Reference Guide

NuMega SoftICE

Windows 3.x

NuMega™
Technologies
Making Software Work™

July 1993

Software License Agreement

Please Read This License Carefully.

You are purchasing a license to use NuMega Technologies, Inc. software. The software is owned by and remains the property of NuMega Technologies, Inc., is protected by international copyrights, and is transferred to the original purchaser and any subsequent owner of the software media for their use only on the license terms set forth below. 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 use you are dissatisfied with the software, return the software, manuals and any partial or whole copies within thirty days of purchase to the party from who you received it for a refund, subject to our restocking fee.

Use Of The Software: NuMega Technologies, Inc. ("NuMega"), grants the original purchaser ("Licensee") the limited rights to possess and use the NuMega Technologies, Inc. Software and User Manual ("Software") for its intended purposes. Licensee agrees that the Software will be used solely for Licensee's internal purposes, and that at any one time, the Software will be installed on a single computer only. If the Software is installed on a networked system, or on a computer connected to a files server or other system that physically allows shared access to the Software, Licensee agrees to provide technical or procedural methods to prevent use of the Software by more than one user.

One machine-readable copy of the Software may be made for BACK UP PURPOSES ONLY, and the copy shall display all proprietary notices, and be labeled externally to show that the back-up copy is the property of NuMega, and that use is subject to this License. Documentation may not be copied in whole or part.

Use of the Software by any department, agency or other entity of the U.S. Federal Government is limited by the terms of the following "Rider for Governmental Entity Users."

Licensee may transfer its rights under this License, PROVIDED that the party to whom such rights are transferred agrees to the terms and conditions of this Licensee, and written notice is provided to NuMega. Upon such transfer, Licensee must transfer or destroy all copies of the Software.

Except as expressly provided in this License, Licensee may not modify, reverse engineer, decompile, disassemble, distribute, sub-license, sell, rent, lease, give or in any way transfer, by any means or in any medium, including telecommunications, the Software. Licensee will use its best efforts and take all reasonable steps to protect the Software from unauthorized use, copying or dissemination, and will maintain all proprietary notices intact.

Limited Warranty And Indemnification: NuMega warrants the Software media to be free of defects in workmanship for a period of ninety days from purchase. During this period, NuMega will replace at no cost any such media returned to NuMega, postage prepaid. This service is NuMega's sole liability under this warranty.

NuMega agrees to indemnify and hold Licensee harmless from all loss, claim or damage to Licensee arising out of any claim, legal action, or suit alleging that the Software infringes a United States patent, copyright, or trade secret. NuMega will defend, at its own expense, any such claim or action against Licensee, and will pay all reasonable costs, expenses, and damages incurred by Licensee in connection therewith, including reasonable attorneys' fees and expenses incurred by Licensee, on condition that NuMega shall be immediately notified in writing by Licensee of any notice of such claim or action or pending claim or action known to Licensee prior to the time when the failure to deliver such notice has injured NuMega; and that NuMega shall have control of the defense against any such claim or action and all negotiations toward the settlement or compromise thereof. In such event, Licensee shall have the right to participate in any such action at NuMega's cost. If the Software becomes the subject of any claim, suit, or proceeding for infringement of any United States patent, copyright or any other right of a third party, or in the event of any adjudication that the Software infringes upon any United States patent, copyright or any other third party, or if the use or license of such Software is enjoined, in addition to other liability which may arise under this provision, NuMega shall at its sole expense and discretion either (a) obtain a license or any rights necessary to make the warranty contained herein true and correct, or (b) refund to Licensee any amounts paid to NuMega for the Software. Specifically excluded from the above covenant are any claims or actions based upon, or portions of claims or actions based upon, those portions of the Software modified or developed by Licensee or third parties.

Disclaimer: LICENSE FEES FOR THE SOFTWARE DO NOT INCLUDE ANY CONSIDERATION FOR ASSUMPTION OF RISK BY NUMEGA, AND NUMEGA DISCLAIMS ANY AND ALL LIABILITY FOR INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR OPERATION OR INABILITY TO USE THE SOFTWARE, EVEN IF ANY OF THESE PARTIES HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. FURTHERMORE, LICENSEE INDEMNIFIES AND AGREES TO HOLD NUMEGA HARMLESS FROM SUCH CLAIMS. THE ENTIRE RISK AS TO THE RESULTS AND PERFORMANCE OF THE SOFTWARE IS ASSUMED BY THE LICENSEE. THE WARRANTIES EXPRESSED IN THIS LICENSE ARE THE ONLY WARRANTIES MADE BY NUMEGA AND ARE IN LIEU OF ALL OTHER WARRANTIES EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO IMPLIED WARRANTIES OF MERCHANTABILITY AND OF FITNESS FOR A PARTICULAR PURPOSE.

THIS WARRANTY GIVES YOU SPECIFIED LEGAL RIGHTS, AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM JURISDICTION TO

JURISDICTION. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF WARRANTIES, SO THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

Term: This License is effective as of the time Licensee receives the Software, and shall continue in effect until Licensee ceases all use of the Software and returns or destroys all copies thereof, or until automatically terminated upon the failure of Licensee to comply with any of the terms of this License.

General: This License is the complete and exclusive statement of the parties' agreement. Should any provision of this License 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 License shall nonetheless remain in full force and effect. This License shall be controlled by the laws of the State of New Hampshire, and the United States of America.

Rider For U.S. Government Entity Users

This is a Rider to the above Software License Agreement, ("License"), and shall take precedence over the License where a conflict occurs.

1.The Software was: developed at private expense; no portion was developed with government funds; is a trade secret of NuMega and its licensor for all purposes of the Freedom of Information Act; is "commercial computer software" subject to limited utilization as provided in any contract between the vendor and the government entity; and in all respects is proprietary data belonging solely to NuMega and its licensor.

2.For units of the DOD, the Software is sold only with "Restricted Rights" as that term is defined in the DOD Supplement to DFAR 252.227-7013 (b)(3)(ii), and use, duplication or disclosure is subject to restrictions set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at 252.227-7013. Manufacturer: NuMega Technologies, Inc. P.O. Box 7780, Nashua, New Hampshire 03060-7780 USA.

3.If the Software was acquired under a GSA Schedule, the Government has agreed to refrain from changing or removing any insignia or lettering from the Software or Documentation or from producing copies of manuals or disks (except for back up purposes) and; (1) Title to and ownership of the Software and Documentation and any reproductions thereof shall remain with NuMega and its licensor; (2) use of the Software shall be limited to the facility for which it is acquired; and (3) if the use of the Software is discontinued at the original installation and the Government wishes to use it at another location, it may do so by giving prior written notice to NuMega, specifying the new location site and class of computer.

4.Government personnel using the Software, other than under a DOD contract or GSA Schedule, are hereby on notice that use of the Software is subject to restrictions that are the same or similar to those specified above.

Contents

1 Introduction

Product Description

SoftICE/W is a powerful, low-level debugger that runs under Windows 3.0 and Windows 3.1 in enhanced mode. It can be used to debug anything running in the Windows environment, including Windows applications, Windows device drivers and Windows virtual device drivers. It can also debug any DOS software running in Windows virtual machines, including applications, drivers and T&SRs. SoftICE/W is based on Nu-Mega's MS-DOS SoftICE product and contains many of the same commands and features. Some of the major features of SoftICE/W are listed below:

- source level debugging of Windows applications, Windows device drivers, Windows VxDs, DOS applications, DOS T&SRs, DOS loadable drivers.
- real time break points on memory reads/writes, port reads/writes, memory ranges, and interrupts.
- break points on Windows messages.
- back trace history ranges.
- full screen windowed user interface.
- the ability to display internal Windows information including VxD map, Windows heap, local heap, exports from Windows USER, GDI and KERNEL, etc.
- a window that can pop up at any time.
- the ability to debug any code, including the Windows kernel itself.
- user-friendly dynamic help.
- the ability to watch variables and multiple data windows.
- programmable function keys.

SoftICE/W itself is an .EXE file that is loaded before enhanced Windows. It in turn automatically loads Windows.

Product Philosophy

SoftICE/W was designed to debug any Windows code while in enhanced mode. This includes debugging in interrupt routines, through processor level changes, through I/O drivers and in other complex areas. While in design and development, special emphasis was placed on the SoftICE/W break point capabilities. Not only are hardware-like break points provided, but SoftICE/W break points follow the memory as windows discards, re-loads and swaps memory. This is something that hardware debuggers do not do.

The SoftICE/W user interface was designed to be functional without compromising system robustness. To prevent re-entrance problems with Windows, SoftICE/W must access the hardware directly to perform its I/O. In order to still use the hardware on your PC for I/O (screen and keyboard) we had to be very cautious in our design decisions. Because of this we have had to forgo features that we would have liked to include. These include use of graphics mode for the display, use of the mouse while in the debugger and use of the file system to page source in and out dynamically.

Since we were restricted to character mode (for the above reasons), we minimized the annoying flash associated with character mode debuggers. While debugging Windows programs, SoftICE/W only flashes when a routine is actually displaying information to the screen.

SoftICE/W Requirements

SoftICE/W has the following software and hardware requirements:

- Microsoft Windows version 3.0 or later.
- A PC capable of running Windows in enhanced mode (a 386 or 486 processor) with at least 256K additional memory over and above Windows memory requirements.
- The SoftICE/W display can be on a serial terminal, a secondary monochrome monitor, or the Windows monitor itself if Windows is run in CGA, EGA, VGA or most super-VGA modes. If you have an 8514 monitor with a VGA as a secondary monitor, SoftICE/W can use the VGA as its alternate display.

Note: Since SoftICE/W does not use the DOS file system, it must keep all symbols and source in memory. The actual memory requirement for SoftICE/W depends on the number of symbol tables and source files that will be loaded at once.

SoftICE/W Installation

To install SoftICE for Windows, run the program SETUP from your SoftICE for Windows diskette 1. You can start from either one of two places in Windows:

- From the **Program Manager** window, pull down the **File** menu and choose the **Run** option.
- Or, from the **Main** window, choose **File Manager**, pull down the **File** menu, and choose the **Run** option.

In either case, complete the **Run** dialog box with: *d:*SETUP, where *d:* is the drive that contains the SIW diskette. This procedure copies all the SoftICE for Windows files over to your hard drive, creates the SIW program group and icons, and allows you to configure your video correctly. For more information on configuring your video, see sections 1.61 USING A SECOND MONITOR FOR OUTPUT, 1.62 RUNNING SoftICE/W ON A SECOND COMPUTER, and 1.63 USING A CGA, EGA, OR SUPER-VGA CONTROLLER.

Steps to a quick start:

If you are running Windows with most standard VGA drivers, you can simply enter WINICE on the DOS command line to load SoftICE/W, then refer to chapter 3 for details on using SoftICE/W. (See page 6 for details on using SoftICE/W with other video adapters including many SVGA adapters.)

Loading SoftICE/W

SoftICE/W must be loaded from DOS. SoftICE/W automatically loads Windows as the last step in its initialization. From that point on, SoftICE/W remains loaded as a resident debugger until you exit from Windows.

Instead of typing WIN to run Windows, you simply enter WINICE to run Windows with SoftICE/W resident in the background.

The full command line syntax for SoftICE/W including the optional parameters and command line switches is:

```
winice [/X] [/HST d] [/tra d] [/sym d] [/load name]     [path]WIN.COM
[WIN-parameters]]
```

There are several optional switches that can be specified on the command line:

/X	If /X is specified, SoftICE willpop up immediately after switchingto protected mode. At this point, no VxD's have been runor initialized, just loaded. This includes the WindowsVirtual Machine Monitor (VMM) itself. This option is usefulwhen debugging VxDs and device drivers. This command ignores the X command in the INIT= statement in WINICE.DAT.
/HST	If /HST (history memory) is specified, SoftICE/W will allocate extra memory for the command windows display history. The number following the /HST switch is the amount in K of extra memory to allocate. This number is always entered in decimal. SoftICE/W automatically allocates 8K for the history buffer. Anything specified by the /HST switch is added to the 8K. Having a large amount of history memory is especially useful when used in conjunction with the WLOG utility to dump large amounts of data to a text file.
/tra	If /tra (trace buffer memory) is specified, SoftICE/W will allocate extra memory for the back trace history buffer. The number following the /tra switch is the amount in K of extra memory to allocate. This number is always entered in decimal. SoftICE/W automatically allocates 8K for the back trace history buffer. Anything specified by the /tra switch is added to the 8K.
/sym	If /sym (symbol table memory) is specified, SoftICE/W will allocate memory for source and symbols. The number following the /sym switch is the amount in K of memory to allocate. This number is always entered in decimal. SoftICE/W automatically allocates 0K for symbol table memory.
/load	If /load (pre-load symbol tables and source files) is specified, SoftICE/W will pre-load the symbol table and referenced source files from the specified program file. The name following the /load switch should be the complete path and file name to a program that contains a symbol table. This switch is most useful when debugging Windows drivers, Windows DLL's, VxDs, DOS loadable drivers or DOS T&SRs. Symbol information for all other types of programs is loaded with WLDR.EXE.
/loadx	The /loadx switch is the same as the /load switch except just symbols are loaded, not source files.
/exp	Adds exports from a specified DLL or Windows application to the SoftICE/W export list. This allows you to symbolically access these exported symbols.

Note: All of the above switches can also be specified in the WINICE.DAT file so they do not have to be repeatedly given on the command line. The **/HST**, **/SYM** and **/TRA** switches on the command line override whatever is in the WINICE.DAT file.

All memory required is allocated from extended memory via XMS calls.

After any desired switches are specified, you may also specify a full path and file name where WIN.COM is located. If the file name WIN.COM is specified with no path, WINICE will search all the default paths. *WIN-parameters* is the command line that is passed to WIN.COM. These fields are optional. They should only be used if you wish to load WIN.COM from an alternate directory, or if you wish to pass parameters to Windows.

After WINICE.EXE is loaded, it will load Windows by executing WIN.COM. SoftICE/W will look in the current directory and then search the default paths for the WIN.COM file, unless a full path-file-name is specified after the WINICE switches.

Once WIN.COM is loaded, SoftICE/W remains in the background until activated by either a key sequence (initially **Ctrl D**), a break point, a fault, or by an INT 1 or INT 3 if **I1HERE** or **I3HERE** mode is turned on.

SoftICE/W can also pop up after WIN.COM is loaded if the **X** command is removed from the **INIT** statement in the WINICE.DAT file. Doing this allows you to begin debugging before your VxD init code is executed. SoftICE/W will pop up immediately after switching to protected mode. At this point, no VxD's have been run or initialized, just loaded. This includes the Windows Virtual Machine Monitor (VMM) itself.

Using a Second Monitor for Output

SoftICE/W is able to display its output on a monochrome monitor separate from the Windows display. To make SoftICE/W display its output on this second monitor, you place the **ALTSCR ON** command in the **INIT** string of your WINICE.DAT file. This causes all SoftICE/W screen output to be redirected to the second monitor. If no second monitor is found, then SoftICE/W will not output anything. **ALTSCR** will only display in 25 line mode. If you are currently using 43 or 50 line mode, SoftICE/W will automatically switch to 25 line mode before switching to the second monitor.

If you wish SoftICE/W output to switch back to the Windows display, enter **ALTSCR OFF** from the SoftICE/W command window.

An example **INIT** string that will enable a monochrome second monitor is:

```
INIT "ALTSCR ON; X;"
```

See page 189 for a complete description of the **ALTSCR** command.

Note: **ALTSCR ON** can also be entered from the SoftICE/W command window if you wish to switch SoftICE/W output to your second monitor.

Note: If you have an 8514 monitor with a VGA as a secondary monitor, SoftICE/W can use the VGA as its alternate display. To do this **ALTSCR** should be **OFF**. Unlike a monochrome monitor, the SoftICE/W output will only be displayed when SoftICE/W is popped up.

Running SoftICE/W on a Second Computer

To run SoftICE/W from a second computer, you will need a second IBM-compatible PC running MSDOS. Any PC will do, including 8088, 8086 or 80286 machines. You must first attach the computer to your Windows computer with a null modem cable attached to the two serial ports. Then run the SERIAL.EXE utility on the second PC. You must also place the **SERIAL ON** command in the WINICE.DAT **INIT** statement. **SERIAL** is only supported in 25-line mode. If you are currently using 43- or 50-line mode, SoftICE/W will automatically switch to 25-line mode before switching to the second terminal.

You must also place the COM*n* keyword on a separate line in the WINICE.DAT file to reserve a specific COM port for the serial connection. "*n*" is a number between 1 and 4 representing the COM port. If this statement is not present in WINICE.DAT, then you must pop SoftICE/W up from the keyboard on the SoftICE/W machine, not the second PC.

The following example will tell the second machine to use COM1 at 19200 baud, and tell SoftICE/W to switch its display to a serial terminal on COM1 at 19200 baud. Type:

SERIAL ON 19200(on the second PC)

SERIAL ON 19200(on the SoftICE/W machine)

See page 179 for a complete description of the **SERIAL** command and the SERIAL.EXE utility.

Using a CGA, EGA or Super-VGA Controller

Since SoftICE/W does not use Windows or the Windows display driver for output, it must access the video display hardware directly. However, With super-VGA there is no true standard at the hardware level so SIW will often not read the card correctly.

When you install SoftICE/W, you are given a list of video cards that SIW currently supports. If your video card is in the list, then simply select it and SoftICE should be all set to go.

If your video card is not in the list, then you have a few choices to get SIW running correctly on your system:

- Run SIW on a second monitor. (See section 1.61)
- Run SIW on a second machine. (See section 1.62)
- Run Windows in standard VGA mode using the standard Windows VGA driver (VGA.DRV). If you do this, you will want to run the Video Setup program and choose VGA. You will then want to put the line DISPLAY=0 in your WINICE.DAT file.
- Obtain a video card that SIW does support.

At any time, if you switch video cards, rerun the video setup program and choose the correct card from the list given.

One other utility provided with SoftICE/W that might help with the video problem is VIDMODE. This utility is a Windows program that must be run while SoftICE/W is resident. VIDMODE forces Windows to switch to character mode and then back to graphics mode. During these two mode switching operations, I/O to the video adapter is recorded and written to a file named WINICE.VID.

The next time you run SoftICE/W, this file is loaded into memory and the I/O is played back to the controller when SoftICE/W pops up.

VIDMODE has three options: standard, graphics to character and character to graphics. The default is standard and should be tried first. If the standard option fails, you can run VIDMODE again, this time checking the boxes for the other options.

If VIDMODE does not get back to graphics mode properly when run with the default options (no boxes checked), then it will not work with any other option either. See the next section for alternatives.

To run VIDMODE you must do the following:

- Run SoftICE/W from command line by entering WINICE.
- Select the VIDMODE icon or select *File* followed by *Run* from the Windows program manager. Then enter *drive:path-name*\VIDMODE.EXE.
- Check the desired VIDMODE check boxes, then select OK.

Warning: When either of VIDMODE's check boxes are checked, there is a danger that VIDMODE will hang Windows. Take care to shut down any Windows applications with work in progress before running VIDMODE with boxes checked.

VIDMODE writes a file named WINICE.VID to the directory that WINICE.EXE was loaded from. This file contains data necessary for SoftICE/W to work with your video controller. The next time you run SoftICE/W, it will read the information from the WINICE.VID and use those video parameters.

The Windows screen may become momentarily corrupted for a few seconds from the time the winice.vid file is processed until Windows actually comes up.

Note: You must run VIDMODE.EXE each time you reconfigure Windows to a different VGA mode.

WINICE.DAT Initialization File

When SoftICE/W is first loaded, it pops up its screen and reads information from the WINICE.DAT initialization file located in the same drive and directory as the WINICE.EXE file. This file is used to specify initialization and customization parameters including the initial programming of the function keys and to specify a string of commands that will be executed immediately.

WINICE.DAT is an ASCII text file where each line is of the following format:

- *keyword* = *string*

Valid *keywords* are the function keys **F1-F12**, **SF1-SF12**, **CF1-CF12**, **AF1-AF12** and the keywords **INIT**, **LOAD**, **LOAD32, LOADX, EXP, TRA**, **SYM** and **HST**. The meaning of *string* changes depending on the keyword. For the function keys and the **INIT** *keyword*, *string* is a list of one or more SoftICE/W commands within quotes. For the **TRA**, **SYM** and **HST** *keywords*, *string* is a decimal number that specifies the amount of memory in K to allocate. For the **LOAD, LOAD32, LOADX and EXP** *keywords*, *string* is the full path, file name and extension of a program file that contains a symbol table, or exports in the case of **EXP** .

All of the above keywords have corresponding command line switches. For a complete description of these keywords and their corresponding switches see *Loading SoftICE/W* on page 3.

There are two additional keywords that are not followed by a string and do not have corresponding command line switches. These keywords are **COM*n*** and **NOLEDS.**

The **COM*n*** keyword tells Soft_ICE/W which COM port to reserve for a serial terminal connection. The "*n*" is replaced by a number between 1 and 4, for example, **COM1**.

NOLEDS is a special keyword that is used on some computers that have keyboards that cause SoftICE/W to hang when it pops up. Placing the **NOLEDS** keyword in WINICE.DAT will eliminate this problem, but has the side effect of disabling the keyboard LED indicators from toggling while in SoftICE/W.

Defining Function Keys

You can assign values to function keys by placing commands in the WINICE.DAT initialization file. A command string can be assigned to any of the 12 function keys or key combinations involving **Shift**, **Ctrl** or **Alt** pressed with a function key. The format of a function key definition is one of the following:

F*n* **=** "*string*" Defines function key *n*

SF*n* = "*string*"	Defines Shift + function key *n*
CF*n* = "*string*"	Defines Ctrl + function key *n*
AF*n* = "*string*	Defines Alt + function key *n*
n	Decimal number from 1 to 12.
string	One or more SoftICE/W commands within quotes. A ';' embedded in the command string represents the *Enter* key. Putting the '^' in front of a command makes the command invisible.

An example of each type of function key definition follows:

```
F1 = "H;"
SF3 = "^FORMAT;"
CF11 = "SHOW B;"
AF5 = "CLS;"
```

If these function key definitions are put in your WINICE.DAT initialization file, the following actions will occur when SoftICE/W pops up. Pressing **F1** will display help information about all the SoftICE/W commands in the command window. Pressing **Shift** and **F3** together will toggle the display format in the data window. Pressing **Ctrl** and **F11** together will begin displaying from the back trace history buffer starting with the oldest instruction in the buffer. Pressing **Alt** and **F5** together will clear the SoftICE/W command window and all display history.

The SoftICE/W Initialization String

SoftICE/W has a provision to execute a series of commands at initialization time. This is useful for altering the **Ctrl D** hot key sequence that pops up SoftICE/W, or for changing the SoftICE/W window sizes. If you will be using a secondary monochrome monitor or using SoftICE/W from a remote computer you must insert the **ALTSCR ON** or **SERIAL ON** commands in the initialization string. See the complete description of each command (pages 189 and 179 respectively) before placing it in the string. An example of an initialization string is:

```
INIT = "ALTSCR ON; ALTKEY ALT Z; WR; X;"
```

This initialization string will switch the SoftICE/W output to a secondary monochrome monitor, change the hot key sequence to **Alt Z** , toggle the register window off, and exit from SoftICE/W.

Note: Always place new items in the **INIT** statement prior to the **X;**. The **X;** exits from SoftICE/W, and if the entries are made after **X;**, they will not be executed until you pop up SoftICE/W for the first time.

Allocating Extra Memory

You can allocate extra memory for the command window display history, the back trace history buffer, and for source and symbols. Do this by specifying the **HST**, **TRA**, or **SYM** statements in the WINICE.DAT initialization file, in the form **HST =** *d*. See *Loading SoftICE/W* on page 3 for more details on these statements.

Pre-loading Symbols and Source Files

Specifying a **LOAD** statement in the WINICE.DAT initialization file lets you pre-load symbol tables and source files for programs that you will later debug. This is especially useful for VxD's, Windows drivers, Windows DLL's, DOS loadable drivers and T&SRs. You must specify the complete path and file name of the program file that contains symbolic information. Specifying the **LOADX** statement allows you to load symbol tables without loading source files.

When a symbol table is loaded with a **LOAD** statement, the symbol segment values are not always adjusted to the right location. If you have loaded a symbol table for a DOS program, T&SR or loadable driver, you must use the **SYMLOC** command to locate the segment addresses.

If you have preloaded a symbol table for a Windows program or DLL, the segments will be automatically located when the Windows program or DLL is loaded. Before the program or DLL is loaded, the segments of the symbols are ordinal numbers.

See *Loading SoftICE/W* on page 3 for more details on the **LOAD** statement.

Default WINICE.DAT Initialization File

The WINICE.DAT initialization file found on the release diskette defines the default settings for functions keys as well as a list of LOAD and EXP statements that are commented out. If UN-commented, these statements will load symbol tables for Windows internal components. The .SYM files loaded by these statements come with the Microsoft Windows SDK.

The default function key assignments emulate the function key actions done by CodeView. The default WINICE.DAT initialization file is provided below:

```
F1 = "H;"
F2 = "^WR;"
F3 = "^SRC;"
F4 = "^RS;"
F5 = "^X;"
F6 = "^EC;"
F7 = "^HERE;"
F8 = "^T;"
F9 = "^BPX;"
F10 = "^P;"
F11 = "^G @SS:ESP;"
F12 = "^P RET;"

SF3 = "^FORMAT;"

CF8 = "^XT;"
CF9 = "TRACE OFF;"
CF10 = "^XP;"
CF11 = "SHOW B;"
CF12 = "TRACE B;"

AF1 = "^WR;"
AF2 = "^WD;"
AF3 = "^WC;"
AF4 = "^WW;"
AF5 = "CLS;"
AF8 = "^XT R;"
```

```
INIT = "X;"
```

A ';' embedded in the command string represents the *Enter* key. Putting the '^' in front of a command makes the command invisible, so the command will not be echoed to the command window.

Note: You must alter the default WINICE.DAT file if you will be using a secondary monochrome monitor, or a second computer attached by a serial cable, as your SoftICE/W display.

2 User Interface

SoftICE/W Screen

SoftICE/W provides a full-screen windowed interface for debugging. There can be up to five windows displayed within the screen at one time:

Register Window	Display/Edit the current state of the registers and flags.
Watch Window	Display the value of any variables that are being watched with the WATCH command.
Data Window	Display/Edit memory.
Code Window	Display unassembled instructions and/or source code.
Command Window	Enter user commands and display information.

The SoftICE/W windows are tiled windows and can be sized and removed. The key sequence **Ctrl D** is the initial hot key sequence that will pop up and pop down the SoftICE/W screen. This hot key sequence can be changed with the **ALTKEY** command. A detailed description of the function and control of each window is given in the following sections.

If you want to see more lines per screen, you can use the **LINES** command to switch to 43 (or 50 lines on VGA cards) per screen instead of the standard 25 lines.

When the SoftICE/W screen is displayed, all background activity on your computer comes to a halt. All interrupts are disabled and SoftICE/W does all screen and keyboard I/O by directly accessing the hardware.

When SoftICE/W pops up, the reason for the popup is displayed. The only time a reason is not displayed is for the **T** and **P** commands.

Register Window

The register window always contains three lines and displays the current value of the system registers and flags. The following registers are displayed:

◊ EAX, EBX, ECX, EDX, ESI, EDI, EBP, ESP, EIP,CS, DS, SS, ES, FS, GS.

For the 15 registers, SoftICE/W highlights the registers that have been altered. This is done for the **T**, **P** and **G** commands. This feature is useful for seeing what registers have been altered by a procedure call.

The second line of the register window also contains the flags. A flag whose value is 0 is displayed as a lower case letter with a normal video attribute. A flag whose value is 1 is displayed as an upper case highlighted letter. The following flags are displayed:

O	Overflow flag
D	Direction flag
I	Interrupt flag
S	Sign flag
Z	Zero flag
A	Auxiliary carry flag
P	Parity flag
C	Carry flag

If the current instruction references a memory location and the register window is visible, the contents of the memory location will be displayed in the third line of the register window beneath the flags field.

The register window is also used for editing the registers and flags. Use the **R** command to move the cursor into the register window. The registers can now be edited in place. The following keys are active when editing the register window:

Tab	Position to the beginning of the next register field.
Shift Tab	Position to the beginning of the previous register field.
Enter	Accept changes and exit edit register mode.
Esc	Exit edit register mode. The register the cursor is currently on will NOT be changed, but other previously modified registers will be changed.
Insert	Toggle the value of a flag when the cursor is positioned in the flags field.
Arrow keys	Move the cursor left and right and up and down in the register window.

The register window cannot be sized since it always contains three lines. However, its visibility can be toggled using the **WR** command. This will free up screen space for the other windows.

Associated commands:

- **WR**, **R**

Watch Window

The watch window is a display-only window that can contain up to eight lines. Each line displays the value of one expression that is being watched. Each watch line contains the following fields:

watch number	This is a number from 0 to 7 that identifies the watch variable index. This number is used when clearing watch variables using the CWATCH command.
expression	This is the actual expression that was typed on the WATCH command. This expression is reevaluated every time the watch window is displayed. If the expression is NOT a symbol and no segment is specified, the following defaults are used: • If it's IP or EIP, CS is used. • If it's BP, EBP, SP or ESP, SS is used • Anything else uses DS, including just hexadecimal addresses
location	This is the hexadecimal address of the watch variable.
value	This is the current value of the variable being watched. This field can display the following data types depending on the type of watch set: • Byte hexadecimal • Word hexadecimal • Dword hexadecimal • Short Real (4 byte real) • Long Real (8 byte real) • 10-Byte Real

The watch window cannot be sized, since it always contains one line for each expression being watched. However, its visibility can be toggled using the **WW** command. If there are no watch expressions set, no window will be displayed.

Associated commands:

- CWATCH
- WATCH
- WATCHB
- WATCHD
- WATCHL

- WATCHS
- WATCHT
- WATCHW
- WW

Data Window

The data window displays the contents of memory. Each line displays 16 bytes of data in the current format if it is either byte, word, dword, short real or long real. If the current format is 10-byte real, each line displays 20 bytes of data. The data bytes are also displayed in ASCII on the right side of the window if the current format is hexadecimal (byte, word, or dword). There can be up to four different data windows, each able to display a different location in a different format. The window number is displayed on the line above the data window on the right hand side. It will always be in the range 0 to 3. Only one data window is displayed at a time. Cycling through the data windows is done with the **DATA** command (default function key is **F12**).

To the left of the window number is the segment type. This can be one of the following two fields:

VM	The displayed data is from a segment from a virtual machine.
PROT	The displayed data is from a protected mode selector.

To the left of the segment type is the data format type. This can be either byte, word, dword, short real, long real or 10-byte real. Use the **FORMAT** command to change the data format display of the current data window.

An expression can be assigned to any of the data windows using the **DEX** command. Whenever SoftICE/W pops up, this expression is evaluated and the resulting location will be displayed in the specified data window. A good use of this function is setting up a window that always displays the contents of the stack. To do this, you could type in the command **DEX 0 SS:ESP**. Every time SoftICE/W pops up after that, the current contents of the stack will be displayed in data window 0.

In addition to containing the window number, the segment type and the data format, the line above the data window also contains one of the following two strings:

- If the window has been assigned an expression with the **DEX** command, the ASCII expression will be displayed on this line.

- The nearest symbol preceding the data location. This can be one of the following strings:
  - ◊ A symbol name followed by the hexadecimal offset from the symbol name.
  - ◊ A VxD name followed the by the hexadecimal offset from the beginning of the VxD.
  - ◊ A Windows module name followed by a type, if the data segment is part of the Windows heap.
  - ◊ The owner name of the data segment if it is part of a virtual machine.
  - ◊ If the location does not have an associated symbol, this field will be blank.

The data window is also used for editing memory. Use one of the **E** commands to move the cursor into the data window. Memory can now be edited either with hexadecimal or ASCII characters. The data window can also be scrolled using the arrow keys. The following keys are active when editing the data window:

`Tab`	Toggle position between numeric and ASCII areas.
`Shift Tab`	Position cursor to the beginning of the previous data field (previous byte, word, or dword in hexadecimal mode, previous character in ASCII mode).
`Shift F3`	Change the format of the data window. Pressing this key combination cycles between the byte, word, dword, short real, long real and 10-byte real formats.
`Enter`	Accept changes and exit edit data mode.
`Esc`	Exit edit data mode. The data field the cursor is currently on will NOT be changed, but other previously modified data fields will be changed.
`Arrow keys`	Move the cursor left and right and up and down in the data window. They are also used to scroll through memory. The PageUp and PageDn keys can be used to scroll the data window a page at a time.

The data window can also be scrolled when the cursor is in the code window or the command window. The following keys are used to scroll the data window:

Alt PageUp	Scroll data window down one page.
Alt PageDn	Scroll data window up one page.
Alt UpArrow	Scroll data window down one line.
Alt DownArrow	Scroll data window up one line.

The data window can be sized by entering the **WD** command followed by the number of lines desired. The **WD** command can also be used to toggle the visibility of the data window.

Associated commands:

- D
- DATA
- DB
- DW

- DD
- DEX
- DS
- DL
- DT
- E
- EB
- ED
- EW
- ES
- EL
- ET
- FORMAT
- S
- WD

Code Window

The code window is used to display the disassembled code and/or source code. The code can be displayed in three different modes:

Source	If source code is available, the actual source file can be displayed in the code window.
Mixed	In mixed mode, both the actual source lines and the disassembled instructions are displayed. Each source line is followed by its assembler instructions.
Code	In code mode, only the disassembled instructions are displayed.

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

Location	This is the hexadecimal address of the instruction. If there is a public code symbol for this location, it is displayed on the line above this instruction.
Code bytes	These are the actual hexadecimal bytes of the instruction. The default is to suppress the code bytes since they are usually not needed. These bytes can be displayed using the CODE command
Instruction	The disassembled mnemonics of the instruction. This is the current assembly language instruction. If any of the memory address parts of the instruction match a symbol, the symbol will be displayed instead of the hexadecimal address.

An example of a disassembled instruction in code or mixed mode is:

```
00FD:00001DA1  56        PUSH     SI
```

When SoftICE/W pops up, the instruction located at the current EIP will always be displayed in reverse video. If the instruction is a relative jump, it will contain either the string (JUMP) or (NO JUMP) indicating whether or not the jump will be taken. If the instruction references a memory location, the contents of the memory location will be displayed in the register window beneath the flags field. If the register window is not visible, this value is displayed on the end of the code line.

If there is a break point set on any instruction in the code window, the line will be displayed with the bold video attribute.

The line above the code window displays information about the code being displayed. At the right of this line is the code segment type. This can be one of the following three fields:

VM	The displayed code segment is from a virtual machine.
PROT16	The displayed code is from a 16-bit protected mode selector.
PROT32	The displayed code is from a 32-bit protected mode selector.

On the left hand side of the line will be one of the following:

- If source code is displayed on the screen, the name of the current source file will be displayed.
- A symbol name followed by the hexadecimal offset from the symbol name. This corresponds to the top line displayed in the code window.
- A VxD name followed by the hexadecimal offset from the beginning of the VxD.
- A Windows module name followed by the segment number in parenthesis, if the code segment is part of the Windows heap.
- The owner name of the code segment if it is part of a virtual machine.
- If the location does not have a preceding symbol, this field will be blank.

You move the cursor into and out of the code window by using the **EC** command (default key is **F6**). When the cursor is in the code window, you can scroll the code window up and down using the arrow keys and the *PageUp*/*PageDn* keys. You can set point-and-shoot style break points in this mode by using the **BPX** (default key is **F9**) and **HERE** (default key is **F7**) commands.

You can still enter commands even when the cursor is in the code window. After you type the first letter of a command, the cursor will move down to the command window. After *Enter* is pressed and the command is completed, the cursor will move back to the code window. You can also use function key commands in this mode.

The instruction at the current EIP can be made visible at any time by using the '**.**' command. Any code address can be unassembled using the **U** command.

The code window can also be scrolled when the cursor is in the command window. The following keys are used to scroll the code window:

Ctrl PageUp	Scroll code window down one page.
Ctrl PageDn	Scroll code window up one page.
Ctrl UpArrow	Scroll code window down one line.
Ctrl DownArrow	Scroll code window up one line.
Ctrl Home	Jump to first line of source file.
Ctrl End	Jump to last line of source file.

The code window can be sized by entering the **WC** command followed by the number of lines desired. The **WC** command can also be used to toggle the visibility of the code window.

Associated commands:

- .
- A
- BPX
- CODE
- EC
- FILE
- HERE
- SRC
- SS
- TABS
- U
- WC

Command Window

The command window is used for command entry and miscellaneous information display. The command window is always visible and is always at least two lines long. It cannot be sized explicitly, but its size changes as the other windows are sized. Commands can be entered whenever the cursor is in the command window or the code window. The description of the command window can be broken down into several functional areas described below.

Line Editing

Commands are typed in from the keyboard and executed when the *Enter* key is pressed. The command line can be edited using the following keys:

Home	Move cursor to column 0 of command line.
End	Move cursor past the last character of the command line.
Insert	Toggle insert mode. When in insert mode the cursor is displayed as a block cursor. The character entered is inserted at the current cursor position and the end of the line is shifted to the right by one. When not in insert mode, the character entered overwrites the character at the current cursor position.
Delete	Delete the character at the current cursor position and shift the end of the line to the left by one.
Bksp	Destructive backspace.
Esc	Cancel command line.
Arrows	The left and right arrow keys move the cursor within the command line.

Command History

SoftICE/W remembers the last 32 commands that have been typed in the command window. These commands can be recalled for editing and execution. When the cursor is in the command window, the following keys are used for command recall:

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

When the cursor is in the code window, the following keys are used for command recall:

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

Information Display

Many SoftICE/W commands display information in the command window. Information is displayed starting at the line beneath the command and continuing downward. If displaying on the last line of the window, the window will scroll. If all the information cannot fit in the window, an 'Any Key To Continue, ESC To Cancel' prompt will appear on the help line. This prompting can be turned off using the **PAUSE** command.

Display History

SoftICE/W reserves memory to save everything that is written into the command window. The command window can then be scrolled up and down. The following keys are used to scroll the command window:

Shift UpArrow	Scroll the display history down by one line.
Shift DownArrow	Scroll the display history up by one line.
PageUp	Scroll the display history down by one page.
PageDn	Scroll the display history up by one page.

SoftICE/W initially reserves enough memory to store 128 lines of data. This can be increased by using the **/HST** switch on the DOS command line when starting WINICE or by using the **HST** statement in the WINICE.DAT initialization file.

The display history can be saved to a disk file with the utility WLOG.EXE. WLOG.EXE can be run from Windows or from DOS. When run from DOS, WLOG.EXE has the following syntax:

```
WLOG [file-name]
```

If the file already exists, it is truncated to length zero before the display history is written to the file. If *file-name* is not specified, the command usage is displayed.

When run from Windows, WLOG.EXE will prompt the user for the output file name.

Some occasions where using WLOG would be very handy include when you are dumping large amounts of data, disassembling code, or listing Windows messages logged by the **BMSG** command. You must first make sure that the data is going to the command window so that WLOG has access to the information. For example, before dumping data, remove the data window so the data is displayed in the command window.

Help Line

The bottom line of the screen always contains the help line. This line is updated as characters are typed on the command line. The help line gives three different levels of help as the command line is entered:

- When the typed characters don't specify a complete command, all valid commands that start with the typed characters are displayed.
- When the typed characters exactly match a command, a description of that command is displayed.
- When a space is entered after a command, the syntax of that command is displayed.

When editing in the register window or the data window, the help line contains the valid editing keys for that window.

Command Completion

As characters are typed, the help line displays the list of valid commands that start with those characters. If only one command is displayed, then the command can be automatically completed by pressing the space bar. SoftICE/W will fill in the remaining characters of the command followed by a trailing space.

Function Keys

You can assign character strings to any of the 48 function keys (**F1**-**F12**, **Shift F1**-**Shift F12**, **Ctrl F1**-**Ctrl F12**, **Alt F1**-**Alt F12**) by using the **FKEY** command or putting them in the WINICE.DAT initialization file. This makes it possible to assign a command or a series of commands to a particular function key. When the function key is pressed, the character string is inserted into the keyboard buffer, so it behaves as if the string had been typed in.

The semicolon '**;**' character represents the *Enter* key. If the string is preceded by the caret '^' character, the command will not be displayed. This makes it possible to repeatedly use function keys without cluttering up the command window.

Command Syntax

SoftICE/W is a command-driven debugging tool. To interact with SoftICE/W, you enter commands, which can optionally be modified by parameters.

All commands are text strings that are one to six characters in length and are case-insensitive. All parameters are either ASCII strings or expressions. A space between the command and its parameters is not required.

An *address* in SoftICE/W can be either a segment:offset address or just an offset.

Expressions in SoftICE/W are combinations of the following items:

Numbers	Numbers are entered in hexadecimal and can be from 1 to eight characters in length (32 bit maximum).
Segments	Any number or register followed by a colon is interpreted as a segment. SoftICE/W will interpret this segment according the current code mode. If the current popup mode is from a virtual machine, the value will be treated as a real mode segment. If the current popup mode is from protected mode, the value will be treated as a selector. This behavior can be overridden by preceding the segment with an override operator. Use & for segments and # for selectors.
Registers	Registers can be used in place of numbers in an expression. SoftICE/W will then use the contents of the registers for these values. The following register names may be used in an expression: • AL, AH, AX, EAX, BL, BH, BX, EBX, CL, CH, CX, ECX, DL, DH, DX, EDX, SI, ESI, DI, EDI, BP, EBP, SP, ESP, DS, ES, SS, CS, FS, GS, IP, EIP or FL If the (E)SP or (E)BP registers are used and no segment is specified, SoftICE/W will automatically use the SS segment. If the (E)IP register is used without specifying a segment, SoftICE/W will automatically use the CS segment. For all other registers, SoftICE/W will use DS. For plain hexadecimal numbers, SoftICE/W will continue to use whatever segment/selector is currently displayed in the data window.
Line numbers	A decimal number preceded by a '.' (period) will be interpreted as a source file line number. It will be converted to the correct segment:offset address.
Symbols	Symbols are case-insensitive ASCII strings representing the address of a symbol. SoftICE/W recognizes the following symbols: • All symbols loaded by WLDR or from WINICE.DAT. • All exported symbols from USER.EXE, GDI.EXE and KERNEL.EXE. • All VxD names- • The names of all exported VxD calls. • The names of all Windows messages in the range 0 to 400h.
Operators	SoftICE/W recognizes the following operators: • +, -, *, / All operators are of equal precedence and are evaluated left to right SoftICE/W also recognizes the following special operators: • The '@' sign typed as the first character of an expression is the indirection operator. If the specified segment is a 32-bit segment, SoftICE/W will do 32-bit near indirection. This means the dword at the supplied address will be treated as a 32-bit offset within the specified segment. If the specified segment is a 16-bit segment, SoftICE/W will do 16-bit far indirection. This means the dword at the supplied address will be treated as a segment:offset address. • An '&' preceding a segment expression will force the segment to be evaluated as a real mode segment regardless of the current popup type. • A '#' preceding a segment expression will force the segment to be evaluated as a protected mode selector regardless of the current popup type.

Expression examples :

```
D DS:EBX*4
```

This command displays memory at the address formed by multiplying the contents of the EBX register by four.

```
G @SS:ESP
```

Assume you are at the first instruction of a called procedure. Entering this command will set a temporary break point at the return address on the stack and skip the procedure.

```
U GetMessage
```

This command will disassemble instructions starting at the Windows GetMessage procedure.

```
D #FD:0
```

This command will display data from the protected mode selector FD.

```
D &FD:0
```

This command will display data from the real mode segment FD.

```
G  .112
```

This command will cause your program to execute until it reaches line number 112.

Notational Conventions Used in this Manual

Throughout this manual, the following conventions are used:

- Command names and function key names are printed in **bold**. Other key names are in *italics*.
- Other words that are in *italics* must be replaced by an actual value, rather than typing in the *italicized* word.
- Items that are in brackets [ ] are optional.
- Items separated by a vertical bar are choices. (*x* | *y* means to use either item *x* or item *y*)

3 Using SoftICE/W

Experimenting with SoftICE/W

SoftICE/W is a Windows resident debugger. This means that while Windows is running, SoftICE/W is always in memory waiting to be activated through its **Ctrl D** hot key sequence. SoftICE/W is useful for many types of debugging, and is used quite differently depending on your debugging problem. A good way to learn about SoftICE/W is to experiment by popping it up in different parts of Windows and using some of the SoftICE/W "Display System Information" commands to tell you where you are and what Windows is currently doing. This chapter assumes that you've installed and loaded SoftICE/W before beginning to experiment. If you haven't yet done so, see Loading SoftICE/W and *SoftICE/W Installation* on page 2 for more information on installing and loading. The examples in this chapter also assume you are using the supplied default WINICE.DAT initialization file for function key definitions.

This chapter shows you how to get started using SoftICE/W. The sub-chapters 3.1 through 3.3 should be read no matter what your intended use of SoftICE/W. Other sub-chapters can be read as you begin to use SoftICE/W on particular debugging problems.

Popping Up SoftICE/W

The default hot key sequence is **Ctrl** pressed simultaneously with **D**. This will pop up the SoftICE/W screen.

Note: When you pop up SoftICE/W, you will typically see assembly language instead of source code displayed in the SoftICE/W code window. This is because SoftICE/W can pop up at whatever point the instruction pointer happened to be within Windows or MS-DOS.

Normally when the SoftICE/W screen pops up, it will replace your Windows display. If SoftICE/W is not compatible with your Windows mode, or if you prefer to see the SoftICE/W screen and the Windows display at the same time, you can use SoftICE/W on a second monitor or on a second computer attached through the serial port.

Changing the Hot Key Sequence

If you wish to use a different hot key sequence to pop up the SoftICE/W screen, you can easily change the hot key sequence with the **ALTKEY** command. You typically place the **ALTKEY** command in the **INIT** statement of the WINICE.DAT initialization file. This will always change the key sequence from the start of Windows. If Windows is already running and you decide that you have a hot key conflict with another program, you can pop up SoftICE/W and change the hot key sequence for the current debugging session by entering the **ALTKEY** command on the SoftICE/W command line.

For example, to change the hot key sequence to **Ctrl Z**, you would enter:

```
ALTKEY CTRL Z
```

See page 171 for a complete description of the **ALTKEY** command.

Loading Systems Level Symbols

SoftICE/W is a Windows resident debugger that is in memory at all times while Windows is running. This gives you the ability to pop up SoftICE/W to trace through any code at any time including system software. SoftICE/W automatically loads exported symbols from GDI, USER and KERNEL as well as module names, VxD names and VxD service routines. There are two ways you can load additional systems level symbols.

You can use the **EXP** command in **WINICE.DAT** to load exported symbols from DLL's and Windows Apps. This will add to the symbols displayed by SoftICE/W's **EXP** command.

You can also load .SYM files supplied by Microsoft in the Windows SDK. These symbols are loaded in the form of program symbol tables and are used in the same way you would use symbols loaded for a program that you are debugging.

There are several examples of the **EXP** directive and the **LOAD** directive commented out in the default **WINICE.DAT**. Un-comment the appropriate lines in **WINICE.DAT** to experiment with these additional system symbols.

Note: If you are loading WIN386.SYM (built for the debugging kernel only), you may want to place the command **'TABLE AUTOOFF'** in the **INIT** statement of **WINICE.DAT**. Otherwise, almost every time you pop up SoftICE/W you will switch to the WIN386.SYM symbol table.

Loading Programs for Debugging

In addition to being a resident debugger, SoftICE/W also provides the capability to load and debug a program as you would with a traditional applications debugger. The following sections describe how to prepare and load your program for debugging at source level.

Preparing a Program for Debugging

Before debugging a DOS or Windows program, you normally compile and link the program with the symbolic switches. With Borland compilers, use the /v switch on both the compile and link. With Microsoft compilers, use the /Zi switch on the compile and the /CO switch on the link.

If you have a DOS program that is a .COM file, you must use an alternate method. You must create a detailed .MAP file, and run the supplied utility MSYM.EXE to create a .SYM file. WLDR will look for the .SYM file if it does not find symbols in the .EXE file.

For Microsoft compilers, you must use the /M and /LI switches when linking to create a detailed .MAP file. For Borland compilers use the /m and /l switches.

When SoftICE/W loads your program into memory it will load all of the source files that were compiled with debug information. In a large program, this may not be practical. SoftICE/W allows you to selectively load source files by using a .SRC file. You create the .SRC file with an editor. When SoftICE/W loads symbols into memory it searches for a file name *program-name*.SRC on the same directory as the program. The .SRC file contains a list of the source files to be loaded regardless of how many files were compiled with debug information.

If the .SRC file is not present, all files are loaded. If the .SRC file is length 0, no files are loaded. The .SRC file is a simple text file with file names (without path) delimited by carriage return/line feed. For example, if you have a C program named FOO.EXE with 10 source files named FILE1 - FILE10 and you only want to load 3, 5 and 8 the FOO.SRC file would contain:

```
FILE3.C
FILE5.C
FILE8.C
```

The final step in preparing a program for debugging is to make sure that you have reserved enough memory with the **SYM** statement in your **WINICE.DAT** file to hold the symbol information and all of the source files.

Preparing a Windows Driver for Debugging

For most drivers, you prepare the program as if it were an application. However, there are some sample assembly language drivers supplied in the Microsoft DDK that will not debug properly at source level with the standard approach. You should try the standard approach first. If you have problems with source synching up with the generated assembly code, see below.

Many of the sample Windows drivers supplied with the DDK are structured in a way that is not fully compatible with the tools with respect to source level debugging. These problems occur for one of the following reasons:

1. Multiple code segments are defined in each module.
2. Code is in include files.

Multiple Code Segments In Module

The debug records included in the .EXE file when the /CODEVIEW switch is used do not handle multiple code segments in a module, therefore we must extract the debug information from the .MAP file with the MSYM utility. MSYM creates a .SYM file that is compatible with SoftICE/W. If you have a driver with multiple code segments defined in each module, then build using the following steps:

1. MASM with /Zi or /Zd.
2. LINK with /LI and /MA (DO NOT USE /CO).
3. Enter MSYM followed by the file name with no extension.

   (DO NOT USE Microsoft's MAPSYM; it does not handle source records properly.)
4. Place the .SYM file created by MSYM.EXE in the same directory as your driver.
5. Place a "LOAD = path/driver-name.DRV" statement in WINICE.DAT.
6. Make sure that you have reserved enough memory for the symbol information and all of the source files with the **SYM** directive in WINICE.DAT.

Source Code In Include Files

Source code in include files is not handled by the linker properly when creating line number records in .MAP files. This problem is present in the sample display drivers that come with the DDK. The worst offender seems to be BITBLT.ASM. If you wish to debug a driver that has code in the include files, there are three alternatives:

1. Eliminate include files by placing the source code from the include file directly in the main module.

2. Eliminate include files by making them separate modules.
3. Fix the .MAP file after linking, by inserting the include file names records into the .MAP file manually with your text editor.

The third method is clumsy and must be done after each link. The problem is that the line number sections in the .MAP file contain the line number information for the include files, but not the include file names. You must look through the line number sections finding places where a line number is less than the one that preceded it. At this place you must inserta line that contains the include file name. See file EXAMPLE.MAP for an example of this.

Fixed LOADONCALL Segments

Some of the segments in the sample display drivers have the FIXED LOADONCALL attribute. Apparently, these segments may be loaded only in certain Windows modes of operation (standard or enhanced). If you have a segment with this attribute, the symbol table will retain its ordinal number rather than being updated with the actual selector value. To remedy this you must enter SYMLOC R after the segment has been loaded (or alternatively modify the .DEF file to change the attribute for debugging).

Preparing a VxD for Debugging

Normally you prepare a VxD by assembling with the /Zi switch and linking with the /CO switch. However, this does not work with some VxDs. If you are debugging a VxD and are having trouble with the source lines matching up with the generated code then read on.

To properly debug a VxD that is being built with the CodeView switch (/CO) on the link line, you may have to alter your source files so that PCODE segments are always placed before INITCODE or REALCODE in the source files. This is not the case with several sample VxD's supplied with the DDK.

If this re-organization is not reasonable in your situation, you can use the MSYM utility to extract the symbol and line number information from the .MAP file. To do this, do the following steps:

1. Assemble with the /Zi or /Zd switch.
2. Link with the /MA /LI switches. (DO NOT USE /CO.)
3. Run MSYM.EXE by entering MSYM program-name (with NO extension).
4. Place the .SYM file created by MSYM.EXE in the same directory as your VxD.
5. Place a "LOAD = path/vxd-name.extension" statement in the WINICE.DAT file.
6. Make sure that you have reserved enough memory for the symbol information and all of the source files with the **SYM** directive in WINICE.DAT.

WLDR Program and Symbol Loader

The SoftICE/W utility WLDR.EXE is used for loading Windows and MS-DOS programs, symbols and source files. WLDR.EXE can be run from a DOS virtual machine to load MS-DOS programs or from Windows to load Windows programs. WLDR.EXE will load your program, its symbols and source into memory, then it will transfer control to SoftICE/W for debugging. WLDR works for .EXE files or .SYM files, including .SYM files for VxDs. WLDR also supports an indirection file on the command line for loading symbols for multiple DLLs. The syntax of the command is:

```
[@filename;] application-file-name application-command-line
```

where *filename* is a text file that contains the full path name of each DLL to load, one name per line.

Loading a Windows Program for Debugging

To load a Windows program, WLDR.EXE must be run from Windows. If the WLDR icon is not already installed, first install WLDR's icon on a Windows menu from the Windows program manager by selecting *File* followed by *New*, then choose *Program Item*. Fill in the command line field with the full path and file name of WLDR.EXE. After the icon is installed, double click on the icon to run WLDR.

When run, WLDR prompts the user for the file name of the program to be loaded and any command line parameters for the Windows program. You can choose *browse* to navigate through the file system to find the program that you want to load. WLDR remembers the last five programs that you have debugged in a file called WLDR.INI.

Other WLDR options let you load symbols only (no program is loaded) or load the program and symbols without loading source files.

If your program has associated DLLs (Dynamically Loadable Libraries) that you want to debug symbolically, you can use one of the following methods to bring in symbols and source:

1. Go to File Properties and enter a command line listing all the DLLs, separated by spaces, and end the list with a semicolon followed by the program name. For example:
   - ◊ DLL1 DLL2 DLL3; PROG.EXE
2. Specify these DLL's in the WINICE.DAT file using the **LOAD** statement.

3 Pre-load each of the DLLs separately with WLDR before you load your main program.

Note: If you are debugging a program frequently you can select the *make icon* option to create an icon that will quickly load a single program. You can then go to File Properties to add DLL names to the command line if desired.

Note: If your program has too many source files to fit into symbol memory, you can instruct WLDR to selectively load source files by using a .SRC file. See page 28 for more information.

Loading a DOS Program for Debugging

To load a DOS program for debugging, WLDR.EXE must be run from a DOS virtual machine. WLDR.EXE can load programs with no source or symbols, programs with source and symbols, or source and symbols without the program.

For symbolically debugging application programs and T&SR programs, you will typically use WLDR.EXE to load the program, symbols and source files in one step. For debugging T&SRs loaded before Windows, DOS loadable device drivers, ROMs and other system components you will typically use WLDR.EXE to load only the symbols and source files, then locate the symbols with the **SYMLOC** command.

The syntax for WLDR.EXE is:

`WLDR` *`program-name`* `[ .SYM | .`*`extension`*`]`

If no file extension is supplied, WLDR.EXE loads your program, symbols and source files by doing the following:

1 Loads program symbols and source into the reserved symbol memory.

2 Loads *program-name*.EXE into memory at the location it would have loaded if it had been loaded directly from the DOS prompt.

3 Brings up SoftICE/W with the instruction pointer at the first instruction of your program.

If .SYM is specified, then only step number 1 above will be performed, which loads only the symbols and source. Specifying .SYM does not mean a file with the extension .SYM; it just means you want only symbols and source loaded.

If the extension .EXE or .COM is specified, then only steps 2 and 3 will be performed, which loads only the program.

Debugging a Program at Source Level

This section describes the basics of debugging a DOS or Windows program at source level. It assumes that the program has been built with symbols and source.

The first step in debugging an application program at source level is to load it with WLDR.EXE, the SoftICE/W symbol table and source loading utility. WLDR.EXE should be run from Windows to debug a Windows program or from a DOS VM to debug a DOS program. WLDR.EXE loads your program into memory with the instruction pointer at the very first instruction of your startup code.

WLDR.EXE prompts for path names if source files can not be found on the current directory or the directories referenced in the symbol tables. If your source files are on other directories, use the **SRC** environment variable to specify the other directories. Set the **SRC** environment variable from DOS before running Windows so the environment variable is valid in all virtual machines. The syntax for setting the **SRC** environment variable is:

SET SRC =*path1*;*path2*;...;*pathn*

Path1 through *pathn* are directories that contain source files referenced in symbol tables that you will be loading.

Note: If your program has too many source files to fit into symbol memory, you can instruct WLDR to selectively load source files with a .SRC file. See page 28 for more information.

Special Note for Debugging C Programs

If you are debugging a C program using Microsoft or Borland tools, then the source file that contains main or WinMain is displayed. At this point you must press the **F8** or **F10** key to cause your program to execute to main or WinMain. In most cases you want to do this.

In the rare case that you want to debug your startup code of a C program, press the **F3** key to go into mixed mode, or assembly mode if you do not have source loaded for your startup code. Now you can debug through your startup code. If you want to get to main or WinMain after pressing **F3**, then enter **G main** or **G WinMain**.

Single Stepping and Tracing

Once your program is loaded you can perform standard debugging operations like stepping through code. To single step, press the **F8** key; to program step, press the **F10** key. In C or other high level languages, pressing **F10** will step to the next source line. **F8** will trace into a called function if source is available for that function, otherwise it will act the same as **F10**.

Point-and-Shoot Break Points

One of the most common debugging operations is setting and clearing sticky break points. With SoftICE/W you can set point-and-shoot break points by placing the cursor on the desired instruction, then pressing the **F9** key. These are sticky break points that can be toggled on and off with the **F9** key. The cursor must be in the code window before setting point-and-shoot break points. If the cursor is in the command window, press the **F6** key to move the cursor to the code window.

If you wish to go to a particular line without setting a sticky break point, place the cursor on the desired line and press the **F7** key. Alternatively, you can enter **G *.line-number*** or **G *symbol-name***.

Navigating Through Your Source Files

While setting point-and-shoot break points or while browsing it is often necessary to move through the source in the code window. SoftICE/W gives you several ways to view different source code in the code window. For local movements use the arrow keys and *PageUp/ PageDown* keys. If you want to start displaying in the current file at a different line number, enter **U *.line-number***. The specified line is displayed as the top line of the code window. If you want to move the display to a particular function enter **U *symbol-name***, where *symbol-name* is the name of the function. This will cause the source code for the specified symbol or routine to be displayed at the top of the code window.

A convenient method for moving within a source file is the source search command (**SS**). Entering **SS *string*** will move the source display to the next occurrence of the specified string.

If you want to view a different source file use the **FILE** command. Enter **FILE *file-name*** to display the specified file. The file name can be a partial-name. If you don't know the name of the source file, entering **FILE *** will display all of the files loaded for the particular symbol table.

Range Break Points

One of SoftICE/W's most powerful features is the memory range break point. These are set from the command line with the **BPR** command. Memory range break points will tell SoftICE/W to pop up whenever memory in the specified range is accessed. This is especially useful if you believe a memory area is being corrupted, but you don't know what program or routine is responsible.

We will experiment with memory ranges by loading a test program with WLDR if you have not already done so. If your test program is written in C, press **F8** to get to main. Now enter **BPR DS:0 DS:200 W**. This will set a memory range break point over the first 200H bytes of your data segment for write. Now press **F5** to run your program. SoftICE/W will pop up whenever anything writes into this area. If it is a Windows program, then chances are you will see assembly language in the SoftICE/W code window as Windows manipulates data in your program's data segment.

To disable the range, enter **BC ***. Now press **F5** to continue running your program.

Back Trace History

Another powerful feature of SoftICE/W is back trace history. When back trace history is enabled, SoftICE/W collects every instruction executed in the specified memory range into a large circular buffer (the buffer size is determined by the **TRA=** statement in the WINICE.DAT initialization file). Back trace history is especially useful if you have a program that crashes, but you don't know where to start looking. If you have a back trace range set over your program's code area when the crash occurs, you can single step backwards in time so you can see where your program lost track.

To experiment with back trace ranges, load a program with WLDR. If your program is written in C, then press **F8** to get to main. Set a back trace range over your code segment by entering **BPR CS:0 CS:FFFF T.** If it is a Windows program, the FFFF limit will most likely be too high; you must use the **LDT** command to get the actual limit. For example, entering **LDT CS** might display an **LDT** line that looks something like:

```
94D Code16 Base=80628CC0 Lim=0000179F DPL=1 P RE
```

The limit in this example is 0000179F.

Now enter **LDT CS** and note the limit that is returned.

Notice that the syntax for a back trace range is the same as that of the range break point above, except that we used a **T** for trace instead of a **W** for write.

Now set a point-and-shoot break point somewhere in your program, then press **F5** to let your program run. SoftICE/W should pop up as soon as the point-and-shoot break point goes off.

Enter **TRACE 1**. This puts you in SoftICE/W's trace simulation mode. It allows you to debug through the back trace history buffer. The parameter "1" tells SoftICE/W that you want to start at the most recent instruction.

To trace backward in the back trace history buffer press **Alt F8**. This will single step one instruction back in time. Press **Alt F8** several times. Now try **Ctrl F8**, which will single step one instruction forward.

Enter **TRACE OFF** to exit trace simulation mode.

Enter **BC *** to disable the back trace and the point-and-shoot break point.

See *Back Trace Ranges* on page 73 for a detailed description of back trace ranges.

Debugging a DOS T&SR in a Virtual Machine

There are two scenarios for debugging a DOS T&SR in the Windows environment. The first case is if the DOS T&SR is loaded in a DOS VM after Windows has been run. In this case, you load and begin debugging your init code just as if it were a DOS application program.

The second case is if the T&SR program is loaded prior to running Windows. In this case you must load the symbol table separately with the WLDR command from the DOS prompt in a Windows VM.

You prepare your T&SR for debugging by placing the appropriate compiler and linker switches just as if you were preparing to debug the program with an application debugger. You can then load the T&SR's symbol table and source files into SoftICE/W's symbol memory with the WLDR utility or you can cause the symbol table and source files to be pre-loaded with the **LOAD** statement in WINICE.DAT (see *WINICE.DAT Initialization File* on page 7 for more information about WINICE.DAT).

To use the WLDR.EXE utility to load your symbol table and source files you must enter the following from a DOS VM:

- **WLDR** *program-name***.SYM**

When **.SYM** is specified, WLDR will extract and load the symbols from your .EXE file. Next it will load into memory all of the source files referenced in the symbol table.

Your symbols must now be located to the address that your T&SR program had been previously loaded. To do this you must first pop up SoftICE/W with its hot key sequence. Now enter **MAP** to display a memory map of the DOS environment. The **MAP** command's display will show the base segment of your T&SR's. The file name of your T&SR should be displayed one or more times in the far right column of the **MAP** command's display. The first entry is usually the environment, the second entry is usually the resident part of your T&SR. Note the starting segment of your T&SR; it is displayed in the left column of the **MAP** command display.

Hint: If you are stopping at an embedded INT 1 or INT 3 in your T&SR you can usually locate the symbols by entering SYMLOC CS.

Note: DOS versions prior to 5.0 or environments with a NetWare shell may not display the name of your T&SR if your T&SR has released its environment. You must determine which entry is the main body of the T&SR by its size or ordering with other T&SRs loaded.

Enter **SYMLOC *base-segment*** to adjust the symbols and source references to match the actual addresses in your T&SR.

You can now use the **FILE** command to bring up a source file for your T&SR. At this point you can set point-and-shoot break points to begin debugging your T&SR.

Note: SoftICE/W cannot debug your T&SR's init code or portions of the code that execute before Windows is run. Use SoftICE for DOS to debug these portions.

Debugging a DOS Loadable Device Driver

DOS loadable device drivers are loaded prior to Windows, so you must load the symbol table separately with WLDR from the DOS prompt in a Windows VM.

You prepare your DOS loadable driver for debugging by placing the appropriate compiler and linker switches just as if you were preparing to debug the program with an application debugger. Many drivers are "EXE2BINed" to create a .COM or .SYM file. It is best to leave your DOS loadable driver as an .EXE file while debugging. This gives you the most complete symbol information. If you cannot leave your driver as an .EXE file for some reason, then you must create a .SYM file with the MSYM.EXE utility. See *Loading a DOS Program for Debugging* on page 33 for details.

You can load your loadable driver's symbol table and source files into SoftICE/W's symbol memory with the WLDR utility or you can cause the symbol table and source files to be pre-loaded with the **LOAD** statement in WINICE.DAT (see *WINICE.DAT Initialization File* on page 7 for more information about WINICE.DAT).

To use the WLDR.EXE utility to load your symbol table and source files you must enter the following from a DOS VM:

WLDR *driver-name***.SYM**

When **.SYM** is specified, WLDR will extract and load the symbols from your .EXE file or search for a **.SYM** file. Next it will load into memory all of the source files referenced in the symbol table.

Your symbols must now be located to the address that your DOS loadable driver had been previously loaded. To do this you must first pop up SoftICE/W with its hot key sequence. Now enter **MAPV86** to display a memory map of the DOS environment. The **MAPV86** command's display will show the base segment of your driver. If your driver is a character device, the **MAPV86** command displays the device name in the far right column of the **MAPV86** command's display. If your driver is a block device, then the name is not present. You must determine which block device is yours by the order of loading or the size. Note the starting segment of your loadable driver: it is displayed in the left column of the **MAPV86** command display.

Enter **SYMLOC *base-segment*** to adjust the symbols and source references to match the actual addresses in your driver.

You can now use the **FILE** command to bring up a source file for your loadable driver. At this point you can set point-and-shoot break points to begin debugging.

Hint: If you are stopping at an embedded INT 1 or INT 3 in your driver you can usually locate the symbols by entering **SYMLOC CS**.

Note: SoftICE/W cannot debug your DOS loadable driver's init code or portions of the code that execute before Windows is run. Use SoftICE for DOS to debug these portions.

Hints for System Level Debugging in DOS VMs

There are a few differences between debugging in DOS and debugging in a Windows DOS VM. These are particularly notable when dealing with interrupts.

Many T&SRs, DOS programs and DOS loadable drivers hook interrupt vectors. With SoftICE/W you can debug any part of your T&SR, even portions that are controlling hardware at interrupt time. You can set point-and-shoot break points in your interrupt handlers, then single step from there.

One difference between SoftICE for DOS and SoftICE/W is the use of break points on interrupts. In most case when debugging an interrupt handler you want to get control when the interrupt occurs. For example, you may want control when the timer interrupt occurs. With SoftICE for DOS you would have set a **BPINT 8**. However, under Windows, there are two problems. First, Windows moves the base of the interrupt controller from 8 to 50H, so the actual hardware timer interrupt is 50H, not 8.

Secondly, if you set a **BPINT 50** instead of a **BPINT 8**, SoftICE/W will pop up in Windows protected mode interrupt 50 handler, not in DOS where your interrupt service routine is. Interrupts go to Windows first which executes hundreds or thousands of instructions before control gets to your T&SR.

The easiest solution to this problem is to set a **BPX** break point on the first instruction of the interrupt handler instead of using a **BPINT** command. For example, to set a break point on the first instruction of the timer interrupt handler in the current virtual machine you would enter: **BPX &0:8*4**.

Debugging a Windows Device Driver

A Windows device driver is structured just like a Windows application. However, you often want to start debugging the Windows driver before the Windows interface is up. This is especially true in the case of a display driver.

SoftICE/W lets you start debugging at source level from the very first instruction of your Windows driver. You prepare your driver for debugging just as you would prepare a Windows application, by building with the proper debugging switches for your compiler and linker.

To debug your driver you must do the following:

1. Place a LOAD statement in the WINICE.DAT initialization file or use the /LOAD switch on the WINICE.EXE command line. The syntax for the LOAD statement is:
   - ◊ **LOAD =** *driver-file-name*
2. Place an INT 1 or INT 3 instruction at the start of your driver, or the first instruction that you would like to debug. Windows C programmers can call the Windows API call DebugBreak() to perform an INT 3.
3. Place the **I1HERE ON** or **I3HERE ON** command in your WINICE.DAT initialization file.

SoftICE/W loads the symbol table and all source files referenced in the symbol table. SoftICE/W will pop up at the INT 1 or INT 3 instruction of the driver. If you need to eliminate the INT 1 or INT 3 instruction so you don't continue popping up there, you can use the **ZAP** command to replace the INT 1 or INT 3 with NOP instructions before debugging.

Note: Your driver is loaded by Windows from the SYSTEM.INI file just as if you were not debugging.

Debugging a Windows VxD

SoftICE/W lets you start debugging from the very first protected mode instruction of your Windows VxD. You prepare your VxD for debugging just as you would prepare a Windows application, by building with the debugging switches. Use the /Zi switch with MASM5, and the /MA and /LIswitches with LINK386, then run Nu-Mega's MSYM utility on the .MAP file to create a .SYM file.

Before you debug your VxD, you must place a **LOAD** statement in the WINICE.DAT initialization file or use the /**LOAD** switch on the WINICE.EXE command line. The syntax for the **LOAD** statement is:

LOAD = *vxd-file-name*

For example, to load the symbols for MYVXD.386 enter:

- LOAD = C:\WINDEV\MYSTUFF\MYVXD.386

The **LOAD** statement causes SoftICE/W to load the symbol table and source files referenced in the symbol information. However, SoftICE/W will not stop at the first instruction of the VxD.

If you wish to debug initialization code, you must use one of the following methods:

1. Place an INT 1 instruction in your VxD at the point you want to SoftICE/W to pop up. You must also place the command **I1HERE ON** in the **INIT** statement of the WINICE.DAT initialization file. This will cause SoftICE/W to pop up when the INT 1 instruction is executed. If you need to eliminate the INT 1 instruction so you don't continue popping up there, you can use the **ZAP** command to replace the INT 1 with NOP instructions before debugging.

2. Remove the **X** command from the **INIT** string in WINICE.DAT. When SoftICE/W pops up, you can set a break point in your code by using the VxD map, and/or with symbols if you preloaded symbols.

SoftICE/W will not allow debugging of your initialization code segments at source level, but once you get to your actual driver code you can debug at source level.

Note: Your VxD is loaded by Windows from the SYSTEM.INI file just as if you were not debugging.

Note: You can use SoftICE for DOS to debug the real mode initialization of your VxD.

Debugging Multiple Programs At Once

SoftICE/W allows multiple symbol tables to be loaded at the same time. This lets you debug complex sets of system software that may contain several different components including Windows applications, Window DLLs, Windows drivers, VxDs,DOS applications, DOS T&SRs and DOS loadable device drivers.

The symbol tables for system level components and VxDs are typically loaded with the **LOAD** directive in the WINICE.DAT file, and the symbol tables for the applications are loaded along with the applications by using the WLDR utility.

The **TABLE** command gives you a list of all the symbol tables currently loaded and lets you select a different symbol table. When you reach a break point in a program that has a corresponding symbol table, enter **TABLE** followed by the first few characters of the symbol table name. This will change the current symbol table to the one that matches your program.

If you are not sure which table is the current table, enter **TABLE** with no parameters. This will show all loaded tables with the current table highlighted.

You can also switch tables to a symbol table that does not match the code where you are currently executing. This is useful when you want to set a break point in a different program than the one you are currently in. This always works for Windows applications, drivers, DLLs and VxDs. It may not work for programs in DOS VM's if the one where you wish to set the break point is not the VM currently mapped in.

Debugging a Win32s Application or DLL

One of the many features of SoftICE/W is the ability to debug Win32s applications at source level. Due to the complex nature of Win32s, special preparations must be made before debugging can begin.

Note: SoftICE/W will only pop up on processor faults if you are running the debug version of WIN32s. The retail version does not notify kernel debuggers that a fault has occurred.

Preparing a Win32s Application for Debugging

SoftICE/W is not currently equipped to properly read the debug information found in Win32s applications. In order for SoftICE/W to debug your Win32s application at source level, first take the following steps:

1. Compile all modules that you want to debug at source level with debug information. Then link your program with full debug information turned on.
2. Run the provided DBG2MAP utility to produce a detailed map file. See the DBG2MAP Utility section on page 70 for more details on this utility.
3. Run the provided MSYM on the map file created by DBG2MAP.

The above steps will create a .SYM file that SoftICE/W can load. Since SoftICE/W is loading a .SYM file, only source and public symbols will be available. SoftICE/W will have no knowledge of local variables.

Loading 32 bit Symbols

Once you have prepared your Win32s application for debugging, you must load the symbol tables into SoftICE/W. .SYM files can be loaded in two different ways:

1. Preloaded from WINICE.DAT:
   - To preload the .SYM files, place a LOAD32= statement in your WINICE.DAT file. Specify either the .EXE file name or the .SYM file name. For example:
     - ◊ LOAD32=c:\windows\system\win32s\kernel32.dll or
     - ◊ LOAD32=c:\windows\system\win32s\kernel32.sym

   You can also use the LOAD= statement, but this will only work for .EXE files, not for .SYM files. If you try to load a .SYM file with a LOAD= statement, SoftICE/W will assume that it is a 16 bit .SYM file.
2. Loaded from the WLDR utility:

- Run WLDR from Windows and specify the executable name. If you are loading DLL symbols, check the symbols only check box, otherwise WLDR will actually load your DLL into memory, which you probably don't want.

Note: When loading an application through WLDR, the start CS:EIP is not yet present in memory when SoftICE/W breaks. If you are viewing the code in assembly mode, you will see nothing but INVALID's. Single stepping once will page the code into memory.

Loading 32 bit Exports

To load 32 bit exports into SoftICE/W, use the **EXP** directive in WINICE.DAT. The WINICE.DAT file contains sample **EXP** lines for Win32s that are commented out. Just change the directory names to wherever your Win32s is installed and remove the preceding ; from the **EXP** statement.

Example: EXP=C:\WINDOWS\SYSTEM\WIN32S\GDI32.DLL

The **EXP** command lists all exported symbols that SoftICE/W knows about. These symbols can be used in any SoftICE/W expression and are automatically displayed when disassembling code.

When displaying exports in SoftICE/W, if the module is not yet loaded, the segment will be displayed as FE and the offset will be the offset from the 32 bit image base. Once the module is loaded into memory, a selector:offset will be displayed and the offset will now have the image base address added in.

Note: If you are using a version of Win32s prior to 1.15, the system DLLs are in three separate DLLS (GDI32.DLL KERNEL32.DLL, and USER32.DLL.) In Win32s version 1.15, these files have all been rolled into one DLL called W32SCOMB.DLL

DBG2MAP Utility

DBG2MAP is a command line utility that accepts a Win32 (PE) executable file with debug information as input, and emits a .MAP file. This .MAP file can then be run through Nu-Mega's MSYM program to create a .SYM file for use by SoftICE/W. At the present time, .SYM files are the only way for SoftICE/W to load 32 bit symbol tables.

DBG2MAP works with executables produced with Microsoft Visual C++ 32 bit Edition, Borland C++ 4.0, and the Microsoft Win32 SDK compiler. The .SYM files generated by DBG2MAP/MSYM can be loaded into SoftICE/W via a LOAD32= statement in the WINICE.DAT file, or by the WLDR program. When using WLDR to load the symbols, specify the name of the EXE or DLL, not the .SYM file name.

To run DBG2MAP.EXE, type the following from a command prompt:

- DBG2MAP [*switches*] *filename*

  /A   Include arguments in C++ functions names (default: no)

  filename is the name of your Win32 EXE or DLL that contains debugging information.

After DBG2MAP finishes, there will be a .MAP file in the current directory with the same base filename as your EXE or DLL.

The "/A" switch tells DBG2MAP to leave in the arguments from the function names of Microsoft C++ programs. By default, DBG2MAP truncates Microsoft C++ function names starting with the '(' character. If you instruct DBG2MAP to leave in the arguments in the symbol names, the symbols may be long and difficult to type in correctly. This could also lead to size problems as SoftICE is limited in the number of symbols it can load from a .SYM file.

DBG2MAP is a console mode Win32 program. If run under a version of Win32 that supports console mode applications, it will run natively. Otherwise, it uses a bound-in version of Phar-Lap's TNT DOS Extender.

When building with the command line tools, you may experience problems in both the Microsoft and Borland environments. For Borland users, Phar-Lap says that Borland's MAKE.EXE is incompatible with other DPMI tools such as the TNT DOS Extender. Phar Lap recommends using the real mode MAKER.EXE program instead of the protected mode MAKE.EXE.

Microsoft users may have problems when running DBG2MAP from within an NMAKE makefile. This is due to memory conflicts between the DBG2MAP version of the TNT DOS Extender, and the older Phar Lap DOS Extender used in the Microsoft tools (CL.EXE and LINK.EXE). To work around this, we suggest running DBG2MAP from a batch file.

Debugging a Universal Thunk

Universal Thunks are a unique feature of Win32s. Using them with a 32 bit application can seamlessly call down into a 16 bit DLL. The actual transition to get from 32 bit code to 16 bit code is quite messy and involves a transition into the 32 bit kernel, then into the Win32s16.dll and then finally into the 16 bit target DLL.

SoftICE/W includes a feature that will automatically step through the thunk so you won't have to see any of the transition code. To make this function work you must have source code loaded for both the 32 bit side and the 16 bit side. When you are sitting on a call to a universal thunk in 32 bit code and you single step, SoftICE/W will determine that it is a universal thunk and set a break point on the 16 bit side so that control will automatically end up in the 16 bit DLL.

Returning from the thunk is not as seamless, but it's still simple. When you single step the return statement in 16 bit code you will be in Win32s16.DLL. To handle this case, SoftICE/W has a "step until return" command, **P RET**. Invoke this command twice to return control to source level code in 32 bit mode. The first step will go from Win32s16.DLL to the 32 bit kernel and the second step will actually return to source level.

Exploring Windows Internals with SoftICE/W

For many people the easiest way to learn a software product is by experimenting. This section is designed to give people a head start with experimenting in assembly mode with SoftICE/W. This section is specifically designed to show you how to use SoftICE/W to explore Windows internals. It is written with the understanding that the reader has some understanding of 8086 assembly language.

Start by pressing the **Ctrl D** key sequence to pop up SoftICE/W. When you pop up SoftICE/W with **Ctrl D**, or if SoftICE/W pops up because it has hit a break point, you may be anywhere in Windows. The first thing to look at is the line above the code window. SoftICE/W displays the owner or nearest symbol at the far left. If the owner displayed is a Windows program, then the ordinal number of the code segment is displayed in parenthesis. If it is a VxD, the owner is the name of the Vxd. If it is a DOS program, the owner is the name of the DOS program.

At the far right you will see PROT16, PROT32 or VM. This tells you the processor mode. If the mode is VM then you are in a DOS virtual machine, if the mode is PROT32 you are most likely in a Windows VxD, and if the mode is PROT16 you are in a Windows program or the Windows kernel.

If you are in the Windows kernel or a Windows program you can enter **STACK** to see what path Windows has taken to reach the current routine. The **STACK** command walks the stack displaying the nesting of previously called Windows routines.

Other commands that may give you useful information about what Windows is currently executing are **TASK**, **HEAP** and **LHEAP**. **TASK** displays the Windows task queue. One of the Windows programs in this list will be marked with an asterisk. This is the currently running Windows program. **HEAP** displays all of the currently allocated memory blocks in the global heap. **LHEAP** will display all of the allocated memory blocks in the local heap if there is a current local heap.

HEAP can also be used to display the heap of a Windows program or DLL. For example if you enter **HEAP KERNEL**, you will get a display something like:

Han/Sel	Address	Length	Owner	Type	Seg/Rsr
00F5	000311C0	000004C0	KERNEL	ModuleDB	
00FD	00031680	00007600	KERNEL	Code	01
0586	00054220	00003640	KERNEL	Alloc	
0106	00083E40	00002660	KERNEL	Code D	02
010E	805089A0	00001300	KERNEL	Code D	03
0096	80520440	00000C20	KERNEL	Alloc	

Total Memory: 62K

Single Stepping and Execution Break Points

Wherever you pop up in Windows, you can single step. The **F8** key is programmed to single step and the **F10** key is programmed to program step (step over calls and loops). You can even single step through processor mode transitions and interrupt routines.

To set execution break points, you first move the cursor into the code window by pressing **F6**. Use the arrow keys to place the cursor on the instruction that you would like a break point on. Press **F9** to highlight the line and set the execution break point on the instruction. To clear the break point, place the cursor back on the same line and press **F9** again. Press **F6** to move the cursor back to the command window.

Press **Ctrl D** to execute to your break point. SoftICE/W will return to Windows, and Windows will continue running until the break point is hit. At this point SoftICE/W will pop up again. Since you probably set a break point on a random memory location, it may not go off. If not, press **Ctrl D** again to pop up SoftICE/W and enter **BC 0** to clear the break point (this assumes you only have one break point set). Now try experimenting by setting break points with **F9** until you get one to go off.

Another way to set a sticky break point is with the **BPX** command. Enter **BPX** followed by an address or the name of a symbol. While experimenting with Windows, try setting execution break points on some Windows kernel routines. Use the SoftICE/W **EXP** command to get a list of exported symbols. Then enter **BPX *symbol-name*** to set a break point. After you have set a few break points, enter **BL** to list the break points. Entering the **BC *** command will clear all break points that are currently set.

SoftICE/W keeps a history of all break points that have been set. You can use the **BH** command to view or re-set break points that have been cleared. Enter **BH** now. If you would like to re-set an old break point, use the cursor keys to be on the line of the break point you would like to re-set. Press the *Insert* key to highlight and select one or more break points. Press *Enter* to set the selected break points and exit from the **BH** command, or *Esc* to exit from the **BH** command without setting any break points.

If you are experimenting by setting break points on kernel routines, you may want to try WINEXEC and BITBLT. Enter **BPX WINEXEC**, and also **BPX BITBLT**. After pressing **Ctrl D** and Windows is running, try executing a Windows program like CALC. This will cause the break point on WINEXEC to go off. After you enter **Ctrl D** again, the program you selected will run until the BITBLT routine is called. When SoftICE/W pops up at the start of the BITBLT routine, you will notice the routine name "BITBLT" is displayed at the top of the assembly language display. At this point enter **STACK**. This will display the nesting of routines called in the sequence to reach BITBLT. Since you did not load the symbol table for the Windows program, you will not get the routine names, but only the addresses.

If you would like to see the call stack of a different Windows program in your system, enter **TASK**. This will give you a list of all Windows programs that are currently running. Notice that the current program is marked with '*' to its left. Now enter **STACK** followed by a program name that is not the current program. This will show the call stack at the last point that the program relinquished control back to Windows.

Now enter **Ctrl D** several times and each time SoftICE/W will come right back. Enter the **STACK** each time. The call stack will be different if BITBLT is called from a different sequence of routines. Since BITBLT is called frequently in most Windows programs, you will eventually have to clear the break point to continue running Windows.

One thing to keep in mind is that SoftICE/W implements **BPX** break points by placing an interrupt 3 instruction at the break address, so don't try to set a **BPX**-style break point on a data object. If you are unsure, use the **LDT** command to see if it is in data or code. Enter **LDT *selector*** where *selector* is the segment portion of the address. The second column of the **LDT** display will show you the segment type. If it is CODE16 or CODE32, then a **BPX**-style break point is safe at this location.

Note that it is always safe to set a break point on execution on exports from the **EXP** command since they are all code entry points.

Don't be afraid to experiment with single stepping and setting break points anywhere in the Windows environment. See *Using Break Points* on page 56 for details on setting more powerful SoftICE/W break points.

Exploring the Windows/DOS Transition

An interesting series of events to watch is the transition from Windows to DOS during a DOS function call by a Windows program. When a Windows program accesses disk files, it usually makes DOS function calls through interrupt 21H. The following paragraphs show how to break on these calls to DOS and then show the transition between protected mode where the Windows program is running and Virtual 86 mode where DOS is running.

Pop up SoftICE/W from Windows using **Ctrl D**, then set a break point on interrupt 21 by entering **BPINT 21**. Enter **Ctrl D** to return to Windows, then run a Windows program like calculator. Windows actually makes DOS interrupt calls, so you'll be seeing Windows' INT 21 calls. Each time SoftICE/W pops up, notice which DOS function call the Windows program is executing. This is displayed next to the INT 21 instruction in the code window.

Press **Ctrl D** a few times and SoftICE/W will immediately pop up each time as new DOS calls occur. Now enter **BPX @ &0:21*4**. This will set a break point in the primary VM at the place where the interrupt 21 will execute when control eventually gets to the VM. The "0:21*4" is the interrupt vector for INT 21, the "&" tells SoftICE/W that the 0 is a virtual 86 segment and the "@" means address pointed to, or contents of.

Now enter **Ctrl D**, and SoftICE/W will pop up in the VM at the first instruction of DOS or at the interrupt 21 handler of a T&SR that is intercepting DOS calls. The name of the T&SR or the name DOS is displayed in the line above the code window. Windows may have executed several hundred instructions between the time the Windows application executed the INT 21 and the time control enters DOS. In future experimentation you may want to single step through this entire transition.

Now enter **U @ SS:SP**. This will disassemble at the return address that will execute when DOS eventually performs an IRET. Notice that this is in the ROM BIOS. The first instruction that will execute is ARPL. In fact this is the last instruction that will execute in the VM. The ARPL will cause an invalid op code fault that will send control back to Windows. This is a trick that Windows uses to make the transition from the VM to protected mode code. Now enter **D @SS:SP**. Look at the ASCII portion of the data window. Notice that the first instruction is a "c". That just happens to be the op-code for ARPL. This "c" could be anywhere in the ROM. In most BIOS's it turns out that control actually returns into the ROM's copyright message.

Press the **F5** key several times (this is equivalent to entering **Ctrl D**). You should bounce back and forth between the two break points that are set. Occasionally you may not go back to the INT 21 in the Windows program, because sometimes Windows does not actually perform an INT instruction to call DOS in the VM. If you look at the EAX register in the register window you will notice it changing each time you pop into SoftICE/W as different values are placed in AH to call DOS with different functions.

Enter **BC *** to clear the two break points before proceeding.

Setting Break Points in VxD's

We have not yet experimented with any 32-bit Windows code. To do this, we will set a break point in a VxD. Before we begin, navigate through Windows to a menu that holds the DOS icon. Pop up SoftICE/W and press **Alt F3**. This will close the code window. Now enter **VCALL**. This will display a list of VxD service routines. Most of these routines are located in Windows supplied VxDs. These are actually vectors to routines, but SoftICE/W automatically performs the indirection for you when you try to set a break point on one of these names.

Press the space bar several times to page through the list, or press *Esc* to stop the display. Now press **Alt F3** to make the code window visible again, then enter **BPX CALL_WHEN_VM_RETURNS**. This will set a break point at a VxD call that will be called when entering and leaving a DOS VM. Now press **F5** to return to Windows. Double click on the DOS icon. This should pop up the SoftICE/W screen. The far right of the line above the code window should show PROT32. This means we have popped up in 32-bit code.

Press **F5** to continue. At the DOS VM command prompt enter **EXIT** to close the VM. This should bring you back to the same break point in SoftICE/W. Now enter **BC *** to clear the break point.

Protected Mode Level Transitions

Windows uses two different 386/486 protection levels. Protection levels is a mechanism that lets less trusted code have less privileges than more trusted code. In Windows enhanced mode, VxDs run at level 0 (most trusted level) and all other code runs at level 3 (least trusted level).

When Windows code calls a VxD it must make a level transition from level 3 to level 0. This transition is performed by calling into a segment filled with INT 30H instructions. The interrupt 30H entry in the IDT is an interrupt gate that causes the processor to transition from level 3 to level 0. The INT 30H interrupt service routine vectors to the appropriate VxD call based on the offset that was called in the segment filled with INT 30H instructions.

Note that in Windows 3.0 this mechanism worked slightly differently. In Windows 3.0, the two levels used were level 0 and level 1. The mechanism to transition to level 0 to make VxD calls was a segment filled with HLT instructions instead of a segment filled with INT 30H instructions. The HLT caused a GP and the GP handler vectored to the appropriate routine based on the offset that cased the GP. The change to INT 30H not only improves performance, but slightly simplifies a very complicated GP handler.

So if you are single stepping through Windows and you get a screen filled with INT30H instructions, don't panic; it's normal.

When Windows wants to make a transition from level 3 to level 0 from a DOS VM, then a different mechanism is used. Instead of an HLT instruction, an ARPL instruction is used.

Memory Addresses in Windows

One of the most confusing things about Windows in enhanced mode is the different 386/486 processor modes that Windows uses. 16-bit Windows programs, 32-bit VxD's and programs in DOS VMs are all different. These next few sections explain these different addressing modes and how Windows addresses memory in general.

Different Modes of Enhanced Windows

When you pop up SoftICE/W, the instruction pointer may be anywhere in Windows or in a DOS VM. You can pop up in any one of the three addressing modes supported by Windows. These modes are 16-bit protected mode, 32-bit protected mode, and 8086 virtual address mode. SoftICE/W displays the 386/486 addressing mode on the line above the code window at the far right. SoftICE/W displays PROT16, PROT32 or VM to identify the three addressing modes.

16-Bit Protected Mode

Windows programs and Windows itself are 16-bit protected mode programs. In 16-bit protected mode, the segment portion of the address is a selector. A selector is an index into a lookup table. The address is calculated by extracting a base address from the lookup table and adding it to the offset portion of the address. There are two lookup tables: the LDT(local descriptor table) and the GDT(Global Descriptor table). The processor decides to use the LDT or GDT based on bit two of the selector: if bit two is a 1 then the LDT is used, otherwise the GDT is used. Selectors for Windows programs are LDT selectors and selectors for VxD's are GDT selectors.

The segments are usually 64K or less. The actual length of a segment can be displayed with the **LDT** or **GDT** command.

32-Bit Protected Mode

Windows VxD's are 32-bit protected mode programs. Like 16-bit protected mode programs, the segment portion of the address is treated as a selector. However there are typically only two selectors used: 28 for code, and 30 for data and stack.

These two selectors each address the entire 4 gigabyte virtual address space of the processor.

The processor interprets instructions differently in 32-bit protected mode than it does in 16-bit protected mode or virtual 8086 mode. SoftICE/W decides how to disassemble instructions based on the type of the selector specified. Therefore if you try to disassemble some 16-bit code with selector 28, the disassembly will not be accurate.

It is possible to develop 32-bit Windows applications. You must have a translation layer that converts 32-bit API calls to 16-bit Windows API calls. As of this writing, Watcom and MetaWare provide tool kits for developing 32-bit Windows programs.

8086 Virtual Address Mode

Any code executing in a DOS virtual machine is using 8086 virtual address mode. In 8086 virtual address mode the processor calculates an address by multiplying the segment portion by 16, then adding the offset. When you pop SoftICE/W in a DOS virtual machine you will see 'VM' displayed on the line above the code window.

Overriding the Default SoftICE/W Addressing Mode

SoftICE/W maintains two default addressing modes: one for the code window and one for the data window. The default for the code window is always the mode that existed when SoftICE/W popped up. The default mode for the data window is 32-bit protected mode until it is changed.

To change the default mode in the code window you must use an override character when entering the **U** command to disassemble code or display source. To change the default in the data window you must use an override character when displaying or editing data with the **D** or **E** commands.

There are two override characters:

&	Force address to be an 8086 virtual mode address.
#	Force address to be a protected mode address. (SoftICE/W determines whether it is a 16-bit or 32-bit protected address from the attributes in the LDT or GDT entry for this selector).

Virtual Addresses

When you use an address in SoftICE/W it is almost always a virtual address. It is not the actual address that goes out on the processor bus (called the physical address). The virtual address and physical address are different because Windows uses paging to alter the physical/ virtual relationship.

To get from a virtual address to a physical address, the 386/486 must perform two separate translations. First it must combine the segment or selector and the offset to get a linear address. If it is an 8086 style segment:offset address this is done by multiplying the segment by 16, then adding the offset. If it is a protected mode selector:offset address then the linear address is calculated by adding the offset to the base address found in the GDT or LDT.

Once the linear address is calculated, the 386/486 uses a set of lookup tables to get the actual physical address. These lookup tables are called page tables.

The paging mechanism allows physical memory to be divided into 4K chunks. Page tables can be set up so that any 4K chunk of physical memory can reside at any address in the 4 gigabyte virtual address range.

Most of the time you can ignore physical addresses. However there are times when you must know physical addresses, and there are also times when you may get confused if you do not understand the concept of physical and virtual addresses. Below is a memory map showing how Windows partitions the 4 gigabytes of virtual address space. This is followed by several sub sections that describe each component of the Windows memory map.

Sample Enhanced Windows Memory Map

00000000 - 000FFFFF	- Current Virtual Machine
00000000 - 000FFFFF	- Windows & Windows Programs (overlap with VM)
00400000 - 7FFFFFFF	- Physical memory contiguously mapped
80000000 - 803FFFFF	- Windows VxD's
80500000 - 80FFFFFF	- Windows Programs
81000000 - FFFFFFFF	- DOS VM's

The above memory map is not guaranteed, but for reference only. All above addresses are in hexadecimal.

The entire 4 gigabyte virtual address space can be accessed in SoftICE/W with GDT selector 30. For example, to display memory in the monochrome video adapter you would enter: D #30:400B0000. The # tells SoftICE/W that this is a protected mode selector and will change the data window default mode to protected mode.

To look at any physical address, use the **PHYS** command as follows:

- PHYS B8000

This shows you a list of virtual addresses that you can precede with #30: to get a physical address.

To see how Windows partitions memory on your computer you can use the SoftICE/W **PAGE** command. See the description of the **PAGE** command on page 123 for more details.

Current Virtual Machine

The first area shown in the Windows memory map above (00000000 - 000FFFFF) holds the current DOS virtual machine. This area is paged in and out with other DOS virtual machines as the Windows user switches VM's. The current virtual machine is addressable from protected segment 30 at two different places, at 0 and at its permanent location somewhere above 8100000. However, you would normally access this memory with virtual 8086 addressing. Also note that Windows pages memory to disk if it does not have enough physical memory so that if you try to look at memory owned by a DOS VM with SoftICE/W, the memory may not currently be paged in. In this case the data window will be filled with '?'.

Also note that the initial VM (which is occupied by loadable drivers and T&SRs that were loaded prior to running Windows) generally has the same physical and virtual addresses. This rule does not hold past 640K with UMB's and video memory. However, most parts of other VMs do not share this one-to-one relationship between physical and logical memory, since their physical memory is in extended memory. Some of the low areas (usually less than 64K) are shared among virtual machines and will have the same physical and virtual addresses.

Windows and Windows Programs

Windows and Windows programs occupy the virtual area between 00000000 and 000FFFFF and also the area between 80500000 and 80FFFFFF. This memory is almost always accessed using a 16-bit protected selector, as Windows' memory management software is continuously changing the virtual addresses of data objects.

Notice that the first megabyte of the Windows area overlaps with the VM area. This can present particular confusion for people doing system level debugging. This overlap occurs because Windows starts as a DOS program, and the real-mode DOS memory area (lower 1 Meg) becomes VM 1 in Windows enhanced mode.

When Windows switches to a VM other than VM 1, it changes the page table that covers the first four megabytes; each VM gets its own page table so that task switches that involve VMs are quick. Windows also invalidates the LDT so all of the LDT selectors that would be used to access Windows segments are not available.

The net result for SoftICE/W users is that most of Windows is invisible when you pop up in a VM. If sufficient memory is available, Windows is still in the same memory it occupied prior to the task switch to the VM. It is simply not addressable.

There are some hints for dealing with this problem below in the section entitled "LDT Addresses".

Physical Memory

Physical memory is mapped contiguously from 00400000 to 7FFFFFFF. This may seem a bit confusing at first but it is quite simple and useful. If you want to look at a particular physical memory address, you simply add 0040000 to that address.

For example, if you would like to look at VGA graphics memory that is at physical address 000A0000 you can always view it with the virtual address 004A0000

Windows VxD's

Windows VxD's are mapped from 80000000 to 803FFFFF. This is one area that it is safe to use the 32-bit virtual addresses all of the time. Windows does not change VxD virtual addresses and they are always available through GDT selectors 28 and 30.

Windows Programs and DOS VM's

Windows normally allocates several megabytes of virtual address space for Windows programs starting at 80400000. Following this is memory for DOS VM's. When a virtual machine is scheduled by the Windows kernel it is also mapped to 0. At this point the VM is visible in two virtual address ranges; one starting at 0, the other starting at the VM's permanent address range above 80400000.

LDT Addresses

Windows programs are generally addressable from LDT selectors. Unfortunately, there are times when the LDT is not valid. This usually happens when a DOS VM other than VM 1 is currently running. So if you pop up SoftICE/W at a place where there is no valid LDT, you will not be able to dump memory or disassemble in Windows programs.

This memory is usually mapped in somewhere between 00000000 and 03FFFFFF but the selectors are not valid and you will typically not know the virtual address.

If you have a symbol table and source loaded for a Windows program, you can set break points using symbols or point-and-shoot break points with source.

There are a few SoftICE/W commands that require a valid LDT. These are **HEAP**, **MOD**, **LHEAP**, **LDT**, and **TASK**.

4 Commands

Introduction

SoftICE/W has break point capability that has traditionally only been available with hardware debuggers. The power and flexibility of the 386/486 chip allows advanced break point capability without additional hardware.

Break points can be set on memory location reads and writes, memory range reads and writes, program execution, port accesses, and ranges of Windows messages. SoftICE/W assigns a hexadecimal number, from 0 to 1F, to each break point. This *break-number* is used to identify break points when you set, delete, disable, enable, or edit them.

All of the SoftICE/W break points are sticky. That means they don't disappear automatically after they've been used; you must intentionally clear or disable them using the **BC** or the **BD** commands. When break points are cleared, they can be recalled with the **BH** command which displays a break point history. This history is saved in the WINICE.BRK file when SoftICE/W is exited. SoftICE/W can handle 32 break points at one time. There is a limit on break points on memory location (**BPM**s), of which you can only have four, due to restrictions of the 386/486 processor.

Break points can be specified with a *count* parameter. The *count* parameter tells SoftICE/W how many times the break point conditions should be ignored before the break point occurs.

In general, any SoftICE/W break point can be set anywhere in memory. The principle exception to this is that I/O break points cannot be set within VxD code. Also there are small areas of memory that memory range break points can not span (see the description of the **BPR** command on page 60 for more information on these areas). These include the GDT (global descriptor table), the IDT (interrupt descriptor table) and level 0 stacks. SoftICE/W will warn you if you try to set a range over the GDT or IDT, but not level 0 stacks. Level 0 stacks are allocated by Windows in system data areas. They will never occur in a user program.

Using Break Points

Setting Break Points

BPM, **BPMB**, **BPMW**, **BPMD**	Set break point on memory access or execution
BPR	Set break point on memory range
BPRW	Set multiple range break points on Windows program or code segment.
BPIO	Set break point on I/O port access
BPINT	Set break point on interrupt
BPX	Set/Clear break point on execution
BMSG	Set break point on Windows message
CSIP	Set CS:EIP (instruction pointer) range qualifier

BPM, BPMB, BPMW, BPMD

Set a break point on memory access or execution.

Syntax

BPM[*size*] *address* [*verb*] [*qualifier value*] [*debug-reg*] [C=*count*]

size	B, W, or D

.

B	Byte
W	Word
D	Double Word

The *size* is actually a range covered by this break point. For example, if double word is used, and the third byte of the double is modified, then a break point will occur. The *size* is also important if the optional *qualifier* is specified (see below).

verb	R, W, RW, or X

.

R	Read
W	Write
RW	Reads and Writes
X	Execute

qualifier	EQ, NE, GT, LT, or M

.

EQ	Equal
NE	Not Equal
GT	Greater Than
LT	Less Than
M	Mask (a bit mask is represented as a combination of 1's, 0's and X's. X's are don't-care bits.)

These *qualifiers* are only applicable to the read and write break points, not the execution break point.

value	A byte, word, or double word *value*, depending on the size specified.
debug-reg	DR0, DR1, DR2 or DR3.

Comments

If a *verb* is not specified, **RW** is the default.

If a *size* is not specified, **B** is the default.

If a *debug-reg* is not specified, SoftICE/W will use the first available debug register starting from **DR3** and working backwards. You can ignore the debug register unless you're debugging an application that uses debug registers itself, such as a debugging tool.

All of the *verb* types except **X** cause the program to execute the instruction that caused the break point. The current CS:EIP will be the instruction after the break point. If the *verb* type is **X**, the current CS:EIP will be the instruction where the break point was set.

If **R** is specified, then the break point will occur on read accesses and on write operations that do not change the value of the memory location.

If the *verb* is **R**, **W** or **RW**, executing an instruction at the specified *address* will not cause the break point to occur.

If **BPMW** is used, the specified *address* must start on a word boundary. If **BPMD** is used, the specified *address* must start on a double word boundary.

BPM break points are set on virtual addresses. This means they will always go off if their address is accessed. This is different than **BPX** and **BPR** break points that are dependent on the page table that was active when they were set.

BPM break points are not affected by swapping or discarding and reloading by Windows.

Examples

BPM ES:DI+1F W EQ 10 C=3

This command defines a break point on memory byte access. The third time that 10 hexadecimal is written to location ES:DI+1F, the break point will occur.

BPM CS:80204D20 X

This command defines a break point on execution. The first time that the instruction at address CS:80204D20H is executed, the break point will occur.

BPMW Foo W EQ M 0XXX XXXX XXXX XXX1

This command defines a word break point on memory write. The break point will occur the first time that location Foo has a value written to it that sets the high order bit to 0 and the low order bit to 1. The other bits can be any value.

BPM DS:80150000 W GT 5

This command defines a byte break point on memory write. The break point will occur the first time that the byte at location DS:80150000H has a value written to it that is greater than 5.

BPR

Set a break point on a memory range.

Syntax

BPR *start-address end-address* [*verb*] [C=*count*]

start-address	Beginning of memory range.
end-address	End point of memory range.
verb	R, W, RW, T or TW.

R	Read
W	Write
RW	Reads and Writes
T	Back Trace on Execution
TW	Back Trace on Memory Writes

Comments

None of the *verb* types cause the program to execute the instruction that caused the break point. The current CS:EIP will be the instruction that caused the break point.

There is no range break point on execution. If a range break point is desired on execution, **R** must be used. An instruction fetch is considered a read for range break points.

If a *verb* is not specified, **W** is the default.

The range break point will degrade system performance in certain circumstances. Any read or write within the 4K page that contains the break point range is analyzed by SoftICE/W. This performance degradation is usually not noticeable, however, degradation could be extreme in exception cases.

The **T** and **TW** *verbs* enable back trace ranges on the specified range. They do not cause break points, but instead log instruction information that can be displayed later with the **SHOW** or **TRACE** commands. See *Back Trace Ranges* on page 73 for more information.

Range break points are always set in the current page tables. If the addresses are below 4 megabytes, the **BPR**s are then tied to the current virtual machine.

Because of this sharing, there are some areas in memory where a range break point is not supported. These include the page tables, the GDT, the IDTs, the LDT, and SoftICE/W. If you try to set a range break point or back trace range over one of these areas, SoftICE/W will warn you.

There are two other data areas that you can not place a range break point over, but SoftICE/W will not warn you. These are Windows level 0 stacks and critical areas in the VMM. Windows level 0 stacks are usually in separately allocated data segments. VMM is a VxD that handles paging in Windows enhanced mode. If you set a range over a level 0 stack or a critical area in VMM you could hang the system.

If the memory that covers the range break point is swapped or moved, the range break point will follow it.

Example

BPR ES:0 ES:1FFF W

This command defines a break point on a memory range. The break point will occur if there are any writes to the memory range between ES:0 and ES:1fff.

BPRW

Set range break points on Windows program or code segment.

Syntax

BPRW *module-name | selector* [*verb*]

module-name	Any valid Windows Module name that contains executable code segments.
selector	A valid selector in a Windows program.
verb	R, W, RW, T or TW

.

R	Read
W	Write
RW	Reads and Writes
T	Back Trace on Execution
TW	Back Trace on Memory Writes

Comments

The **BPRW** command is a short-hand way of setting range break points on either all of the code segments of a Windows program or on a single segment.

The **BPRW** command actually sets **BPR** style break points. If you enter the **BL** command after entering a **BPRW** command, you can see all of the separate range break points that were set.

Each of these range break points must be cleared separately with the **BC** command.

Usage

A common reason to use **BPRW** is setting a back trace history range over an entire Windows application or DLL. This is done by specifying the module name and the **T** verb.

Another use is to break whenever control returns to a program. Use the **R** verb to do this. This works because the **R** verb breaks on execution as well as reads.

Specifying the selector is useful because you do not have to look up the segment limit with the **LDT** or **GDT** commands.

Note

The **BPRW** command can become very slow when using the **T** verb to back trace or when using the command in conjunction with a **CSIP** qualifying range.

Example

BPRW PROGMAN T

This command sets up a back trace range on all of the code segments in the program manager. All instructions executed by the program manager will be logged to the back trace history buffer.

BPIO

Set a break point on an I/O port access.

Syntax

BPIO ***port*** **[*verb*] [*qualifier value*] [C=*count*]**

port	A byte or word value.
verb	R, W, or RW

R	Read (IN)
W	Write (OUT)
RW	Reads and Writes

qualifier	EQ, NE, GT, LT, or M

EQ	Equal
NE	Not Equal
GT	Greater Than
LT	Less Than
M	Mask (a bit mask is represented as a combination of 1's, 0's and X's. X's are don't-care bits.)

value	A byte, word or dword value

Comments

If *value* is specified, it is compared with the actual data value read or written by the IN or OUT instruction causing the break point. The *value* may be a byte, a word, or a dword.

The instruction pointer (CS:EIP) will point to the instruction after the IN or OUT instruction that caused the break point.

If a *verb* is not specified, **RW** is the default.

Windows virtualizes many of the system I/O ports. To get a list of these, use the **TSS** command. This will show each hooked I/O port plus the address of the I/O handler and the name of the VxD that owns it. If you wish to see how a particular port is virtualized, set a **BPX** on the address of the I/O handler.

Note

BPIO break points do not go off in 32-bit VxD code.

Examples

BPIO 21 W NE FF

This command defines a break point on I/O port access. The break point will occur if the interrupt controller one mask register is written with a value other than FFH.

BPIO 3FE R EQ M 11XX XXXX

This command defines a byte break point on I/O port read. The break point will occur the first time that I/O port 3FE is read with a value that has the two high order bits set to 1. The other bits can be any value.

BPINT

Set a break point on an interrupt.

Syntax

BPINT *int-number* [AL|AH|AX=*value*] [C=*count*]

int-number	Interrupt number from 0 - 5F hex.
value	A byte or a word value.

Comments

The **BPINT** command allows breaking on the execution of a hardware or software interrupt or a processor exception. By optionally qualifying the **AX** register with a *value*, specific DOS or BIOS calls can be easily isolated.

If no *value* is specified, a break point will occur when the interrupt specified by *int-number* occurs. This interrupt can be a hardware, software, or internal interrupt.

The optional *value* is compared with the specified register (**AH**, **AL** or **AX**) when the interrupt occurs. If the *value* matches the specified register, then the break point will occur.

When the break point occurs, if the interrupt was a hardware interrupt or processor exception, the instruction pointer (CS:EIP) will point to the first instruction within the interrupt routine. If the interrupt was a software interrupt, when the break point occurs the instruction pointer (CS:EIP) will point to the INT instruction causing the interrupt.

BPINT only works for interrupts that are handled through the IDT. Currently the IDT contains only interrupts 0-5FH. Interrupts above this are dispatched through general protection faults. Also Windows remaps the hardware interrupts from their usual vectors. The primary interrupt controller is mapped from vector 50H-57H. The secondary interrupt controller is mapped from vector 58H-5FH. For example, IRQ0 is INT50H and IRQ8 is INT58H.

Note

If a **BPINT** goes off due to a software interrupt instruction in a DOS VM, then single stepping will go into Windows protected mode interrupt handlers, and then eventually control will return to the DOS VM's interrupt handle. If you want to go directly to the DOS VM's interrupt handler after the **BPINT** has occurred on a software interrupt instruction, enter **G @ &0:**int-number***4**.

Windows only accommodates interrupts 0 - 5FH in its interrupt descriptor table. Interrupts above 5FH cause a general protection violation, and are vectored into DOS VM's by Windows as simulated interrupts. If you want to set a break point on an interrupt above 5FH in a DOS VM, then you must set a **BPX** break point on the first instruction of your interrupt handler. The easiest way to do this is to enter **BPX @ &0:**int-number***4**.

Example

BPINT 21 AH=4C

This command defines a break point on interrupt 21H. The break point will occur when DOS function call 4CH (terminate program) is called.

BPX

Set or clear a break point on execution.

Syntax

BPX [*address*] [C=*count*]

Comments

The **BPX** command allows you to set or clear a point-and-shoot execution break point in the code window. When the cursor is in the code window, the *address* is not required. Instead, when you enter the **BPX** command, the execution break point is set at the address of the current cursor location. If an execution break point has already been set at the address of the current cursor location, then that break point is cleared when you enter the **BPX** command.

If the code window is not visible or the cursor is not in the code window, then the *address* must be specified. If an offset only is specified, then the current CS register value is used as the segment. Use the **EC** command (default key is **F6**) to move the cursor into the code window.

Address must be the first byte of an instruction opcode.

Examples

BPX EIP+10

This sets an execution break point at the instruction 10H bytes past the current instruction pointer (CS:EIP).

BPX .1234

This sets an execution break point at source line 1234.

Special Technical Information

BPX normally places an INT 3 instruction at the break point address. This is used instead of a break point register to make more execution break points available. If your circumstances require the use of a break point register for some reason (code not yet loaded in a DOS VM, for example) you can set an execution break point with the **BPM** command and specify **X** as the *verb*.

If you try to set a **BPX** at an address that is in ROM, a break point register is used instead of an INT 3.

BPX break points in DOS VMs are tied to the VM they were set in. This is normally what you would like while debugging a DOS program in a DOS VM. However, there are situations when you may want the break point to go off at a certain address no matter what VM is currently mapped in. This is usually true when debugging in DOS or a T&SR that was run before Windows. In this case, use a **BPM** break point with the **X** *verb*.

The BPX command will also except a windows module name as a parameter. When a module name is entered, SoftICE/W will set a BPX style breakpoint on every exported entry point in that module. For example, **BPX KERNEL** will set a breakpoint on every function in the Windows module KRNL386.EXE. This can be very useful is you need to break the next time any function in a DLL is called.

Note SoftICE/W supports a maximum of 256 break points so it is possible to run out of break points when using this command.

Default Function Key

F9

BMSG

Set a break point on one or more Windows messages.

Syntax

BMSG *window-handle* [L] [*message-range*] [C=*count*]

window-handle	A 16-bit handle returned when the window is created.
message-range	Either a single Windows message or a range of Windows messages specified by entering the lower message number followed by a space followed by the higher message number. Message numbers can be specified either in hexadecimal or by using the actual ASCII names of the messages, for example, WM_QUIT.
L	Logs messages to the SoftICE/W command window.

Comments

The **BMSG** command allows you to set a break point on the message handler of any window. Break points can be set either on a single message or on a range of messages to a specific window. If the **L** parameter is specified, SoftICE/W will log the messages into the command window instead of popping up when the message occurs.

If no *message-range* is specified, the break point applies to ALL Windows messages.

You can use the **HWND** command to get the window handle.

You may set multiple **BMSG** break points on one *window-handle*, although the ranges may not overlap.

When SoftICE/W does pop up on a **BMSG** break point, the instruction pointer (CS:EIP) will be on the first instruction of the message handling procedure. Each time SoftICE/W breaks, the current message will displayed in the following format:

```
hWnd=xxxx wParam=xxxx lParam=xxxxxxxx msg=xxxx ASCII
string
```

These are the parameters that are passed to the message procedure. All numbers are hexadecimal. The ASCII string is the name of the message, for example, WM_PAINT.

Note

To get a list of all valid Windows messages enter the **WMSG** command with no parameters.

Examples

BMSG 9BC WM_MOUSEFIRST WM_MOUSELAST

This sets a break point on the Window message handler whose window handle is 9BCH. The break point will be triggered on any Mouse message.

BMSG F4C L 0 WM_CREATE

This will cause SoftICE/W to log all messages numbered from 0 up to and including WM_CREATE for the window whose handle is F4CH. SoftICE/W will NOT pop up for these messages. The next time SoftICE/W is popped up, the messages will be displayed in the command window.

CSIP

Set CS:EIP (instruction pointer) memory range qualifier for all break points.

Syntax

CSIP [OFF | [NOT] *start-address end-address* | *Windows-module-name*

NOT	When NOT is specified, the break point will only occur if the CS:EIP is outside the specified range.
OFF	Turns off CSIP checking.
start-address	Beginning of memory range.
end-address	End point of memory range.
Windows-module-name	If a valid Windows module name is specified instead of a memory range, then the range covers all code areas in the specified Windows module.

Comments

The **CSIP** command qualifies break points so that the code that causes the break point must come from a specified memory range. This function is often useful when a program is suspected of accidentally modifying memory outside of its boundaries.

When break point conditions are met, the instruction pointer (CS:EIP) is compared with the specified memory range. If it is within the range, the break point is activated. To activate the break point only when the instruction pointer (CS:EIP) is outside the range, use the **NOT** parameter.

Since Windows programs are typically broken into several code segments scattered throughout memory, a Windows module name can be input as the range. If a module name is entered, then the range covers all code segments in the specified Windows program or DLL.

When a **CSIP** range is specified, it applies to ALL break points that are currently active.

If no parameters are specified, the current memory range is displayed.

Example

CSIP NOT &F000:0 &FFFF:0

This causes break points to occur only if the CS:EIP is NOT in the ROM BIOS when the break point conditions are met.

CSIP CALC

This causes break points to occur only if they are caused by the Windows program CALC.

Back Trace Ranges

Introduction

SoftICE/W can collect instruction information in a back trace history buffer as your program executes. These instructions can then be displayed after a bug has occurred. This allows you to go back and retrace a program's action to determine the actual flow of instructions preceding a break point.

Instruction information is collected on memory accesses within a specified address range, rather than system wide. Using specific ranges rather than collecting all instructions is useful for two reasons:

1. The back trace history buffer is not cluttered by extraneous information that you are not interested in. For example, you may be interested in collecting all the instructions that execute within a particular program in a DOS VM. But you may not be interested in interrupt activity and instruction execution within MS-DOS itself.
2. Back trace ranges degrade system performance while they are active. By limiting the range to an area that you are interested in, you can improve system performance greatly.

SoftICE/W has two methods of utilizing the instructions in the back trace history buffer:

1. The **SHOW** command allows you to display instructions from the back trace history buffer. You must specify how many instructions you wish to go back in the buffer.
2. The **TRACE** command allows you to single step forward or backward in time, replaying the actual program flow. This way you can see the instruction flow within the context of the surrounding program code or source code.

Using Back Trace Ranges Across Code Areas

To use back trace ranges you must do the following:

1. Allocate a back trace history buffer of the desired size by inserting the **TRA** statement in your WINICE.DAT initialization file or by placing the /**TRA** switch on the WINICE.EXE command line. For example, to create a back trace history buffer of 100K you might have the following line in your WINICE.DAT file:

   - **TRA** = 92

   A back trace history buffer of 8K is allocated by default. Anything specified by the /TRA switch is added to the 8K. If 8K is suitable for your needs you do not have to allocate a larger buffer.

   The history buffer size is limited only by the amount of extended memory available.
2. Enable back trace ranges by creating a memory range break point with the **T** verb. For example:

- **BPR** &1000:0 &2000:0 **T**

  When the **T** verb is used with the **BPR** command, it does not cause break points; instead instruction information is logged in the back trace history buffer, and can be displayed later with the **SHOW** or **TRACE** commands

3 Set any other break points if desired.

4 Exit from SoftICE/W with the **X** command.

5 After a break point has occurred, or you have popped SoftICE/W up with the **Ctrl D** hot key sequence, you can display instructions in the buffer with the **SHOW** command. For example, to go back 50 instructions in the buffer and display instructions, enter:

  SHOW 50

6 To replay a series of instructions you must first enter trace simulation mode with the **TRACE** command. To begin replaying the sequence of instructions starting back 50 instructions in the buffer, enter:

  - **TRACE 50**

7 After you have entered trace simulation mode, you can trace through the sequence of instructions by using the **XT**, **XP** or **XG** commands. This allows you to re-enact the program flow. For example, you can single step through the sequence of instructions in the buffer, starting at the instruction specified by the **TRACE** command, by entering the **XT** command several times (default key is **Ctrl F8**). To single step backwards, enter **XT R** (default key is **Alt F8**).

  If you would like to skip over a call use the **XP** command (default key is **Ctrl F10**), or if you would like to go to a specific address use the **XG** command. The **XG** command searches the buffer for the next occurrence of the specified address.

8 To exit from trace simulation mode enter:

  - **TRACE OFF**

9 To reset the back trace history buffer, use the **XRSET** command.

Using Back Trace Ranges Across Data

Back trace ranges can be set up to watch memory accesses on a specified data area. This is useful if a particular data range is being accessed by several different sources and you wish to analyze the pattern of access.

To set a back trace range over a data area you must follow the instructions above shown for back trace range over code areas, except you must use the **TW** verb instead of **T**. When you set a range with the **TW** verb, then every instruction that accesses data in the specified range is collected in the back trace history buffer.

You can analyze this data with either the **SHOW** or **TRACE** commands as described above, however the **TRACE** method can become confusing because there is usually no flow to the instructions collected.

Note: SoftICE for DOS does not allow back trace ranges over data areas. SoftICE for DOS does allow the **TW** attribute, but this enables "coarse" mode, which SoftICE/W does not support.

Special Notes

While in trace simulation mode, most SoftICE/W commands work as normal, including displaying the memory map, and displaying and editing data. The exceptions are:

1. Register information is not logged in the back trace history buffer, so the register values do not change as you trace through the buffer, except for CS and EIP.

2. Commands that normally exit from SoftICE/W do not work while in trace simulation mode. These are **X**, **T**, **P**, **G , HERE, GENINT,** and **EXIT**.

As you peruse instructions from the back trace history buffer with the **SHOW** and **TRACE** commands, you may notice peculiarities in instruction execution. These are caused by jumps into and out of the specified range. These usually occur at jumps, calls, returns and entry points.

When you have a hang problem or other difficult bug that requires back trace ranges, you may have to use large ranges to narrow the scope of the problem. Once you have a better idea of the specific problem area, you can go to smaller ranges.

Large ranges can be very slow, in some cases so slow that you can not run your program. If this occurs, you should break each large range up into smaller ranges, then set them one at a time. Each time you set one of the small ranges, you must duplicate the bug you are trying to solve.

Warning: Ranges that cover interrupt service routines can stop forward execution of your program, and in some cases cause your program or Windows to overflow its stack. Forward execution stops if the interrupting source is frequent enough to have another interrupt present before the previous interrupt has completed. If the interrupt service routine allows nested interrupts (most Windows internal interrupt service routines allow this), then you can get stack over-flows as well.

Implementation Details and Caveats

SoftICE/W back trace ranges are implemented with 386/486 paging. This allows memory to be divided into 4K pages, and each page can be set to have different attributes. SoftICE/W marks pages within the specified range with the not present attribute so a processor exception occurs when the memory is accessed. SoftICE/W fields this exception, then collects the data in the back trace history buffer.

In enhanced mode, Windows is using 386/486 paging quite extensively. SoftICE/W shares the page tables and exception handling with Windows. This adds much more overhead than SoftICE for DOS back trace ranges. Because of this sharing, there are some areas in memory where a back trace range is not supported. These include the page tables, the GDT, the IDTs, the LDT, and SoftICE/W. If you try to set a range break point or back trace range over one of these areas, SoftICE/W will warn you.

Since back trace ranges are a form of memory range break point, any restriction that applies to range break points also applies to back trace ranges. See the description of the **BPR** command on page 60 for more information on this topic.

SoftICE/W only collects addresses of instructions, not the actual instruction itself, in the back trace buffer. Later, when you view the back trace data with the **SHOW** or **TRACE** command, the data is retrieved to show you Dis-assembled instructions. If the program you were tracing has been terminated, or the memory re-used, then the data may no longer be valid and the display may appear garbled. If you had source loaded for the area being traced, it will still be valid.

Manipulating Break Points

SoftICE/W provides several commands for manipulating break points. Manipulation commands allow listing, modifying, deleting, enabling, disabling and recalling of break points. Break points are identified by *break-numbers*, which are hexadecimal digits from 0 to 1F. The break point manipulation commands are:

BD	Disable break points
BE	Enable break points
BL	List break points
BPE	Edit break point
BPT	Use break point as a template
BC	Clear break points
BH	Break point history

BD

Disable one or more break points.

Syntax

BD *list* | *

list	A series of *break-numbers* separated by commas or spaces.
*	Disables all break points.

Comments

The **BD** command is used to temporarily deactivate break points. The break points can be reactivated with the **BE** (Enable break points) command.

You can tell which of the break points are disabled by listing the break points with the **BL** command. A break point that is disabled will have an asterisk (*) after the *break-number.*

Example

BD 1,3

This command temporarily disables break points 1 and 3.

BE

Enable one or more break points.

Syntax

BE *list* | *

*li*Comments

st	A series of *break-numbers* separated by commas or spaces.
*	Enables all break points.

The **BE** command is used to reactivate break points that were deactivated by the **BD** (Disable break points) command.

Note that a break point is automatically enabled when it is first defined or when edited.

Example

BE 3

This command enables break point 3.

BL

List all break points.

Syntax

BL

Comments

The **BL** command displays all break points that are currently set. For each break point, **BL** lists the *break-number,* break point conditions, break point state, and count.

The state of a break point is either enabled or disabled. If the break point is disabled, an asterisk (*****) is displayed after its *break-number.* The break point that most recently caused an action to occur is highlighted.

The **BL** command has no parameters.

Example

BL

This command displays all the break points that have been defined. A sample display, which shows four break points, follows:

```
0)   BPMB #30:123400 W EQ 0010 DR3 C=03
1) * BPR #30:80022800 #30:80022FFF W C=01
2)   BPIO 0021 W NE 00FF C=01
3)   BPINT 21 AH=3D C=01
```

Note that in this example, break point 1 is preceded with an asterisk (*), showing that it has been disabled.

BPE

Edit a break point description.

Syntax

BPE *Êbreak-number*

Comments

The **BPE** command loads the break point description into the edit line for modification. The break point description can then be edited using the editing keys, and re-entered by pressing the *Enter* key. This command offers a quick way to modify the parameters of an existing break point.

Example

```
BPE 1
```

This command moves a description of break point 1 into the edit line and removes break point 1. Pressing the *Enter* key will cause the break point to be re-entered.

BPT

Use a break point description as a template.

Syntax

BPT *Êbreak-number*

Comments

The **BPT** command uses an existing break point description as a template for a new break point.

A description of the existing break point is loaded into the edit line. The break point description can then be edited using the editing keys, and entered by pressing the *Enter* key. The break point referenced by *break-number* is not altered. This command offers a quick way to create a new break point that is similar to an existing break point.

Example

```
BPT 3
```

This command moves a template of break point 3 into the edit line. When the *Enter* key is pressed, a new break point is added.

BC

Clear one or more break points.

Syntax

BC *list* | *

list A series of *break-numbers* separated by commas or spaces.

*Clears all break points.

Example

BC *

This command clears all break points. After this, a **BL** command will show no break points until more are defined.

BH

List and allow you to select previously set break points from the break point history.

Syntax

BH

Comments

The **BH** command is used to recall break points that have been set in both the current and previous SoftICE/W sessions. All saved break points will be displayed in the command window and can be selected using the following keys:

UpArrow	This positions the cursor one line up. If the cursor is on the top line of the command window the list is scrolled.
DownArrow	This positions the cursor one line down. If the cursor is on the bottom line of the command window, the list is scrolled.
Insert	This key selects the break point at the current cursor line, or deselects it if already selected.
Enter	This key sets all selected break points.
Esc	This key exits break point history without setting any break points.

The last 32 break points are saved by SoftICE/W. Each time Windows exits normally, these break points are written to the WINICE.BRK file in the same directory as WINICE.EXE. Every time SoftICE/W is loaded, it reads the break point history from the WINICE.BRK file.

Example

BH

This command allows selection of any of the last 32 break points from current and previous SoftICE/W sessions.

Using Other Commands

Display and Edit Commands

R	Display or change registers
U	Unassemble instructions
D	Display memory in the most recently specified format
DB	Display memory in byte format
DW	Display memory in word format
DD	Display memory in double word format
DS	Display memory in short real format
DL	Display memory in long real format
DT	Display memory in 10-byte real format
E	Edit memory in the most recently specified format
EB	Edit memory bytes
EW	Edit memory words
ED	Edit memory double words
ES	Edit memory short reals
EL	Edit memory long reals
ET	Edit memory 10-byte reals
? or H	Display help information
VER	Display SoftICE/W version number
WATCH	Add watch on byte variable
WATCHB	Add watch on byte variable
WATCHW	Add watch on word variable
WATCHD	Add watch on double word variable
WATCHS	Add watch on short real variable
WATCHL	Add watch on long real variable
WATCHT	Add watch on 10-byte real variable
CWATCH	Clear watch on expression
FORMAT	Change data window format
DATA	Change data window

R

Display or change the register values.

Syntax

R [*register-name* [[=]*value*]]

register-name	Any of the following: AL, AH, AX, EAX, BL, BH, BX, EBX, CL, CH, CX, ECX,DL, DH, DX, EDX, DI, EDI,SI, ESI, BP, EBP, SP, ESP, IP, EIP, FL, DS, ES, SS, CSFS or GS.
value	If *register-name* is any name other than FL, *value* is a hex value or an expression. If *register-name* is FL, *value* is a series of one or more of the following flag symbols, each optionally preceded by a plus or minus sign: • O (Overflow flag) • D (Direction flag) • I (Interrupt flag) • S (Sign flag) • Z (Zero flag) • A (Auxiliary carry flag) • P (Parity flag) • C (Carry flag)

Comments

If no parameters are supplied, the cursor moves up to the register window, and the registers can be edited in place. If the register window is not currently visible, it is made visible. If *register-name* is supplied without a *value*, the cursor moves up to the register window positioned at the beginning of the appropriate register field. For a complete description of editing in the register window, see *Register Window* on page 14 for more information.

If both *register-name* and *value* are supplied, the specified register's contents are changed to the *value*.

To change a flag value, use **FL** as the *register-name*, followed by the symbols of the flag whose values you want to toggle. To turn a flag on, precede the flag symbol with a plus sign. To turn a flag off, precede the flag symbol with a minus sign. The flags can be listed in any order.

Examples

```
R AH=5
```

This command sets the **AH** register equal to 5.

R FL=OZP

This command toggles the **O**, **Z**, and **P** flag values.

R FL

This command moves the cursor into the register window position under the first flag field.

R FL=O+A-C

This command toggles the **O** flag value, turns on the **A** flag value, and turns off the **C** flag value.

U

Unassemble instructions.

Syntax

U [*address*]

Comments

The **U** command displays either source code or unassembled code at the specified *address*. The code will be displayed in the current mode of the code window, either code, mixed or source. Source can be displayed only if it is available for the specified *address*. To change the mode of the code window, use the **SRC** command (default function key = **F3**).

If *address* is not specified, the command unassembles at the address starting at the first byte after the last byte unassembled by a previous unassemble command, or the instruction following the last instruction in the code window.

If the code window is visible, the instructions are displayed in the code window, otherwise they are displayed in the command window. In the command window either eight lines will be displayed, or one less than the length of the command window.

If you wish to make the code window visible, use the **WC** command (default key = **Alt F3**). If you wish to move the cursor to the code window, use the **EC** command (default key = **F6**).

If the instruction is at the current CS:EIP, it is displayed using the reverse video attribute. If the current CS:EIP instruction is a relative jump, it will contain either the string (JUMP) or (NO JUMP) indicating whether or not the jump will be taken. If the current CS:EIP instruction references a memory location, the contents of the memory location will be displayed in the register window beneath the flags field. If the register window is not visible, this value is displayed on the end of the code line.

If a break point is set on an instruction being displayed, then it is displayed using the bold attribute.

If any of the memory addresses within an instruction have a corresponding symbol, then the symbol is displayed instead of the hexadecimal address. If an instruction is located at a code symbol, then the symbol name is displayed on the line above the instruction.

The actual hexadecimal bytes of the instruction can be viewed or suppressed using the **CODE** command.

Example

U EIP-10

This command unassembles instructions beginning at 10 hexadecimal bytes before the current address.

U .121

This command displays source in the code window starting at line number 121.

D, DB, DW, DD, DS, DL, DT

Display memory.

Syntax

D[*size*] [*address*]

size	B, W, D, S, L, or T

B	Byte
W	Word
D	Double Word
S	Short Real
L	Long Real
T	10-Byte Real

Comments

The **D** command displays the memory contents at the specified *address*.

The contents are displayed in the format of the *size* specified. If no *size* is specified, the last *size* used will be displayed. The ASCII representation is also displayed for the byte, word, and double word hexadecimal formats.

For the double word format, data can be specified in two different ways. If the displayed segment is a 32-bit segment the dwords will be displayed as 32-bit hexadecimals (eight hexadecimal digits). If the displayed segment is a 16-bit segment (VM segment or LDT selector) the dwords will be displayed as 16:16 pointers (four hexadecimal digits ':' four more hexadecimal digits).

If *address* is not specified, the command displays memory at the address starting at the first byte after the last byte displayed in the current data window.

If the data window is visible, the data is displayed there, otherwise it is displayed in the command window. In the command window either eight lines will be displayed, or one less than the length of the window.

For floating point values, numbers can be displayed in the following format:

[leading sign] decimal-digits . decimal-digits E sign exponent

The following ASCII strings can also be displayed for real formats.

String	Exponent	Mantissa	Sign
Not A Number	all 1's	NOT 0	+/-
Denormal	all 0's	NOT 0	+/-
Invalid	10 byte only with mantissa=0		
Infinity	all 1's	0	+/-

Example

DW ES:1000

This command displays, in word format and in ASCII format, the memory starting at address ES:1000H.

Also see the **DEX**, **DATA** and **WD** commands.

E, EB, EW, ED, ES, EL, ET

Edit memory.

Syntax

E[*size*] [*address* [*data-list*]]

size	B, W, D, S, L, or T.	

B	Byte
W	Word
D	Double Word
S	Short Real
L	Long Real
T	10-Byte Real

data-list	List of data objects of the specified *size* (bytes, words, double words, short reals, long reals, or 10-byte reals) or quoted strings separated by commas or spaces. The quoted string can be enclosed with single quotes or double quotes.

Comments

If no *data-list* is specified, the cursor moves into the data window and the memory can be edited in place. If the data window is not currently visible, it is made visible. Both ASCII and hexadecimal edit modes are supported. To toggle between the ASCII and hexadecimal display areas, press the *Tab* key. For a complete description of editing in the data window, see *Data Window* on page 16 for more information.

If no *size* is specified, the last *size* used will be assumed.

If a *data-list* is specified, the memory is immediately changed to its new values.

Valid floating point numbers can be entered in the following format:

[leading sign] decimal-digits . decimal-digits E sign exponent

An example of a valid floating point number is -1.123456 E-19 .

Examples

EB DS:1000

This command will move the cursor into the data window for editing. The starting address in the data window will be at DS:1000H, and the data will be displayed in hexadecimal byte format as well as in ASCII. The initial edit mode will be hexadecimal.

EB DS:1000 'Test String',0

This command will move the null terminated ASCII string 'Test String' into memory at location DS:1000H.

ES DS:1000 3.1415

This command will move the short real number 3.1415 into the memory location DS:1000H.

? or H

Display help information.

Syntax

```
?   [command | expression]
```

or

```
H   [command | expression]
```

Comments

The **?** command and the **H** command both display help information.

If no parameters are specified, help displays short descriptions of all the commands and operators.

If *command* is specified, help displays more detailed information on the specified *command*, including the command syntax and an example.

If *expression* is specified, the *expression* is evaluated and the result is displayed in hexadecimal, decimal and ASCII.

Examples

```
? ALTKEY
```

This command displays information about the **ALTKEY** command, including its syntax and an example.

```
H 10*4+3
```

This command displays '43' , '67' and 'C'. These are the hexadecimal, decimal, and ASCII representations of the value of the expression "10*4+3".

Default Function Key

F1

VER

Display the SoftICE/W version number.

Syntax

VER

Example

VER

This command displays the SoftICE/W version number, the Nu-Mega Technologies,Inc. copyright message, then the name of the registered user and the product serial number.

WATCH, WATCHB, WATCHW, WATCHD, WATCHS, WATCHL, WATCHT

Add a watch expression.

Syntax

WATCH [*size*] *expression*

size	B, W, D, S, L, or T.

B	Byte
W	Word
D	Double Word
S	Short real
L	Long real
T	10-Byte Real

Comments

The **WATCH** commands are used to display the results of *expressions*. The results of the *expression* are displayed in the format of the *size* specified. If no *size* is specified, byte will be assumed. The *expressions* being watched are displayed in the watch window. There can be up to eight watch *expressions* at a time. Every time the SoftICE/W screen is popped up, the watch window will display the *expressions'* current values.

Each line in the watch window contains the following information:

- A *watch-number* from 0 through 7. The only purpose of this number is for use by the clear watch command (**CWATCH**).
- The *expression* being evaluated.
- The hexadecimal address of the *expression*.
- The current value of the *expression* displayed in the appropriate format.

If the address corresponding to the *expression* is marked not present in the page tables, then ?? will be displayed in the value field.

Example

WATCHW FooVariable

This command creates a word-size entry in the watch window for the variable FooVariable. A sample of what would appear in the watch window follows:

```
0)  FooVariable     #93D:288           0080
```

This line indicates that FooVariable's current value is 80H and the current address is 93D:288.

WATCHD DS:ESI

This command creates a dword-size entry in the watch window and displays the dword pointed to by the DS:ESI registers as shown below:

```
0)  DS:ESI          #0CE5:0000153B     0704:0000
```

Also see the **CWATCH** and **WW** commands.

CWATCH

Clear a watch expression.

Syntax

CWATCH *list* | *

list	This is a *list* of *watch-numbers* from 0-7 separated by commas. *Watch-numbers* are the numbers displayed on the beginning of each line in the watch window.
*	Clear all watch expressions.

Comments

The **CWATCH** command is used to clear one or more watch expressions from the watch window. After clearing the expressions, the ones still remaining in the window are renumbered sequentially starting at 0. If there are no more watch expressions remaining, the window disappears.

Example

CWATCH 1,3

This command clears the first and third watch expressions from the watch window.

FORMAT

Change the format of the data window.

Syntax

```
FORMAT
```

Comments

The **FORMAT** command is used to change the display format in the currently displayed data window. The formats are changed in the order byte, word, dword, short real, long real, 10-byte real and then starting back at byte. This command is most useful when assigned to a function key. The default function key assignment is **Shift F3**. **Shift F3** is also supported when editing in the data window.

Example

FORMAT

This command changes the data window to the next data format.

Default Function Key

Shift F3

DATA

Change to display another data window.

Syntax

DATA [*window-number*]

window-number	The number of the data window you want to view. This can be 0, 1, 2 or 3.

Comments

SoftICE/W supports up to four data windows. Each data window can display a different address in any format. Only one data window is visible at any time. The **DATA** command is used to change the current data window. Specifying **DATA** without a parameter just switches to display the next data window. The windows are numbered from 0 to 3 and this number is displayed on the right hand side of the line above the data window. If a *window-number* is specified after the **DATA** command, SoftICE/W switches to display that window. **DATA** is probably most useful when assigned to a function key. Its default function key assignment is **F12**.

Example

DATA 3

This command changes the data window to data window number 3.

Display System Information Commands

GDT	Display Global Descriptor Table
LDT	Display Local Descriptor Table
IDT	Display Interrupt Descriptor Table
TSS	Display Task State Segment & I/O port hooks
CR	Display control registers
MOD	Display Windows module list
HEAP	Display Windows global heap
LHEAP	Display Windows local heap
VXD	Display Windows VxD map
TASK	Display Windows task list
STACK	Display a call stack
VCALL	Display VxD calls
WMSG	Display Windows messages
PAGE	Display page table information
PHYS	Display all virtual addresses for a physical address
MAPV86	Display virtual machine memory map
MAP32	Display a memory map for 32 bit modules
HWND	Display information on Windows handles.
CLASS	Display information on Windows classes.
VM	Display information on virtual machines.

GDT

Display the Global Descriptor Table.

Syntax

GDT [*selector*]

selector	This is the starting GDT selector to display.

Comments

This command displays the contents of the Global Descriptor Table. If an optional *selector* is specified, the display will begin at that *selector*. If the starting *selector* is an LDT selector (bit 2 is a 1) SoftICE/W will automatically display the LDT rather than the GDT. At the top of the display, the flat base address of the GDT along with the limit will be displayed. Each line of the display contains the following information:

selector value	The lower two bits of this value will reflect the descriptor privilege level.
selector type:	This can be one of the following:

• Code16	16-bit code selector
• Data16	16-bit data selector
• Code32	32-bit code selector
• Data32	32-bit data selector
• LDT	Local Descriptor Table selector
• TSS32	32-bit Task State Segment selector
• TSS16	16-bit Task State Segment selector
• CallG32	32-bit Call Gate selector
• CallG16	16-bit Call Gate selector
• TaskG32	32-bit Task Gate selector
• TaskG16	16-bit Task Gate selector
• TrapG16	16-bit Trap Gate selector
• IntG32	32-bit Interrupt Gate selector
• IntG16	16-bit Interrupt Gate selector
• Reserved	Reserved selector

selector base	Flat virtual base address of the selector
selector limit	Size of this selector
selector DPL	The selector's descriptor privilege level (DPL), which can be either 0, 1, 2 or 3.
present bit	A 'P' or 'NP' indicating whether the selector is present or not present.
segment attributes	One of the following:

• RW	Data selector is readable and writeable.
• RO	Data selector is read only.
• RE	Code selector is readable and executable.
• EO	Code selector is execute only.
• B	TSS's busy bit is set.

Example

GDT

This command will display the Global Descriptor Table in the command window.

LDT

Display the Local Descriptor Table.

Syntax

LDT [*selector*]

selector	This is the starting LDT selector to display.

Comments

This command displays the contents of the Local Descriptor Table by reading the LDT register. If there is no LDT, an error message will be displayed. If an optional *selector* is specified, the display will begin at that *selector*. If the starting *selector* is a GDT selector (bit 2 is 0) then the GDT is displayed rather than the LDT. At the top of the display, the flat base address of the LDT along with the limit will be displayed. Each line of the display contains the following information:

selector value	The lower two bits of this value will reflect the descriptor privilege level.
selector type	This can be one of the following:

Code16	16-bit code selector
Data16	16-bit data selector
Code32	32-bit code selector
Data32	32-bit data selector
CallG32	32-bit Call Gate selector
CallG16	16-bit Call Gate selector
TaskG32	32-bit Task Gate selector
TaskG16	16-bit Task Gate selector
TrapG16	16-bit Trap Gate selector
IntG32	32-bit Interrupt Gate selector
IntG16	16-bit Interrupt Gate selector
Reserved	Reserved selector

selector base	Flat virtual base address of the selector.

selector limit	Size of this selector.
selector DPL	The selector's descriptor privilege level (DPL), which can be either 0, 1, 2 or 3.
present bit	A 'P' or 'NP' indicating whether the selector is present or not present.
segment attributes	One of the following:

RW	Data selector is readable and writeable.
RO	Data selector is read only.
RE	Code selector is readable and executable.
EO	Code selector is execute only.
B	TSS's busy bit is set.

Example

LDT

This command will display the Local Descriptor Table in the command window.

IDT

Display the Interrupt Descriptor Table.

Syntax

IDT [*interrupt-number*]

interrupt-number	The starting *interrupt-number* to display.

Comments

This command displays the contents of the Interrupt Descriptor Table by reading the IDT register. If an optional *interrupt-number* is specified, the display will begin at that entry. At the top of the display the flat base address of the IDT along with the limit will be displayed. Each line of the display contains the following information:

interrupt number	0 - 0FFH.
interrupt type:	One of the following:

CallG32	32-bit Call Gate
CallG16	16-bit Call Gate
TaskG	Task Gate
TrapG16	16-bit Trap Gate
TrapG32	32-bit Trap Gate
IntG32	32-bit Interrupt Gate
IntG16	16-bit Interrupt Gate

address	Selector:offset of the interrupt handler.
selector's DPL	The selector's descriptor privilege level (DPL), which can be either 0, 1, 2 or 3.
present bit	A 'P' or 'NP' indicating whether the entry is present or not present.

Example

IDT

This command will display the Interrupt Descriptor Table in the command window.

TSS

Display task state segment & I/O port hooks.

Syntax

TSS

Comments

This command displays the contents of the task state segment by reading the task register (TR). The following information is displayed:

TSS selector value	TSS selector number.
selector base	Flat virtual base address of the TSS.
selector limit	Size of the TSS.

The next four lines of the display show the contents of the register fields in the TSS. The following registers are displayed:

```
LDT, GS, FS, DS, SS, CS, ES, CR3
EAX, EBX, ECX, EDX, EIP
ESI, EDI, EBP, ESP, EFLAGS
Level 0, 1 and 2 stack SS:ESP
```

The next portion of the display is the TSS bit mask, which shows each I/O port that has been hooked by a Windows virtual device driver (VxD). For each hooked port, the following information is displayed:

port number	The 16-bit port number.
handler address	The 32-bit flat address of the I/O handler. All I/O instructions on the port will be reflected to this handler.
handler name	The symbolic name of the handler. If symbols are available for the VxD, the nearest symbol will be displayed, otherwise the name of the VxD followed by the offset within the VxD will be displayed.

If we are interested in which VxD has hooked port 21H (interrupt mask register) we would look at the TSS bit mask portion of the TSS display and see something like the following:

```
0021                  800792B4                   PICD+0AF8
```

This indicates that port 21H is hooked by the virtual PIC device and the handler is at offset 800792B4 in the flat code segment. The handler is offset 0AF8H bytes from the beginning of VPICD's code segment.

Example

TSS

This command displays the task state segment in the command window.

CR

Display the control registers.

Syntax

CR

Comments

This command displays the contents of the three control registers CR0, CR2 and CR3, and the debug registers in the command window.

Example

CR

This command will display the control registers in the command window. A sample display follows:

```
CR0=FFFFFFE1
CR2=000CC985
CR3=002FE000
DR1=00000000
DR2=00000000
DR3=00000000
DR6=FFFF0FF0
DR7=00000400
```

MOD

Display the Windows module list.

Syntax

MOD [*partial name]**

Comments

This command displays the Windows module list in the command window. A module is a Windows application or DLL. All 16 bit modules will be displayed first, followed by all 32 bit modules. If a *partial name* is specified, only those modules that begin with the name will be displayed. For each loaded module the following information is displayed:

module handle	A 16-bit handle that Windows assigns to each module. It is actually a 16-bit selector of the module database record which is similar in format to the EXE header of the module file.
pe-header	Selector:offset of the PE File header for that module. Note: A value will only be displayed in this column for 32 bit modules.
module name	The name specified in the .DEF file using the 'NAME' or 'LIBRARY' keyword.
file name	The full path and file name of the module's executable file.

Example

MOD

An abbreviated sample follows:

```
hMod  PEHeader        Module Name  .EXE File Name
0117                  KERNEL       C:\WINDOWS\SYSTEM\KRNL386.EXE
0147                  SYSTEM       C:\WINDOWS\SYSTEM\SYSTEM.DRV
014F                  KEYBOARD     C:\WINDOWS\SYSTEM\KEYBOARD.DRV
0167                  MOUSE        C:\WINDOWS\SYSTEM\LMOUSE.DRV
01C7                  DISPLAY      C:\WINDOWS\SYSTEM\VGA.DRV
01E7                  SOUND        C:\WINDOWS\SYSTEM\MMSOUND.DRV
0237                  COMM         C:\WINDOWS\SYSTEM\COMM.DRV
0000  2987:80756080   W32SKRNL     C:\WINDOWS\SYSTEM\win32s\w32skrnl.dll
12C7  2987:86C20080   FREECELL     C:\WIN32APP\FREECELL\FREECELL.EXE
1FC7  2987:86C40080   CARDS        C:\WIN32APP\FREECELL\CARDS.dll
1FDF  2987:86C70080   w32scomb     C:\WINDOWS\SYSTEM\win32s\w32scomb.dll
```

HEAP

Display the Windows global heap.

Syntax

HEAP [FREE | *module-name* | *selector*]

FREE	If FREE is specified, only heap entries marked as free will be displayed.
module-name	This is the name of the module. If supplied only heap entries belonging to the module are displayed.
selector	This is an LDT selector. Only a single heap entry will be displayed.

Comments

The **HEAP** command displays the Windows global heap in the command window. If no parameters are specified, the entire global heap is displayed. If **FREE** is specified, only the heap entries marked **FREE** are displayed. If *module-name* is specified, only heap entries belonging to the module will be displayed. If *selector* is specified, only the single heap entry corresponding to the *selector* will be displayed. At the end of the listing, the total amount of memory used by the heap entries that were displayed is shown. If the current CS:EIP belongs to one of the heap entries, that entry will be displayed with the bold video attribute.

For each heap entry the following information is displayed:

selector or *handle*	In Windows 3.0 this is almost the same thing. Heap selectors all have a dpl of 1 while the corresponding handle is the same selector with a dpl of 2. For example, if the handle was 106H the selector would be 105H. Either of these can be used in an expression. If 106:0 were used in a SoftICE/W expression, SoftICE/W would convert it to 105:0 when displaying it.
address	The 32-bit flat virtual address.
size	The size of the heap entry in bytes.
module name	The module name of the owner of the heap entry.
type	The type of entry. This can be one of the following:

Code	Non-discardable code segment
Code D	Discardable code segment
Data	Data segment
ModuleDB	Module data base segment
TaskDB	Task data base segment
BurgerM	Burger Master (The heap itself)
Alloc	Allocated memory.
Resource	Windows Resource

Additional type information. If the heap entry was a code or a data segment, the segment number from the .EXE file will be displayed. If the heap entry was a resource, one of the following fields can be displayed:

UserDef

Cursor

Bitmap

Icon

Menu

Dialog

String

FontGrp

Font

Accel

ErrTable

CursGrp

IconGrp

NameTabl

Note

If there is no current LDT, then the **HEAP** command is unable to display heap information.

Example

HEAP kernel

This command would display all heap entries belonging to the KERNEL module. This would look something like the following:

Han/Sel	Address	Length	Owner	Type	Seg/Rsr
00F5	000311C0	000004C0	KERNEL	ModuleDB	
00FD	00031680	00007600	KERNEL	Code	01
0575	00054220	00003640	KERNEL	Alloc	
0106	00083E40	00002660	KERNEL	Code D	02
010E	805089A0	00001300	KERNEL	Code D	03
0096	80520440	00000C20	KERNEL	Alloc	

Total Memory:62K

LHEAP

Display the Windows local heap.

Syntax

LHEAP [*selector*]

selector	This is an LDT data selector.

Comments:

The **LHEAP** command displays the data objects that a Windows program has allocated on the local heap. If *selector* is not specified, the value of the current DS register is used. The specified *selector* is usually the Windows program's data selector. To find this, do a **HEAP** command of the Windows program you are interested in and look for an entry of type data.

If no selector is specified, SoftICE/W will use DS.

There are cases when a Windows program will place its local heap in an allocated memory region or have more than one local heap. In this case you must find the selector of the segment that contains the local heap by other means. Segments marked as alloc in the **HEAP** command could contain a local heap.

For each local heap entry the following information is displayed:

offset	The 16-bit offset relative to the specified selector base address.
size	The size of the heap entry in bytes.
type	The type of entry. This can be one of the following:

FIX	Fixed (not moveable)
MOV	Moveable
FREE	Available memory

handle	The handle associated with each element. For fixed elements, the handle is equal to the address that is returned from LocalAlloc(). For moveable elements, the handle is the address that will be passed to LocalLock().

At the end of the list, the total amount of memory in the local heap is displayed.

Example

LHEAP GDI

This command would display all local heap entries belonging to GDI's default local heap. The display would look something like the following:

```
Offset      Size        Type        Handle
93D2        0046        Mov         0DFA
941E        0046        Mov         0C52
946A        0046        Mov         40DA
94B6        004E        Mov         0C66
950A        4A52        Mov         0E52
Used: 19.3K
```

VXD

Display the Windows VxD map.

Syntax

VXD [*VxD-name*]

VxD-name	The name of a virtual device driver.

Comments

This command displays a map of all Windows virtual device drivers in the command window. If no parameters are specified, all VxD's are displayed. If a *VxD-name* is specified, only that VxD will be displayed. Information that will be shown includes the control procedure address, the Protected Mode API address, the V86 API address and the addresses of all VxD services. If the current CS:EIP belongs to one of the VxD's in the map, that line will be displayed with the bold video attribute.

If no parameters are specified, then each entry in the VxD map contains the following information:

VxD name	Name specified in the .DEF file when the VxD was built.
address	The flat 32-bit address of one VxD section. VxDs are comprised of multiple sections where each section contains both code and data. (i.e. LockCode, LockData would be one section.)
size	The length of the VxD. This includes both the code and the data of the VxD group.
code selector	The flat code selector.
data selector	The flat data selector.
type	The section number from the .386 file.
id	The VxD ID number. VxD ID numbers are used to obtain the Protected Mode and V86 API addresses that applications call.
DDB	Address of the VxDs Device Descriptor Block (DDB). This is a control block that contains information about the VxD like the address of the Control Procedure and addresses of APIs.

If a VxD name is specified, the following information is displayed in addition to the above information:

Control Procedure	Where all VxD messages are dispatched to.
Protected Mode API	The address of the routine where all services called by protected mode applications are processed.
V86 API Address	The address of the routine where all services called by V86 applications are processed.
VxD Services	A list of all VxD services that are callable from other VxDs. For the Windows system VxDs, both the name and the address of the routines will be displayed.

Example

VXD

This command displays the VxD map in the command window. The first few lines of the display would look something like the following:

VxDName	Address	Length	Code	Data	Type	ID	DDB
VMM	80001000	000193D0	0028	0030	LGRP	01	
VMM	80200000	00002F1C	0028	0030	IGRP		
LoadHi	8001A3d0	000007E8	0028	0030	LGRP	02	
LoadHi	80202F1C	00000788	0028	0030	IGRP		
WINICE	8001ABB8	00027875	0028	0030	LGRP		
CV1	80042430	0000036B	0028	0030	LGRP		
VDDVGA	8004279C	00007AD8	0028	0030	LGRP		
VDDVGA	802036A8	000005EC	0028	0030	IGRP		

The VxD names in the table above can be used as symbol names. This is especially useful when setting break points at the entry points of these VxD service routines.

TASK

Display the Windows task list.

Syntax

TASK

Comments

This command displays information about all tasks that are running in the enhanced Windows environment. The task that currently has the focus is displayed with an asterisk after the task name. For each running task, the following information is displayed:

- The task name.
- The stack address (SS:SP) of the task when it last relinquished control. This is not useful for the current task since SP probably has changed since the last time control was relinquished.
- The top of stack offset.
- The bottom of stack offset. SP cannot go below this.
- The lowest SP has ever been when control was relinquished.
- The selector for the task data base segment.
- The queue handle for the task. This is just the selector for the queue.

The **TASK** command also works for 32 bit tasks. However, the following fields change in this case:

- The StackBottom field will contain the highest legal address of the stack shown as a 32 bit flat offset.
- The StackTop field will contain the lowest legal address of the stack shown as a 32 bit flat offset.
- The StackLow field is not used.
- The SS:SP field will contain the 16 bit selector:offset address of the stack. If you examine the base address of the 16 bit selector, you will see that this points into the exact same memory as the flat 32 bit pointers used with the 32 bit data selector.

If you want SoftICE/W to pop up when a non-active task is re-started, you can use the **STACK** command to find the address to set the execution break point. To do this enter **STACK** followed by the task name. The bottom line of the call stack will show an address preceded by the word 'at'. This is the address of the CALL instruction the program made to Windows, but it has not yet returned. You must set an execution break point at the address following this call.

You can also use this technique to stop at other routines higher on the call stack. This is useful when you don't want to single step through all of the library code until execution resumes in your program's code.

The **TASK** command is also very useful when you get a Windows general protection fault. Using the **TASK** command will let you know which program caused the fault.

Example

TASK

Sample output follows:

```
TaskNm      SS:SP          StackTop    StackBot    Low     TaskDB  hQueue  Events
FREECELL    21BF:7D96      86CE0000    86D00000            10FF    121F    0000
PROGMAN     17A7:200A      0936        2070        14CE    064F    07D7    0000
CLOCK       1427:1916      02E4        1A4E        143E    144F    1437    0000
MSWORD      * 29AF:913E    5956        93A4        7ADE    1F67    1F47    0000
```

STACK

Display the call stack for a DOS program or Windows task.

Syntax

STACK [*task-name* | *SS:BP*]

task-name	The name of the task as displayed by the task command.
SS:BP	The SS:BP of a valid stack frame.

Comments

This command displays a call stack for a DOS program or a Windows task. If you enter **STACK** with no parameters, then the current SS:BP will be used as a base for the stack frame. If you are using **STACK** to display a the stack of a Windows task that is not the current task, then either a *task-name* or a valid *SS:BP* stack frame must be specified. A list of running tasks can be obtained with the **TASK** command. The current task (marked with an '*') should not be used since the last known SS:SP is no longer correct.

The **STACK** command walks the stack showing the address of each routine. If the routine is found in the current symbol table then its name is displayed. If it is not in the symbol table then the export list and module name list are searched in that order. If stack variables are present, then they are displayed as well.

Each entry of the call stack contains the following information:

- Symbol name or module name.
- The CS:IP value of this entry.
- The source line number if available.
- The address of the first line of this routine or the name of the routine that was called to reach this routine.

If stack variables are available for this entry then the following information about each is displayed:

- BP relative offset.
- Stack variable name.
- Data in the stack variable if it is of type char, int or long.

The **STACK** command will also work in 32 bit code. Since 32 bit support is limited to .SYM files, local variables will not be displayed in the call stack. For each line in the call stack, both the nearest symbol to the address and the actual address are displayed. If there is no symbol available, the module name and object/section name are displayed instead.

The 32 bit call stack support is not limited to applications. It will also work for VxD code at ring 0. However, since most VxDs are written in assembly language, many times there is not a valid call stack to walk.

The call stack code will not trace through thunks or level changes.

Example

STACK

This is the output of the **STACK** command after a break point is set in the message handler of a Windows program.

```
__astart at 0935:1021  [?]
WinMain at 0935:0d76 [00750]
   [BP+000C]hInstance  0935
   [BP+000A]hPrev  0000
   [BP+0006]lpszCmdLine
   [BP+0004]CmdShow
   [BP-0002]width  00DD
   [BP-0004]hWnd  00E5
USER!SENDMESSAGE+004F at 05CD:06A7
USER(01) at 0595:04A0 [?]  0595:048b
USER(06) at 05BD:1A83 [?]
=>ClockWndProc at 0935:006F [0179]
   [BP+000E]hWnd  1954
   [BP+000C]message  0024
   [BP+000A]wParam 0000
   [BP+0006]lParam  06ED:07A4
   [BP-0022]ps  0000
```

This is an example of the **STACK** command in 32 bit mode. Execution has been stopped within the C library DLL's memset routine:

```
W32SCOMB!DispatchCB32+01FF at 2197:86C5003B
   UTSAMP!.text+01A4 at 2197:86C211A4
   _MyGetFreeSpace@0+0016 at 2197:86C7113B
   => MSVCRT10!memset+0005 at 2197:86C94F89
```

VCALL

Display the names and addresses of VxD callable routines.

Syntax

VCALL [*partial-name]**

partial-name	A VxD callable routine name or the first few characters of the name followed by '*'. If '*' is the last character of the string then all routines that start with the specified characters will be displayed.

Comments

This command displays the names and addresses of Windows VxD callable routines. These are Windows services provided to VxD's. All of these routines are located in enhanced Windows standard VxD's. Most of the routines are located in VMM. If the VxD callable routine is not in VMM, its name is prefaced by the name of the VxD.

The addresses displayed are not valid until the VMM VxD has been initialized. If the **X** is removed from the **INIT** string, and SoftICE/W pops up when loading, the addresses are not valid at that point.

Example

VCALL Call*

This would display all VxD calls that start with "Call". A sample output for this command follows:

```
80006E04Call_When_VM_Returns
80009FD4Call_Global_Event
80009FF4Call_VM_Event
8000A018Call_Priority_VM_Event
8000969CCall_When_VM_Ints_Enabled
800082C0Call_When_Not_Critical
8000889FCall_When_Task_Switched
8000898CCall_When_Idle
```

WMSG

Display the names and message numbers of Windows messages.

Syntax

WMSG [*partial-name]**

partial-name	A Windows message name or the first few characters of a Windows message name followed by '*'. If '*' is the last character of the string then all the Windows messages that start with the specified characters will be displayed.

Comments

This command displays the names and message numbers of Windows messages. This command is useful when logging or setting break points on Windows messages with the **BMSG** command.

Example

WMSG WM_GET*

This command displays the names and message numbers of all Windows messages that start with "WM_GET". A sample output for this command follows:

```
000D   WM_GETTEXT
000E   WM_GETTEXTLENGTH
0024   WM_GETMINMAXINFO
0031   WM_GETFONT
0087   WM_GETDLGCODE
```

PAGE

Display page table information.

Syntax

PAGE [*address* [L *length*]]

address	A virtual address, segment:offset address, or selector:offset address that you wish to know page table information about, including the virtual and physical address.
length	Number of pages to be displayed.

Comments

The **PAGE** command has many uses. When *address* is specified as a parameter it shows the internal data of a page table entry. This includes the following:

- The virtual address of a segment:offset or selector:offset address.
- The physical address of a virtual address, segment:offset address or selector:offset address.

When *length* is added as a parameter, the **PAGE** command shows multiple page table entries.

If no parameters are specified, the **PAGE** command shows the page directory. The is a high-level memory map of windows. The first line displayed is the address of the page directory itself. Each following line displays the information in each page directory entry.

Technical Note

In the 386/486 architecture each page directory entry refers to a single page table, and each page table contains 1024 entries. Each entry represents a 4K page, so each page table controls four megabytes of memory.

The following specific information is displayed by the **PAGE** command:

physical address	If the page directory is being displayed then this is the physical address of the page table that this page director entry refers to. Each page directory entry references one page table which controls 4 megabytes of memory. If specific pages are being displayed, then this is the physical address that corresponds to *address*. If *length* was entered, then the physical addresses for each entry are the physical addresses of start of the page.
linear address	If the page directory is being displayed then this is the virtual address of the page table entry. This is the address you would use in SoftICE/W if you wanted to display the page table entry with the D command. If specified pages are being displayed, this is virtual *address*. If *length* was entered then this is the virtual address of the start of the page.
attribute	This is the attribute of the page directory or page table entry. Valid attributes are:

P	Present
D	Dirty
A	Accessed
U	User

type — Each page directory entry has a three bit field that can be used by the operating system to classify page tables. Windows classifies page tables into the following six categories:

- System
- Instance
- VM
- Private
- Relock
- Hooked

If a page is marked Not Present, then all that is displayed is NP followed by the 32-bit contents of the page table entry.

Examples

PAGE

PAGE with no parameters displays page directory information. The following is a sample **PAGE** command output :

```
Page Directory  Physical = 002B6000  Linear = 006B600
```

Physical	Linear	Attribute			Type	Linear Add.Rnge
002B7000	006B7000	P	A	U	System	00000000-003FFFFF
00109000	00509000	P	A	U	System	00400000-007FFFFF
0010A000	0050A000	P		U	System	00800000-00BFFFFF
0010B000	0050B000	P		U	System	00C00000-00FFFFFF
0010C000	0050C000	P		U	System	01000000-013FFFFF
002B8000	006B8000	P	A	U	System	80000000-803FFFFF
00106000	00506000	P	A	U	System	80400000-807FFFFF
00107000	00507000	P		U	System	80800000-80BFFFFF
00108000	00508000	P		U	System	80C00000-80FFFFFF
002B7000	006B7000	P	A	U	System	81000000-813FFFFF

PAGE 00106018 L 3

PAGE with an address specified displays the page table entry that corresponds to that address. In this example, three page table entries will be displayed starting with the page table entry that corresponds to address 00106018. Notice that when the *length* parameter is specified, the linear address is truncated to the base address of the memory page that contains *address*.

Linearl	Physical	Attribute		Type
00106000	00006000	P	U	VM
00107000	00007000	P	U	VM
00108000	00008000	P	U	VM

PAGE #585:263C

In this example **PAGE** can be used to find both the virtual and physical address of a selector:offset address.

Linearl	Physical	Attribute		Type
0004A89C	00218442	P	U	Instance

PHYS

Display all virtual addresses that correspond to a physical address.

Syntax

PHYS ***physical-address***

physical-address	This is an actual memory address that the 386/486 generates after a virtual address has been translated by its paging unit. This is the address that appears on the Computer's BUS and is most important to the programmer when dealing with memory mapped hardware devices like video memory.

Comments

Windows uses 386/486 paging to alter the relationship between virtual addresses and physical addresses. In many cases a physical address range may appear in more than one page table entry, and therefore more than one virtual address range.

Example

PHYS A0000

Physical address A0000 is the start of VGA video memory. Video memory often shows up in multiple virtual address in Windows. In this example there are three different virtual addresses that correspond to physical A0000 as shown:

```
000A0000
004A0000
80CA0000
```

MAPV86

Display the memory map of the current Virtual Machine.

Syntax

MAPV86 [*address*]

address	A segment:offset type address.

Comments

If *address* is specified, only one map line is displayed. If *address* is not specified then the top line of the display contains information about the current virtual machine:

VM id	The virtual machine ID. ID1 is the system VM.
VM handle	The 32-bit virtual machine handle.
CRS pointer	The 32-bit client register structure pointer.
VM address	The 32-bit linear address of the virtual machine. This is the "high" address of the virtual machine that is also mapped to linear address 0.

If the current CS:IP belongs to a **MAPV86** entry, that line will be displayed with the bold video attribute. Each line of the **MAPV86** display contains the following information:

- The segment:offset start address of the component.
- The length of the component in paragraphs.
- The owner name of the component.

Note

Windows may have certain pages of the DOS VM memory mapped out when you enter the **MAPV86** command. If this occurs, the output from the MAPV86 command will terminate with a PAGE NOT PRESENT message. Often, just hot-keying out of SoftICE/W and right back in will cause Windows to map those pages back in.

Example

MAPV86

An abbreviated sample output follows:

```
ID=01 Handle=80441000 CRS Ptr=80013390 Linear=80C00000
```

Start	Length	Name
0000:0000	0040	Interrupt Vector Table
0040:0000	0030	ROM BIOS Variables
0070:0000	025D	I/O System

02CD:0000	08E6	DOS
0BB5:0012	0000	NU-MEGA
0C8B:0000	00E8	SOFTICE1
0D41:0000	00B6	XMSXXXX0
10D0:0000	038F	SMARTAAR

MAP32

Display a memory map of all 32 bit modules currently loaded in memory.

Syntax

MAP32 [module name | module handle]

Comments

MAP32 with no parameters lists all 32 bit modules. If either a module-name or module-handle is specified, only sections from that one module will be displayed. For each module, one line is displayed for every section owned by that module. Each line contains the following information:

Owner	The module name.
Name	The section name from the executable file.
Obj#	The section number from the executable file.
Address	The selector:offset address of the section.
Size	The memory size in bytes.
Type	The type and attributes of the section.

CODE	Code
IDATA	Initialized Data
UDATA	Uninitialized Data
RO	Read Only
RW	Read/Write
SHARED	Object is shared

Example

MAP32 MSVCRT10

Sample output for MAP32 on a single module.

```
Owner       Obj Name   Obj#    Address          Size        Type
MSVCRT10    .text      0001    2197:86C81000    00024A00    CODE  RO
MSVCRT10    .bss       0002    219F:86CA6000    00001A00    UDATA RW
MSVCRT10    .rdata     0003    219F:86CA8000    00000200    IDATA RO
```

```
MSVCRT10   .edata   0004   219F:86CA9000   00005C00   IDATA RO
MSVCRT10   .data    0005   219F:86CAF000   00006A00   IDATA RW
MSVCRT10   .idata   0006   219F:86CB6000   00000A00   IDATA RW
MSVCRT10   .reloc   0007   219F:86CB7000   00001800   IDATA RO
```

HWND

Display information on Window handles.

Syntax

HWND [window-level] [task-name]

window-level	Windows hierarchy number. 0 is the top level, 1 is the next level and so on. The window levels represent a parent child relationship. For example, a level 1 window has a level 0 parent.
task-name	Any currently loaded Windows Task. These names are available with the TASK command.

Comments

This command displays information about all window handles that are currently in use in the Windows environment. For each window handle, the following information is displayed:

Window Handle	The window handle is actually an offset into a data segment in USER where information is stored about a window.
Queue Handle	A queue handle is actually a selector of a segment that contains the message queue for a window. A standard message queue can hold up to six messages.
Queue Owner	Task name of the task that owns this queue.
Class Name	Class name or atom of class that this window belongs to.
Window Procedure	Address of the window procedure for this window.

A common use of the HWND command is to find the window handle for setting a break point on a window message. See the BMSG command.

Example

HWND msword

Sample output follows:

```
Handle       hQueue      QOwner      Class        Procedure
0F4C(0)       087D       MSWORD      #32769       DESKTOP
 0FD4(1)      080D       MSWORD      #32768       MENUWND
 22C4(1)      087D       MSWORD      OpusApp      0925:0378
   53E0(2)    087D       MSWORD      OpusPmt      0945:1514
   2764(2)    087D       MSWORD      a_sdm_Msft   0F85:0010
```

2800(3)	087D	MSWORD	OpusFedt	0F85:0020
2844(3)	087D	MSWORD	OpusFedt	0F85:0020
2428(2)	087D	MSWORD	OpusIconBar	0945:14FE
2888(2)	087D	MSWORD	OpusFedt	0945:14D2

Note that the level number is shown in parenthesis following each handle.

CLASS

Display information on Window classes.

Syntax

`CLASS [module-name]`

module-name	Any currently loaded Windows Module. Not all Windows Modules have classes registered.

Comments

This command displays information about all window classes that have been registered. For each class, the following information is displayed:

Class Handle	The class handle is actually an offset of a data structure within USER. It is used to refer to windows of this class.
Class Name	Name that was passed when the class was registered. If no name was passed the atom is displayed.
Owner	Module that has registered this window class.
Window Procedure	Address of the window procedure for this window class.

Example

CLASS msword

Sample output follows:

Handle	Name	Owner	Window Procedure
0F24	#32772	USER	TITLEWNDPROC
0EFC	#32771	USER	SWITCHWNDPROC
0ED4	#32769	USER	DESKTOPWNDPROC
0E18)	MDIClient	USER	MDICLNTWNDPROC
0DDC	ComboBox	USER	COMBOBXWNDPROC
0DA0	ComboLBox	USER	LBBOXTLWNDPROC
0D64	ScrollBar	USER	SBWNDPROC
0D28	ListBox	USER	LBOXCTLWNDPROC
0CF0	Edit	USER	EDITWNDPROC

Note that in this case we have symbols for all of the window procedures because SoftICE/W includes all of the exported symbols from USER.EXE. If a symbol was not available for the window procedure you would see a hexadecimal address.

VM

Display information on virtual machines.

Syntax

VM [VM-ID]

VM-ID	Index number of this virtual machine. These numbers start at 1 and 1 is always assigned to the VM that Windows Apps run in.

Comments

If no parameters are specified, this command displays information about each virtual machine. For each virtual machine, the following information is displayed:

VM Handle	The vm handle is actually a flat offset of the data structure that holds information about the VM.
Status	This is a bit mask that shows current state information about the VxD. The values are:

0001H	Exclusive mode
0002H	Runs in background
0004H	In process of creating
0008H	Suspended
0010H	Partially destroyed
0020H	Executing protected mode code
0040H	Executing protected mode app
0080H	Executing 32-bit protected app
0100H	Executing call from VxD
0200H	High priority background
0400H	Blocked on semaphore
0800H	Woke up after blocked
1000H	Part of V86 App is pageable
2000H	Rest of V86 is locked
4000H	Scheduled by time-slices
8000H	Idle - has released time slice

High Address	Alternate address space for VM. This is where a VxD typically accesses VM memory (instead of 0). Note that it is likely for parts of the VxD to be paged out at any one time that you pop up SoftICE/W.
VM-ID	Index number of this VxD, starting at 1.
Client Registers	The address of the saved registers of this VM. This address actually points into the level 0 stack for this VM.

If a VM-ID is specified as a parameter to the VM command, then the register values of this VxD are displayed. These registers are those found in the client registers area and may not be valid for the current VM or if the VM is in the process of being interrupted or re-scheduled. If you pop up while the current VM is executing, then the registers displayed in the SoftICE/W register window are valid. If you are in the first few instructions of an interrupt routine, the CS:IP may be the only registers valid (the others have not been saved yet).

There are two sets of segment registers displayed plus EIP and SP. These are for the protected mode context and the real address mode context of the VM. If the VM was executing in protected mode last, then the protected mode registers will be on top, or visa-versa for V86 mode. The general purpose registers (displayed below the segment registers) pertain to the version of the segment registers on top.

In Windows enhanced mode the VM is a unit of scheduling for the kernel. It can have one protected mode thread of execution and one V86 mode thread of execution. Windows, Windows applications and DLL's all run in the protected mode thread of execution of VM 1. Therefore a Windows application can not preempt another Windows application, but a DOS program running in V86 mode of a separate VM can preempt a Windows application.

VM's other than VM 1 normally have a V86 thread of execution only. However, DPMI applications launched from these VM's can execute in the protected mode thread.

Usage

The VM command is very useful while debugging VxD's, DPMI programs and DOS programs running in VM's. For example, if the system hangs up while running a DOS program, you can often find the address of the last instruction it executed with the VM command (the CS:EIP shown).

Another more esoteric, but highly valuable use is when Windows faults all the way back to DOS. There are times when Windows can not handle a fault and exits Windows and you end up back at the DOS prompt.

If this happens, duplicate the problem with **I1HERE ON** in SoftICE/W (Windows executes an INT 1 prior to returning to DOS). When the fault happens, SoftICE/W will pop up. Use the VM command to find out the last address of execution and use the CR command to find the fault address (CR2 contains the fault address). The ESI register usually points to an error message at this point.

Example

VM

Sample output follows:

```
VM Handle   Status     High Addr  VM-ID   Client Regs
806A1000    00004000   81800000   3       806A8F94
8061A000    00000008   81400000   2       80515F94
80461000    00007060   81000000   1       80013390
```

I/O Port Commands

I or **IB**	Input from byte I/O port
IW	Input from word I/O port
O or **OB**	Output to byte I/O port
OW	Output to word I/O port

I, IB, IW

Input a value from an I/O port.

Syntax

I[*size*] *Êport*

size	B or W
B	Byte
W	Word
port	A byte or word value.

Comments

The input from port commands are used to read and display a value from a hardware port. Input can be done from byte or word ports. If no *size* is specified, the default is **B**.

Example

I 21

This command performs an input from port 21, which is the mask register for interrupt controller one.

O, OB, OW

Output a value to an I/O port.

Syntax

O[*size*] *Êport* *value*

size	B or W
B	Byte
W	Word
port	A byte or word value.
value	A byte for a byte port or a word for a word port.

Comments

The output to port commands are used to write a *value* to a hardware port. Output can be done to byte or word ports. If no *size* is specified, the default is **B**.

Example

O 21 FF

This command performs an out to port 21, which masks off all the interrupts for interrupt controller one.

Transfer Control Commands

X	Exit from the SoftICE/W screen
G	Go to an address
T	Trace one instruction
P	Program step
HERE	Go to the current cursor line
EXIT	Force an exit of current program
GENINT	Force an interrupt to occur
HBOOT	Hard system boot (total reset)

X

Exit from the SoftICE/W screen.

Syntax

```
X
```

Comments

The **X** command exits SoftICE/W and restores control to the program that was interrupted to bring up SoftICE/W. The SoftICE/W screen disappears. If any break points have been set, they become active.

If the register window is visible when SoftICE/W pops up, all registers that have been altered since the **X** command was issued will be displayed with the bold video attribute.

Note

While in SoftICE/W, pressing the hot key sequence is equivalent to entering the **X** command.

Default Function Key

F5

G

Go to an address.

Syntax

G [=*start-address*]Ê[*break-address*]

Comments

The **G** command exits from SoftICE/W. If *break-address* is specified, a single one-time execution break point is set on that address. In addition, all sticky break points are armed.

Execution begins at the current CS:EIP unless the *start-address* parameter is supplied. In that case execution begins at *start-address*. Execution continues until *break-address* is encountered, the window pop-up key sequence is used, or a sticky break point occurs. When SoftICE/W pops up, for any reason, the one-time execution break point is cleared.

The *break-address* must be the first byte of an instruction opcode.

When the specified address is reached, the current CS:EIP will be the instruction where the break point was set.

The **G** command with no parameters behaves the same as the **X** command.

The non-sticky execution break point uses an INT 3 style break point.

If the register window is visible when SoftICE/W pops up, all registers that have been altered since the **G** command was issued will be displayed with the bold video attribute.

Example

G CS:80123456

This command sets a one-time break point at address CS:80123456H.

T

Trace one instruction.

Syntax

`T [=start-address] [count]`

count	Specifies how many times SoftICE/W should single step before stopping.

Comments

The **T** command single steps one instruction by utilizing the single step flag.

Execution begins at the current CS:EIP unless the *start-address* parameter is specified. If *start-address* is specified, CS:EIP is changed to *start-address* prior to single stepping.

If *count* is specified, then SoftICE/W single steps *count* times. When single stepping with a *count*, pressing the *Esc* key will terminate stepping.

If the register window is visible when SoftICE/W pops up, all registers that have been altered since the **T** command was issued will be displayed with the bold video attribute.

If the code window is in source mode, this command will single step to the next source statement.

Example

T = CS:1112 8

This command single steps through eight instructions starting at memory location CS:1112.

Default Function Key

F8

P

Execute one program step.

Syntax

P [RET]

Comments

The **P** command is a logical program step. In assembly mode, one instruction at the current CS:EIP is executed unless the instruction is a call, interrupt, loop, or repeated string instruction. In those cases, the entire routine or iteration is completed before control is returned to SoftICE/W.

If **RET** is specified, SoftICE/W will step until it finds a return or return from interrupt instruction. This function works in either 16 or 32 bit code and also works in VxD code. The default key for this function of the **P** command is **F12**.

Note

If you are in an unusually long procedure, there can be a noticeable delay since SoftICE/W is single stepping every instruction.

The **P** command uses the single step flag for most instructions. For call, interrupt, loop, or repeated string instructions, a one-time execution break point is used. In that case, an INT 3 style break point is implemented.

In source mode one source statement is executed. If the source statement involves calling another procedure, the call is not followed. The called procedure is treated like a single statement.

If the register window is visible when SoftICE/W pops up, all registers that have been altered since the **P** command was issued will be displayed with the bold video attribute. For call instructions, this will show what registers a subroutine has not preserved.

Example

P

This command executes one program step.

Default Function Key (P)

F10

Default Function Key (P RET)

F12

HERE

Go to the current cursor line.

Syntax

HERE

Comments

The **HERE** command executes until the program reaches the current cursor line. **HERE** is only available when the cursor is in the code window. If the code window is not visible or the cursor is not in the code window, use the **G** command instead. Use the **EC** command (default key is **F6**) if you want to move the cursor into the code window.

To use the **HERE** command, you place the cursor on the source statement or assembly instruction that you wish to execute to. Then enter **HERE** or press the function key that **HERE** is programmed to (the default is **F7**).

The **HERE** command exits from SoftICE/W with a single one-time execution break point set. In addition, all sticky break points are armed.

Execution begins at the current CS:EIP and continues until the address of the current cursor position in the code window is encountered, the window pop-up key sequence is used, or a sticky break point occurs. When SoftICE/W pops up, for any reason, the one-time execution break point is cleared.

The non-sticky execution break point uses an INT 3 style break point.

If the register window is visible when SoftICE/W pops up, all registers that have been altered since the **HERE** command was issued will be displayed with the bold video attribute.

Example

HERE

This example sets an execution break point at the current cursor position, then exits from SoftICE/W and begins execution at the current CS:EIP.

Default Function Key

F7

EXIT

Force an exit of the current DOS or Windows program.

Syntax

```
EXIT
```

Comments

The **EXIT** command attempts to abort the current DOS or Windows program by forcing a DOS exit function (INT 21H, function 4CH). This command will only work if DOS is in a state where it is able to accept the exit function call. If this call is made from certain interrupt routines, or other times when the DOS is not ready , the system may behave unpredictably. This call can only be used when SoftICE/W pops up in VM mode or 16/32-bit protected mode running at ring 3. In 32-bit ring 0 protected mode code, an error will be displayed.

Caution

The **EXIT** command should be used with care. Since SoftICE/W can be popped up at any time, a situation can occur where the DOS is not in a state to accept an exit function call. Also, the **EXIT** command does not do any program specific resetting. For instance, the **EXIT** command does not reset the video mode or interrupt vectors. For Windows programs, the **EXIT** command does not free resources.

Note

If running under WIN32s, the **EXIT** command will sometimes cause WIN32s to pop up with an unhandled exception occurred dialog box. Pressing OK will then terminate the application.

Example

```
EXIT
```

This command will cause the current DOS or Windows program to exit.

GENINT

Force an interrupt to occur.

Syntax

GENINT [NMI | INT1 | INT3 | *interrupt-number*]

interrupt-number	A valid interrupt number between 0 and 5FH.

Comments

The **GENINT** command forces an interrupt to occur. This function can be used to hand off control to another debugger when using SoftICE/W with another software debugger. It can also be used to test interrupt routines.

The **GENINT** command simulates the processing sequence of a hardware interrupt or an INT instruction. It vectors control through the current IDT entry for the specified interrupt number.

Example

GENINT NMI

This forces a non-maskable interrupt. This will give control back to CodeView for DOS if SoftICE/W is being used as an assistant to CodeView for DOS. If using CodeView for Windows, use **GENINT 0**. For other debuggers, experiment with *interrupt-numbers* 0, 1, 2 and 3.

HBOOT

Do a hard system boot (total reset).

Syntax

HBOOT

Comments

The **HBOOT** command resets the computer system. SoftICE/W is not retained in the reset process. **HBOOT** is sufficient unless an adapter card requires a power-on reset. In those rare cases, the machine power must be recycled.

HBOOT performs the same level of system reset as pressing *Ctrl Alt Delete* when not in SoftICE/W.

Example

```
HBOOT
```

This command makes the system reboot.

Debug Mode Commands

ACTION	Set action after break point is reached
I1HERE	Pop up on embedded INT 1 instructions.
I3HERE	Pop up on INT 3 instructions.
ZAP	Replace embedded INT 1 or INT 3 with NOP

ACTION

Set action after break point is reached.

Syntax

ACTION [NMI | INT1 | INT3 | HERE | *interrupt-number* | *debugger-name*]

interrupt-number	A valid interrupt number between 0 and 5FH.
debugger-name	The module name of the Windows application debugger you wish to gain control on a SoftICE/W break point.

Comments

The **ACTION** command determines where control is given when break point conditions have been met. In most cases, the **ACTION** command is used to pass control to an application debugger you are using in conjunction with SoftICE/W. **HERE** is used when it is desired to return to SoftICE/W when break conditions have been met.

INT1, **INT3** and **NMI** are alternatives for activating DOS debuggers when break conditions are met. *Debugger-name* is used to activate Windows debuggers. The module name of your debugger can be found with the SoftICE/W **MOD** command.

If *debugger-name* is specified, an INT 0 will be used to trigger the Windows debugger. SoftICE/W will ignore break points that the Windows debugger causes if the debugger accesses memory that is covered by a memory location or range break point. When SoftICE/W passes control to the Windows debugger with an INT 0, the Windows debugger will respond as if a divide overflow occurred and display a message. Since the INT 0 was not caused by an actual divide overflow, you can ignore this message.

Example

ACTION NMI

This will cause SoftICE/W to generate a non-maskable interrupt when break conditions are met. This will give control to CodeView for DOS if SoftICE/W is being used as an assistant to CodeView for DOS. If using CodeView for Windows use **ACTION** CVW. If using Borland's Turbo Debugger for Windows use **ACTION** TDW. If using Multiscope's Debugger for Windows, use **ACTION** RTD.

I1HERE

Pop up on embedded INT 1 instructions.

Syntax

```
I1HERE  [ON | OFF]
```

Comments

The **I1HERE** command lets you specify that any embedded interrupt 1 will bring up the SoftICE/W screen. This feature is useful for stopping your program in a specific location. Before popping up, SoftICE/W checks to see that there is really an INT 1 in the code. If there is not, SoftICE/W will not pop up.

To use this feature, place an INT 1 into your code immediately before the location where you want to stop. When the INT 1 occurs, it will bring up the SoftICE/W screen. At this point, the current EIP will be the instruction after the INT 1 instruction.

If no parameter is specified, the current state of **I1HERE** is displayed.

The default is **I1HERE** mode **OFF**.

VMM, the Windows memory management VxD, executes INT 1 instructions prior to certain fatal exits. If you have **I1HERE ON**, you can trap these. The INT 1's generated by VMM are most often caused by a page fault with the registers set up as follows: EAX=faulting address, ESI points to an ASCII message and EBP points to a CRS (Client Register Structure).

Example

I1HERE ON

This command turns on **I1HERE** mode. Any INT 1's generated after this point will bring up the SoftICE/W screen.

I3HERE

Pop up on INT 3 instructions.

Syntax

```
I3HERE  [ON | OFF]
```

Comments

The **I3HERE** command lets you specify that any interrupt 3 will bring up the SoftICE/W screen. This feature is useful for stopping your program in a specific location.

To use this feature, place an INT 3 into your code immediately before the location where you want to stop. When the INT 3 occurs, it will bring up the SoftICE/W screen. At this point, the current EIP will be the instruction after the INT 3 instruction.

If you are developing a Windows program, the DebugBreak() Windows API routine will perform an INT 3.

If no parameter is specified, the current state of **I3HERE** is displayed.

The default is **I3HERE** mode **OFF**.

Example

```
I3HERE ON
```

This command turns on **I3HERE** mode. Any INT 3's generated after this point will bring up the SoftICE/W screen.

ZAP

Replace an embedded interrupt 1 or 3 with a NOP.

Syntax

ZAP

Comments

The **ZAP** command replaces an embedded interrupt 1 or interrupt 3 with the appropriate number of NOP instructions. This is useful when the INT 1 or INT 3 is placed in code that is repeatedly executed and you no longer want SoftICE/W to pop up. This command will only work if the INT 1 or INT 3 instruction is the one before the current CS:EIP.

Example

```
ZAP
```

The embedded interrupt 1 or interrupt 3 will be replaced with NOP instructions.

Utility Commands

S	Search memory for data
F	Fill memory with data
M	Move data
C	Compare two data blocks
A	Assemble code

S

Search memory for data.

Syntax

S [*address* L *length data-list*]

data-list	List of bytes or quoted strings separated by commas or spaces. A quoted string can be enclosed with single quotes or double quotes.
length	Length in bytes.

Comments

Memory is searched for a series of bytes or characters that matches the *data-list*. The search begins at the specified *address* and continues for the *length* specified. When a match is found, the memory at that address is displayed in the data window. and the message "PATTERN FOUND AT *location*" is displayed in the command window. If the data window is not visible, it is made visible.

To search for subsequent occurrences of the *data-list*, use the **S** command with no parameters. The search will continue from the address where the *data-list* was last found, until it finds another occurrence of *data-list* or the *length* is exhausted.

Note

The **S** command ignores pages that are marked not present. This makes it possible to search large areas of address space using the flat data selector (30:).

Examples

S ES:DI+10 L ECX 'Hello',12,34

This command searches for the string 'Hello' followed by the bytes 12H and 34H starting at offset ES:DI+10 for a *length* of ECX bytes.

S 30:0 L FFFFFFFF 'String'

This command searches the entire 4 gigabyte virtual address range for 'string'.

F

Fill memory with data.

Syntax

F *address* L *length data-list*

data-list	List of bytes or quoted strings separated by commas or spaces. A quoted string can be enclosed with single quotes or double quotes.
length	Length in bytes.

Comments

Memory is filled with the series of bytes or characters specified in the *data-list*. Memory is filled starting at the specified *address* and continues for the specified *length*. If the *data-list* length is less than the specified *length*, the *data-list* is repeated as many times as necessary.

Example

F DS:8000 L 100 'Test'

This command fills memory starting at location DS:8000H for a length of 100H bytes with the string 'Test'. The string 'Test' is repeated until the fill length is exhausted.

M

Move data.

Syntax

M *start-address* L *length end-address*

length	Length in bytes.

Comments

The specified number of bytes are moved from the *start-address* to the *end-address*.

Example

M DS:1000 L 2000 ES:5000

This command moves 2000H bytes (8K) from memory location DS:1000H to ES:5000H.

C

Compare two data blocks.

Syntax

C *start-address* L *length end-address*

length	Length in bytes.

Comments

The memory block specified by *start-address* and *length* is compared with the memory block specified by the *end-address* and *length*.

When a byte from the first data block does not match a byte from the second data block, both bytes are displayed, along with their addresses.

Example

C DS:805FF000 L 10 DS:806FF000

This command compares 10H bytes starting at memory location DS:805FF000H with the 10H bytes starting at memory location DS:806FF000H.

A

Assemble code.

Syntax

A [*address*]

Comments

The SoftICE/W assembler allows you to assemble instructions directly into memory. The assembler supports the standard 386 instruction set. Numeric co-processor instructions can NOT be assembled.

If *address* is not specified, then assembly will occur at the last address where instructions were assembled. If the **A** command has not been entered before and *address* is not specified then the current CS:EIP address is used.

The **A** command enters the SoftICE/W interactive assembler. An address is displayed as a prompt for each assembly line. After an assembly language instruction is typed in and *Enter* is pressed, the instructions are assembled into memory at the specified address. Instructions must be entered with standard Intel format. Press *Enter* at an address prompt to exit assembler mode.

If the address range in which you are assembling instructions is visible in the code window, the instructions will change interactively as you assemble.

The SoftICE/W assembler supports the standard 386 family mnemonics; however, there are some special additions:

- USE16 or USE32 entered on a separate line will cause subsequent instructions to be assembled as 16-bit or 32-bit respectively. If USE16 or USE32 is not specified, the default is the same as the mode of the current CS register.
- The DB mnemonic is used to define bytes of data directly into memory. The DB mnemonic is followed by a list of bytes and/or quoted strings separated by spaces or commas.
- The RETF mnemonic represents a far return.
- Override instructions can optionally be placed on a separate line. For example a code segment override would be entered as "CS:".
- WORD PTR, BYTE PTR, DWORD PTR, and FWORD PTR are used to determine data size if there is no register argument, for example, MOV BYTE PTR ES:[1234.],1.
- Use FAR and NEAR to explicitly assemble far and near jumps and calls. If FAR or NEAR is not specified then all jumps and calls are near.
- Operands referring to memory locations should be placed in square brackets, for example: MOV AX,[1234].

Example

A CS:1234

This command prompts you for assembly instructions, then assembles them beginning at offset 1234H within the current code segment. Press *Enter* at the address prompt after entering the last instruction.

Windowing Commands

Four window types may be created with SoftICE/W (register, watch, data and code). Any of these windows can be toggled on or off at any time. The watch, data and code windows can be of variable size; the register window is fixed in size. The windows always remain in a fixed order. Starting from the top of the screen, the order is register window, watch window, data window and code window.

WR	Toggle the register window
WC	Toggle/set the size of the code window
WD	Toggle/set the size of the data window
WW	Toggle the watch window
EC	Enter/exit the code window
.	Locate current instruction in code window

WR

Toggle the register window.

Syntax

WR

Comments

The **WR** command makes the register window visible if it is not currently visible. If the register window is currently visible, **WR** removes the register window.

The register window displays the 80386 register set and the processor flags.

When the register window is made invisible, the extra screen lines are added to the command window.

When the register window is made visible, the lines are taken from the other windows in the following order: command, code, data.

Default Function Key

F2

WC

Toggle/set the size of the code window.

Syntax

WC [*window-size*]

window-size	A decimal number.

Comments

If *window-size* is not specified, this command toggles the code window. If the code window was not visible it is made visible, and if it was visible it is removed.

If *window-size* is specified the code window is resized, or if it was not visible it is made visible with the specified size.

When the code window is made invisible, the extra screen lines are added to the command window.

When the code window is made visible, the lines are taken from the other windows in the following order: command, data.

If you wish to move the cursor to the code window, use the **EC** command (default key = **F6**). See the description of the **EC** command on page 166 for more details.

Example

WC 12

If no code window is present, a code window 12 lines in length is created. If the code window is currently on the screen, it is resized to 12 lines.

Default Function Key

Alt F3

WD

Toggle/set the size of the data window.

Syntax

`WD [window-size]`

window-size	A decimal number.

Comments

If *window-size* is not specified, this command toggles the data window. If the data window was not visible it is made visible, and if it was visible it is removed.

If *window-size* is specified the data window is resized, or if it was not visible it is made visible with the specified size.

When the data window is made invisible, the extra screen lines are added to the command window.

When the data window is made visible, the lines are taken from the other windows in the following order: command, code.

If you wish to move the cursor to the data window to edit data, use the **E** command. See the description of the **E** command on page 92 for more details.

Example

WD 1

If no data window is present, a data window of one line is created. If the data window is currently on the screen, it is resized to one line.

Default Function Key

Alt F2

WW

Toggle the watch window.

Syntax

```
WW
```

Comments

The **WW** command makes the watch window visible if it is not currently visible. If the watch window is currently visible **WW** removes it. If there are no watch expressions declared, **WW** does nothing.

The watch window's size is fixed and always contains one line for each watch expression.

When the watch window is made invisible, the extra screen lines are added to the command window.

When the watch window is made visible, the lines are taken from the other windows in the following order: command, code, data.

If you want to add a watch expression, use one of the **WATCH** commands. See the description of the **WATCH** commands on page 96 for more details.

Default Function Key

Alt F4

EC

Enter or exit the code window.

Syntax

EC

Comments

The **EC** command toggles the cursor location between the code window and the command window. If the cursor was in the command window it is moved to the code window, and if the cursor was in the code window it is moved to the command window.

The code window must be visible for this command to work. If you wish to make the code window visible, use the **WC** command. See the description of the **WC** command on page 163 for more details.

When the cursor is in the code window, several options become available that make debugging much easier. These options are:

Point-and-shoot break points	Point-and-shoot break points are set with the BPX command. If no parameters are specified with the BPX command, an execution break point is set at the location of the cursor position in the code window. The default function key for BPX is F9.
Go to cursor line	You can set a temporary break point at the cursor line and begin executing with the HERE command. The default function key for HERE is F7.
Scrolling the code window	The code window can be scrolled while the cursor is in the code window. The scrolling keys (*UpArrow*, *DownArrow*, *PageUp* and *PageDn*) are redefined while the cursor is in the code window. When the cursor is in the code window the scrolling keys do the following:

UpArrow	Scroll code window up one line.
DownArrow	Scroll code window down one line.
PageUp	Scroll window up one window.
PageDn	Scroll window down one window.

The code window can also be scrolled from the command window with the use of the CTRL key in conjuction with one of the already mentioned cursor keys.

In addition to the above keys, these others have special meaning:

CTRL-HOME	Jumps to line 1 of current source file.
CTRL-END	Jumps to the last line of the current source file.

Default Function Key

F6

.

Locate the current instruction in the code window.

Syntax

```
.
```

Comments

When the code window is visible, this command makes the instruction at the current CS:EIP visible and highlights it.

To make the code window visible, use the **WC** command. See the description of the **WC** command on page 163 for more details.

Debugger Customization Commands

PAUSE	Pause after each screen
ALTKEY	Set alternate key sequence to invoke SoftICE/W
FKEY	Show and edit function key assignments
DEX	Display/assign data window expression
CODE	Display instruction bytes
COLOR	Display/set screen colors
TABS	Display/set tab settings
SERIAL	Redirect console to serial monitor
LINES	Change number of lines of SoftICE/W display
Print-Screen	Print contents of screen
PRN	Set printer output port
FAULTS	Turn off trapping of processor faults

PAUSE

Pause after each screen.

Syntax

```
PAUSE  [ON | OFF]
```

Comments

The **PAUSE** command controls screen pause at the end of each page. If **PAUSE** is **ON**, you are prompted to press any key before information is scrolled out of the command window. The prompt is displayed in the status line at the bottom of the command window.

If no parameter is specified, the current state of **PAUSE** is displayed.

The default is **PAUSE** mode **ON**.

Example

```
PAUSE ON
```

This command specifies that subsequent command window display will not be automatically scrolled off the screen. You will be prompted to press a key before information is scrolled out of the window.

ALTKEY

Set an alternate key sequence to invoke SoftICE/W.

Syntax

ALTKEY [ALT *letter* | CTRL *letter*]

letter	Any letter (A - Z).

Comments

The **ALTKEY** command allows the key sequence for popping up SoftICE/W to be changed. The key sequence can be changed to **Ctrl** + *letter* or **Alt** + *letter.*

Occasionally you may be using a program that conflicts with the **Ctrl D** hot key sequence that brings up the SoftICE/W screen. One way to circumvent this possible problem is to use the **ALTKEY** command to change the hot key sequence.

If no parameter is specified, the current hot key sequence is displayed.

The default hot key sequence is **Ctrl D**.

Hint

If you want to change your hot key sequence every time you run SoftICE/W then you should place the **ALTKEY** command in the **INIT** statement of the WINICE.DAT initialization file.

Example

ALTKEY ALT Z

This command specifies that the key sequence **Alt Z** will now be used to pop up the SoftICE/W screen.

FKEY

Show and edit the function key assignments.

Syntax

FKEY [*function-key string*]

function-key	F1 - F12 (unshifted function key), SF1 - SF12 (shifted function key), CF1 - CF12 (Ctrl plus function key), or AF1 - AF12 (Alt plus function key).
string	The *string* consists of any valid SoftICE/W commands and the special characters (^ and ;). A ^ is placed in the *string* to make a command invisible. A ; is placed in the *string* in place of the *Enter* key.

Comments

The **FKEY** command is used to assign a *string* of one or more commands to a *function-key.*

If no parameters are specified, then the current *function-key* assignments are displayed.

To unassign a specified *function-key,* use the **FKEY** command with these parameters: a *function-key* name followed by a null *string*.

Using carriage return symbols in a *function-key* assignment *string* allows you to assign a *function-key* a series of commands. A carriage return is represented by a ';' (semi-colon).

If you put a '^' (**Shift 6**) in front of a *function-key* definition, the subsequent command will be invisible. The command will function as normal, but all information that would normally be displayed in the command window (excluding error messages) is suppressed. The invisible mode is useful when a command changes information in a window (code, register or data) but you do not want to clutter the command window.

SoftICE/W implements the *function-keys* by inserting the entire *string* into its keyboard buffer. The *function-keys* can therefore be used anyplace where a valid command could be typed.

If you want a function key assignment to be in effect every time you use SoftICE/W, you can pre-initialize a *function-key* in the initialization file WINICE.DAT. For more information on *function-key* definitions in WINICE.DAT, see *WINICE.DAT Initialization File* on page 7.

Example

FKEY F2 ^WR;

This command will assign the toggle register window command to the **F2** *function-key.* The ^ makes the function invisible, and the ; ends the function with a carriage return. After this command is entered, pressing the **F2** key will then toggle the register window on or off.

FKEY CF1 G CS:8028F000;D SS:ESP;U CS:EIP+

This example shows that multiple commands can be assigned to a single function and that partial commands can be assigned for the user to complete. After this command is entered, pressing the **Ctrl F1** key sequence causes the program to execute until location CS:8028F000H is reached, display the stack contents and start the **U** command for the user to complete.

FKEY F1 WD 3;D 100;

After this command is entered, pressing the **F1** key would make the data window 3 lines long and dump data starting at 100H in the segment currently displayed in the data window.

WINICE.DAT Example

F1 = "WR;WD 2;WC 10;"

If this line is placed in WINICE.DAT, when SoftICE/W is loaded it will assign the string to **F1**. If **F1** is pressed while in SoftICE/W, it will toggle the register window, create a data window of length 2 and a code window of length 10. For more information on WINICE.DAT, see *WINICE.DAT Initialization File* on page 7.

DEX

Display or assign a data window expression.

Syntax

DEX [*data-window-number* [*expression*]]

data-window-number	A number from 0 to 3 indicating which data window to use. This number is displayed on the right hand side of the line above the data window.

Comments

The **DEX** command assigns a data *expression* to any of the four SoftICE/W data windows. Every time SoftICE/W pops up, the *expressions* are re-evaluated and the memory at that location is displayed in the appropriate data window. This is useful for displaying changing memory locations where there is always a pointer to the memory in either a register or a variable. The data is displayed in the current format of the data window: either byte, word, dword, short real, long real or 10-byte real. The effect of this command is as if you were to reenter the command **D** *expression* every time SoftICE/W pops up.

Typing **DEX** with no parameters will display all the *expressions* currently assigned to the data windows.

To unassign an *expression* from a data window, type **DEX** followed by the *data-window-number* followed by the *Enter* key.

To cycle through the four data windows, use the **DATA** command. See the description of the **DATA** command on page 100 for more details.

Example

DEX 0 SS:ESP

This example would create a stack window in data window 0. Every time SoftICE/W pops up, data window 0 would contain the contents of the stack.

DEX 1 @PointerVariable

Every time SoftICE/W pops up, data window 1 would contain the contents of the memory pointed at by the public variable PointerVariable.

CODE

Display instruction bytes.

Syntax

```
CODE  [ON | OFF]
```

Comments

The **CODE** command controls whether or not the actual hexadecimal bytes of an instruction are displayed when the instruction is unassembled. If **CODE** is **ON**, the instruction bytes are displayed. If **CODE** is **OFF**, the instruction bytes are not displayed.

Typing **CODE** with no parameters displays the current state of **CODE**.

The default is **CODE** mode **OFF**.

Example

```
CODE ON
```

This will cause the actual hexadecimal bytes of an instruction to be displayed when the instruction is unassembled.

COLOR

Display or set the screen colors.

Syntax

COLOR [*normal bold reverse help line*]

normal	This is the foreground/background attribute that is used to display normal text. (default = 07H grey on black)
bold	This is the foreground/background attribute that is used to display bold text. (default = 0FH white on black)
reverse	This is the foreground/background attribute that is used to display reverse video text. (default = 71H blue on grey)
help	This is the foreground/background attribute that is used to display the help line underneath the command window. (default = 30H black on cyan)
line	This is the foreground/background attribute that is used to display the horizontal lines between the SoftICE/W windows. (default = 02H green on black)

Comments

The **COLOR** command is used to customize the SoftICE/W screen colors on a color monitor. Each of the five specified colors is a hexadecimal byte where the foreground color is in bits 0-3 and the background color is in bits 4-6. This is identical to the standard CGA attribute format where there are 16 foreground colors and 8 background colors.

The actual colors represented by the 16 possible codes are given below:

0	black
1	blue
2	green
3	cyan
4	red
5	magenta
6	brown
7	grey
8	dark grey
9	light blue
A	light green
B	light cyan

C	light red
D	light magenta
E	yellow
F	white

Example

COLOR 7 F 71 30 2

This command will cause the following color assignments:

normal text	grey on black
bold text	white on black
reverse video text	blue on grey
help line	black on cyan
horizontal line	green on black

TABS

Display or set the tab settings for source display.

Syntax

TABS [*tab-setting*]

tab-setting	This can be a number from 1 through 8 that specifies how many columns between tab stops.

Comments

The **TABS** command is used to display or set *tab-settings* for the display of source files. Tab stops can be anywhere from 1 to 8 columns. The default **TABS** setting is 8. **TABS** with no parameters will display the current *tab-setting*. Specifying a *tab-setting* of 1 will allow the most source to be viewed since each tab will be replaced by a single space.

Example

```
TABS 4
```

This will cause the tabs setting to be changed to every fourth column starting at the first display column.

SERIAL

Redirect console to serial terminal.

Syntax

SERIAL [ON [*com-port*] [*baud-rate*] | OFF]

com-port	This is a number from 1 to 4 that corresponds to COM1, COM2, COM3 or COM4. The default is COM1.
baud-rate	This is the *baud-rate* to use for serial communications. The default is to have SoftICE/W automatically determine the fastest possible *baud-rate* that can be used. The allowable rates are listed below: • 1200, 2400, 4800, 9600,19200, 23040, 28800, 38400,57000, 115000.

Comments

Debugging on a serial console requires a second IBM compatible PC running MSDOS. Any PC will do, including 8088, 8086 or 80286 machines. You must first attach the computer to your Windows computer with a null modem cable attached to the two serial ports. Before using the **SERIAL** command, you must run the SERIAL.EXE program on the second PC. The syntax of SERIAL.EXE is as follows:

SERIAL ON *com-port baud-rate*

SERIAL.EXE has two optional parameters. The first parameter is used to specify which *com-port* on the second PC to use. The allowable *com-ports* are the same as those listed above. The second parameter is used to specify a *baud-rate*. The allowable *baud-rates* are the same as those listed above. For example, to use COM2 at 19200 baud you would specify:

```
SERIAL ON 2 19200
```

If no *com-port* is specified, SERIAL.EXE will use COM1. If no *baud-rate* is specified, SoftICE/W will attempt to determine the fastest possible *baud-rate* that can be used. If a *baud-rate* is specified for SERIAL.EXE, the same *baud-rate* must also be specified in the **SERIAL** command.

Prior to using the **SERIAL** command you must also place the **COMn** keyword on a separate line in the WINICE.DAT file to reserve a specific COM port for the serial connection. "n" is a number between 1 and 4 representing the COM port. If this statement is not present in WINICE.DAT, then you must pop SoftICE/W up from the keyboard on the SoftICE/W machine, not the second PC.

The SERIAL.EXE program allows SoftICE/W to use the same user interface on a serial machine that it uses on the Windows monitor. Once the serial connection is established, using SoftICE/W is identical to debugging on the local machine.

If *baud-rate* is not specified, SoftICE/W will automatically figure out the highest possible rate. It does this by starting at the lowest rate and incrementing that rate until characters begin to be lost. At this point SoftICE/W will use the next lowest *baud-rate*. This process takes a few seconds to establish a connection. During the process, the SERIAL.EXE program displays the *baud-rate* it is trying to use. Once the highest *baud-rate* is established, you can switch to specifying this rate in the **SERIAL** command and on the SERIAL.EXE command line. This will result in an immediate connection.

Once a connection is made between the two computers, the Windows screen will be restored and the serial terminal will always contain the SoftICE/W display.

Ctrl D is always the hot key sequence from the serial terminal keyboard. You can also pop up SoftICE/W from the main system keyboard with the standard hot key sequence.

If the computer you are using as the serial terminal has a monochrome display, you will probably want to use the color command to adjust the attributes. Entering **COLOR 07 0F 70 70 07** is recommended.

If the serial monitor starts losing characters and the display becomes corrupted, pressing **Shift \** (backslash) at any time will completely repaint the screen.

When finished debugging on the serial machine, SERIAL.EXE is terminated by pressing **Ctrl Z**.

Specifying **OFF** after the **SERIAL** command will switch back to debugging on the local display. **SERIAL** with no parameters will display the current serial state plus the *com-port* and *baud-rate* being used if **SERIAL** is **ON**.

Note

If you place the **SERIAL** command in the **INIT** statement of the WINICE.DAT initialization file, then SERIAL.EXE must be running on the serial terminal prior to running SoftICE/W.

Examples

SERIAL ON 19200 (on the second PC)

SERIAL ON 2 19200 (on the SoftICE/W machine)

This will cause SoftICE/W to switch its display to a serial terminal on COM2 at a *baud-rate* of 19200. The second machine will use COM1 at a *baud-rate* of 19200.

SERIAL ON 2 (on the second PC)

SERIAL ON (on the SoftICE/W machine)

This will cause SoftICE/W to switch its display to a serial terminal on COM1. SoftICE/W will automatically calculate the highest *baud-rate* that can be used. The secondary PC will use COM2.

LINES

Change number of lines of the SoftICE/W display.

Syntax

```
LINES  [25 | 43 | 50]
```

Comments

The **LINES** command changes SoftICE/W's character display mode. It allows three different display modes: 25-line, 43-line or 50-line mode. 43-line mode is only valid on VGA and EGA display adapters, and 50-line mode is only valid on VGA display adapters.

Typing **LINES** with no parameters displays the current state of **LINES**.

The default is number of display lines is 25. If your SoftICE/W display is on another computer connected by a serial cable, the display is fixed at 25 lines.

Note

If you enter the **SERIAL** command or the **ALTSCR** command, SoftICE/W changes to 25-line mode automatically. If you change back to your EGA or VGA display and want a larger line mode, you must enter the **LINES** command again.

Example

```
LINES 43
```

This will change the SoftICE/W display to 43-line mode if you have an EGA or VGA display.

Print-Screen

Print contents of screen.

Syntax

```
Print-Screen
```

Comments

Depressing the **Print-Screen** key does a screen dump to your printer. All information from the SoftICE/W screen is sent to the printer. The default printer port is LPT1, however this can be changed with the **PRN** command.

If you don't wish to dump to a printer, an alternative to **Print-Screen** is using the WLOG utility. WLOG can be run from Windows or from a DOS VM and writes the SoftICE/W command line window history to a file.

Note

Since SoftICE/W accesses the hardware directly for all of its I/O, **Print-Screen** works only on printers connected directly to a COM or LPT port. It does not work on network printers.

Also see the **PRN** command.

PRN

Set printer output port.

Syntax

PRN [LPT*x* | COM*x*]

x	A decimal number between 1 and 4 for COM or a decimal number between 1 and 2 for LPT.

Comments

The **PRN** command allows you to send output from **Print-Screen** to a different printer port.

If no parameters are supplied, **PRN** displays the currently assigned printer port.

Example

```
PRN COM1
```

This command causes **Print-Screen** output to go to the COM1 port.

FAULTS

Turn fault trapping on or off.

Syntax

`Faults [ on | off ]`

The FAULTS command allows SoftICE/W processor fault trapping to be turned on or off. This is mainly intended for users of both SoftICE/W and BOUNDS-CHECKER/W. For example, if you were Bounds-Checking a program and a fault occurred, SoftICE/W would get control. When you hit 'C' to continue, Bounds-Checker will pop up on the same fault. Users who wish processor faults to go directly into BOUNDS-CHECKER/W can do a FAULTS OFF command.

Example

FAULTS OFF

This command turns off fault trapping in SoftICE/W.

Screen Control Commands

FLASH	Restore screen during **P** and **T**
RS	Restore the program screen
CLS	Clear the command window
ALTSCR	Change to an alternate screen

FLASH

Restore the Windows screen during **P** and **T** commands.

Syntax

```
FLASH  [ON | OFF]
```

Comments

The **FLASH** command lets you specify whether the Windows screen will be restored during any **T** (trace) and **P** (program step) commands. If you specify that the Windows screen is to be restored, it is restored for the brief time period that the **P** or **T** command is executing. This feature is needed to debug sections of code that access video memory directly.

If the **P** command executes across a call, the screen will be restored, because the routine being called may write to the Windows screen. When debugging protected mode applications such as VxD's or Windows applications, this is not true. SoftICE/W will restore the screen only if the display driver is called before the call is completed.

If no parameter is specified, the current state of **FLASH** is displayed.

The default is **FLASH** mode **OFF**.

Example

```
FLASH ON
```

This command turns on **FLASH** mode. The Windows screen will be restored during any subsequent **P** or **T** commands.

RS

Restore the program screen.

Syntax

RS

Comments

The **RS** command allows you to restore the program screen temporarily.

This feature is useful when debugging programs that update the screen frequently. When the **RS** command is used, the program screen is redisplayed. When a key is pressed, the SoftICE/W screen is redisplayed.

Default Function Key

F4

CLS

Clear the command window.

Syntax

CLS

Comments

The **CLS** command clears the SoftICE/W command window and all display history and moves the prompt and the cursor to the upper left-hand corner of the command window.

Default Function Key

Alt F5

ALTSCR

Change to an alternate screen.

Syntax

```
ALTSCR  [ON | OFF]
```

Comments

The **ALTSCR** command allows you to redirect the SoftICE/W output from your default screen to the alternate monochrome monitor. This feature is useful, for instance, when you want to debug a program with heavy screen output.

ALTSCR requires the system to have two monitors attached. The alternate monitor should be a monochrome monitor in a character mode, which is the default mode for monitors.

The default is **ALTSCR** mode **OFF**.

Hint

If you want to change your SoftICE/W display screen every time you run SoftICE/W then you should place the **ALTSCR** command in the **INIT** statement of the WINICE.DAT initialization file.

Example

```
ALTSCR ON
```

This command redirects screen output to the alternate monitor.

Back Trace History Commands

TRACE	Enter or exit trace simulation mode
SHOW	Display from back trace history buffer
XT	Single step in trace simulation mode
XP	Program step in trace simulation mode
XG	Go to address in trace simulation mode
XRSET	Reset the back trace history buffer

TRACE

Enter or exit trace simulation mode.

Syntax

TRACE [**B** | **OFF** | *start*]

start	A hexadecimal number specifying the index within the back trace history buffer to start tracing from. An index of 1 corresponds to the newest instruction in the buffer.

Comments

The **TRACE** command is used to enter, exit and display the current state of trace simulation mode. **TRACE** with no parameters displays the current state of trace simulation mode. **TRACE** followed by **OFF** exits from trace simulation mode and returns to regular debugging mode. **TRACE B** enters trace simulation mode starting from the oldest instruction in the back trace history buffer. **TRACE** followed by a *start* number enters trace simulation mode at the specified index within the back trace history buffer.

Trace simulation mode can only be used if there are instructions in the back trace history buffer. The back trace history buffer must first be filled by specifying a range break point (**BPR**) with either the **T** or **TW** parameters.

When trace simulation mode is active, the help line on the bottom of the screen will show this, as well as the index of the current instruction within the back trace history buffer.

Once in trace simulation mode, the instructions in the back trace history buffer can be stepped through using the **XT**, **XP** and **XG** commands. When stepping through the back trace history buffer, the only register that will change is the EIP register since back trace ranges do NOT record the contents of all the registers. All SoftICE/W commands can be used in trace simulation mode except the following: **X**, **T**, **G**, **P**, **HERE**, and **XRSET**.

Example

TRACE 8

This command will enter trace simulation mode starting at the eighth instruction in the back trace history buffer.

TRACE OFF Default Function Key

Ctrl F9

TRACE B Default Function Key

Ctrl F12

SHOW

Display instructions from the back trace history buffer.

Syntax

SHOW [B | *start*] [L *length*]

start	A hexadecimal number specifying the index within the back trace history buffer to start disassembling from. An index of 1 corresponds to the newest instruction in the buffer.
length	The number of instructions to display.

Comments

The **SHOW** command is used to display instructions from the back trace history buffer. If source is available for the instructions then the display is in mixed mode, otherwise only code is displayed.

All instructions and source are displayed in the command window. Each instruction is preceded by its index within the back trace history buffer. The instruction whose index is 1 is the newest instruction in the buffer. Once **SHOW** has been entered, the contents of the back trace history buffer can be scrolled forward and backwards using the up and down arrow keys. The *Esc* key must be pressed to exit from **SHOW**.

SHOW with no parameters or **SHOW B** will begin displaying from the back trace history buffer starting with the oldest instruction in the buffer. **SHOW** followed by a *start* number begins displaying instructions starting at the specified index within the back trace history buffer.

SHOW can only be used if there are instructions in the back trace history buffer. The back trace history buffer must first be filled by specifying a range break point (**BPR**) with either the **T** or **TW** parameters.

Example

SHOW B

This command will start displaying instructions in the command window, starting at the oldest instruction in the back trace history buffer.

SHOW B Default Function Key

Ctrl F11

XT

Single step in trace simulation mode.

Syntax

XT [R]

Comments

The **XT** command is used to single step the current instruction in the back trace history buffer. It can only be used in trace simulation mode. This command will step to the next instruction contained in the back trace history buffer. **XT R** single steps backwards within the back trace history buffer.

Example

XT

This command single steps one instruction forward in the back trace history buffer.

XT **Default Function Key**

Ctrl F8

XT R **Default Function Key**

Alt F8

XP

Program step in trace simulation mode.

Syntax

XP

Comments

The **XP** command does a program step of the current instruction in the back trace history buffer. It can only be used in trace simulation mode. This command can be used to skip over calls to procedures and rep string instructions.

Example

XP

This command does a program step over the current instruction in the back trace history buffer.

Default Function Key

Ctrl F10

XG

Go to an address in trace simulation mode.

Syntax

```
XG  [R]  address
```

Comments

XG does a go to a specific code *address* within the back trace history buffer. It can only be used in trace simulation mode. The **R** parameter makes **XG** go backwards within the back trace history buffer. If the specified *address* is not found within the back trace history buffer, an error will be displayed.

Example

```
XG 2ff000
```

This command will make the instruction at address CS:2ff000 the current instruction in the back trace history buffer.

XRSET

Reset the back trace history buffer.

Syntax

XRSET

Comments

XRSET clears all information from the back trace history buffer. It can only be used when NOT in trace simulation mode.

Example

```
XRSET
```

This will clear the back trace history buffer.

Symbol and Source Line Commands

TABLE	Change or display current symbol table
EXP	Display export symbols
SYM	Display or set symbol
SYMLOC	Relocate the symbol base
SRC	Toggle between source, mixed and code
FILE	Change or display current source file
SS	Search current source file for a string

TABLE

Change or display the current symbol table.

Syntax

TABLE [[R] ***partial-table-name*****] | AUTOON | AUTOFF | $**

partial-table-name	A symbol table name or enough of the first few characters to define a unique name.
AUTOON	Key word that turns auto table switching on.
AUTOOFF	Key word that turns auto table switching off.
$	When '$' is specified, the current table becomes the table where the current instruction pointer is.

Comments

If no parameters are specified, then all of the currently loaded symbol tables are displayed with the current symbol table highlighted. If *partial-table-name* is specified, the specified table becomes the current symbol table. You do not have to specify the entire table name, but only enough characters to identify a unique table. If the **R** parameter precedes *partial-table-name*, the specified table is removed.

The **TABLE** command is used when you have multiple symbol tables loaded. SoftICE/W supports symbol tables for 16-bit Windows programs, 32-bit Windows VxDs, DOS programs, loadable device drivers, and T&SR's.

Note

Tables are not automatically removed when your program exits. If you re-load your program with WLDR.EXE then the symbol table corresponding to the loaded program is replaced with the new one.

Symbols are only accessible from one symbol table at time. You must use the **TABLE** command to switch to a symbol table before using symbols from that table.

If you use the **AUTOON** keyword, SoftICE/W will switch to auto table switching mode. This will cause the current table to become whichever table the instruction pointer is in when SoftICE/W pops up. **AUTOOFF** turns off this mode.

Examples

TABLE

```
MYTSR.EXE
MYAPP.EXE
MYVXD
GENERIC


006412 bytes of symbol table memory available
```

Since no parameters were specified, all loaded symbol tables are listed. GENERIC is highlighted, because it is the current table.   The amount of available symbol table memory is displayed at the bottom.

TABLE myt

The current table is changed to MYTSR.EXE. Notice that only enough characters to identify a unique table were entered.

EXP

Display export symbols from USER, GDI and KERNEL.

Syntax

EXP [*partial-name]**

partial-name	An export symbol or the first few characters of the name of an export symbol followed by '*'. If '*' is the last character of the string then all the exports that start with the specified characters will be displayed.

Comments

This command displays the export symbols from three of the Windows internal DLLs: USER, GDI and KERNEL. These exported symbols make up the bulk of the Windows API. If no parameters are specified, then all symbols from USER, GDI and KERNEL are displayed.

Example

EXP DELETE*

This would display all exported symbols from GDI, KERNEL and USER that start with "DELETE". A sample output for this command follows:

```
KERNEL
   010D:030E  DELETEPATHNAME00FD:412B  DELETEATOM
USER
   05DD:0583  DELETEMENU
GDI
   0465:0238  DELETELINEFONTS043D:07AD  DELETEDC
   050D:0FB7  DELETEJOB043D:16D0  DELETEOBJECT
   0505:0207  DELETEPQ04C5:169E  DELETEMETAFILE
```

SYM

Display or set symbol.

Syntax

SYM [[*module-name*] !] *symbol-name* [*value*]]

module-name	A valid module name. This can be a partial module name. This allows displaying symbols in a particular module. If *module-name* is specified, it must be followed by !
!	If ! is the only parameter specified, then the modules in this symbol table will be listed.
symbol-name	A valid symbol name. The *symbol-name* can end with an * (asterisk). This allows searching if only the first part of the *symbol-name* is known. The , (comma) character can be used as a wild card character in place of any character in the *symbol-name*.
value	This is a word *value* that is used if you want to set a symbol to a specific *value*.

Comments

The **SYM** command allows displaying and setting of symbols. If **SYM** is entered with no parameters, all symbols are displayed. The *value* of each symbol is displayed next to the *symbol-name*.

If *symbol-name* is specified with no *value*, then the *symbol-name* and its *value* are displayed. If the *symbol-name* was not found then nothing is displayed.

If *module-name*! precedes *symbol-name* or *, then only symbols from the specified module will be displayed.

The **SYM** command is often useful for finding a symbol when you can only remember a portion of the name. Two wild card methods are available for locating symbols. If *symbol-name* ends with an *, then all symbols that match the actual characters typed prior to the * will be displayed, regardless of their ending characters. If a , is used in place of a specific character in *symbol-name*, that character is a wild card character.

If *value* is specified, all symbols that match *symbol-name* are set to *value*. All symbols have word values.

Note

If an address is placed between square brackets as a parameter to the SYM command, then the closest symbol above and below the address will be displayed.

Examples

SYM FOO*

All symbols that start with FOO are displayed.

SYM FOO* 6000

All symbols that start with FOO are given the value 6000.

SYM !

All modules for the current symbol table are displayed.

SYM MAIN!FOO*

All symbols in module 'MAIN" that start with FOO are displayed.

SYMLOC

Relocate the symbol base.

Syntax

SYMLOC [*segment-address*]

Comments

The **SYMLOC** command adjusts the segment or selector symbols in a Windows or a DOS program. In a Windows program, **SYMLOC** converts all selectors in the symbol table from ordinal numbers to the actual selector values. **SYMLOC** is only necessary for a Windows program if the symbol table was loaded by the **LOAD** statement in the WINICE.DAT initialization file. The **SYMLOC** command should only be done once on a Windows program and is not reversible.

In a DOS program **SYMLOC** relocates the segment components of all symbols relative to the specified *segment-address*. This function is necessary when debugging loadable device drivers or other programs that can not be loaded directly with WLDR.EXE.

When relocating for a loadable device driver, use the value of the base address of the driver as found in the **MAP** command. When relocating for an .EXE program, the value is 10H greater than that found as the base in the **MAP** command. When relocating for a .COM program, use the base segment address that is found in the **MAP** command.

The **MAP** command will display at least two entries for each program. The first is typically the environment and the second is typically the program. The base address of the program is the relocation value.

Example

SYMLOC 1244+10

This will relocate all segments in the symbol table relative to 1244. The +10 is used to relocate a T&SR that was originally a .EXE file. If it is a .COM file or a DOS loadable device driver, the +10 is not necessary.

SRC

Toggle between displaying source, mixed and code in the code window.

Syntax

SRC

Comments

The **SRC** command toggles between source mode, mixed mode and code mode in the code window.

Example

SRC

This command changes the current mode of the code window. If the mode was source, it becomes mixed. If the mode was mixed, it becomes code. If the mode was code, it becomes source.

Default Function Key

F3

FILE

Change or display the current source file.

Syntax

FILE [[*]*file-name*]

Comments

If *file-name* is specified, that file becomes the current file and the start of the file is displayed in the code window. If no *file-name* is specified, the name of the current source file, if any, is displayed. If '*' is specified, then all files in the current symbol table are displayed.

The **FILE** command is often useful when setting a break point on a line that has no associated symbol. Use file to bring the desired file into the code window, use the **SS** command to locate the specific line, move the cursor to the specific line, then enter **BPX** or press **F9** to set the break point.

Note

Only source files that have been loaded into extended memory with WLDR.EXE are available with the **FILE** command.

Example

FILE MAIN.C

If MAIN.C had been loaded with WLDR.EXE, this command brings it up in the code window starting with line 1.

SS

Search the current source file for a string.

Syntax

SS **[*line-number*] ['*string*']**

line-number	A decimal number.
string	A character *string* surrounded by quotes.

Comments

The **SS** command searches the current source file for the specified character *string*. If there is a match, the line that contained the *string* will be displayed as the top line in the code window.

The search starts at the specified *line-number.* If no *line-number* is specified, the search starts at the top line displayed in the code window.

If no parameters are specified, the search continues for the previously specified *string*.

The code window must be visible and in source mode before using the **SS** command. To make the code window visible, use the **WC** command. To make the code window display source, use the **SRC** command.

Example

SS 1 'if (i==3)'

The current source file is searched starting at line 1 for the string 'if (i==3)'. The line containing the next occurrence of the string becomes the top line displayed in the code window.

A Running SoftICE/W and SoftICE for DOS Together

SoftICE for DOS v 2.52 (or above) can co-exist with SoftICE/W. Prior versions of SoftICE for DOS cannot reside in memory with SoftICE/W. When SoftICE and SoftICE/W are both resident in memory, they are two completely independent debuggers. When you enter "WINICE" to enter SoftICE/W and Windows, SoftICE for DOS becomes dormant until you exit from Windows.

SoftICE 2.52 Features

The following features have been added to SoftICE for DOS in version 2.52:

- The ability to co-exist with Windows in enhanced mode. When Windows is run in enhanced mode (with or without SoftICE/W), SoftICE for DOS becomes dormant until Windows exits.
- T&SRs and DOS loadable device drivers that are loaded high, with the SoftICE for DOS built-in memory manager, continue to be loaded high in Windows.
- A separate driver called UMB.SYS is provided that allows using MS-DOS version 5 load high commands in place of the SoftICE load high utilities.

Benefits of using SoftICE for DOS and SoftICE/W together.

When using both debuggers, you can have the SoftICE pop-up debugging capability any time in your system whether you are in or out of Windows.

You can debug entirely through complex DOS/Windows based system architectures that include both DOS and Windows components. This includes DOS loadable device drivers and T&SR's that come in prior to Windows but remain active all of the time. You can also debug Windows before it enters protected mode, including the real mode portions of your VxDs.

Requirements

- SoftICE for DOS must be installed in your CONFIG.SYS prior to HIMEM.SYS.
- Your memory requirement is the total memory required for SoftICE for DOS plus the total memory required for SoftICE/W.
- The symbol table and back trace memory are independent for each debugger. Therefore, if you are debugging a T&SR at source level that you must debug both prior to Windows and in Windows, you must have the symbols and source loaded separately for both debuggers.
- The user interfaces for the products are slightly different. For the most part SoftICE/W is a superset of SoftICE for DOS, though there are some features that SoftICE/W does not support, including small window mode and debugging overlaid programs.

B Using SoftICE/W with the Windows Debugging Kernels

Two separate debugging kernels are available from Microsoft: the standard debugging kernel and the VxD debugging version of WIN386.EXE. These debugging kernels provide additional error checking and diagnostic message generation.

If you are using one of these debugging kernels, SoftICE/W recognizes they are loaded and provides two additional capabilities: 1) the re-direction of diagnostic messages and 2) the ability to process "." commands.

When either of the debugging kernels are loaded, most diagnostic messages that normally go out to a serial port are re-directed to the SoftICE/W command window.

In addition, the "." commands in the VxD debugging version of WIN386.EXE can be entered from the SoftICE/W command line. Simply enter the command preceded by a ".", and the command will function like other SoftICE/W commands. Entering ".?" will give a list of the commands available in the VMM VxD. Also, if you are writing your own VxD you can use the debugging calls to use this interface.

Warning: If you are using the VxD debugging version of WIN386.EXE, you must use the /K switch on the WINICE command line or the KBD=TRUE statement in WINICE.DAT initialization file. This is a special switch required for the VxD debugging version of WIN386.EXE. It enables an alternate keystroke handler within SoftICE/W. If you do not use this switch then Windows will hang after popping up SoftICE/W a few times.

C Troubleshooting Guide

This appendix gives solutions to some possible problems that you could encounter when using SoftICE/W. If you do not find the problem here, check the README.SIW file on your distribution diskette for any troubleshooting hints that may not have made it into this manual.

Time does not show the correct time at the end of the day.

SoftICE/W does not let any interrupts go through to the system when the SoftICE/W screen is up. This does not affect the real time clock at all, so the next time you reboot, the time will be displayed correctly again. You can also correct the time by running the program UPTIME.EXE. This gets the time from the real time clock and calls DOS to set the time.

SoftICE/W does not display properly on your monitor.

SoftICE/W does not use the ROM BIOS or Windows drivers for its output, so it must go directly to the hardware. SoftICE/W was designed to work with 100% compatible VGA displays. SoftICE/W can also work with many CGA, EGA and super-VGA displays if you run the VIDMODE utility first. If you have a VGA or super-VGA display that SoftICE/W does not work properly on after running VIDMODE, try re-configuring Windows to use standard VGA mode.

If you have a display adapter that SoftICE/W is not compatible with, then you must use a secondary monochrome monitor or a second computer linked with a serial cable as your debugging screen. For information on a secondary monitor refer to the **ALTSCR** command on page 189, and for a serial terminal refer to the **SERIAL** command on page 179.

If the SERIAL display gets corrupted or characters are missing.

Use shift backslash (\) to repaint the screen. If the problem persists, try running at a slower baud rate.

SoftICE/W will not pop up from the SERIAL display.

The hot key sequence is always CTRL D from the serial display. If this does not work try placing the COMn statement in WINICE.DAT. Refer to the **SERIAL** command on page 179 for more information.

The key sequence used to bring up SoftICE/W conflicts with an existing program that you are running.

You can set a different key sequence to bring up SoftICE/W by using the **ALTKEY** command. Refer to the **ALTKEY** command on page 171.

You just exited from SoftICE/W back to Windows and the keyboard appears to be hung or behaves oddly.

Occasionally the *Ctrl* or *Alt* keys will be logically stuck in the down position after you return from SoftICE/W. This is due to timing issues related with SoftICE/W's key stroke handler. If this occurs, press and release all *Alt* keys and all *Ctrl* keys. This will reset the logical states of these keys to the up position.

Your program does not accept keystrokes, but the keyboard is still active.

A shift state key may be logically stuck down. Try pressing and releasing each *Shift*, *Ctl* and *Alt* key.

SoftICE/W hangs when it is popped up.

Some computers have keyboards that cause SoftICE/W to hang when it pops up. Placing the **NOLEDS** keyword in WINICE.DAT will eliminate this problem, but has the side effect of disabling the keyboard LED indicators from toggling while in SoftICE/W.

You were unassembling instructions, or editing or displaying memory in a software debugger that you were using as an assistant to SoftICE/W, when your debugger crashed.

You accessed an address that triggered a SoftICE/W break point, and **ACTION** was not set to **HERE** or the debugger *module-name*. When SoftICE/W brings you to the point where you want to look around in memory with your debugger, you should disable the SoftICE/W break points. If you don't, you could set off a break point unintentionally. This would cause your debugger to trigger itself, which can be a fatal problem with debuggers that cannot be re-entered. If you use the **ACTION** command with the *module-name* of the debugger specified, SoftICE/W will watch and prevent this situation.

After you exited from a software debugger that you were using as an assistant to SoftICE/W, the system crashed.

This problem could have many causes, but one possible cause is that you may have forgotten to disable the SoftICE/W break points, and **ACTION** is still set to trigger your debugger. When the break point occurs, **ACTION** will attempt to trigger your debugger, but your debugger is no longer loaded.

When Windows appears to be hung, SoftICE/W will not come up.

Situations can occur where SoftICE/W is unable to pop up. This is most likely to occur because of bugs in VxD's (VxD's run at protection level 0). Specific reasons include:

- Interrupts disabled in a VxD.
- The keyboard has been accidently disabled by sending illegal commands to the keyboard controller.
- Key portions of the SoftICE/W debugger have been corrupted by an errant VxD.

You want to debug a program in Windows standard mode with SoftICE/W.

SoftICE/W is not compatible with Windows standard mode. Windows must be in enhanced mode to run SoftICE/W.

The MAP, S, F, C or M commands do not complete properly.

Each of these commands accesses a range of memory. Occasionally you may do one of these commands when Windows has mapped out one or more pages within this range. The **S** command will skip over these pages without searching within them. The **MAP**, **F**, **C** and **M** commands will terminate at the point they reached the mapped-out page with an error message.

The system hangs or is corrupted after program stepping over a CALL instruction in Windows assembly language code.

This can occur from trying to step over a Windows thunk. A thunk is a call to an intermediate routine that sets up the stack frame and does other bookkeeping chores required for both real and protected mode operation. The call to the thunk is followed by data bytes. When you attempt to program step over the thunk, the data byte following the call instruction is replaced with an INT 3. This can cause the thunk to corrupt the stack.

If you attempt to step over a thunk, you will not return to SoftICE/W, because the INT 3 is never executed.

You can not pop up SoftICE/W from a remote machine attached through a serial connection.

You must place the COMn keyword on a separate line in the WINICE.DAT file to reserve a specific COM port for the serial connection. "n" is a number between 1 and 4 representing the COM port. If this statement is not present in WINICE.DAT, then you must pop SoftICE/W up from the keyboard on the SoftICE/W machine, not the second PC.

You pop up SoftICE/W or step through assembly code and you encounter a segment filled with HLT instructions, INT 30HS or an ARPL instruction.

Windows uses INT 30HS, HLT instructions and ARPL instructions to make protected mode level transitions. This is normal Windows operation.

You run out of SYM memory while loading a symbol table with many source files.

If your program has too many source files to fit into symbol memory, you can instruct WLDR to selectively load source files with a .SRC file; see page 28 for more information.

D Error Messages

Address Not Found

Either the **XG** command specified an *address* that was not found in the back trace history buffer, or the **XT** or **XP** reached the end of the back trace history buffer.

All Break Registers Used, Use In RAM Only

You were trying to set a **BPX** break point in ROM and all the debug registers were already used. **BPX** will still work in RAM since it uses the INT 3 method.

Attach to serial device has FAILED

You typed in the **SERIAL** command, and a timeout happened before the connection to the remote PC could complete

Backtrace Buffer Is Empty

You attempted to use a **SHOW** or **TRACE** command with an empty back trace history buffer. You must first use the **BPR** command with either the **T** or the **TW** parameter to fill the back trace history buffer.

BadSelector

You specified an invalid LDT *selector* in aHEAP or LHEAP command.

BPM Break Point Limit Exceeded

Only four **BPM**-style break points are allowed, due to restrictions of the 386/486 processor.

BPMD Address Must Be On DWord Boundary

The *address* specified in **BPMD** did not start on a word 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.

The following messages are displayed as reasons for SoftICE/W popping up:

Break Due to Debug Keyboard Request

Break Due to Embedded INT 1

Break Due to Embedded INT 3

Break Due to G

Break Due to General Protection Fault (0Dh). Fault=dddd

Break Due to HERE

Break Due to Hot Key

Break Due to Invalid Opcode Fault (06h)

Break Due to LDR

Break Due to Page Fault (0Eh). Fault=dddd

Break Due to Stack Fault (0Ch). Fault=dddd

Break Points Not Allowed Within SoftICE/W

You cannot set break points over SoftICE/W code.

Cannot Interrupt To A Less Privileged Level

You cannot **GENINT** from a level to a higher level. This is a restriction of the 386/486 processor.

The following messages are given when attempting to set a **BPR** over an invalid range.

Cannot Use Range: Overlaps GDT

Cannot Use Range: Overlaps IDT

Cannot Use Range: Overlaps LDT

Cannot Use Range: Overlaps Page Table

Cannot Use Range: Overlaps SoftICE/W

Command Is Not Valid In 32 Bit Mode

The **EXIT** command is only valid in PROT16 or VM mode.

Count Too Large

The specified *count* on a break point command was larger than 0FFH.

Debug Register Is Already Being Used

The *debug -reg* specified in the **BPM** command was already used in a previous **BPM** command.

Debugging Version Of WIN386 Is Not Loaded

A VxD "." command was used when the win386.exe debug version was not loaded.

Divide By Zero Error

A division was entered in the command that resulted in a divide by zero error.

Duplicate Break Point

The specified break point already exists.

Error Walking Global Heap

Windows has been corrupted so the global heap can no longer be walked.

Error Walking Module List

Windows has been corrupted so the module list can no longer be walked.

Error Walking Task List

Windows has been corrupted so the task list can no longer be walked.

Illegal Bit Mask

The *mask* specified for the **BPIO** or **BPM** command was invalid.

Illegal Count

An invalid *count* field was specified in a break point command.

Int0D Fault in SoftICE/W at address XXXXX offset XXXXX

Fault Code=XXXX

or

Int0E Fault in SoftICE/W at address XXXXX offset XXXXX

Fault Code=XXXX

These two messages are internal SoftICE/W errors. They mean that code within SoftICE/W 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 call us.

Invalid Address

The *address* specified in the current command was not valid.

Invalid Debug Register

A BPM *debug-reg* greater than 3 was specified. Valid debug registers are DR0, DR1, DR2 and DR3.

Invalid Expression

The *address* specified in the current command was not valid.

Invalid Indirection

The @ operator was attempted for the current expression, but the address pointed to by the indirection was not a valid address.

Invalid Interrupt Number

A **BPINT** *int-number* larger than 5FH was specified. **BPINT** works only for interrupts that are handled through the IDT. Currently, the IDT contains only interrupts 0-5FH. Interrupts above this are dispatched through general protection faults.

Invalid Range

An invalid *address* range was entered in the **BPR** command. See 60 for more information

Invalid Selector

The protected mode selector used was not valid. Use the **LDT** and **GDT** commands to determine valid selectors.

Invalid TSS

The BPIO command was specified when there was not a valid TSS in the system.

Invalid Window Handle

The *window-handle* specified in the **BMSG** command does not exist or is an invalid number.

I/O Port Not Found In I/O Bitmap

The **BPIO** command specified an invalid I/O *port* number.

Module Has No Code Segments

The *module-name* specified in the **CSIP** command must contain code segments.Use the **HEAP** ***module-name*** command to see what types of segments the module contains.

Module Not Found

The HEAPcommand specified a *module-name* that was not found. Use the MOD command to list the valid module names.

Must Be In Trace Simulation Mode

The **XT**, **XP** and **XG** commands can only be used when in trace simulation mode. You must first use the **BPR** command with either the **T** or the **TW** parameter to fill the back trace history buffer, then use the **TRACE** command to enter trace simulation mode.

No Backtrace Memory Allocated

You attempted to use a **BPR T** or **BPR TW** command without allocating back trace memory. You have changed the back trace memory size to 0. You must first allocate back trace memory by using the **/TRA** switch when loading SoftICE/W or specifying the **TRA** statement in the WINICE.DAT initialization file.

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 will only work 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 is displayed if you use certain Windows information commands (**HEAP**, **MOD**, **LHEAP**, **LDT**, and **TASK**) when there is no LDT. Refer to page 104 for more information.

No Local Heap

The **LHEAP** command specified a *selector* that has no local heap.

No More Break Points Available

A maximum of 32 break points are allowed in SoftICE/W.

No More Watch Variables Allowed

A maximum of 8 watch variables are allowed.

No Search In Progress

The **S** command was specified without parameters and no search was in progress. You must have first specified **S** with an *address* and a *data-list* for parameters. To search for subsequent occurrences of the *data-list* , you can then 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

This message occurs if you enter the **SYM**, **SS** or **FILE** command and there are no symbols currently present.

No TSS

At the time the TSS command was entered, there was no valid task state segment in the system.

Only Valid In Source Mode

The **SS** command cannot be used in mixed mode or code mode.

Only Valid In VM Mode

The **SYMLOC** command is only valid while you are in VM mode.

Page Not Present

The specified *address* was marked not present in the page tables.

This means that when SoftICE/W was trying to access some information, it accessed some memory that was in a page marked not present.

Parameter Is Wrong Size

One of the parameters entered in the command was the wrong size. For example, you would get this message if you use the **EB** or **BPMB** commands with a word value instead of a byte value.

Parameters Required

The specified command requires parameters. For a description and a syntax example of the command, specify **? *command*** in the SoftICE/W command window.

Pattern Not Found

The **S** command did not find a match in its search for the *data-list.*

Press 'C' to Continue, 'R' to Return to SoftICE

SoftICE/W popped up due to a fault (06, 0c, 0d, 0e). Press 'R' to return control to SoftICE/W. Press 'C' to pass the fault on to the Windows fault handler which will usually display the UAE box and terminate the application. When you exit SoftICE/W this message will reoccur because the fault has not been handled.

Press 'Z' for SoftICE/W or pass back any other key

SoftICE/W popped up due to a Debug Keyboard Request.

Windows requested a key through in_debug_chr. An example of this is FatalExit. Press 'Z' to return to SoftICE/W. Any other key will be passed on to Windows to satisfy the keyboard request. Normally there will be a prompt in the command window, such as "Abort, Break, or Ignore?" When you exit SoftICE/W this message will reoccur because the key request has not been satisfied.

Range Too Large

BPR *address* ranges cannot be larger than 4 meg.

Segment Limit Exceeded

When using the F,E,C, or M commands in a protected mode region, the selector limit has been exceeded. Use the LDT command with the selector to find its limit.

Selectors Must Be The Same

The **BPR** *start-address* and *end-address* selectors must be the same. If the range is over a protected mode region, then the starting address and the ending address must use the same selector as the segment portion of the address.

SoftICE/W Is NOT ACTIVE

This message is displayed on the help line on monochrome and serial displays when SoftICE/W is no longer in control.

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.

Syntax Error

The specified command had a syntax error. For a description and a syntax example of the command, specify **?** ***command*** in the SoftICE/W command window.

Type TRACE OFF To Exit Trace Simulation Mode

Commands that normally exit from SoftICE/W (**X,T,P,G,EXIT,GENINT**) and the hot key sequence (default is **Ctrl D**) do not work while in trace simulation mode. You must exit trace simulation mode with the **TRACE OFF** command before trying to exit SoftICE/W.

VxD Not Found

The VXD command specified a *VxD-name* that was not found. Type the command VXD with no parameters to get a list of valid VxD names.

E

Alphabetical Command List

?	Display help information	94
.	Locate current instruction in code window	168
A	Assemble code	159
ACTION	Set action after break point is reached	150
ALTKEY	Set alternate key sequence to invoke SoftICE/W	171
ALTSCR	Change to an alternate screen	189
BC	Clear break points	83
BD	Disable break points	78
BE	Enable break points	79
BH	Break point history	84
BL	List break points	80
BMSG	Set break point on Windows message	70
BPE	Edit break point	81
BPINT	Set break point on interrupt	66
BPIO	Set break point on I/O port access	64
BPM	Set break point on memory access or execution	57
BPR	Set break point on memory range	60
BPRW	Set multiple ranges on Windows program or code segment	62
BPT	Use break point as a template	82
BPX	Set/Clear break point on execution	68

C	Compare two data blocks	158
CLASS	Display information on Window classes.	133
CLS	Clear the command window	188
CODE	Display instruction bytes	175
COLOR	Display/set screen colors	176
CR	Display control registers	109
CSIP	Set CS:EIP (instruction pointer) range qualifier	72
CWATCH	Clear watch on expression	98
D	Display memory	90
DATA	Change data window	100
DEX	Display/assign data window expression	174
E	Edit memory	92
EC	Enter/exit the code window	166
EXIT	Force an exit of current program	146
EXP	Display export symbols	200
F	Fill memory with data	156
FAULTS	Turn on/off trapping of processor faults	184
FILE	Change or display current source file	205
FKEY	Show and edit function key assignments	172
FLASH	Restore screen during P and T	186
FORMAT	Change data window format	99
G	Go to an address	142
GDT	Display Global Descriptor Table	102
GENINT	Force an interrupt to occur	147
HBOOT	Hard system boot (total reset)	148
HEAP	Display Windows global heap	111
H	Display help information	94
HERE	Go to the current cursor line	145
HWND	Display information on Window handles.	131
I	Input from I/O port	138
I1HERE	Pop up on embedded INT 1 instructions.	151
I3HERE	Pop up on INT 3 instructions.	152
IDT	Display Interrupt Descriptor Table	106
LDT	Display Local Descriptor Table	104
LHEAP	Display Windows local heap	113

LINES	Change number of lines of SoftICE/W display	181
M	Move data	157
MAPV86	Display Virtual Machine memory map	127
MAP32	Display memory map of 32 bit modules	129
MOD	Display Windows module list	110
O	Output to I/O port	139
P	Program step	144
PAGE	Display page table information	123
PAUSE	Pause after each screen	170
PHYS	Display all virtual addresses for a physical address	126
Print-Screen	Print contents of screen	182
PRN	Set printer output port	183
R	Display or change registers	86
RS	Restore the program screen	187
S	Search memory for data	155
SERIAL	Redirect console to serial monitor	179
SHOW	Display from back trace history buffer	192
SRC	Toggle between source, mixed and code	204
SS	Search current source file for a string	206
STACK	Display a call stack	119
SYM	Display or set symbol	201
SYMLOC	Relocate the symbol base	203
T	Trace one instruction	143
TABLE	Change or display current symbol table	198
TABS	Display/set tab settings	178
TASK	Display Windows task list	117
TRACE	Enter or exit trace simulation mode	191
TSS	Display task state segment & I/O port hooks	107
U	Unassemble instructions	88
VCALL	Display VxD calls	121
VER	Display SoftICE/W version number	95
VM	Display information on virtual machines.	134
VXD	Display Windows VxD map	115
WATCH	Add watch expression	96
WC	Toggle/set the size of the code window	163

WD	Toggle/set the size of the data window	164
WMSG	Display Windows messages	122
WR	Toggle the register window	162
WW	Toggle the watch window	165
X	Exit from the SoftICE/W screen	141
XG	Go to address in trace simulation mode	195
XP	Program step in trace simulation mode	194
XRSET	Reset the back trace history buffer	196
XT	Single step in trace simulation mode	193
ZAP	Replace embedded INT1 or INT3 with NOP	153

F Hints On Debugging Win32s Applications

All Win32s applications and DLLs use one flat code selector and one flat data selector. However, the base of these selectors is not zero; it is actually 0ffff0000h (4gig-64K). You can verify this using the LDT command by entering LDT followed by your programs code selector. When using 32 bit offsets you must be careful to always use the Win32s selector and not the ring 0 flat vxd selector. The ring 0 selectors (28h and 30h) are based at zero, so using them with a Win32s offset will yield the wrong address.

The Win32s flat data selector is an expand down segment. If you do an LDT command on this selector its limit will be shown as 0fffh and an 'ED' will be displayed in the attribute field. What this means is that valid offsets for this selector are 0fffh to 4 gigabytes. This allows Win32s to trap null pointer accesses. So if SoftICE/W ever pops up with a fault and you are accessing an offset below 4K, this is the reason.

If you load exports or symbols for W32SKRNL but the addresses never get fixed up, you must use the debug version of Win32s. SoftICE/W uses Windows segment load notifications to fix up addresses of 32 bit modules. Unfortunately, Win32s only sends load notifications for W32SKRNL if the debug version of Win32s is loaded.

Sometimes in 32 bit code in assembly mode, SoftICE/W will show all

INVALID's as the instructions. This is because the code is not yet paged into memory. Single stepping once will page the code into memory.

If your application suddenly terminates for no reason it is probably causing a processor fault. Only the debugging version of Win32s passes

faults to a kernel debugger so you must install the debug version of Win32s if you want SoftICE/W to pop up on the faulting instruction.

Index