for Symbol) |
|-----------------|-----------------------|--------------------------|
| ? Symbol : 8000 | ? *Symbol : 8000 | contents-of |
| ? -Symbol | ? -*Symbol | contents-of |
| ? !Symbol | ? !*Symbol | contents-of |
| ? Symbol->4 | ? *(&Symbol+4) | address-of |

The following operations *cannot* be directly performed on or between address-types:

| Invalid Expression Form | Example |
|---|--|
| address-type [* , / , % , << , >>] any-type | &Symbol * 4 |
| address-type [+ , & , , ^] address-type | ds:80ff ^ &Symbol |
| any-type [-> , .] address-type | ebp->&Symbol2 |
| address-type [:] any-type | &Symbol : 8000 |
| [- , . , &] address-type | - &Symbol , . &Symbol (line number) |
| address-type - address-type | 23:8fff - 23:4ff0 (legal) |
| Note: This expression is illegal only if address selectors do not have the same value and type. | 1b::0 - 23:0 (illegal) |

Note: Unlike symbol-types, SoftICE does not automatically indirect an address-type. You must explicitly indirect the address-type using one of the indirection operators.

Indirection

There is a subtle difference between the indirection operators (->) and (.) and the indirection operators (*) and (@). The result of an (->) or (.) operator is a plain Dword value, while the result of (*) and (@) is an address-type.

The following expression is illegal, because multiplication is not a valid operation for addresses:

```
? (*Symbol)*3
```

If you try this, you receive the error message `Expecting value, not address.`

However, the following expression is perfectly legal, because the result of Symbol->0 is a plain value, not an address-type:

```
? (Symbol->0)*3
```

This distinction is useful when performing multiple indirections in 16-bit code, because address-type values retain segment/selector information.

Operand Sizes

The SoftICE expression evaluator treats all operand types as Dword (unsigned long) values. This means that you must manually indicate the size of a type using type casting or one of the conversion functions such as `byte()` or `word()`.

Example: If you de-reference memory, SoftICE always returns a Dword value. This may not be suitable, for example, if you are interested in a byte value. To correctly compare a byte-value in a conditional expression, it is necessary to mask off the upper 24-bits, leaving the lower 8-bits intact. In the following expression, assume `Symbol` is a byte value:

```
BPX EIP IF (Symbol == 32)
```

This expression is likely to fail because SoftICE reads a full 32-bit value and compares that to (DWORD) 32, or 0x00000032. This is probably not what you want. The following expressions work correctly:

```
BPX EIP IF ((Symbol & FF) == 32)
or
BPX EIP IF (byte(Symbol) == 32)
```

Use whichever form you prefer; they are equivalent.

Type Casting

The expression evaluator supports the following:

- C++ style type casting

You can use the following form to cast any value to a defined type:

```
TypeName (expression)
```

Note: *TypeName* is case sensitive because a hash lookup is performed instead of a linear search.

- Structure and class indirection through members

```
TypeName (expression)->member
```

After the indirection performs, the new type of the expression is automatically type cast to the type of member. This allows multiple indirections to occur.

```
TypeName (expression)->member->member->member
```

At each indirection, the value of member is evaluated, the automatic type cast applied, and the next member evaluated and type cast until the expression is resolved.

- Taking the address of a member or type

You can use the & (address-of) operator to take the address of a structure or structure member.

```
&TypeName(expression)->member[->member[->member]]
```

This allows you to set BPM style breakpoints on structure members.

- Displaying typed expressions

Wherever possible, the ? (evaluate expression) command displays the result of an expression as a type. Many normal expressions, like registers, have default types.

For complex types, the class or structure members are expanded. Only members at the root level of the object are expanded. Note that base and virtual base classes are considered to be root objects.

```
Example: :? LPSTR (*(ebp-30))
char * = 0x009D000C
<"C:\TOMSDEV\WINICE\NTICE"> char = 0x43 , 'C'
```

```
Example: :? STHashTable (a7bcb0)
class STHashTable = {...}
struct STHashNode * * pHashTable = 0x0089000C <{...}>
unsigned long bucketSize = 0x25
class GrowableArray * pHashEntries = 0x00A7BCC0 <{...}>
```

```
Example: :? STHashTable (a7bcb0)->pHashEntries
class GrowableArray * =0x00A7BCC0 <{...}>
unsigned char * arrayBase = 0x00790078 <">
unsigned char * nextItem = 0x00790078 <">
unsigned long memAvail = 0x1000
unsigned long elementSize = 0x10
```

- Displaying pointers to pointers

Types that are pointers to pointers display the value pointed to.

```
typedef LPSTR *LPLPSTR ;
```

```
? LPLPSTR (eax)
```

```
char **eax = 0x127894 <0x434000>
```

where 0x127894 represents the pointer value and 0x434000 represents the value of the pointer that it points to.

- Displaying unicode strings

Use theWSTR type cast operator to display unicode strings.

```
?WSTR (eax)
```

```
short *eax = <"Company Name">
```

Evaluating Symbols

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

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

To get the address of MyVariable, use the following:

```
? &MyVariable
```

If you use a symbol in conjunction with a command other than ?, the symbol yields the address of the symbol instead of its contents. For example, if you enter DD MyVariable, the Data window displays at the address of MyVariable and the first dword is the number 5.

Using Indirection With Symbols

When you create your symbol file with complete type and symbol information, the expression evaluator supports the ability to dereference through a symbol name using that symbol's type. You can also take the address of a member through a symbol.

```
typedef struct Test
{
    DWORD    dword ;
    LPSTR    lpstr ;
} Test ;

Test test={ 1, "test String" } ;

? test->dword
unsigned long dword=1

? test->lpstr
char *lpstr=0x123456 ,"Test String">

? &test
void * =0x123440

? &test->dword
void * =0x123440

? &test->lpstr
void * =0x123444
```

You can do the same thing through type casting, as follows:

```
Test (eax)->dword or Test (eax)->lpstr
and
&Test (eax)->dword or &Test (eax)->lpstr
```


8 Loading Symbols for System Components

Loading Export Symbols for DLLs and EXEs 127

Using Unnamed Entry Points 127

Using Export Names in Expressions 128

Loading 32-bit DLL Exports Dynamically 128

Using Windows NT Symbol (DBG) Files with SoftICE 129

Using Windows 95 and Windows 98 Symbol (.SYM) Files with SoftICE 129

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 139 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.

Note: 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.

Example: 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.

Example: 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 32-bit DLL Exports Dynamically

SoftICE lets you load 32-bit exports without having to restart the system. 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. The Load Exports window appears.
- 3 Select the files you want to load and click OPEN.

Using Windows NT Symbol (DBG) Files with SoftICE

Microsoft supplies debugging information for key Windows NT components. This debugging information takes the form of .DBG files, which contain COFF debug data for the corresponding .EXE or .DLL. You can find the .DBG files on the Windows NT CD-ROM or you can download them with the associated service pack. To use a .DBG file with SoftICE, use Symbol Loader to translate it to a .NMS file and load it.

Using Windows 95 and Windows 98 Symbol (.SYM) Files with SoftICE

The Windows 95 and Windows 98 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.

9 Using SoftICE with a Modem

| | |
|------------------------------------|------------|
| Introduction | 133 |
| Hardware Requirements | 133 |
| Establishing a Connection | 133 |
| Using SERIAL32.EXE by Modem | 134 |
| DIAL Command | 135 |
| ANSWER Command | 135 |

Introduction

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 accepts the industry-standard AT commands such as ATZ and ATDT, and returns standard result codes such as RING and CONNECT.
- The modem executes 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 dials the remote user:

- 1 Remote user runs SERIAL32.EXE.
- 2 SoftICE user invokes the DIAL command.

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 SERIAL32.EXE.

A connection is established and the remote user is in control of SoftICE.

The following sections describe how to use SERIAL32.EXE and the DIAL and ANSWER commands. Refer to the *SoftICE Command Reference* for more information on the commands.

Using SERIAL32.EXE by Modem

SERIAL32.EXE is an MS-DOS program that acts as a dumb terminal for SoftICE, displaying output and accepting keyboard input. The screen that SERIAL32.EXE shows is exactly what you would see if you were running SoftICE on a local machine.

Notes: You can use the program SERIAL32.EXE with a serial connection or by modem. If you want to use SoftICE with a serial connection, refer to *Installing SoftICE in Getting Started with DriverStudio*.

If the remote machine is running MSDOS, the old MSDOS version of SERIAL may still be used. (The DOS SERIAL.EXE program has been relocated to the UTIL16 directory).

The command-line syntax of SERIAL32.EXE is

```
serial32 [-l=25|43|50|60] [<port name> [<baud rate>]]
```

The command-line syntax of SERIAL.EXE is

```
SERIAL.EXE [r] [com-port] [baud-rate] [I"init-string"] [Pnumber]
```

| | |
|--------------------|--|
| <i>r</i> | Use the <i>r</i> option when you are running in a DOS box in Windows NT on the remote machine. The option disables FIFO and resets the baud-rate, stop bits, and parity. |
| <i>com-port</i> | Is a serial port number, 1-4; it must be specified. |
| <i>baud-rate</i> | Is the rate at which SERIAL32.EXE communicates with the modem. It must be specified. It does not need to match the rate specified with the DIAL and ANSWER commands at the SoftICE end of a connection. |
| <i>I</i> | Use to specify the modem initialization commands. |
| <i>init-string</i> | Is a modem command that must be used to initialize the modem. |
| <i>P</i> | Is used to specify a phone number. If the P option is used to specify a phone number, SERIAL dials the number and attempts to establish a connection with the SoftICE user who must have already issued the ANSWER command in SoftICE. For example: SERIAL 1 57000 p1-603-555-1212. If the P option is not specified, SERIAL operates in an answer mode, waiting for an incoming call. For example, SERIAL 1 57000. |
| <i>number</i> | Is the phone number to dial (when SERIAL is used in dial mode). The string sent to the modem is "ATDT <i>number</i> ". For pulse dialing, use a P as the first digit of the number (this may not work with all modems). |

When a connection is established, the person using SERIAL32.EXE sees the familiar SoftICE text-mode screen and is able to control SoftICE remotely.

DIAL Command

Use the DIAL command within SoftICE to place an outgoing call. The remote user must already be running SERIAL32.EXE in answer mode.

The command syntax is

```
DIAL [ON [com-port] [baud-rate] [I=init-string] [P=number]|OFF]
```

| | |
|--------------------|--|
| <i>ON</i> | Starts dialing. |
| <i>OFF</i> | Terminates a remote session. |
| <i>com-port</i> | Is a serial port number, 1-4; default is 1. |
| <i>baud-rate</i> | Is the rate at which SoftICE communicates with the modem. The default is 57000. It does not need to match the rate specified with SERIAL32.EXE at the other end of the connection. If <i>baud-rate</i> is specified, <i>com-port</i> must also be specified. |
| <i>I</i> | Use to specify the modem initialization commands. |
| <i>init-string</i> | Is a modem command that can be used to initialize the modem. If no init string is specified on the command line, the last init string specified (if any), or the init string specified by the SoftICE initialization settings (if set) is used. |
| <i>P</i> | Is used to specify a phone number. If the P option is used to specify a phone number, SoftICE dials the number and attempts to establish a connection with the SERIAL32.EXE user who must be waiting in answer mode. |
| <i>number</i> | Is the phone number to dial. The string sent to the modem is "ATDT <i>number</i> ". For pulse dialing, use a P as the first digit of the number (this may not work with all modems). If no number is specified on the command line, the last number specified is used (if any), or the number specified by the SoftICE initialization setting is used (if set). For example, DIAL ON 1 57000 p=603-555-1212. |

ANSWER Command

The ANSWER command configures SoftICE to accept an incoming call at any time from another computer running SERIAL32.EXE.

The command syntax is

```
ANSWER [ON [com-port] [baud-rate] [I=init-string]|OFF]
```

| | |
|-----------|----------------------|
| <i>ON</i> | Enables answer mode. |
|-----------|----------------------|

| | |
|--------------------|---|
| <i>OFF</i> | Disables answer mode. |
| <i>com-port</i> | Is a serial port number, 1-4; default is 1. |
| <i>baud-rate</i> | Is the rate at which SoftICE communicates with the modem. The default is 57000. It does not need to match the rate specified with SERIAL32.EXE at the other end of the connection. If baud-rate is specified, com-port must also be specified. |
| <i>init-string</i> | Is a modem command that can be used to initialize the modem. If no init string is specified on the command line, the last init string specified (if any), or the init string specified by the SoftICE initialization settings (if set) is used. For example, ANSWER ON 1 57000. |

10 Customizing SoftICE

| | |
|--|------------|
| Modifying SoftICE Initialization Settings | 139 |
| Modifying General Settings | 139 |
| Pre-loading Symbols and Source Code | 141 |
| Pre-loading Exports | 143 |
| Configuring Remote Debugging - Internet Control | 143 |
| Requirements for Remote SoftICE Support | 143 |
| Setting Up SoftICE for Remote Debugging | 144 |
| Enabling Remote Debugging from the Target Side | 144 |
| Starting the Remote Debugging Session | 145 |
| Configuring Remote Debugging - Dial up Control | 146 |
| Modifying Keyboard Mappings | 147 |
| Working with Persistent Macros | 148 |
| Setting Troubleshooting Options | 151 |

Modifying SoftICE Initialization Settings

SoftICE provides a variety of user-defined settings 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.
- **Remote Debugging: Internet Control**— Define parameters for internet access remote debugging over standard TCP/IP ethernet connection.
- **Remote Debugging: Dial up Control**— Sets a default telephone number and modem initialization strings for remote debugging over a serial port.
- **Keyboard Mappings** — Assigns SoftICE commands to function keys.
- **Macro Definitions** — Defines your own commands to use within SoftICE.
- **Troubleshooting** — Provides solutions to potential problems.

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 SoftICE Initialization Settings window as follows:

SOFTICE INIT SETTINGS DIALOG BOX HERE.

- 3 Select the tab that represents the settings you want to modify.
- 4 Modify the settings and click OK.
The following sections describe these settings.
- 5 Reboot your computer and run SoftICE to apply your changes.

Modifying General Settings

Modify the General SoftICE initialization settings as follows:

Initialization string

INITIALIZATION STRING executes a series of commands when SoftICE initializes. By default, **INITIALIZATION STRING** contains the X (exit) command delimited with a semi-colon, as follows:

X;

You might want to add additional commands to 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 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.

Example: 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 65.

Trace buffer size (Windows 95 and Windows 98 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 95 and Windows 98 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 95 and Windows 98. 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 NuMega 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.

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.
- 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.

Adding Symbol Files to the Symbols List

From the Symbols tab 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.

Hint: Normally, your .NMS symbol file has the same base name as the file you translated. With Windows 95 and Windows 98, 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.

Alternately, you can use the Symbol Loader command-line utility, NMSYM, to specify the output file name.

Hint: 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 95 and Windows 98.

- 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 95 and Windows 98, this memory region also serves as a buffer for holding .NMS images at boot time. By default, SoftICE reserves 1024KB for Windows 95, and Windows 98, and 512KB for Windows NT.

Typically 512KB is adequate for Windows 95, Windows 98, and Windows NT. 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 95 and Windows 98, 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. Note that if you are connected to a network, you can click NETWORK to display the contents of networked drives.

- 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.

Configuring Remote Debugging - Internet Control

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 95, Windows 98, and Windows NT.

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)

The machine that controls the target machine is called the **remote** machine.

- The remote machine must be connected to an IP network that is directly or indirectly connected to the IP network of the target machine. The remote machine also must be running Windows 95, Windows 98, or Windows NT.

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 NT, it's in the \WINNT\SYSTEM32\DRIVERS directory; for Win95, 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 <IP address|DHCP> [MASK=<subnet mask>] [GATEWAY=<IP address>]
NET ALLOW <IP address|ANY> [AUTO] [PASSWORD=<password>]
NET PING <IP address>
NET RESET
NET STOP
NET HELP
NET STATUS
```

NET START <IP address|DHCP> [MASK=<subnet mask>] [GATEWAY=<IP address>]

The NET START command 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 <IP address|ANY> [AUTO] [PASSWORD=<password>]

The NET ALLOW command 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 <IP address>

The NET PING command 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 4 seconds.

NET RESET

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

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

The NET HELP command shows a list of the available network commands with their respective syntax.

NET STATUS

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 current 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 SINET command. Following is the syntax for the SINET command.

```
SINET <target IP address> [<password>]
```

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.

SINET tries to create a connection to the target machine. If the target machine responds, SINET 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 SINET 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.

Configuring Remote Debugging - Dial up Control

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.

Telephone number

TELEPHONE NUMBER presets a phone number for the DIAL command, for example: 412-555-1212.

Serial connection (Windows 95 and Windows 98 only)

If you are using SoftICE for Windows 95 and Windows 98, 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 Windows NT, SoftICE automatically determines your serial connection.

DIAL initialization string

DIAL INITIALIZATION STRING presets the modem initialization string for the DIAL command, for example: ATX0.

ANSWER initialization string

ANSWER INITIALIZATION STRING presets the modem initialization string for the ANSWER command, for example: ATX0.

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 12 function keys or key combinations involving Shift, Ctrl, or Alt and a function key.

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

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.

Note: SoftICE uses the following abbreviations for the Function, Alt, Ctrl, and Shift keys:

| Key | Abbreviation | Example |
|----------|--------------|---------|
| Function | F | F1 |
| Alt | A | AF1 |
| Ctrl | C | CF1 |
| Shift | S | SF1 |

- 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:

| Default Function Key Assignments | |
|----------------------------------|----------------------|
| F1 = H; | F11 = ^G @SS:ESP; |
| F2 = ^WR; | F12 = ^P RET; |
| F3 = ^SRC; | SF3 = ^FORMAT; |
| F4 = ^RS; | AF1 = ^WR; |
| F5 = ^X; | AF2 = ^WD; |
| F6 = ^EC; | AF3 = ^WC; |
| F7 = ^HERE; | AF4 = ^WW; |
| F8 = ^T; | AF5 = CLS; |
| F9 = ^BPX; | AF11=dd dataaddr->0; |
| F10 = ^P; | AF12=dd dataaddr->4; |

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 64 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 alphanumeric 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.

Example: 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).

- 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.

| Legal Name | Legal Definition | Example |
|------------|--------------------------------------|-------------------------------|
| Qexp | <code>addr explorer; Query %1</code> | Qexp Qexp 140000 |
| lshot | <code>bpx %1 do \"bc bindex\"</code> | lshot eip or lshot @esp |

| Legal Name | Legal Definition | Example |
|------------|-------------------------------------|------------------------------------|
| 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:

| Illegal Name or Definition | Explanation |
|---|--|
| Name: $\text{\textcircled{D}}$ 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 the ESC key.

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

These settings let you troubleshoot SoftICE. Modify these settings only when directed to do so by NuMega Technical Support or to remedy the specific situations described within this documentation. By default, the Troubleshooting settings are all turned off.

Hint: If you want to return all the troubleshooting settings to their original states, click RESTORE DEFAULTS.

Hint: 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.

Disable mouse support

If you are having problems using your mouse in SoftICE, select DISABLE MOUSE SUPPORT.

Disable Num Lock and Caps Lock programming

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 Windows NT, try the DO NOT PATCH KEYBOARD DRIVER setting.

Do not patch keyboard driver (Windows NT 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.

11 Exploring Windows NT

| | |
|---|------------|
| Overview | 155 |
| Resources for Advanced Debugging | 155 |
| Inside the Windows NT Kernel | 158 |
| Managing the Intel Architecture | 159 |
| Windows NT System Memory Map | 163 |
| Win32 Subsystem | 170 |
| Inside CSRSS | 170 |
| USER and GDI Objects | 172 |
| Process Address Space | 177 |
| Heap API | 178 |

Overview

Without qualification, the Windows NT operating system is an incredible feat of software engineering and system design. It is hard to imagine any system of such complexity reaching all of its design goals, including three of the most difficult: portability, reliability, and extensibility, without compromising its interfaces or implementation. Yet, somehow the system engineers at MicroSoft who design and develop Windows NT manage to keep each and every component of the system smoothly interlocked, not unlike the precision gears of a finely made watch. If you are going to write Windows NT applications, you should explore what lies beneath your application code: the Windows NT 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 your application or driver.

This chapter provides a quick overview of the more pertinent and interesting aspects of Windows NT. It focuses on areas where little or no documentation currently exists. 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 pieces of Windows NT fit together.

Resources for Advanced Debugging

Microsoft provides several resources for advanced debugging including: checked build, the Windows NT DDK, .DBG files, and Kernel Debugger Extensions. The following sections discuss these resources:

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. 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 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.

.DBG Files

Microsoft provides a separate DBG 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). Advanced type information such as structures and locals is not provided, but having access to a public definition for each API makes debugging through system calls a lot easier.

Hint: Using .DBG 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 file and use Symbol Loader to create a .NMS file that SoftICE can load.

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

| | |
|-----------------------|--|
| NTOSKRNL . EXE | The Windows NT Kernel. (Most of the operating system resides here.) |
| HAL . DLL | The Hardware Abstraction Layer. Important primitives for NTOSKRNL. |
| NTDLL . DLL | Basic implementation of the Win32 API, and functionality traditionally attributed to KERNEL. Also the interface between USER and SYSTEM mode. Essentially replaces KERNEL32.DLL. |
| CSRSS . EXE | The Win32 subsystem server process. Most subsystem calls are routed through this process. |
| WINSRV . DLL | Under Windows NT 3.51, the core implementation of USER and GDI functionality. Only loaded in the context of CSRSS. |
| WIN32K . SYS | A system device driver that replaces WINSRV.DLL and minimizes inter-process communication between applications and CSRSS. May not be available for all versions of the OS. |
| USER32 . DLL | Basic implementation of USER functionality. Mostly stubs to WINSRV.DLL (via LPC to CSRSS). More recent versions contain more implementation to minimize context switches. |
| KERNEL32 . DLL . | Some basic implementation of traditional KERNEL functionality, but mostly stubs to NTDLL.DLL. |

Resources

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

- *Microsoft Developers Network* (MSDN)
MSDN is published quarterly, on CD-ROM, and it contains a wealth of information and articles on all aspects of programming Microsoft operating systems. This is one of the only places where you can find practical information on writing Windows NT device drivers.
- *Inside Windows NT* - Helen Custer, Microsoft Press
Inside Windows NT provides a high-level view of the design for the Windows NT operating system. Each of the major sub-systems is thoroughly discussed, and many block diagrams illuminate internal data structures, policies, and algorithms. Although the contents of this book may seem highly abstracted from the actual operating system implementation, once you step into OS code with SoftICE, many of the higher level relationships become clear. Currently, this is the most valuable set of information on Windows NT operating system internals. You will gain the most benefit from the information in this book if you use SoftICE to explore the actual implementation of the system design.

- *Advanced Windows*, 2nd Edition - Jeffery Richter, Microsoft Press
Advanced Windows is an excellent resource for the systems programmer developing Win32 applications and system code. Richter presents extensive discussions of processes, threads, memory management, and synchronization objects. Relevant sample code and utilities are also provided.

Inside the Windows 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 Manager
- ◊ Process and Thread creation and manipulation
- ◊ Process and Thread scheduling
- ◊ Local Procedure Call (LPC) facilities
- ◊ Security Management
- ◊ Exception handling
- ◊ VDM hardware emulation
- ◊ Synchronization primitives, such as Semaphores and Mutants
- ◊ Run Time Library
- ◊ File System
- ◊ I/O subsystems

Managing the Intel 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.

| 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 |

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.

| 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 <i>Blue Screen</i> . |
| 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 <i>Blue Screen</i> . |
| 21 | MS-DOS Int 21 trap. Only used for Virtual DOS Machines (VMD) and WOW. |
| 2A | Service to get the current tick count. |
| 2B,2C | Direct thread switch services. |
| 2D | Debug service. |
| 2E | Execute System Service. Windows NT transitions from user mode to system mode using INT 2E. For more information, refer to the NTCALL command in the <i>SoftICE Command Reference</i> . |
| 30-37 | Primary Interrupt Controller (IRQ0-IRQ7). 30 - HAL clock interrupt (IRQ0). |
| 38-3F | Secondary Interrupt Controller (IRQ8-IRQ15). |

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

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

Interrupts for Virtual DOS machines (VDM), which include the WOW (16-bit Windows on Window) subsystem, do not vector directly through the IDT. For a VDM, interrupts are emulated by triggering a general protection fault that special VDM code within NTOSKRNL handles. In most cases, the interrupt is eventually reflected back to the VDM for servicing. MS-DOS Interrupt 21 is handled as a special case (since an actual IDT entry exists). This could be for performance reasons, compatibility issues, or both.

Drivers may install and uninstall interrupt handlers as necessary, using `IoConnectInterrupt` 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 one base TSS that all processes share. 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. Under Windows NT 3.51, when executing in a VDM, the TSS contains a valid offset to a I/O bitmap that traps direct I/O for subsequent emulation by the operating system. 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=FFDF0000  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 always mapped at a base address of 0xFFDF000. 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. 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.

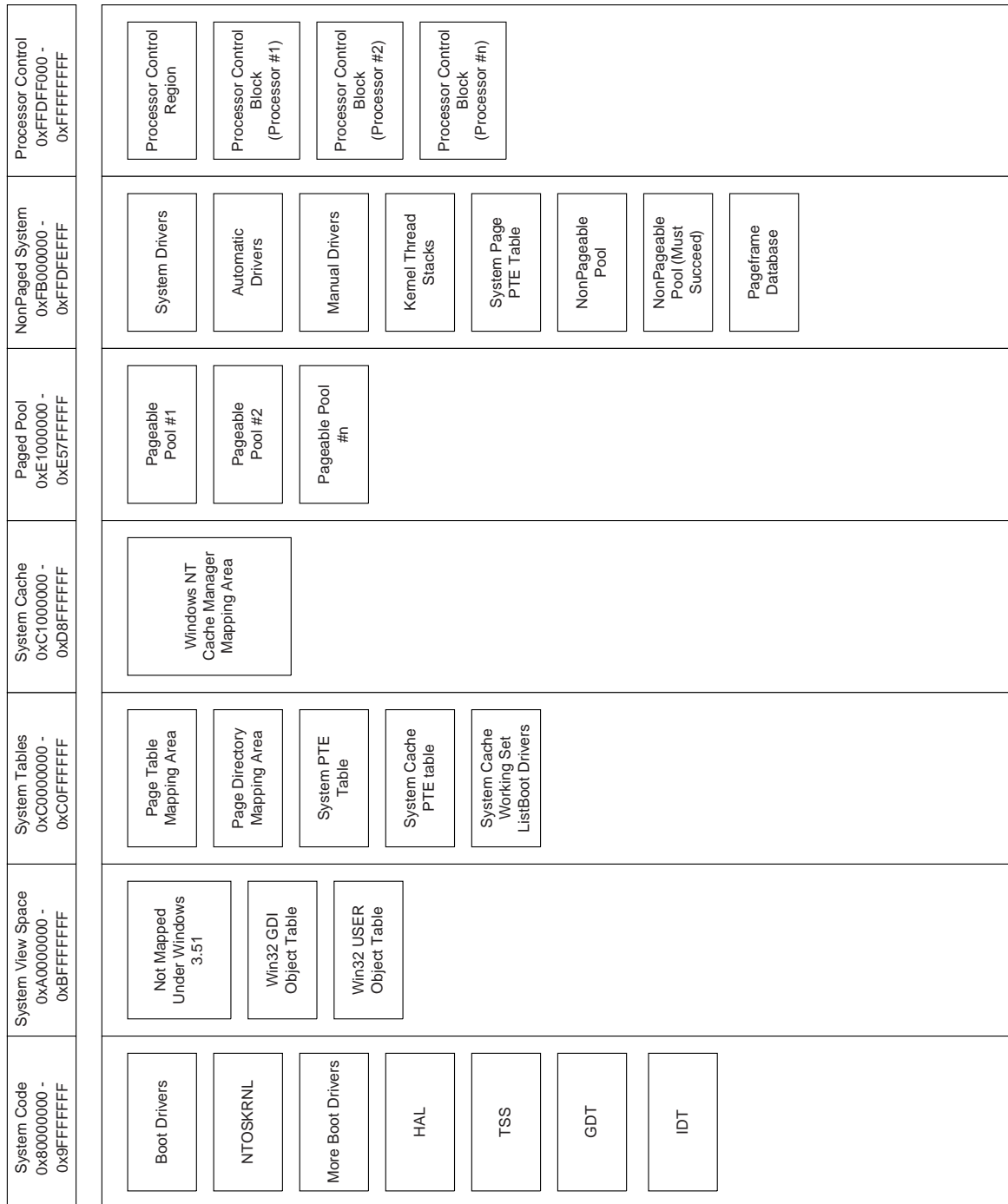
- System View area

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.

The following diagram shows the system memory map for Windows NT.



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.

In this scheme, the Page Directory is actually performing two functions. In addition to being the Page Directory, representing 4GB, it also serves as a page table, representing 4MB in the address range 0xC0000000 - 0xC03FFFFFFF. The Page Directory maps the 4MB region where the process page tables are mapped (0xC0000000-0xC03FFFFFFF), 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 - 0xC03FFFFFFF 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 - A03FFFFFF
017F0000             P   A S RW      A0400000 - A07FFFFFF
01727000             P   A S RW      A0800000 - A0BFFFFFF
01F6E000            P   A S RW      C0000000 - C03FFFFFF
0066F000             P   A S RW      C0400000 - C07FFFFFF
00041000             P   A S RW      C0C00000 - C0FFFFFFF
00042000             P   A S RW      C1000000 - C13FFFFFF

```

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 NonPaged pool area. NonPaged 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 NonPaged 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.

- NonPaged 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 NonPaged memory pools, and the Page Frame Database. Although it is contradictory to say that items in the NonPaged 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 NonPaged pool is similar to the Paged Pool with the exception that objects created in the NonPaged pool are not discarded from memory for any reason. The NonPaged pool is used to allocate key system data structures such as kernel process and thread environment blocks. There is a second NonPaged 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 (PCRB) data structures for each processor within the system and a global data structure, the Processor Control Region that reflects the current state of the system. The Processor Control Region (PCR) contains key pieces of information about the current state of the system, such as the currently running kernel thread; the current interrupt request level (IRQL); the current exception frame; base addresses of the IDT, TSS, and GDT; and kernel thread stack pointers. Small portions of the PCR and PCRB data structures are documented in NTDDK.H.

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

```
? IRQL
00000002h
```

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

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

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

```
? thread
FF088E90h
? tid
71h
```

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

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

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

The current process is not stored as part of the PCR or PCRB. Windows 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 3.51, and to a lesser degree under new versions of the operating system, most of the core functionality is actually implemented in WINSRV.DLL within the CSRSS process. Calls that are traditionally associated with one of the standard Windows components are typically implemented as stubs that call other modules, for example, NTDLL.DLL, or use inter-process communication to CSRSS for servicing.

Most USER and GDI API calls are routed through the appropriate 32-bit module in the process address space. There, they are packaged as Local Procedure Call (LPC) messages and routed to CSRSS for processing. As you might imagine, this LPC mechanism, although much more optimized than a true Remote Procedure Call (RPC), has much more overhead than a simple function call. It is surprising to think that every time your application calls the `IsWindow` function in `USER32.DLL`, it must be packaged for LPC and sent as a subsystem message to CSRSS. For CSRSS to be able to process this message, a process switch must occur and a worker thread must be awoken and dispatched. The specific service must be

determined, parameters must be validated, and finally the service must be executed. When everything is complete on the CSRSS side, a LPC reply must be made to the client (your application), which involves another process switch and unpackaging of the LPC reply. Whew! All that just to determine if a handle represents a valid window.

In their design of a forthcoming version of Windows NT, Microsoft is working to remove as much of this overhead as possible. First, they are moving much of the functionality of WINSRV.DLL into the actual USER32 and GDI32 modules that are loaded into your application's address space. This allows the most common services to execute as simple function calls; no LPC is necessary. Second, rather than making a context switch into CSRSS to access functionality in WINSRV.DLL, a new system driver, WIN32K.SYS allows USER and GDI services to execute more efficiently through a simple transition from user to system mode. Having WIN32K.SYS as a device driver that provides application services allows Windows NT to maintain a high level of encapsulation and robustness, while providing a much more efficient pseudo client-server service architecture.

Although CSRSS executes as a separate process, it still has a big impact on the address space of every Win32 application. If you use the 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

Under Windows NT 3.51, the protected Win32 subsystem process, CSRSS, provides a majority of the traditional USER functionality. APIs and data structures provided by the WINSRV.DLL module manage window classes, and window data structures, as well as many other USER data types.

Under Windows NT 3.51, the following USER object types exist. 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. |
| QUEUE (7) | Windows message queue. |
| CPD (8) | Call Procedure Data thunk. |
| ACCELERATOR (9) | Accelerator Table Object. |
| WINDOW STATION (0xA) | |
| DESKTOP (0xB) | Object representing a desktop window hierarchy. |
| DDEOBJECT (0xC) | DDE Objects such as strings. |

Newer versions of Window NT add/define the following USER object types.

| | |
|------------------------------|---|
| DESKTOP (---) | This Object type has been removed. This type is now a kernel object that is managed by the Kernel Object Manager. |
| QUEUE (---) | This Object type has been removed. |
| WINDOW STATION (0xD) | Changed Object type ID. Also exists as a kernel object. |
| DDEOBJECT (0xA) | Changed Object type ID. |
| KEYBOARD LAYOUT (0xE) | New Object type. Object to describe a keyboard layout. |
| CLIPBOARD FORMAT (7) | New Object type. Registered Clipboard Formats. |

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 HMAAllocObject 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.

| <i>Entry</i> | Object Pointer (DWORD) | Owner (DWORD) | Type (BYTE) | Flags (BYTE) | Instance Count (WORD) | <i>Handle Value</i> |
|--------------|-------------------------------|----------------------|--------------------|---------------------|------------------------------|---------------------|
| 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:E12E9EA | 0001001C | 00000006 | 00000000 | FF0E45D8 |
| 8 | | | | |
| 0010:E12E9EB | 00000000 | E12E7008 | 00000000 | 00000000 |
| 8 | | | | |

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 1 20
0010:E12E700 0001001B 00000000 00000000 E12E9C34
8
0010:E12E701 E17DB714 00000000 00000000 00000000
8
```

- 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 HMAAllocObject 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!HMAAllocObject if (esp->c == %1)"
```

The HMAAllocObject 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!HMAAllocObject if (esp->c == 1)
```

When this breakpoint is instantiated, it traps all calls to HMAAllocObject 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.

| 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. |
| 0x7FFF0000 - 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.

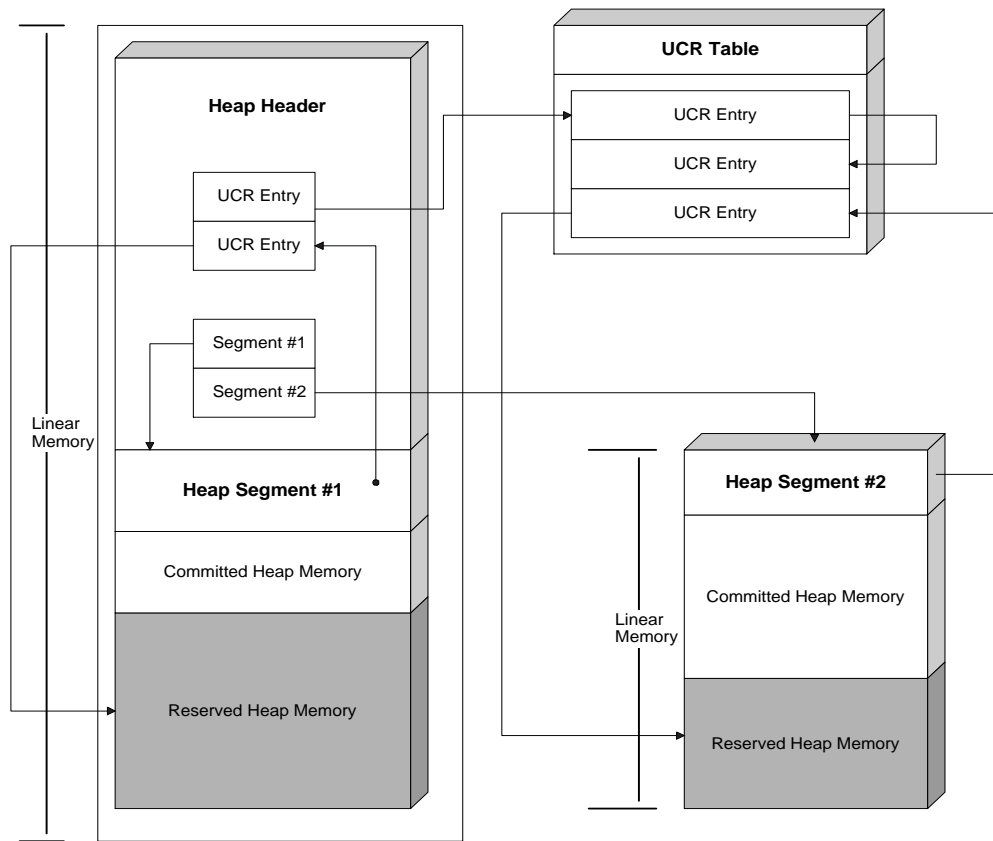
To understand the design and implementation of the Heap API, it is important to realize that a Win32 heap is not necessarily composed of one section of contiguous linear memory. For growable heaps, it might be necessary to allocate many linear regions, using `VirtualAlloc`, which will generally be non-contiguous. Special data structures track all the linear address regions that comprise the heap. These data structures are called 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 #1 is 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 following diagram 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.

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 | Commt/Psnt/Rsvd | Segments | Flags | Process |
|----------|-------------------|-----------------|----------|----------|----------|
| 00140000 | 01 | 001C/0018/00E4 | 1 | 00000002 | Explorer |
| 01 | 00140000-00240000 | 001C/0018/00E4 | | E4000 | |

Diagram annotations:

- Heap segment count (points to the 'Segments' column value '1')
- Heap segment memory range (points to the 'Base' and 'Id' columns for the second row)
- Largest UCR (points to the 'Flags' column value '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 |

Default size of a heap segment
Default size for commits
VM threshold

- 3 Use the `-B` option of the `HEAP32` command to display the base addresses of heap blocks within the default heap:

```
:heap32 -b 140000
```

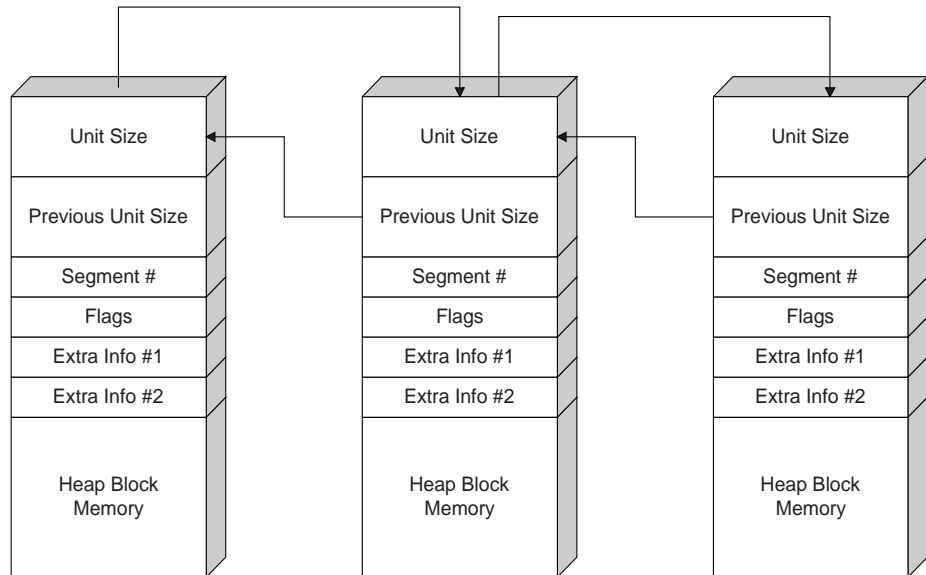
| Base | Type | Size | Seg# | Flags |
|----------|---------|------|------|-------|
| 00140000 | HEAP | 580 | 01 | |
| 00140580 | SEGMENT | 38 | 01 | |
| 001405B8 | ALLOC | 30 | 01 | |

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 following diagram illustrates how the heap manager tracks blocks of *contiguous* memory. The heap manager also tracks non-contiguous free blocks in doubly-linked lists, but the node pointers for the next and previous links are not stored in the block header. Instead, the heap manager uses the first two Dwords within the heap block memory area.



As shown in the preceding diagram, each block stores its unit size as well as the unit size of the previous block. The unit size represents the number of heap units occupied by the heap block. The previous unit size is the number of heap units occupied by the previous heap block. Using these two values, the heap manager is able to walk contiguous heap blocks.

Heap units represent the base granularity of allocations made from a heap. The size of an allocation request is rounded upwards as necessary, so that it is an even multiple of this granularity. Rather than using a granularity of 1 byte, the heap manager uses a granularity of 8 bytes. This means that all allocations are an even multiple of 8 bytes, and that allocation sizes can be converted to units by round up and dividing by 8.

For example, if a process requests an allocation of 32 bytes, the number of units is $32 / 8 = 4$. If the allocation request was 34 bytes, the allocation size is rounded upward to an even multiple of 8. In this example, the 34 bytes requested would be rounded to an allocation of 40 bytes, or 5 units. The process requesting the allocation is unaware of any rounding to satisfy unit granularity and proceeds as if the allocation request of 34 bytes was actually 34 bytes.

By using a unit size of 8, the types of allocation made by most applications can be recorded using one word value with the restriction that the maximum size of a heap block, in units, is the largest unsigned short or 0xFFFF. This makes the theoretical maximum size of a heap block in bytes, $0xFFFF * 8$, or 524,280 bytes. (This limitation is documented in the Win32 HeapAlloc API documentation.) Does that mean that a program cannot allocate a heap block greater than 512k? Well, yes and no. A heap block larger than 512k cannot be allocated, but there is nothing to prevent the Heap API from using VirtualAlloc to allocate a region of linear memory to satisfy the request. This is exactly what the heap manager does if the size of the

requested allocation exceeds the `heaps VMThreshold`. The value of `VMThreshold` is stored in the heap header and by default is 520,192 bytes (or `0xFE000` units). When the heap manager allocates a large heap block using `VirtualAlloc`, the resulting structure is referred to as a `Virtually Allocated Block (VAB)`.

The heap manager walks contiguous heap blocks by converting the current heap block's unit size into bytes and adding that to the heap block's base address. The address of the previous heap block is calculated in a similar manner, converting the unit size of the previous block to bytes and subtracting it from the heap block's base address. The heap manager walks contiguous heap blocks during coalescing free blocks, sub-allocating a smaller block from a larger free block, and when validating a heap or heap entry.

Unit sizes are important for free block list management as the array of 128 doubly-linked lists inside the heap header track free blocks by unit size. Free blocks that have a unit size in the range from 1 to 127 are stored in the free list at the corresponding array index. Thus, all free blocks of unit size 32 are stored in `Heap->FreeLists[32]`. Because it is not possible to have a heap block that is 0 units, the free list at array index zero stores all heap blocks that are larger than 127 units; these entries are sorted by size in ascending order. Because a majority of allocations made by a process are less than 128 units (1024 bytes or 1K), this is a fast way to find an exact or best fit block to satisfy an allocation. Blocks of 128 units or greater are allocated much less frequently, so the overhead of doing a linear search of one free list does not have a large impact on the overall performance of most applications.

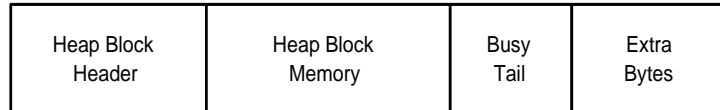
The `flags` field within the heap block header denotes special attributes of the block. One bit is used to mark a block as allocated versus free. Another is used if it is a VAB. Another is used to mark the last block within a committed region. The last block within a committed region is referred to as a sentinel block, and indicates that no more contiguous blocks follow. Using this flag is much faster than determining if a heap block address is valid by walking the heap segment's UCR chain. Another flag is used to mark a block for free or busy-tail checking. When a process is debugged, the heap manager marks the block in certain ways. Thus, when an allocated block is released or a free block is reallocated, the heap manager can determine if the heap block was overwritten in any way.

The extra info fields of the heap block header have different usage depending on whether the block is allocated or free. In an allocated block, the first field records the number of extra bytes that were allocated to satisfy granularity or alignment requirements. The second field is a pseudo-tag. Heap tags and pseudo tags are beyond the scope of this discussion.

For a free block, the extra info fields hold byte and bit-mask values that access a free-list-in-use bit-field maintained within the heap header. This bit-field provides quicker lookups when a small block needs to be allocated. Each bit within the bit-field represents one of the 127 small block free lists, and if the corresponding bit is set, that free list contains one or more free entries. A zero bit means that a free entry of that size is not available and a larger block will need to be sub-allocated from. The first extra info field holds the byte index into the bit-field array. The second extra info field holds the inverted mask of the bit position within the bit-field. Note that this applies to Windows NT 3.51 only. Newer versions of Windows NT still use the free list bit-field, but do not store the byte index or bit-mask values.

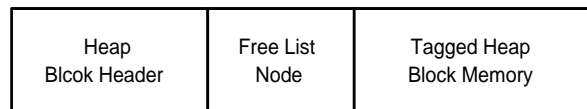
The heap block memory array is also different depending on the allocated state of the free block. For allocated blocks, this is the actual memory used by your application. For free blocks, the first two Dwords (1 unit) are used as next and previous pointers that link free blocks together in a doubly-linked list. If the process that allocated the heap block is being debugged, an allocated heap block also contains a busy-tail signature at the end of the block. Free blocks are marked with a special tag that can detect if a stray pointer writes into the heap memory area, or the process continues to use the block after it was deallocated.

The following diagram shows the basic architecture of an allocated heap block.



In the diagram, 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.



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 manager 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.

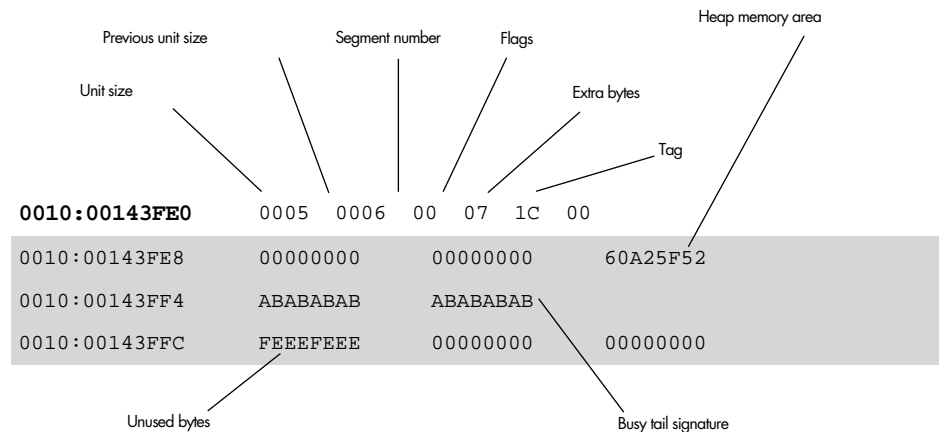
The following examples show how to use the SoftICE HEAP32 command to aid in monitoring and debugging Win32 heap issues.

The following 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 subsequent example.

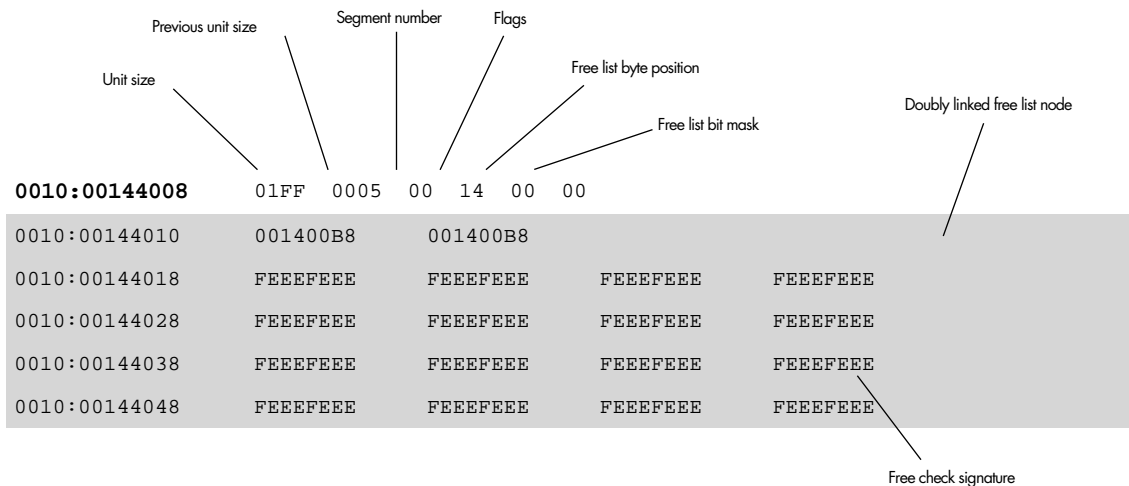
```
:HEAP32 -b 140000
Base          Type          Size      Seg#      Flags
00140000     HEAP             580       01
00140580     SEGMENT         38        01        TAGGED | BUSYTAIL
001405B8     ALLOC           40        01
. . .
00143FE0     ALLOC           28        01        TAGGED | BUSYTAIL
00144008     FREE            FF8       01        FREECHECK | SENTINEL
```

To examine the contents of an allocated heap block and a free block, the following 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.



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). 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.



12 Using BoundsChecker Driver Edition

| | |
|---|------------|
| Configuring BoundsChecker | 191 |
| Using the EVENT Command | 192 |
| Viewing Events in the Event Window | 193 |
| Viewing Events in the Command Window | 194 |
| Reviewing Event Data | 195 |
| Understanding Special Display Characteristics | 195 |
| Finding Events | 195 |
| Reading Summary Information | 196 |
| Reading Detail Information | 197 |
| Filtering Events | 198 |
| Filtering by Event Type | 199 |
| Filtering by Parameters | 199 |

What is BoundsChecker Driver Edition?

BoundsChecker Driver Edition helps you monitor interaction between your device driver and the operating system. It is available in DriverStudio. The core of BoundsChecker is a boot-load driver (BCHKD.SYS) that can monitor and log all calls to the operating system kernel, and detect errors. The driver loads before all other drivers, monitors and collects *events*, and writes them to a circular buffer.

You can view these events in SoftICE (with the system stopped) or in DriverWorkbench. SoftICE displays the event data collected by BoundsChecker with summary or detail information. The event filtering capabilities of the viewer allow you to review only the events that are important to you.

For information about the events BoundsChecker monitors, refer to *Appendix E: Events Monitored by BoundsChecker* on page 213.

Note: For more information about viewing events in DriverWorkbench, refer to the DriverWorkbench help system.

Configuring BoundsChecker

When you install DriverStudio, you configure BoundsChecker to collect data for specific drivers and events. Complete the following steps to modify those settings after you have installed DriverStudio.

- 1 Start DriverWorkbench.
- 2 On the **BoundsChecker** menu, select **Configure**.
- 3 Select a configuration to update.
If you choose Open an existing BoundsChecker configuration, DriverWorkbench changes the current configuration to the selected configuration when you complete the wizard.
- 4 Select the drivers you want BoundsChecker to monitor and click **Next**.
- 5 Select categories of events and specific events you want BoundsChecker to monitor and click **Next**.
- 6 Click **Finish** to complete the configuration process.

Note: If you enable event logging for a driver that is already loaded, you must restart your system for BoundsChecker to start the logging. However, you can enable logging for drivers that are not loaded without restarting.

7 Click **Finish** to complete the configuration process.

Note: If you enable event logging for a driver that is already loaded, you must restart your system for BoundsChecker to start the logging. However, you can enable logging for drivers that are not loaded without restarting.

Using the EVENT Command

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

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

| | |
|-----|---|
| -? | Displays descriptions of the supported command switches |
| -a | Turns API return display on or off. The default setting is on. When this setting is off, SoftICE does not display API return events. |
| -lx | Specifies the stack-checking level (0x40 - 0x4000). The default setting is 0x800. |
| -nd | Specifies the nesting depth used to display events. Legal values are 0 to 32 (decimal format). The default nesting level is 10. If events nest past the specified nesting depth, SoftICE does not display them as indented. |
| -o | Turns event logging on or off. The default setting is on. |
| -pd | Specifies the SoftICE pop-up level for BoundsChecker events. The default setting is 0. 0 - SoftICE does not pop up on BoundsChecker events 1 - SoftICE pops up on errors only 2 - SoftICE pops up on all errors and warnings |
| -r | Clears the event buffer |
| -s | Displays the current status of event viewing and logging. The number of logged events is the total that have been trapped since the system was started. It is displayed in decimal format. |

| | |
|--------------------------|--|
| -t | Turns display of thread switches on or off. The default setting is on. When this option is on and event n-1 is in a different thread than event n, SoftICE displays event n in reverse video indicating a thread switch has occurred. When this option is off, SoftICE does not display thread switches. |
| -x | Displays all events with their parameters, as well as general summary information for each event, including elapsed time, current thread and current IRQL. If you do not specify this switch, SoftICE displays a single summary line for each event. |
| <i>start-event-index</i> | Displays events starting at the specified event index |
| <i>Levent-count</i> | Displays the logged events in the Command window, starting from the specified <i>start-event-index</i> for a length of <i>event-count</i> events. If you do not specify a length, SoftICE displays the events in a scrollable window starting from <i>start-event-index</i> (if one is specified). |

Viewing Events in the Event Window

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

```
EVENT
```

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

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

The Event window supports the following keys.

| | |
|------------|--|
| Enter | Toggles the display state of the event at the current cursor position between summary information and detail information |
| Esc | Closes the Event window. When you re-open the Event window, SoftICE preserves the previous window state (i.e. current event, expansion state, and filters are the same). |
| PageUp | Scrolls the screen up one page |
| PageDown | Scrolls the screen down one page |
| Up Arrow | Moves cursor up one line. If on the top line, it scrolls the window up one line. |
| Down Arrow | Moves cursor down one line. If on bottom line, it scrolls window down one line. |

| | |
|-------------------|---|
| Shift-Left Arrow | Scrolls the window left one column |
| Shift-Right Arrow | Scrolls the window right one column |
| Home | Moves the cursor to the top row. If the cursor is already on the top row, starts display at the first event. |
| End | Moves the cursor to the bottom row. If the cursor is already on the bottom, starts display at the last event. |
| * | Undoes the last Home or End operation |
| F1 | Toggles the display state of all events between summary information and detail information |
| F2 | Displays the Event filtering dialog |
| F3 | Displays the Parameter filtering dialog |
| F4 | Displays error events only |
| F | Closes the Event window and returns focus to the Command window. Use this key if you want to use other SoftICE commands on data that is displayed in the Event window. If you bring up the Event window again, SoftICE preserves the previous window state (i.e. current top event, expansion state, and filters are the same). |
| R | Toggles the display state of API returns between showing all API returns and showing no API returns |
| T | Toggles the highlighting of thread switches. Thread switches are indicated by displaying the summary line of the first event in the new thread in reverse video. |
| E | Toggles the highlighting of errors on API returns. SoftICE displays the summary line of API return errors in bold |
| S | Displays the event at the current cursor position at the top of the Event window |
| N | Finds the next event that matches the search criteria selected with the right mouse button |
| P | Finds the previous event that matches the search criteria selected with the right mouse button |
| 0 - 7 | Filters events by CPU number on SMP machines. Each key acts as a toggle for displaying all events that occurred on a specific CPU. These keys also appear as buttons on the top line of the Event window. |

Viewing Events in the Command Window

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

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

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

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

Saving a SoftICE History File

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

Reviewing Event Data

The following sections describe features that can help you review event data more effectively.

Understanding Special Display Characteristics

In the Event window, SoftICE displays the following types of events with special characteristics.

- **Thread switches** - SoftICE displays thread switches in reverse video. For example, when event n-1 is in a different thread than event n, SoftICE displays event n in reverse video to indicate that a thread switch has occurred. You can use the -t switch to turn this display option on and off.
- **Error events** - SoftICE displays error events in bold.

Finding Events

When you review event data in the Event window, you can right-click and use Find Next and Find Prev to find the other events that include a match for the selected value. Right-click any string in the format *name:value*, where *name* can be any of the following types of values.

- Parameter name

- IRQL
- THREAD
- TargetEIP

After you select a string, you can press N to find the next event, and P to find the previous event.

Reading Summary Information

SoftICE displays every event in the event log with one line of summary information. There are two types of summary lines: API calls and API returns. The following example uses `ExAllocatePoolWithTag` to illustrate the two event types.

```
0001E   ExAllocatePoolWithTag ULONG NumberOfBytes:42
0001F   ExAllocatePoolWithTag returns pvoid:E116F568
```

A description of each field in this example follows.

| | |
|-------------------|--|
| Sequence Number | Hexadecimal sequence number of the event. Sequence numbers run from 0 to the number of events - 1. Sequence numbers become important when display filtering is used. |
| Event Name | Name of the API. (i.e. <code>ExAllocatePoolWithTag</code>) |
| Summary Parameter | <i>Present only for API calls.</i> Displays the most important parameter for the API. The parameter type is displayed followed by the parameter name, followed by the parameter value. For the example API, the summary parameter is the <code>ULONG NumberOfBytes</code> , and the value passed in is <code>0x42</code> . |
| Return Value | <i>Present only for API returns.</i> Displays the return type followed by the value returned from the API. In the example the value returned is of type <code>pvoid</code> and is the address of the allocated buffer <code>0xE116F568</code> . |

Reading Detail Information

In addition to the summary line, you can also display each event with detail information. The following example includes the detail information for the `ExAllocatePoolWithTag` example described in the summary information section.

```
0001E ExAllocatePoolWithTag ULONG NumberOfBytes:42
      Called From: 80218183 Fastfat!PAGE+4423
      IRQL:00 PASSIVE_LEVEL CPU:00 Thread:8071DB60 System(2)
      Elapsed time:(11.86 microseconds)
      POOL_TYPE PoolType:11
      ULONG NumberOfBytes:42
      ULONG Tag:20746146

0001F ExAllocatePoolWithTag returns pvoid:E116F568
      Called From: 80218183 Fastfat!PAGE+4423
      IRQL:00 PASSIVE_LEVEL CPU:00 Thread:8071DB60 System(2)
      Elapsed time:(7.98 microseconds)
```

For every event (both calls and returns), SoftICE displays the following generic information.

| | |
|-------------|---|
| Called From | Address the API returns to. The address is displayed in hexadecimal and symbolic format. If an actual symbol is not available, SoftICE displays the address as <code>Module!Section + offset</code> . |
| IRQL | Numeric IRQL at which the API was called. SoftICE displays the hexadecimal value and the description. (i.e. <code>PASSIVE</code> , <code>DISPATCH</code> etc.) |

| | |
|----------------|---|
| CPU | For SMP machines, this value is the number of the CPU that executed the API. For uni-processor machines, this value is always zero. |
| Thread | ID of the current thread. SoftICE displays the address of the thread control block, followed by the process name and process ID. |
| Elapsed time | Actual elapsed time since the last displayed event. The time is displayed as microseconds, milliseconds or seconds (whichever makes the most sense based on the value). |
| Parameter List | <p>List of parameters passed to the API. For each parameter, SoftICE displays the parameter type, parameter name, and parameter value. For calls, SoftICE displays all parameters. For returns, SoftICE displays only output parameters. For parameters that are pointers to structures, SoftICE displays the entire structure. SoftICE expands the following structures.</p> <ul style="list-style-type: none">• IRP• File Object• IO status block• Register frame for faults and interrupts.• Surface Object (SURFOBJ)• Point (POINTL)• Rectangle (RECTL)• Clipping object (CLPOBJ)• Brush object (BRUSHOBJ)• String Object (STROBJ)• Font Object (FONTOBJ)• USB request block (URB) <p>In addition to these structures, SoftICE also displays all string parameters. This applies to ASCII, ANSI and UNICODE strings.</p> |

Filtering Events

When you filter events, BoundChecker still collects information for the events that are not displayed. Filtering changes only what SoftICE displays. You must reconfigure BoundsChecker in DriverWorkbench to change the events monitored by BoundsChecker.

You can filter events in the Event window by event type or by parameters. You can set combinations of event and parameter filters. For example, you can view all FastIO APIs that operate on a specific FileObject. You can select a *name:value* and right-click to quickly add a new parameter filter. SoftICE adds the filter to the current filter list and immediately enables it.

Filtering by Event Type

When you are viewing event data in the Event window, you can press F2 or click Events at the top of the window to change the types of events SoftICE displays. Complete the following steps to filter by events.

- 1 Press **F2** or click **Events** at the top of the Event window to open the Events dialog.
- 2 Select or deselect a category of events from the Category list on the left side of the dialog. You can select or deselect a category by clicking the + or - at the beginning of the line, or using the Insert key.
- 3 Select or deselect an event from the Event list on the right side of the dialog. All events are selected or deselected by default, but you can change them individually. You can select or deselect an event by clicking the + or - at the beginning of the line, or using the Insert key.
- 4 Press **Enter** or click **OK** to close the Events dialog and apply your changes. Press **Esc** or click **Cancel** to close the Events dialog without saving your changes.

Filtering by Parameters

To filter the Event window by parameter values, you can use any of the following methods to open the Parameters dialog.

- Press **F3**
- Click **Parameters** at the top of the window
- Right-click and select **Add Filter**
To use the Add Filter menu item, select any string in the format *name:value* where *name* can be any of the following types of values.
 - ◇ Parameter name
 - ◇ IRQL
 - ◇ THREAD
 - ◇ TargetEIP

The following sections describe how to use the Parameters dialog to manage parameter filters.

To Add a Filter

- 1 Press **F3** or click **Parameters** at the top of the Event window.
- 2 Select the type of filter you want to add from the Available Filters list.
- 3 Press **Insert**.

- 4 Enter the parameter value in the Value field.
- 5 Press **Enter**.
- 6 Press **Enter** again to enable all filters listed. Press **Esc** to disable the listed filters.

To Delete a Filter

- 1 Press **F3** or click **Parameters** at the top of the Event window.
- 2 Select the filter you want to delete in the Current Filters list.
- 3 Press **Delete**.
- 4 Press **Enter** to enable all filters listed. Press **Esc** to disable the listed filters.

To Disable Filters

- 1 Press **F3** or click **Parameters** at the top of the Event window.
- 2 Press **Esc** to disable all filters listed in the Parameters dialog.

To Enable Filters

- 1 Press **F3** or click **Parameters** at the top of the Event window.
- 2 Press **Enter** to enable all filters listed in the Parameters dialog.

When you enable a parameter filter, SoftICE displays only the events that match that filter. If you define multiple parameter filters, SoftICE displays only events that match all selected filter criteria.

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 *Operators* on page 111 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.

It is the common wonder of all men, how among so many millions of faces, there should be none alike.

◇ Sir Thomas Browne

B Supported Display Adapters

The following table lists the display adaptors SoftICE supported when the product most recently shipped. However, NuMega regularly adds new display adaptor support to enhance SoftICE. You can download the latest support files from the NuMega FTP or BBS sites. Refer to *Installing SoftICE* in *Getting Started 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 |

Supported Display Adaptors

| | | |
|----------------------------------|-----------------------------|-----------------------------------|
| DEC PC76H-EA | DEC PC76H-EB | DEC PC76H-EC |
| DEC PCXAG-AJ | DEC PCXAG-AK | DEC PCXAG-AN |
| 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 |
| 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 |

Supported Display Adaptors

| | | |
|------------------------------------|-------------------|------------------------|
| 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 | |

C Troubleshooting SoftICE

If you encounter the following problems, try the corresponding solutions. If you encounter further difficulties, contact the NuMega Technical Support Center.

| 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 <i>Appendix B: Supported Display Adapters</i> on page 205. |
| 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 <i>Setting Troubleshooting Options</i> on page 151. |
| The PC crashes when you run SoftICE for Windows 95 and Windows 98. | 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, COMNNECT, 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 <i>Setting Troubleshooting Options</i> on page 151. |
| Windows 95 and Windows 98 crash 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 95 and Windows 98 initializes. When Windows 95 and Windows 98 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. |

There must be a beginning of any great matter, but the continuing unto the end until it be thoroughly finished yields the true glory.

◇ Sir Francis Drake

D Kernel Debugger Extensions

SoftICE for NT now supports Kernel Debugger 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 Kernel Debugger Extension for use with SoftICE;

- 1 Use the `KD2SYS` or `KD2SYSXLAT` program to convert the DLL to a system driver. This program:
 - ◇ Copies the DLL to the `\SYSTEMROOT\SYSTEM32\DRIVERS` directory and gives it an extension of `.SYS`
 - ◇ Modifies the file to tell the system that the file can be loaded as a system driver and redirect many API calls to SoftICE
 - ◇ 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.

Requirements for Using Kernel Debugger Extensions

- 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 KDExtension. 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 KDExtension. 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.

E Events Monitored by BoundsChecker

BoundsChecker Driver Edition can monitor and log the following types of events.

- Windows NT kernel-mode API calls
- Standard driver routines and callbacks
- System-wide events
- Errors

The following sections describe the events monitored by BoundsChecker in more detail.

Windows NT Kernel-mode API Calls

BoundsChecker can monitor the following core Windows NT subsystem API calls.

| | |
|----------------------------|-----|
| Executive Support | Ex |
| Hardware Abstraction Layer | Hal |
| I/O Manager | Io |
| Kernel | Ke |
| Memory Manager | Mm |
| Native | Nt |
| Object Manager | Ob |
| Process Structure | Ps |
| Run-time Library | Rtl |
| Security Reference Monitor | Se |
| Zw | Zw |

Specialized Windows NT subsystems also export APIs that are used by higher-level drivers. BoundsChecker monitors APIs from the following types of drivers.

- Network Drivers
NDIS
- Graphics Drivers
GDI
- SCSI Drivers
SCSI PortSCSI Class
CLASSPNP
- WDM Drivers
USB

BoundsChecker monitors all APIs independently for each driver. If you do not enable logging for an API that registers a standard driver routine or callback, BoundsChecker does not monitor that standard driver routine or callback. For example, if you do not enable monitoring for IoConnectInterrupt, BoundsChecker does not monitor any PKSERVICE_ROUTINE callbacks.

Standard Driver Routines and Callbacks

Though Windows NT is composed of many different subsystems and drivers tailored to specific needs, most drivers are very similar in structure and design. A driver contains a combination of standard driver routines, fast I/O and subsystem-specific callbacks.

The following sections include lists of the specific routines and callbacks monitored by BoundsChecker.

Standard Driver Routines

- BoundsChecker monitors the following standard driver routines.
- DriverEntry
- Dispatch (IRP_MJ_XXX)
- StartIo
- Reinitialize
- InterruptServiceRoutine
- DPCRoutine
- SynchCritSection
- ControlRoutine
- CancelRoutine

- IoCompletionRoutine
- TimerRoutine
- UnloadRoutine
- WorkerThreadRoutine
- StartRoutine
- AddDevice

Subsystem-specific Callbacks

All network, graphics, SCSI and WDM drivers contain APIs that register callbacks. BoundsChecker monitors the following groups of callbacks for these specialized subsystems.

- Network Drivers
 - MiniportXxx functions
 - ProtocolXxx functions
- Graphics Drivers
 - DDI
 - DDRAW
- SCSI Drivers
 - SCSI Miniport routines
 - SCSI Driver routines
- WDM
 - USB D IOCTL functions

BoundsChecker monitors all events independently for each driver.

Fast I/O Routines

BoundsChecker monitors fast I/O routines.

DriverWorks Routines

BoundsChecker logs events that are reported by DriverWorks routines.

System-wide Events

Unlike other types of events, BoundsChecker monitors system-wide events globally, not independently for each driver. BoundsChecker monitors the following system-wide events.

- DebugPrint
- Error events

- Windows NT system calls
- Divide-by-zero exception
- Double faults
- Page faults

DebugPrint

When BoundsChecker is enabled for DebugPrint, all user and kernel-mode DebugPrints are added to the BoundsChecker event stream.

Error Events

Error events detected and identified by BoundsChecker.

Windows NT System Calls

Windows NT uses Int2E to perform native system services. BoundsChecker saves a register frame and flags for this event. If the Int2E occurs in user mode, BoundsChecker also saves SS:ESP.

Divide-by-zero Exception

BoundsChecker saves a register frame and flags for all user and kernel-mode exceptions of this type. If the exception occurred in user mode, BoundsChecker also saves SS:ESP.

Double Faults

This type of fault can occur when a driver uses its entire kernel-mode stack. The following error message is displayed at the blue screen.

```
**STOP 0x0000007F (0x00000008, 0x00000000, 0x00000000, 0x00000000)
UNEXPECTED_KERNEL_MODE_TRAP
```

For double faults, BoundsChecker saves a register frame and flags to the event stream. In the Event window, the called from address is the address where the double fault occurred.

Page Faults (Int0E)

If enabled, BoundsChecker can record all kernel and user mode page faults. BoundsChecker can log faults that are fatal or non-fatal. When a page fault occurs, BoundsChecker saves a register frame and flags to the event stream.

Errors

In addition to logging APIs, BoundsChecker also actively checks for certain types of common driver errors. When BoundsChecker detects an error condition, SoftICE can be configured to pop up in the code that caused the error. BoundsChecker provides a description of the error, which SoftICE displays. BoundsChecker also writes an error event to the event log.

Note: The `-p` switch of the EVENT command controls the SoftICE error pop-up level.

When SoftICE first pops up on a BoundsChecker error, SoftICE displays the following prompt on the help line.

Press 'A' to acknowledge error, 'S' to suppress error

When you acknowledge the error, BoundsChecker ignores it and continues. If it occurs in the future, SoftICE continues to pop up. When you suppress the error, BoundsChecker adds it to a suppression list and SoftICE no longer pops up for that specific error. The suppression is not global for the API; it is tied to the address that the API was called from.

BoundsChecker checks for the following classes of errors.

- API return errors
- API parameter errors
- IRQL violations
- Stack threshold

API Return Errors

Drivers generate this type of error when an API returns a NTSTATUS error code or a Boolean error code. The following is an example of an API return error message.

```
Break due to Error at FDB05461, STATUS_OBJECT_NAME_NOT_FOUND is an
invalid return value for NtCreateFile
```

Common status codes that are considered warnings include STATUS_BUFFER_TOO_SMALL, STATUS_OBJECT_NAME_NOT_FOUND, and STATUS_NO_MORE. You can use the `-pd` switch with the EVENT command to set the SoftICE pop up level for these errors.

API Parameter Errors

Drivers generate this type of error when a bad parameter value is passed to an API. A common example of this would be a null pointer passed to a function that requires a valid pointer. The following is an example of an API parameter error message.

```
Break due to Error at F15D46D0, PKDEFERRED_ROUTINE DeferredRoutine:0 is
an invalid parameter for KeInitializeDpc.
```

Common API parameter errors are null pointers and incorrect object types (for example, DEVICE_OBJECT, IRP, PKDPC).

IRQL Violations

Drivers generate this type of error when an API is called at an improper IRQL level. The following is an example of an IRQL violation.

```
Break due to IRQL violation at FDB26ECC, current IRQL at DISPATCH_LEVEL,  
KeInitializeDpc must be running at IRQL PASSIVE_LEVEL
```

Stack Threshold Errors

The default stack threshold is 2K. If a driver uses more than the remaining stack space (total stack space minus the threshold amount), BoundsChecker generates an error event. You can use the `-lx` switch with the `EVENT` command to set the stack threshold in SoftICE.

SoftICE pops up only once per thread for stack threshold errors, and cannot be suppressed.

Some atypical drivers switch to their own stack. For these drivers, BoundsChecker ignores errors generated when the current stack pointer is more than 4K from the top or bottom of the stack.

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

SYMBOLS

+ (plus sign) 71, 73

. (dot) command 70

A

A command 70

ADDR command 163, 166

Address

space 177

type 119

Alt-C 66

Alt-D 76

ALTKEY command 54

Alt-L 14, 70

Alt-R 74

Alt-W 72

ANSWER command 133

modem connection 135

ANSWER initialization string 146

Applications

building 30

debugging 27

Arrays

collapsing 14

expanding 14

Assigning expressions 78

B

BC command 22, 108

BD command 22, 108

BE command 108

BH command 108

Bitwise operators 112

BL command 16, 21, 22, 108

BMSG command 91, 96

Borland compiler 30

BoundsChecker Driver Edition 191

BPCOUNT function 100

BPE command 21, 108

BPINDEX expression function 103

BPINT command 91, 94

BPIO command 91, 95

BPLOG expression function 102

BPM command 91, 93

BPMD command 22

BPMISS expression function 101

BPT command 108

BPTOTAL expression function 102

BPX

breakpoint 19

command 15, 70, 91, 92

Breakpoint action 92

setting 98

Breakpoint index 107, 108

Breakpoints

BPCOUNT function 100

BPINDEX 103

BPLOG function 102

BPMISS function 101

BPTOTAL function 102

BPX 19

clearing 22

conditional 19, 98

conditional expression 92

context 97

criteria to trigger 97

disabling 22

duplicate 106

elapsed time 106

embedded 108

execution 91, 92

expressions 107

I/O 91, 95

INT 1 and INT 3 108

interrupt 91, 94

manipulating 108

memory 21, 91, 93

one-shot 15

point-and-shoot 15

statistics 107

sticky 15, 91

types 91

using 89

virtual 97

window message 91, 96

BSTAT command 102, 103, 107

Building

applications 30

debug information 10

Built-in functions 116

C

Character constants 114

Checked build 155

CLASS command 17

- Clearing
 - breakpoints 22
- Closing
 - Code window 66
 - Data window 76
 - FPU Stack window 80
 - Locals window 70
 - Register window 74
 - SoftICE windows 56
 - Watch window 72
- Code mode 68
- Code window 12, 55, 66
 - closing 66
 - disassembled instruction 68
 - entering commands 69
 - JUMP 69
 - modes 68
 - moving the cursor to 57, 66
 - NO JUMP 69
 - opening 66
 - resizing 66
 - scrolling 67
 - strings 69
- Collapsing
 - arrays 14
 - stacks 71
 - strings 14
 - structures 14
 - typed expressions 73
- Command history
 - recalling 63
- Command line arguments
 - passing 34
- Command window 55, 60
 - associated commands 66
 - history buffer 65
 - scrolling 60
- Commands
 - . (dot) 70
 - A 70
 - ALTKEY 54
 - ANSWER 133
 - BC 22, 108
 - BD 22, 108
 - BE 108
 - BH 108
 - BL 16, 21, 22, 108
 - BMSG 91, 96
 - BPE 21, 108
 - BPINT 91, 94
 - BPIO 91, 95
 - BPM 91
 - BPMD 22
 - BPX 15, 70, 91, 92
 - BSTAT 107
 - CLASS 17
 - CR 76
 - D 76, 78, 79
 - DATA 76
 - DEX 78, 79
 - DIAL 133
 - E 79
 - editing 63
 - entering 58, 60
 - FILE 12, 70
 - FORMAT 76, 78
 - G 15, 21, 74, 76
 - H 17, 59
 - HERE 15, 70, 92
 - HWND 20, 96
 - IDT 94
 - informational 17
 - LINES 56
 - LOADER32 39, 40
 - LOCALS 71
 - MACRO 64
 - P 14, 74, 76, 151
 - recalling 63
 - S 79
 - SET 61, 66, 70
 - SRC 14, 70
 - SS 70
 - SYM 18
 - syntax 61
 - T 76
 - TABLE 18
 - TABS 70
 - TYPES 71
 - U 13, 15, 70
 - WATCH 73
 - WC 66
 - WD 76
 - WF 80
 - WL 70
 - WR 74
 - WS 79
 - WW 72
 - WX 80
 - X 21
- Commands T 74
- commands, Universal Video Driver 53
- Compiler options
 - 32-bit 30
- Compilers
 - Borland 30
 - Delphi 30
 - MASM 31
 - Microsoft Visual C++ 31
 - Symantec C++ 31
 - Watcom C++ 31
- Conditional breakpoints 98
 - count functions 100
 - performance 106
 - setting 19
- Conditional expression
 - breakpoints 92
- Controlling SoftICE windows 56
- Controlling the SoftICE screen 12
- Copying data 58
- Count functions
 - conditional expressions 100

- CPU flags 75
- CR command 76
- Creating
 - Persistent Macros 148
- CSRSS 170
- Ctrl-D 54
- Cursor
 - moving among windows 57
- Customizing SoftICE 137
- Cycling Data windows 76
- D**
- D command 76, 78, 79
- Data
 - copying 58
 - pasting 58
- DATA command 76
- Data window 55, 76
 - assigning expressions 78
 - associated commands 79
 - closing 76
 - cycling through 76
 - fields 77
 - format 76
 - moving the cursor to 57, 76
 - opening 76
 - resizing 76
 - scrolling 77
 - viewing addresses 76
- DBG files 129, 156
- Debug information
 - building 10
- Debugging
 - applications 27
 - device drivers 28
 - features 3
 - generating information 30
 - preparing to 125
 - resources 155
- Deleting
 - symbol tables 38
 - watch 73
- Delphi compiler 30
- DEVICE command 156
- Device drivers
 - debugging 28
- DEX command 78, 79
- DIAL command 133
 - modem connection 135
- DIAL initialization string 146
- Disable mapping of non-present pages 151
- Disable mouse support 151
- Disable Num Lock and Caps Lock programming 151
- Disable Pentium support 151
- Disable thread-specific stepping 151
- Disabling
 - breakpoints 22
 - SoftICE 54
- Disassembled instruction
 - Code window 68
- Display adapters
 - supported 205
- Display command 58
- Display diagnostic messages 141
- Displaying registers 80
- DLL exports 127
 - loading 32-bit 128
- Do not patch keyboard driver 151
- DRIVER command 156
- DriverStudio
 - BoundsChecker Driver Edition 191
- Duplicate breakpoints 106
- E**
- E command 79
- Eaddr function 117
- EBP register 105
- Editing
 - commands 63
 - flags 75
 - memory 78
 - registers 75
- Effective address 74
- Embedded breakpoints 108
- Entering commands 58, 60
 - syntax 61
- Entry points 127
 - unnamed 127
- Error messages 201, 211
- ESP register 105
- Evaluate function 118
- Execution breakpoints 91, 92
- Expanding
 - arrays 14
 - stacks 71
 - strings 14
 - structures 14
 - typed expressions 73
- Export Information 143
- Export names
 - expressions 128
- Exports 139
 - DLL 127, 128
- Expression evaluator 111
 - built-in functions 116
 - character constants 114
 - expression values 118
 - forming expressions 114
 - indirection operators 120
 - numbers 114
 - operands 121

- operators 111
- registers 115
- symbols 115
- Expression types 118
- Expression values
 - address-type 118
 - literal-type 118
 - register-type 118
 - symbol-type 118
- Expressions 111
 - assigning 78
 - breakpoints 107
 - export names 128
 - forming 114
 - watching 73
- F**
- Fault trapping 83
- Faults
 - trapping 83
- Fields
 - Data window 77
- FILE command 12, 70
- Flags 74
 - editing 75
- FORMAT command 76, 78
- Formatting
 - Data window 76
- Forming expressions 114
- FPU Stack window 55, 80
 - closing 80
 - displaying registers 80
 - moving the cursor to 57
 - opening 80
- Function keys 62, 147
 - modifying 147
- Functions
 - built-in 116
 - expression evaluator 116
- G**
- G command 15, 21, 74, 76
- GDI objects 172
- GDIDEMO application 9, 10
- GDT command 161
- General settings 139
 - modifying 139
- Global Descriptor Table 159, 161
- H**
- H command 17, 59
- Handle values 174
- Heap
 - API 178
 - architecture 178
 - blocks 182
- HEAP32 command 171, 181
- Help
 - for SoftICE xiii, 59
 - for Symbol Loader xiii
- Help line 12, 55, 59
- HERE command 15, 70, 92
- History buffer 65
- History buffer size 140
- HWND command 20, 96
- I**
- I/O breakpoints 91, 95
- IDT command 94, 160
- Indirection operators 111, 120
- Information
 - Help line 59
- Informational commands 17
- Initialization file 139
- Initialization settings
 - Remote Debugging 139
- Initialization string 139
- Initialization strings
 - modem 146
- INT 1 instruction
 - breakpoints 108
- INT 3 instruction
 - breakpoints 108
- Intel architecture 159
- Internet parameters
 - Remote Debugging 139
- Interrupt
 - breakpoints 91, 94
 - Descriptor Table 159, 160
- J**
- JUMP string 69
- K**
- Kernel
 - Windows NT 158
- Keyboard Mappings 139
 - modifying 147
- L**
- LDT command 162
- LINES command 56
- LOADER32 39, 40
- LOADER32.EXE 38
- Loading
 - 32-bit DLL exports 128
 - GDIDEMO 11
 - modules 31
 - SoftICE 9, 28
 - source 11, 31
 - symbols 18
- Local data
 - viewing 14
- Local Descriptor Table 159, 162

- LOCALS command 71
- Locals window 14, 55, 70
 - associated commands 71
 - closing 70
 - moving the cursor to 57, 70
 - opening 70
 - resizing 70
 - scrolling 71
- Logical operators 112
- Lowercase disassembly 141
- M**
- MACRO command 64, 176
- Macro Definitions 139
- Macro limit 150
- Macros
 - definitions 148
 - recursion 64, 149
 - Run-time 64
- Manipulating breakpoints 108
- MAP32 command 163, 177
- MASM compiler 31
- Math operators 111
- Memory
 - breakpoints 21, 91, 93
 - editing 78
 - map of system memory 163
- Messages
 - error 201, 211
- Microsoft Visual C++ compiler 31
- Mixed mode 68
- MMX registers 80
- MOD command 156, 177
- Modem 133
 - ANSWER command 135
 - connection 133
 - DIAL command 135
 - hardware requirements 133
 - initialization strings 146
 - SERIAL.EXE 134
- Modes
 - Code 68
 - Code window 68
 - Mixed 68
 - Source 68
- Modifying
 - function keys 147
 - General settings 139
 - Keyboard Mappings 147
 - SoftICE Initialization settings 139
- Modules
 - loading 31
 - translating 31
- Mouse commands
 - Display 58
 - Previous 58
 - Un-Assemble 58
 - What 58
- Moving the cursor 57
- Moving the SoftICE Window 56
- N**
- Navigating
 - SoftICE 51, 81
- Nesting limit 64
- NET ALLOW 144
- NET HELP 145
- NET PING 145
- NET RESET 145
- NET START 144
- NET STATUS 145
- NET STOP 145
- NMAKE command 10
- NMS file 32
- NMSYM.EXE 40
- NO JUMP string 69
- NonPaged System area 168
- NTCALL command 160
- NTOSKRNL.EXE 159
- O**
- OBJDIR command 156
- OBJTAB command 163, 175
- One-shot breakpoints 15
- Opening
 - Code window 66
 - Data window 76
 - FPU Stack window 80
 - Locals window 70
 - Register window 74
 - SoftICE windows 56
 - Watch window 72
- Operand sizes 121
- Operators
 - bitwise 112
 - expression evaluator 111
 - indirection 111, 120
 - logical 112
 - math 111
 - precedence 113
 - special 112
- P**
- P command 14, 16, 74, 76, 151
- Packaging source files 36
- PAGE command 165
- Page Table Entry 167
- Paged Pool System area 168
- Passing command line arguments 34
- Pasting data 58
- Persistent Macros 148
- PHYS command 165
- Point-and-shoot breakpoints 15

- Precedence operators 113
- Pre-loading
 - source 141
 - symbols 141
- Preparing to debug 125
- Previous command 58
- Process address space 177
- Processor Control Region 169
- ProtoPTEs 167
- PTE 167
- Q**
- QUERY command 171, 177, 178
- R**
- Recalling
 - command history 63
- Register window 55, 74
 - associated commands 76
 - closing 74
 - CPU flags 75
 - moving the cursor to 57, 74
 - opening 74
- Registers 74, 115
 - editing 75
- Remote Debugging 139, 146
- Remote Debugging, NET commands 144
- Remote Debugging, start session 145
- Requirements, Remote Debugging 143
- Reserving
 - symbol memory 142
- Resizing
 - Code window 66
 - Data window 76
 - Locals window 70
- SoftICE screen 56
- SoftICE windows 57
- Watch window 72
- Run-time macros 64
- S**
- S command 79
- Scrolling
 - Code window 67
 - Command window 60
 - Data window 77
 - Locals window 71
 - Watch window 72
 - windows 57
- Serial
 - connection 146
- SERIAL.EXE 133
 - modem 134
- SET command 61, 66, 70
- Setting
 - breakpoint actions 98
 - breakpoints 15
 - conditional breakpoints 19, 98
 - execution breakpoints 92
 - I/O breakpoints 95
 - interrupt breakpoints 94
 - memory breakpoints 21, 93
 - point-and-shoot breakpoints 15
 - source file search path 34
 - window message breakpoints 96
- Setting Video Memory size 54
- SoftICE
 - customizing 137
 - disabling 54
 - features 3
 - implementation 4
 - informational commands 17
 - initialization file 139
 - loading 9, 28
 - modem connection 133
 - navigating through 51, 81
 - overview 1
 - product overview xi, 3
 - user interface 5, 55
- SoftICE Initialization settings
 - Exports 139
 - General 139
 - Keyboard Mappings 139
 - Macro Definitions 139
 - modifying 139
 - Symbols 139
 - Troubleshooting 139
- SoftICE screen 55
 - controlling 12
 - resizing 56
- SoftICE Tutorial 7
- SoftICE windows
 - closing 56
 - Code 55, 66
 - Command 55, 60
 - controlling 56
 - Data 55, 76
 - FPU Stack 55, 80
 - Locals 55
 - opening 56
 - Register 55, 74
 - resizing 57
 - Watch 55, 72
- Sorting symbol tables 38
- Source
 - loading 11, 31
 - mode 68
 - packaging 36
 - pre-loading 141
 - specifying 37
 - stepping 13
 - tracing 13
 - translating 31
- Special operators 112
- Specifying Source Files 37

- SRC
 - command 14, 68, 70
 - file 37
 - SS command 70
 - Stack frame 14, 105
 - Stacks
 - collapsing 71
 - expanding 71
 - Stepping
 - source code 13
 - Sticky breakpoints 15, 91
 - Strings
 - Code window 69
 - collapsing 14
 - expanding 14
 - Structures
 - collapsing 14
 - expanding 14
 - SYM command 18, 168
 - Symantec C++ compiler 31
 - Symbol buffer size 142
 - Symbol Loader 5, 11, 31, 139
 - command line interface 38
 - command-line utility 40
 - Symbol tables
 - deleting 38
 - sorting 38
 - Symbols 115, 139
 - pre-loading 141
 - reserving memory 142
 - tables 18
 - type 119
 - System
 - Code area 163
 - memory map 163
 - Tables System area 163
 - View System area 163
 - System Page Table Entries 167
- T**
- T command 74, 76
 - TABLE command 18
 - Tables 18
 - TABS command 70
 - Tail recursion 64
 - Task State Segment 159, 161
 - Telephone number 146
 - THREAD command 178
 - Time stamp counter 106
 - Total RAM 140
 - Trace buffer size 140
 - Tracing
 - source code 13
 - Translating
 - modules 31
 - source 31
 - Trap NMI 141
 - Triggering
 - breakpoints 97
 - Troubleshooting 139
 - error messages 201, 211
 - SoftICE 209
 - Troubleshooting Options 151
 - TSS command 161
 - Typed expressions
 - collapsing 73
 - expanding 73
 - TYPES command 71
- U**
- U command 13, 15, 70
 - Un-Assemble command 58
 - Universal Video Driver 53
 - USER
 - object creation 176
 - Object Table 175
 - objects 172
 - User-defined
 - commands 148
 - settings 139
- V**
- Viewing
 - addresses 76
 - local data 14
 - Virtual breakpoints 97
- W**
- Watch
 - deleting 73
 - WATCH command 73
 - Watch window 55, 72
 - associated commands 74
 - closing 72
 - fields 73
 - moving the cursor to 57, 72
 - opening 72
 - resizing 72
 - scrolling 72
 - Watching
 - expressions 73
 - Watcom C++ compiler 31
 - WC command 66
 - WD command 76
 - WF command 80
 - WHAT command 175
 - What command 58
 - Win32 subsystem 170
 - Window message breakpoints 91, 96
 - Windows
 - Code 12, 55, 66
 - Command 55
 - components 170

Data 55, 76

FPU Stack 55, 80

Locals 14, 55, 70

moving the cursor among 57

Register 55, 74

scrolling 57

Watch 55, 72

Windows NT

DDK 156

exploring 153

kernel 158

references 157

system memory map 163

WL command 70

WR command 74

WS command 79

WW command 72

WX command 80

X

X command 21