# Soft-Scope®

Remote-Target Debugger
for Windows 95 and Windows NT

## Copyright and Trademark Information

Copyright 1994, 1997 Concurrent Sciences, Inc. All rights reserved.
Third edition, first printing May 1997.

No part of this publication may be reproduced, translated into another language, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written consent of Concurrent Sciences, Inc.

*Other brands and names are marked with an asterisk and are the property of their respective owners.

Concurrent Sciences, Inc. makes no warranty of any kind with regard to this material, including but not limited to the implied warranties of merchantability and fitness for a particular purpose. Concurrent Sciences, Inc. assumes no responsibility for any errors that may appear in this document. Concurrent Sciences, Inc. makes no commitment to update or to keep current the information contained in this document.

**Concurrent Sciences, Inc.**
PO Box 9666 • Moscow, ID 83843 USA
(208) 882-0445 • Fax (208) 882-9774
info@consci.com • www.debugger.com

# Quick Contents

1. INTRODUCTION ........ 1-1

2. GETTING STARTED WITH SOFT-SCOPE ........ 2-1

3. SOFT-SCOPE BASICS ........ 3-1

4. CONTROLLING PROGRAM EXECUTION WITH SOFT-SCOPE ........ 4-1

5. EXAMINING DATA WITH SOFT-SCOPE ........ 5-1

6. CONFIGURING SOFT-SCOPE ........ 6-1

7. CREATING AND USING SOFT-SCOPE MACROS ........ 7-1

8. TOOLS THAT SOFT-SCOPE SUPPORTS ........ 8-1

A. DATA TYPES, OPERATORS, REGISTERS, AND DESCRIPTORS ........ A-1

1 2 3 4 5 6 7 8 A

## Documentation Conventions

**Computer output and code examples:** `Courier`, usually in a separate paragraph.

**Computer input:** **`Courier 11 bold`**, usually in a separate paragraph.

**Dialog box prompt:** Times, (data reference: or command:), in lower case. Prompt is followed by computer input.

**Command names and function names:** Bold italic, as in ***LOAD*** command or ***main()*** function.

**Variables**: Courier 11 italic (*`mt_busy`*).

**File names, configuration options**: Times bold (configuration option **targ.dev**), in lower case.

**Mouse buttons, keyboard keys and names**: Initial capital, in angle brackets, as in press <Enter> or double-click <Button-L>.

**Key presses:** An example of concurrent key presses is <Alt>+<Tab>.

**Menu names and selections, dialog box names, button names, window titles**: Times bold, as in **File** menu**.**

**Pull-down menu subfunctions:** Times bold, as in **File/Load**. The pull-down menu subfunctions are named by the selection path used to invoke them. The dots may be omitted in text.

**NOTES**: Indicate important information.

**CAUTION**: Indicate potential damage to hardware or data.

# Quick Contents (*continued*)

**B. ERROR MESSAGES ........................................................... B-1**

**C. DEBUGGING .EXE EXECUTABLE FILES ............................... C-1**

**D. HELPFUL HINTS ...........................................................D-1**

**E. ADD ONS ........................................................................ E-1**

**F. INTEL FLOATING-POINT EMULATION ................................. F-1**

**INDEX ........................................................................ INDEX-1**

B
C
D
E
F
I

(This page blank)

# Contents

**1. INTRODUCTION ........ 1-1**
**Overview ........ 1-2**
**FAQs: Frequently Asked Questions ........ 1-3**
**Chapter Summaries ........ 1-5**

**2. GETTING STARTED WITH SOFT-SCOPE ........ 2-1**
**Overview ........ 2-2**
**Installing Soft-Scope on the Host ........ 2-3**
Host-system Requirements ........ 2-3
Soft-Scope Distribution Disks ........ 2-3
Soft-Scope Installation for Windows 95 and Windows NT ........ 2-4
Figure 2-1: Changing baud rate using Options window ........ 2-5
**Invoking Soft-Scope ........ 2-6**
Message Window ........ 2-6
Figure 2-2: Initial Soft-Scope window with connect message ........ 2-7
**Loading Your First Application ........ 2-8**
Figure 2-3: File-Load dialog box ........ 2-9
Figure 2-4: Soft-Scope display after application load ........ 2-9
Figure 2-5: Run application to first line of main() ........ 2-10
**Troubleshooting ........ 2-11**
Symptoms of Problems ........ 2-11
Checklist of Corrective Actions ........ 2-11

**3. SOFT-SCOPE BASICS ........ 3-1**
**Overview ........ 3-3**
**Pull-Down Menu Map ........ 3-4**
Table 3-1: Pull-Down Menu Map ........ 3-4

**Window Pull-Down Menu** ........ **3-7**
Finding a String ........ 3-7
Figure 3-1: Find dialog box ........ 3-8
Capturing a Window to a Log File ........ 3-9
Figure 3-2: Log window showing capture of Trace window ........ 3-10
Saving Window Layout ........ 3-11
Open Window List ........ 3-11
**Accelerator Keys** ........ **3-12**
**Double-click Function** ........ **3-14**
Double-click in the Code Window ........ 3-14
Double-click on Data References ........ 3-14
Double-click on Pointers ........ 3-15
**Online Help** ........ **3-16**
**Commands and Command Line** ........ **3-17**
Figure 3-3: Command line dialog box ........ 3-17
Command Syntax Elements ........ 3-19
**Loading an Application** ........ **3-21**
Load ........ 3-21
Figure 3-4: File-Load dialog box ........ 3-22
Symbol Load ........ 3-24
Figure 3-5: File-Symbol load dialog box ........ 3-24
Restart ........ 3-25
Figure 3-6: File-Restart dialog box ........ 3-26
After the Load ........ 3-28
Figure 3-7: Soft-Scope after an application load ........ 3-28
Soft-Scope .tmp Files ........ 3-29
Command Line ........ 3-30

**4. CONTROLLING PROGRAM EXECUTION WITH SOFT-SCOPE ........ 4-1**

**Overview ........ 4-3**

**Controlling Program Execution ........ 4-3**

Stepping through Code ........ 4-4

Single Step ........ 4-4

Specify a Number of Steps ........ 4-5

Step Command via the Command Line ........ 4-5

**Code Window ........ 4-6**

Figure 4-1: Code window in Source mode ........ 4-7

Figure 4-2: Code reference dialog box ........ 4-8

Toolbar Buttons ........ 4-8

Figure 4-3: Display modes dialog box ........ 4-9

Figure 4-4: Code window in Assembly mode with logical addresses4-10

Code Window Execution Pointers ........ 4-11

**Code References ........ 4-12**

Line Numbers ........ 4-12

Symbol Names ........ 4-12

Guidelines ........ 4-13

**Locating Code ........ 4-14**

**Breakpoints Window ........ 4-16**

Figure 4-5: Breakpoints window ........ 4-17

Toolbar Buttons ........ 4-18

Command Line ........ 4-19

**Editing Breakpoints ........ 4-20**

Figure 4-6: Breakpoint edit dialog box ........ 4-20

**Software Breakpoints ........ 4-22**

Permanent Software Breakpoints ........ 4-22

Temporary Software Breakpoints ........ 4-23

**Hardware Breakpoints** .......... **4-24**
Data Breakpoints .......... 4-24
Command Line .......... 4-24
Debug Registers .......... 4-25
Exec Breakpoints .......... 4-26
Command Line .......... 4-26
**Executing to a Location** .......... **4-27**
Go .......... 4-27
Go to a Specific Location .......... 4-27
Return from a Procedure Call .......... 4-28
Go to a Cursor Position .......... 4-28
Stop .......... 4-28
**Procedure Call Sequence** .......... **4-30**
Calls Window .......... 4-30
Figure 4-7: Calls window .......... 4-31
Command Line .......... 4-31
**Stack Information** .......... **4-32**
**Trace Window** .......... **4-33**
Figure 4-8: Trace window displaying procedures .......... 4-33
Toolbar Buttons .......... 4-34
Figure 4-9: Assembly display modes dialog box .......... 4-35
Figure 4-10: Trace window displaying procedures and source .......... 4-36
Command Line .......... 4-37
Figure 4-11: Trace window displaying procedures, source, and assembly code .......... 4-37
Trace Buffer .......... 4-38
Trace File Size .......... 4-38

**5. EXAMINING DATA WITH SOFT-SCOPE .................................5-1**
Chapter Contents ................................................................... 5-1
**Overview .............................................................................5-3**
**Numbers ..............................................................................5-3**
Setting the Default Base ......................................................... 5-4
Table 5-1: Default number bases ............................................. 5-5
**Operators ............................................................................5-6**
Symbolic Operator Examples ................................................... 5-6
Arithmetic Operators Return Numeric Values ............................. 5-6
Logical Operator Examples ...................................................... 5-7
Table 5-2: C operators ............................................................ 5-8
Table 5-3: Soft-Scope specific operators and functions ................. 5-9
**Strings ...............................................................................5-10**
Escape Sequences ................................................................ 5-10
Where to Enter Strings ...........................................................5-11
Table 5-4: String escape sequences ..........................................5-11
**Reference Summary .............................................................5-12**
Table 5-5: Reference summary ............................................... 5-12
**The Data Window ..................................................................5-14**
Figure 5-1: Data reference dialog box ...................................... 5-14
Toolbar Buttons .................................................................... 5-15
Figure 5-2: Display modes dialog box ...................................... 5-15
Command Line ...................................................................... 5-16
Figure 5-3: Data window in Eval mode ..................................... 5-17
Double-click for Quick References ........................................... 5-17
Figure 5-4: Data window in expanded format ............................ 5-18
**Data References ...................................................................5-19**
Simple Variables ................................................................... 5-19
Referencing Arrays ................................................................ 5-20
Displaying an Entire Array .................................................. 5-20
Displaying a Single Element of an Array ............................... 5-20
Displaying a Selected Number of Arrays ................................ 5-20

Variables as Subscripts ........ 5-21
Referencing Structures ........ 5-21
Referencing Unions ........ 5-22
Referencing Bitfields ........ 5-22
Referencing Pointers ........ 5-23
Dereferencing Pointers ........ 5-23
Figure 5-5: Before double-click on “->” ........ 5-23
Figure 5-6: After double-click on “->” ........ 5-23
Selector Is Not Stored in Memory ........ 5-24
Making Complex Assignments ........ 5-24
**Referencing Memory ........ 5-25**
Using the Symbols Window to Find Code References ........ 5-26
**Reference Scoping ........ 5-27**
Examples ........ 5-27
Table 5-6: Reference Scoping ........ 5-28
Referencing Automatic (Stacked-based) Variables ........ 5-28
Referencing Register Variables ........ 5-29
**The Watch Window ........ 5-30**
Toolbar Buttons ........ 5-31
Figure 5-7: Display modes dialog box ........ 5-31
Figure 5-8: Watch window in Normal display mode ........ 5-32
Command Line ........ 5-33
Watching a Pointer ........ 5-33
Watching Memory ........ 5-33
**The Symbols Window ........ 5-34**
Toolbar Buttons ........ 5-34
Command Line ........ 5-35
Displaying Global Symbols ........ 5-36
Figure 5-9: Symbols window in Procedures mode ........ 5-36

**Built-in Functions** .......... **5-37**
Determining Addresses .......... 5-37
Using Return as a Memory Reference .......... 5-38
Determining How Many Elements in an Array .......... 5-38
Reading and Writing to Port Addresses .......... 5-38
**Type Overrides** .......... **5-40**
Applying a Type Override to a Variable .......... 5-40
Applying a Type Override to an Address .......... 5-41
Using a Variable to Superimpose its Data Type over the Address of Another Variable .......... 5-42
Using a User-declared Variable to Define a Type Override .......... 5-42
Changing the Amount of Memory Displayed .......... 5-43
Using Expressions in Type Overrides To Do Mathematical Operations 5-43
Assigning Values Using Type Overrides .......... 5-44
Displaying Data in its Most Useful Format .......... 5-44
**The Dump Window** .......... **5-46**
Toolbar Buttons .......... 5-47
Figure 5-10: Dump modes dialog box .......... 5-47
Command Line .......... 5-49
Figure 5-11: Dump window in Byte mode, 8 bytes per line .......... 5-49
**Uploading Memory and Registers** .......... **5-50**
Command Line .......... 5-51
Format of Upload Files .......... 5-51
**The Registers Window** .......... **5-52**
Toolbar Buttons .......... 5-52
Command Line .......... 5-53
Accessing Registers When the Target is Running .......... 5-53
Figure 5-12: Registers window for 80386EX target .......... 5-54
Contents of the Registers Window .......... 5-55

**CPU Structures** ........ **5-56**
Figure 5-13: IDT descriptors ........ 5-56
Figure 5-14: Data window in Normal mode ........ 5-58
Figure 5-15: Data window in Eval mode ........ 5-58
Command Line ........ 5-58
Table 5-7: Descriptor abbreviations ........ 5-59
Modifying a Descriptor Element ........ 5-59
**Real-Mode Structures** ........ **5-60**
Table 5-8: Peripheral Control Block ........ 5-60
Table 5-8: Peripheral Control Block (continued) ........ 5-61
$VECTOR[ ] Array ........ 5-64
**Application Input/Output** ........ **5-64**

**6. CONFIGURING SOFT-SCOPE** ........ **6-1**
**Overview** ........ **6-3**
**Options Window** ........ **6-3**
Toolbar Buttons ........ 6-4
Save and Restore Options ........ 6-4
Command Line ........ 6-5
Figure 6-1: Options window showing default values ........ 6-5
**Soft-Scope Configuration Options** ........ **6-6**
Table 6-1: Soft-Scope configuration options ........ 6-6
Control Default Number Base ........ 6-7
Change Log File Name ........ 6-7
Define Initial Command ........ 6-7
Define Initial Macro File ........ 6-7
Configure Host To Target Communications ........ 6-8
Control Screen Refresh Rate ........ 6-8
Control Command Delay ........ 6-8
Define Command ........ 6-9
Change Log File Size ........ 6-10
Define Path To Application Files ........ 6-10

Define Tab Spaces .......... 6-10
Define Case for Symbol Search .......... 6-10
Access CPU-specific Data Types .......... 6-11
Display LDTR register value .......... 6-11
Define Pointer Type Override Display .......... 6-12
Specify Integer Data Type Size .......... 6-13
Specify Floating Point Emulation Parameter .......... 6-14
Control Memory Caching .......... 6-14
Control Code Memory Cache Flush .......... 6-14
Define Host Communication Device .......... 6-15
Specify Where To Search For Memory Control Block .......... 6-15
Specify Where To Search for the NULL Device .......... 6-16
Specify Size of Memory Reads .......... 6-16
Tell Soft-Scope that Interrupts are Disabled .......... 6-16
Verify Memory Writes .......... 6-17
Specify temporary file location .......... 6-17
Specify the Size of the Trace File .......... 6-17
Preserve Trace Data across Applications .......... 6-18

**7. CREATING AND USING SOFT-SCOPE MACROS .......... 7-1**
**Overview .......... 7-3**
**Creating a Macro .......... 7-3**
Compiled Macro Files .......... 7-4
Built-in CPU Variables .......... 7-5
**Macros Window .......... 7-6**
Loading a Macro File .......... 7-6
Toolbar Buttons .......... 7-6
Figure 7-1: Macros window .......... 7-7
Command Line .......... 7-7
Example Use of cmd.macro and load.init_command .......... 7-8
Identify Macros in the Macros Window .......... 7-9

**Macro Parameters ........ 7-10**
Optional Parameters ........ 7-10
Integer Type ........ 7-10
LITERAL Parameter ........ 7-11
TEXT Parameter ........ 7-12
EXPRESSION Parameter ........ 7-12
REFERENCE Parameter ........ 7-12
ADDRESS Parameter ........ 7-13
LINE Parameter ........ 7-13
MODULE and PROCEDURE Types ........ 7-13
**Local Variables ........ 7-14**
Declaring Local Variables ........ 7-14
Defining One-dimensional Arrays ........ 7-15
Assigning Numeric Values to Arrays ........ 7-15
Assigning Pointer Values from Your Application ........ 7-16
**Macro Statements ........ 7-17**
ABORT ........ 7-17
BREAK ........ 7-17
IF, IF...ELSE ........ 7-17
RESPOND ........ 7-18
RETURN ........ 7-18
WHILE ........ 7-18
MACRO SUSPEND ........ 7-19
MACRO RESUME ........ 7-19
Custom Commands with an Extended Monitor ........ 7-20
**Manipulating Windows from Macros ........ 7-22**
WMOVE ........ 7-23
WRESIZE ........ 7-23
WFUNCTION ........ 7-23
Examples ........ 7-24

**Macro Print Function ........ 7-25**
PRINT ........ 7-25
Conversion Specifiers ........ 7-25
Table 7-1: Conversion specifiers ........ 7-26
$ Parameter Prefix in Control Strings ........ 7-27
Escape Sequences ........ 7-27
Directed Output from Macros ........ 7-27
Using Field-width Specifiers with PRINT or WPRINTF ........ 7-28
Specifying the Leading Zero Flag ........ 7-28

**8. TOOLS THAT SOFT-SCOPE SUPPORTS ........ 8-1**
**Tool Summary ........ 8-2**
Table 8-1: Supported tools ........ 8-2
Sample Files ........ 8-4
**Linking Your Application ........ 8-5**
CSi-Link™ ........ 8-5
**Generating Symbolic Information ........ 8-6**
SSBUG ........ 8-6
**Tool Directives ........ 8-7**
Borland ........ 8-7
Intel ........ 8-7
ASM86, ASM286 and ASM386 ........ 8-7
BND286/386 and BLD286/386 ........ 8-8
Intel iC-86, iC-286 and iC-386 ........ 8-9
Intel LINK86/LOC86 ........ 8-9
Intel PL/M-86, PL/M-286 and PL/M-386 ........ 8-10
MetaWare ........ 8-10
Microsoft ........ 8-10
Phar Lap ........ 8-11
Phar Lap LinkLoc ........ 8-11
Phar Lap 386/ASM ........ 8-12
Watcom ........ 8-12

**A. DATA TYPES, OPERATORS, REGISTERS, AND DESCRIPTORS ........................................................................................... A-1**
**Data Types ............................................................................... A-2**
Table A-1: Data types for use in type overrides ............................. A-2
**Operators ............................................................................... A-8**
Table A-2: Soft-Scope operators .................................................. A-8
**General-Purpose Registers ............................................................. A-10**
Figure A-1: General-purpose registers ........................................ A-10
Figure A-2: Flags register ........................................................... A-11
Figure A-3: Segment registers ..................................................... A-12
**NPX Registers ............................................................................ A-13**
Figure A-4: NPX registers ........................................................... A-13
**Protected-Mode Registers ............................................................. A-14**
Figure A-5: Control registers ...................................................... A-14
Figure A-6: Protected-mode registers ......................................... A-14
**Descriptors and Subfields ............................................................. A-15**
Table A-3: 386 protected-mode variables ................................... A-15
Table A-4: Page table entries ...................................................... A-15
Table A-5: Descriptor subfields .................................................... A-16
Table A-6: TSS386 subfields ....................................................... A-17
Table A-6: TSS386 subfields (continued) .................................... A-18

**B. ERROR MESSAGES ................................................................... B-1**
**Overview ................................................................................... B-2**
**Address Error Messages ................................................................ B-3**
Example Address Error Message ..................................................... B-3
Explanation ......................................................................................... B-3
How To Interpret Address Errors .................................................... B-4
Table B-1: Conversion entry codes ............................................... B-4
Table B-2: Address error messages .............................................. B-5
**Error Messages ........................................................................... B-7**

**C. DEBUGGING .EXE EXECUTABLE FILES ............................... C-1**
**Overview ......................................................................................... C-2**
**Debugging .exe Files .............................................................................. C-2**
Preparing Your Application ........................................................... C-2
Using the Special Monitor ............................................................ C-3
Loading an .exe Application .......................................................... C-3

**D. HELPFUL HINTS ............................................................................ D-1**
**Overview ......................................................................................... D-2**
**Helpful Hints ................................................................................... D-3**
Changing the Execution Point ........................................................ D-3
Source Line Address ..................................................................... D-3
Changing an Executable Instruction ............................................... D-4
Bypassing Start-up Code ............................................................... D-5
Copying Memory .......................................................................... D-5
Receiver Timeouts ........................................................................ D-6
Segment Limit Exceeded ................................................................ D-6

**E. ADD ONS ......................................................................................... E-1**
**Real-Time Operating Systems Support ............................................ E-2**
Kernel Objects .................................................................................. E-3
Figure E-1: SuperTask! kernel objects dialog box ......................... E-3
Task List ......................................................................................... E-4
Figure E-2: SuperTask! task list dialog box .................................. E-4
Current Task ................................................................................... E-4
Figure E-3: SuperTask! current task dialog box ............................ E-4

**F. INTEL FLOATING-POINT EMULATION ................................ F-1**
**Overview ........................................................................................ F-2**
**Intel Floating-Point Emulation ............................................................ F-2**

**INDEX ........................................................................................ INDEX-1**

# 1. Introduction

1

## Chapter Contents

**Overview .......................................................................................1-2**
**FAQs: Frequently Asked Questions ....................................................1-3**
**Chapter Summaries ..................................................................1-5**

# Overview

Soft-Scope is a remote-target, source-level debugger for embedded-system development. It contains basic features found in other Windows-based debuggers, such as pull-down menus, dialog and text boxes, display and modification of symbols and CPU structures, source-code display, execution trace, and single- or multiple- instruction execution control. This version of Soft-Scope works with our CSi-Mon target-resident monitor. See the *CSi-Mon Monitor User's Guide* for information about installing the monitor.

In the following pages, you will find answers to questions frequently asked about Soft-Scope and CSi-Mon. The chapter closes with a brief summary of each chapter in the manual.

# FAQs: Frequently Asked Questions



### How do I get technical support?

If you have a current maintenance contract, contact our technical support staff by telephone at (208) 882-0445 (9am - 5pm, Pacific Time), by email at tech@consci.com, or by fax at (208) 882-9774. If you need to purchase a maintenance contract, contact our sales staff at (800) 897-3001, (208) 882-0445, or by email at info@consci.com.

### What are the host-system requirements?

For you to install and run Soft-Scope properly, your host computer must be able to run Windows 95 or NT version 3.5x or 4.0 and have 2MB of free RAM and 6MB of free disk space.

For a list of the development tools (compilers, assemblers, linkers, and locators) that Soft-Scope supports, see the chapter, *Tools that Soft-Scope Supports*.

### What are the target-system requirements?

The CSi-Mon monitor can be configured to support most of the x86 16- and 32-bit processors running in real or protected mode. For a complete list, refer to the *CSi-Mon Monitor User's Guide*. For protected-mode applications, the monitor requires approximately 8KB of code space and 20KB of combined data and stack space. For real mode, it requires 4KB code space and 14KB data and stack space.

### What are the communication requirements for Soft-Scope and CSi-Mon?

Typically, the CSi-Mon monitor communicates with Soft-Scope via an RS-232 serial link. CSi-Mon supports the NS16550, NS16450, 8251, and 8274 UARTs. Other UARTS can be supported by modifying the source

file **siuart.c** and the header file **siuart.h**, which can be found among the CSi-Mon source files. Soft-Scope uses the host PC's serial port.

### What other hardware and software can be used with Soft-Scope?

Soft-Scope supports a variety of in-circuit emulators, logic analyzers, evaluation boards, ROM emulators and RTOS kernels. Please contact our technical sales department at (800) 897-3001, (208) 882-0445, and info@consci.com for a complete list.

# Chapter Summaries



This *Soft-Scope User's Guide* contains the following chapters:

### 1. Introduction

This chapter contains some frequently asked questions and provides basic information that will help you use this manual.

### 2. Getting Started with Soft-Scope

Read this chapter to learn how to install and invoke Soft-Scope, and how to load your first application. There is a troubleshooting section at the end of the chapter to help you resolve problems with getting Soft-Scope up and running.

### 3. Soft-Scope Basics

This chapter contains general descriptions of Soft-Scope's pull-down menus, windows, and commands. It also discusses how to load an application.

### 4. Controlling Program Execution in Soft-Scope

This chapter describes how to execute your application and view its source code. It discusses in detail how to reference the source code, single step, step to a specified location, use breakpoints, and examine procedure-call nesting.

### 5. Examining Data with Soft-Scope

Read this chapter to learn how to access data, as well as how to use some of the more advanced features of Soft-Scope. For example, in this chapter you will learn how to directly reference memory, how to use type overrides to display the most useful information, and how to use Soft-Scope's built-in functions.

### 6. Configuring Soft-Scope

Soft-Scope allows you to configure many of its functions and commands to best fit your needs. This chapter provides detailed information about each configuration option available.

### 7. Creating and Using Soft-Scope Macros

This chapter describes Soft-Scope's macro language, which allows you to customize the debugger to meet your specific needs.

### 8. Tools that Soft-Scope Supports

You should read this chapter before you start debugging an application. It is a tool-by-tool explanation of how to prepare an application for debugging so that it is fully compatible with Soft-Scope.

### Appendix A: Data Types, Operators, Registers, and Descriptors

This appendix contains tables and figures of supported data types, registers, and CPU structures.

### Appendix B: Error Messages

Refer to this appendix for a list of error messages and what they mean.

### Appendix C: Debugging .exe Executable Files

Read this appendix for information about debugging **.exe** files on a target PC.

### Appendix D: Helpful Hints

Refer to this appendix for some helpful hints.

### Appendix E: Add Ons

Includes information about RTOS support using the Soft-Scope Kernel Awareness Standard.



### Appendix F: Intel Floating-Point Emulation

This appendix describes how to configure Soft-Scope to recognize Intel 8087 floating-point emulation instructions.

### Index

# 2. Getting Started with Soft-Scope

2

## Chapter Contents

**Overview** ........................................ **2-2**
**Installing Soft-Scope on the Host** ........................................ **2-3**
Host-system Requirements ........................................ 2-3
Soft-Scope Distribution Disks ........................................ 2-3
Soft-Scope Installation for Windows 95 and Windows NT ........................................ 2-4
Figure 2-1: Changing baud rate using Options window ........................................ 2-5
**Invoking Soft-Scope** ........................................ **2-6**
Message Window ........................................ 2-6
Figure 2-2: Initial Soft-Scope window with connect message ........................................ 2-7
**Loading Your First Application** ........................................ **2-8**
Figure 2-3: File-Load dialog box ........................................ 2-9
Figure 2-4: Soft-Scope display after application load ........................................ 2-10
Figure 2-5: Run application to first line of main() ........................................ 2-11
**Troubleshooting** ........................................ **2-12**
Symptoms of Problems ........................................ 2-12
Checklist of Corrective Actions ........................................ 2-12

# Overview

This chapter describes procedures for installing and invoking Soft-Scope and loading your first application. A troubleshooting section is included to help you resolve commonly encountered problems.

It is assumed that you have already installed the CSi-Mon monitor on your target board or PC. If not, see the *CSi-Mon Monitor User's Guide* for instructions on how to install the CSi-Mon monitor.

We recommend you read the **readme.wri** file included on distribution disk number one for any information about your version of Soft-Scope that became available after this manual went to press.

# Installing Soft-Scope on the Host

## Host-system Requirements



Your host computer must have at least an 80486 processor, a hard drive with at least 6MB of free disk space, and at least 2MB of free RAM. We recommend you use a VGA monitor with 800x600 resolution.

A serial port is required to connect your host PC to the CSi-Mon monitor running on your target board. See the *CSi-Mon Monitor User's Guide* for more details on communicating with your target board. Windows NT version 3.5x, 4.0, or Windows 95 must be installed on the host.

## Soft-Scope Distribution Disks

The Soft-Scope software comes on four distribution disks:

Disk 1 The Soft-Scope for Windows executable file (**sswin32.exe**), support files, and **readme.wri**.

Disk 2 Soft-Scope **.dll** files.

Disk 3 Support files and sample programs in directory **\samp**.

Disk 4 A ROMmable sample program.

For a detailed listing of installed files and where they are installed, see **contents.wri** on disk one. After installation, this file and **readme.wri** will be in the directory containing **sswin32.exe** (default = **sswin**).

# Soft-Scope Installation for Windows 95 and Windows NT

To install Soft-Scope to run under Windows 95/NT on your host computer, follow these six steps:

1. Invoke Microsoft Windows.
2. Place disk 1 in the floppy disk drive from which you will install Soft-Scope.
3. Choose **Start/Run** from the Windows taskbar. The **Run** dialog box will open.
4. Type **`x:install`** in the Open text box, where *x* is the disk drive from which you are installing. Choose OK.
5. Insert disks 2, 3, and 4 when prompted.
6. Serial communication parameters are defined in the **sswin32.ini** file found in the directory where you installed Soft-Scope (default = **sswin**). By default Soft-Scope will use the standard Windows serial device driver (**comm.drv**), 9600 baud and the com2 port. If you need to change these values, use the **Display** command from the **Options** pull-down menu. Soft-Scope supports baud rates up to 115200. To select another baud rate, double-click <Button-L> on connect.baudrate=9600 in the **Options** window and enter the new baud rate in the text box. To select another com port, double-click <Button-L> on connect.comport=com2 and enter the new com port. See figure 2-1 for an example of changing the baud rate to 115200.

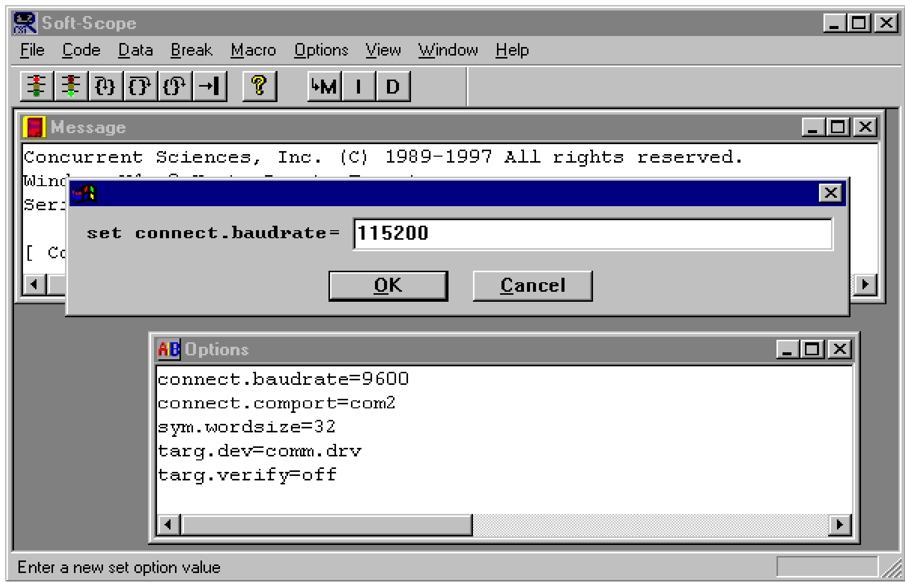



Figure 2-1: Changing baud rate using Options window

# Invoking Soft-Scope

Before invoking Soft-Scope for the first time, you will need to startup the CSi-Mon monitor running on your target (board or PC) and connect your host PC to your target's serial port. Make sure you are using the correct com ports and both the host PC and target are using the same baud rate. Use a serial communication program such as Kermit or HyperTerminal to determine if your host PC can talk to your target.

To invoke Soft-Scope, select **Programs/Soft-Scope/Soft-Scope** from the Windows taskbar or create a Windows shortcut. Upon execution, the Soft-Scope main window will open followed by the **Message** window.

# Message Window

The initial Soft-Scope screen, as shown in figure 2-2, contains a menu bar and the **Message** window. Notice the messages inside the window include version and copyright information, serial number, and a message about establishing contact with the target. If Soft-Scope cannot make contact with the target, you'll see the error message "Remote - Target not responding" in the middle of your screen. In that case you'll need to troubleshoot your serial connection to the target. See the *Troubleshooting* section later in this chapter and in the *CSi-Mon Monitor User's Guide*.

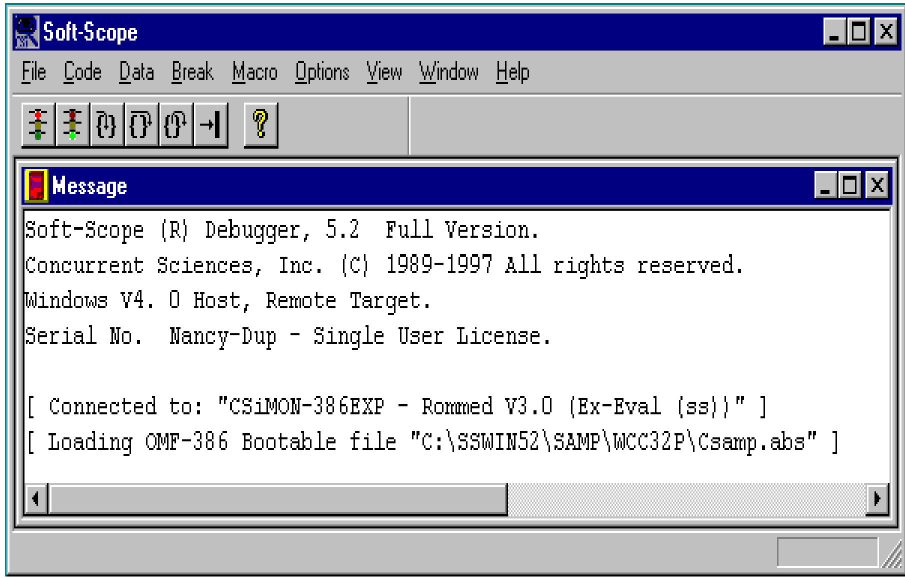
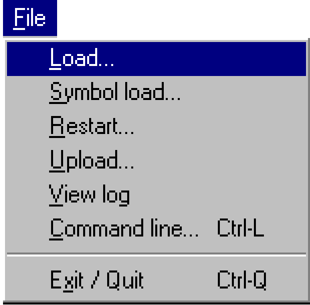



Figure 2-2: Initial Soft-Scope window with connect message

# Loading Your First Application

Soft-Scope can be used to debug absolutely located, bootable files, prepared with tools discussed in the *Tools that Soft-Scope Supports* chapter. A loadable application image for Soft-Scope contains both executable instructions and associated symbolic information.

In this section, we will load one of the **csamp** programs found in the **\samp** subdirectory. For a complete discussion of loading applications, see the section *Loading an Application* in the chapter *Soft-Scope Basics*.




Use Soft-Scope to download bootable absolute files to the target by following these steps:

1. Choose the **Load...** command from the **File** pull-down menu to open the dialog box shown in figure 2-3.
2. Enter the file name, or choose the **Browse...** button to select a file from directory listings.

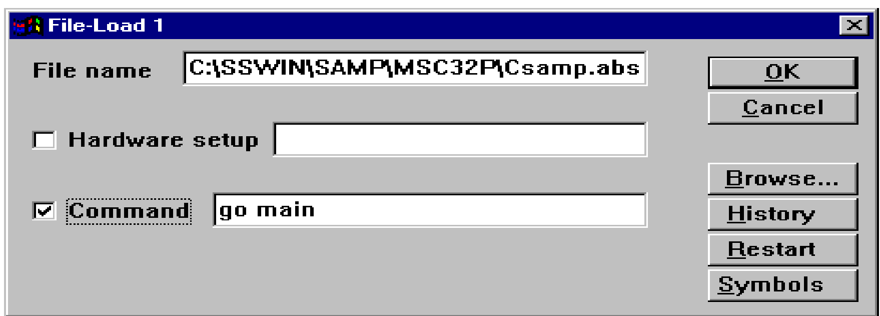



Figure 2-3: File-Load dialog box



3. After making all of your selections in the **File-Load** dialog box, click on the **OK** button. You should see the status line at the bottom of the **Soft-Scope** window recording the percentage of file loaded as your application is loaded. The **Code** window will then open showing your application's startup code in source mode as shown in figure 2-4.

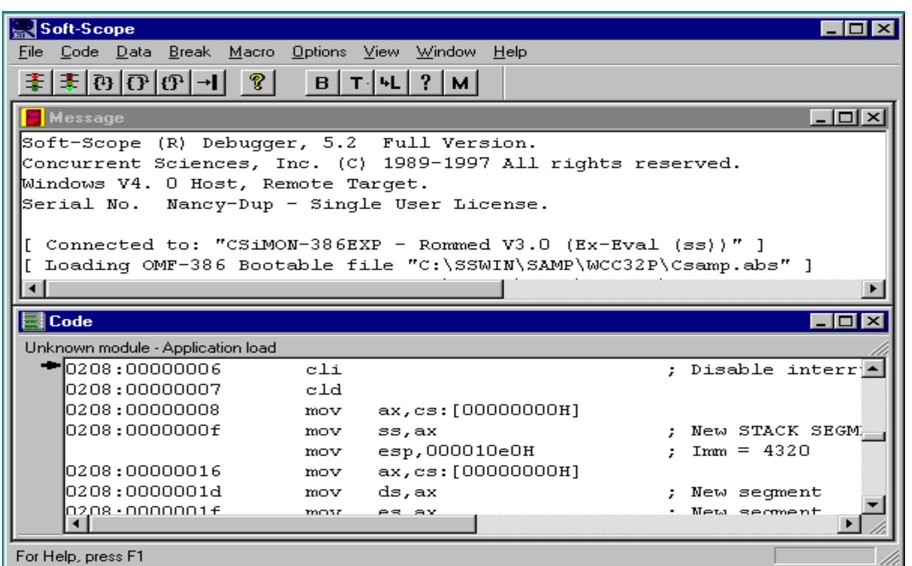



Figure 2-4: Soft-Scope display after application load

The arrow in the **Code** window shows which line of code is referenced by the instruction pointer. To run your application to the first line in `main.c`, select the **Go to...** command from the **Code** pull-down menu and enter **`go main`** in the text box as shown in figure 2-5.

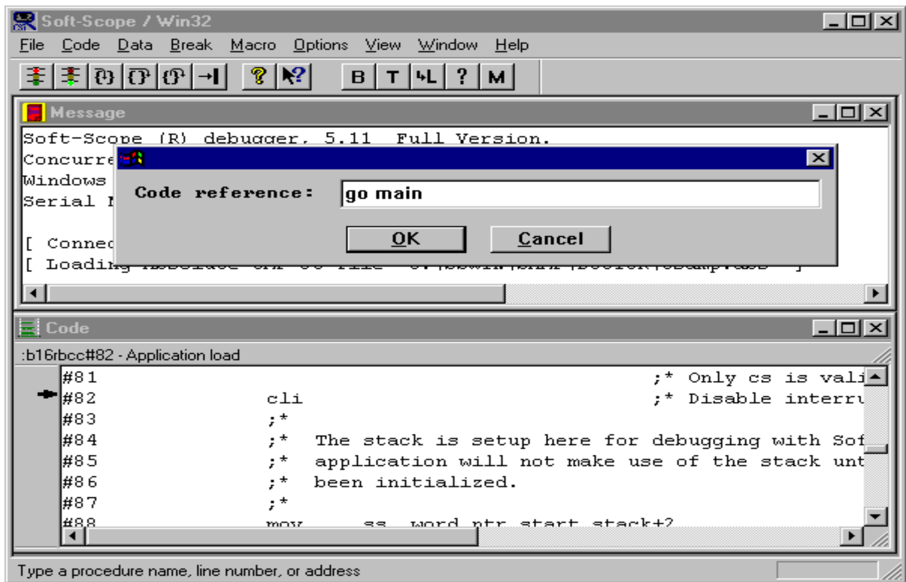



Figure 2-5: Run application to first line of main()

You are now ready to set breakpoints and step through your application. Detailed information about these and other commands are found later in this user's guide and in Soft-Scope's online help.

# Troubleshooting

This section will help you identify problems that may arise during your first Soft-Scope session. For more information, see the *Troubleshooting* section of the *CSi-Mon Monitor User's Guide*.



## Symptoms of Problems

- The Soft-Scope initial display doesn't appear as expected.
- Soft-Scope reports it can not communicate with the CSi-Mon monitor.
- Soft-Scope printed an error message about a configuration option.
- The target application won't load.
- The target application loads, but won't execute properly.

## Checklist of Corrective Actions

1. Have you accidentally altered the directory structure that Soft-Scope created when it was installed (**contents.wri** lists that directory structure)? If so, Soft-Scope won't know where to find information that it needs to operate.
2. Are all of your cable connections tight? If you are using a PC target or if your hardware requires it, are you sure that your serial connection has a null-modem configuration? [An easy way to affect this configuration is to attach an inexpensive null-modem adapter to your serial cable.]
3. Make sure you are using the correct com port and baud rate. Try talking to your target using a terminal program such as Kermit or HyperTerminal to confirm your serial connection is working properly.

4. If you are loading a real-mode application, you must be running a real-mode version of the CSi-Mon monitor on your target board or PC. A protected-mode application requires a protected-mode monitor.

# 3. Soft-Scope Basics

## Chapter Contents

**Overview** ........................................ **3-3**
**Pull-Down Menu Map** ........................................ **3-4**
Table 3-1: Pull-Down Menu Map ........................................ 3-4
**Window Pull-Down Menu** ........................................ **3-7**
Finding a String ........................................ 3-7
Figure 3-1: Find dialog box ........................................ 3-8
Capturing a Window to a Log File ........................................ 3-9
Figure 3-2: Log window showing capture of Trace window .......... 3-10
Saving Window Layout ........................................ 3-11
Open Window List ........................................ 3-11
**Accelerator Keys** ........................................ **3-12**
**Double-click Function** ........................................ **3-14**
Double-click in the Code Window ........................................ 3-14
Double-click on Data References ........................................ 3-14
Double-click on Pointers ........................................ 3-15
**Online Help** ........................................ **3-16**
**Commands and Command Line** ........................................ **3-17**
Figure 3-3: Command line dialog box ........................................ 3-17
Command Syntax Elements ........................................ 3-19
**Loading an Application** ........................................ **3-21**
Load ........................................ 3-21
Figure 3-4: File-Load dialog box ........................................ 3-22
Symbol Load ........................................ 3-24
Figure 3-5: File-Symbol load dialog box ........................................ 3-24
Restart ........................................ 3-25
Figure 3-6: File-Restart dialog box ........................................ 3-26

After the Load .................................................................... 3-28
Figure 3-7: Soft-Scope after an application load ........................... 3-28
Soft-Scope .tmp Files ...................................................................... 3-29
Command Line .................................................................................. 3-30

# Overview

Soft-Scope for the Microsoft Windows 95/NT operating system uses Windows conventions whenever possible, so getting around in Soft-Scope is similar to using your other Windows applications. For details on how to manipulate windows and use the PC keyboard and mouse, see Soft-Scope's online help and your Microsoft Windows user's guide.

Soft-Scope offers features specific to debugging embedded applications. Many of them, such as the window-capture feature, require special understanding. This chapter describes these features and the application loading process.

# Pull-Down Menu Map

Table 3-1 lists the Soft-Scope menu map items and associated pull-down commands. A brief summary is given for each command. Window, File, and Help commands will be discussed in this chapter. The chapters that follow will cover the rest of the pull-down commands.

Table 3-1: Pull-Down Menu Map

| File | Code | Data |
|---|---|---|
| **Load...**<br>Download application symbols and data. | **Display...**<br>Enter a reference to activate the Code window. | **Examine...**<br>Evaluate a data expression. |
| **Symbol load...**<br>Download symbolic information only. | **Module**<br>Display program modules in the Symbols window. | **Watch...**<br>Place a variable into the Watch window. |
| **Restart...**<br>Reset registers and reload descriptor tables. | **Calls**<br>Display procedure call nesting. | **Symbols**<br>Display application symbols in the Symbols window. |
| **Upload...**<br>Save memory/registers to a file for later debugging. | **Trace**<br>Display execution trace. | **Registers**<br>Display CPU registers in the Registers window. |
| **View log**<br>View the contents of the log file. | **Step into**<br>**Step over**<br>Step into or over a procedure call. | **Dump...**<br>Display memory in the Dump window. |
| **Command line...**<br>Enter a command. | **Go to return**<br>Return from a procedure call. | **CPU structures...**<br>View a second menu listing the CPU structures specific to your target system. |
| **Exit/Quit**<br>Terminate Soft-Scope. | **Go to...**<br>Execute to the referenced location. | |
| Recently loaded file list. | **Stop**<br>Stop target execution. | |

Table 3-1: Pull-Down Menu Map *(continued)*

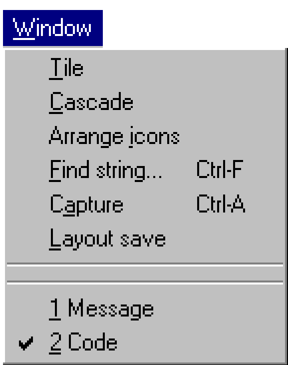
| Break | Macro | Options |
|---|---|---|
| **Display**<br>Display and set breakpoints. | **Display**<br>Display a list of loaded macros. | **Display**<br>Display a list of current options. |
| **Execution...**<br>Set a software-execution breakpoint. | **Load...**<br>Load and compile a macro. | **Reload settings**<br>Reload the options file. |
| **Access...**<br>Set a hardware-access breakpoint. | **Resume**<br>Resume a suspended macro. | **Save settings**<br>Save current options to the options file. |
| **Write...**<br>Set a hardware-write breakpoint. | | |
| **Exec...**<br>Set a hardware-execution breakpoint using debug registers. | | |

Table continued on next page.

Table 3-1: Pull-Down Menu Map *(continued)*

| View | Window | Help |
|---|---|---|
| **Toolbar**<br>Turn toolbar on/off. | **Tile**<br>Arrange open windows so borders don't overlap. | **Index**<br>Display an Index of help topics. |
| **Status bar**<br>Turn status bar on/off. | **Cascade**<br>Arrange open windows in an overlapping pattern. | **Using help**<br>Describes how to use help. |
| | **Arrange icons**<br>Organize the icons displayed at the bottom of the window. | **About Soft-Scope...**<br>Displays Soft-Scope's version number. |
| | **Find string...**<br>Search for a specified string. | |
| | **Capture**<br>Save the contents of the active window to the log file. | |
| | **Layout save**<br>Save the window configuration. | |
| | List of open Soft-Scope windows. Select a name to activate that window. | |

# Window Pull-Down Menu






The **Window** pull-down menu includes commands for doing standard window icon manipulations such as **Tile**, **Cascade** and **Arrange** icons. Consult Soft-Scope's online help or your Microsoft Windows user's guide if you need instructions on using these functions. This section will discuss **Find string...**, **Capture**, and **Layout save**.

## Finding a String

Select the **Find string...** command from the **Window** pull-down menu to search for a text string in a window. The search function completes a search for a specific character string of not more than 40 characters and works in any window *except the **Code** window when it is in assembly mode.*

Enter the string you want to find in the **Find** dialog box shown in figure 3-1. When the string is found, the cursor moves to the first character in the string and the string is displayed in the currently active window.

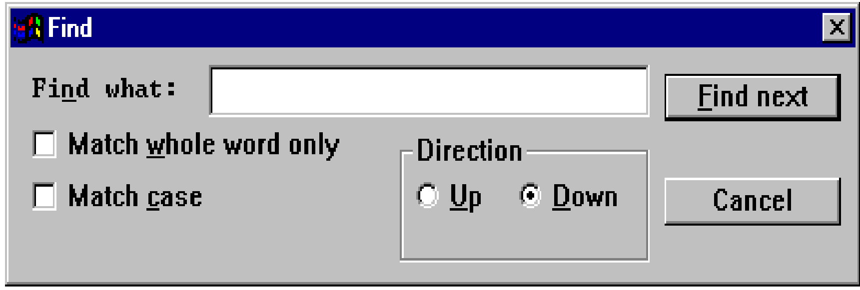



Figure 3-1: Find dialog box

The **Find** dialog box gives you several options:

| | |
|---|---|
| Match whole word only | The search function finds strings that match only what you enter into the text box. It doesn't find strings that contain your search string as a proper substring. For example, if you typed *mod* in the text box, the word *module* would not be considered a match. |
| Match case | The search function finds only strings that match the case of the characters you enter. |
| Direction | Controls the search direction (up/down). |
| Find next | Searches for the next occurrence. |
| Cancel | Cancels the search. |

# Capturing a Window to a Log File

Using the **Capture** command from the **Window** pull-down menu, you can capture the contents of the current window to a log file. All of the data displayed in the current window is copied. See figure 3-2 for an example of a **Log** window.

Specify a *count* followed by the accelerator key <Ctrl>+<A> to capture a *count* number of lines:

```
25 <Ctrl>+<A>
```

Specify the log file name and path with the **cmd.file** configuration option, which is explained in more detail in the *Configuring Soft-Scope* chapter. The default log file name is **sswin32.log**.

If the file specified by **cmd.file** already exists, Soft-Scope gives you the option to append your capture to the end of the file or to start over and rewrite the file.

You can append any sort of data you want to the **Log** window and log file with the ***WPRINTF*** macro command, which is discussed fully in the *Macro Print Functions* section of the *Creating and Using Soft-Scope Macros* chapter:

```
wprintf (log, "%s", "Print this in the log
window")
```

3

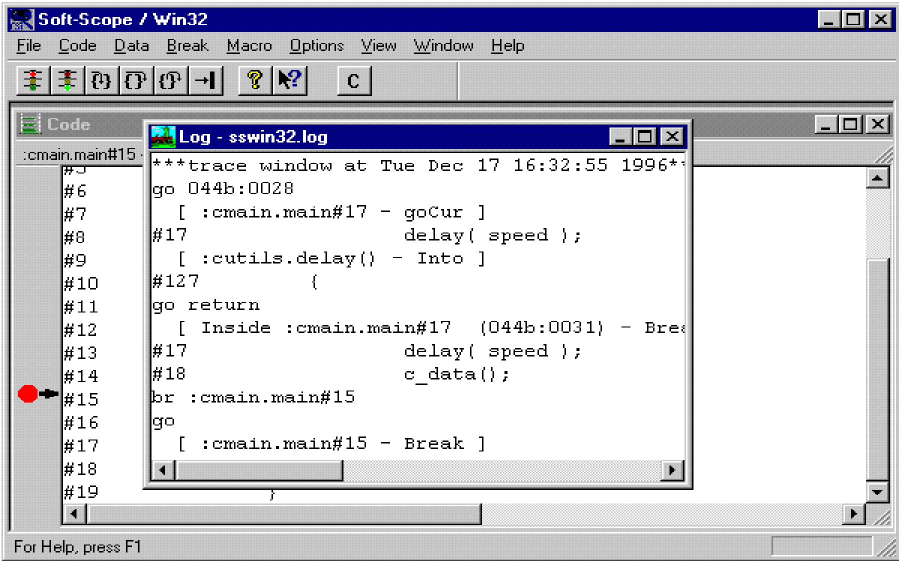



Figure 3-2: Log window showing capture of Trace window

To view the contents of the log file, use the **View log** command from the **File** pull-down menu to open the **Log** window. Although everything you write to your log file is stored on disk, the **Log** window can display only the last 500 lines of the log file. See the *Configuring Soft-Scope* chapter for a discussion of the configuration option **log.winsize**, which enables you to alter the number of lines in the **Log** window.

To clear the **Log** window and erase the entire contents of the current log file, use the **Clear** toolbar button.

## Saving Window Layout

Use the **Layout save** command from the **Window** pull-down menu to save the size and location of windows that have been moved or resized during the current Soft-Scope session.

## Open Window List

At the bottom of the **Window** pull-down menu is a list of open windows. A checkmark identifies the active window. To make another window in the list the active window, click on it with your mouse <Button-L>.

# Accelerator Keys

Soft-Scope pull-down menus and some of the commands can be invoked with accelerator keys as shown below. Following the Windows convention, the letter following the <Alt> key is usually the first letter of the pull-down menu title as identified by the underscore. Soft-Scope commands use the <Ctrl> key followed by a letter.

These are the accelerator keys:

| | |
|---|---|
| <Alt>+<B> | Activates the **Break** pull-down menu |
| <Alt>+<C> | Activates the **Code** pull-down menu |
| <Alt>+<D> | Activates the **Data** pull-down menu |
| <Alt>+<F> | Activates the **File** pull-down menu |
| <Alt>+<H> | Activates the **Help** pull-down menu |
| <Alt>+<M> | Activates the **Macro** pull-down menu |
| <Alt>+<O> | Activates the **Options** pull-down menu |
| <Alt>+<W> | Activates the **Window** pull-down menu |
| <Ctrl>+<A> | Capture the current window to a file |
| <Ctrl>+<C> | Cancels the current operation |
| <Ctrl>+<F> | Opens the **Find** dialog box |
| <Ctrl>+<L> | Opens **Command line** dialog box |
| <Ctrl>+<Q> | Quits/Exits Soft-Scope; all work files are erased except the temporary quick-reload file |
| <Ctrl>+<X> | Closes the active window |
| <Ctrl>+<End> | Displays last page of the current window |
| <Ctrl>+<Home> | Displays first page of the current window |

| | |
|---|---|
| <Ctrl>+<PgDn> | Pages down one-half of the current window |
| <Ctrl>+<PgUp> | Pages up one-half of the current window |
| <Ctrl>+<Shift>+<Tab> | Activates previous window in window queue |
| <Ctrl>+<Tab> | Activates next window in window queue |

# Double-click Function

One of the most useful tools of the Soft-Scope user interface is the left mouse button (<Button-L>) double-click. Double-clicking <Button-L> allows you to accomplish a variety of tasks without having to touch your keyboard or move elsewhere in the **Soft-Scope** window.

In general, you can initiate the default function in any window except the **Breakpoints** window, by double-clicking <Button-L>. For example, double-clicking <Button-L> on an item in the **Watch** window will open the **Modify** dialog box (the default function) and place it in the text box.

# Double-click in the Code Window

Double-clicking <Button-L> evaluates the expression identified by the cursor and places it in the appropriate window. For data references, double-clicking <Button-L> will open the **Data** window and display the reference in normal mode. For code references, double-clicking <Button-L> will open the **Code** window and display the code associated with the reference.

# Double-click on Data References

By double-clicking <Button-L> on data references in the **Data** and **Watch** windows you can manipulate the way you view structures, unions, pointers and classes. Assume the following structure:

```
struc1 structure (...)
```

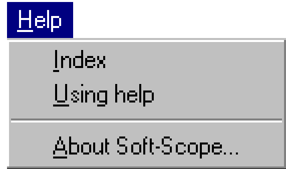
Double-click <Button-L> on or after the word "structure" to display the entire structure. Double-click <Button-L> before the word "structure" to place the structure in a dialog box for modification.

# Double-click on Pointers

Double-click <Button-L> to reference pointers. Double-click <Button-L> before the “->“ to display the pointer in a dialog box for modification. Double-click on the “->“ to dereference the pointer and display the dereferenced data. To display the indirect data in a dialog box for modification, double-click <Button-L> after the “->“.

# Online Help

Soft-Scope's online help contains much of the information found in this user's guide. The hypertext links between help topics provide an excellent way to find the information you are looking for.




The **Help** pull-down menu contains several options:

| | |
|---|---|
| Index | Displays an index of topics. Put the cursor on the item you want and click the left mouse button.<br>Soft-Scope's online help uses the standard windows help engine. The menu bar and toolbar contain functions to search for, print, set bookmarks in and annotate a help topic. |
| Using help | Displays the standard windows information about using **Help**. |
| About Soft-Scope... | Displays Soft-Scope's version number and copyright information. |

# Commands and Command Line

Soft-Scope commands are commonly invoked via the **Command line** dialog box. The dialog box is activated by entering <Crtl>+<L>. Commands are entered in the text box. Figure 3-3 shows the dialog box and command for causing Soft-Scope to execute to the function ***main***.

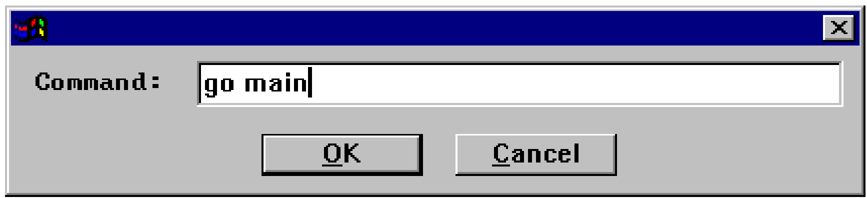



Figure 3-3: Command line dialog box



Soft-Scope commands can also be used in macros as discussed in the *Creating and Using Soft-Scope Macros* chapter.

These are the Soft-Scope commands:

```
BR[EAKPT] [-] [[EXEC] coderef  [when-then]]
BR[EAKPT] [-] [(ACCESS | WRITE) memref  [when-then]]

CALLS

DIS[ASM] [[TO] coderef ]

DUMP [[TO] memref ]

EVAL (memref | coderef) [, (memref | coderef)]*

EXIT

G[O] [EXEC] coderef
G[O] [(WRITE | ACCESS) memref ]
G[O] RETURN

HELP [keyword ]

LINE [coderef ]
```

```
L[IST] [[TO] lineref ]

LOAD filename
LOAD (RESTART | SYMBOLS) filename

MACRO   [LIST] [[TO] macroname]]
MACRO   LOAD filename
MACRO   DELETE [macroname]
MACRO   RESUME
MACRO   SUSPEND

MESSAGE[S]

MODULE  [[TO] :modname]
MODULE  :modname = filename

PROCEDURES [[TO] coderef ]

QUIT

REG

SET [[TO] optionname ]
SET [RELOAD | SAVE]
SET [optionname = optionvalue]

STACK [USAGE | RESET]

S[TEP] [INTO | OVER]

STOP

SYMBOLS [[TO] coderef  ]

TRACE

TYPE (memref | coderef) [, (memref | coderef)]*

UPLOAD memref [REGISTER[S]] filename
UPLOAD REGISTER[S] filename

VER[SION]

WATCH [memref [, memref ]*]
```

# Command Syntax Elements

The command syntax elements listed below are used in both command-line commands and in menu-selection dialog boxes. Optional entries are defined by brackets ( [ ] ), and a vertical line ( | ) indicates a choice between the items on either side of the line.

These are the command syntax elements:

| | |
|---|---|
| *address* | A logical, physical, or linear address |
| *coderef* | *address* \| [:*modname*]#*linenum* \| [:*modname*.]*codesym* |
| *codesym* | The name of a procedure or label |
| *dataref* | *coderef* \| *memref* \| *lineref* |
| *datasym* | The name of a symbol |
| *f:* | Block device driver specification |
| *filename* | A system-dependent identifier for a disk file |
| *filename.bug* | A **.bug** file name associated with a relocatable DOS or OMF86 program, including a path to the file |
| *hexnumber16* | A 16-bit hexadecimal number |
| *keyword* | A word to use for a Help search |
| *linenum* | A line number found in the current module or in *modname* |
| *lineref* | :*modname* \| [:*modname*]#*linenum* \| [:*modname*.]*codesym* |
| *macroname* | The name of a macro from the currently loaded macros |
| *memref* | *address* \| *lineref* \| [:*modname*.][*codesym*.]*datasym* |

| | |
|---|---|
| *modname* | A module name |
| *optionname* | The name of a configuration option |
| *optionvalue* | The value of a configuration option |
| TO | Places the reference at the bottom of the window, and fills the upper part of the window with what is before the reference. |

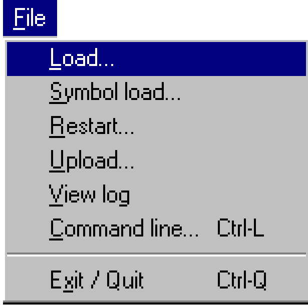
# Loading an Application

Soft-Scope can be used to debug real- and protected-mode absolutely located bootable files prepared with tools discussed in the chapter *Tools that Soft-Scope Supports*. A loadable application image for Soft-Scope contains both executable instructions and associated symbolic information.

Soft-Scope's format of choice for loadable files is the **.abs** file which is an extended version of an **.omf** file produced by our linker, CSi-Link. Other formats can be used as long as Soft-Scope can access their symbolics. For example, Soft-Scope can be used with **.exe** files prepared in the special way discussed in the appendix *Debugging **.exe** Files*, as well as with files in OMF-86, OMF-286, HEX, and other formats (see the chapter *Tools that Soft-Scope Supports*).



## Load




To load a bootable absolute file to your target, select **Load...** from the **File** pull-down menu to open the dialog box shown in figure 3-4.

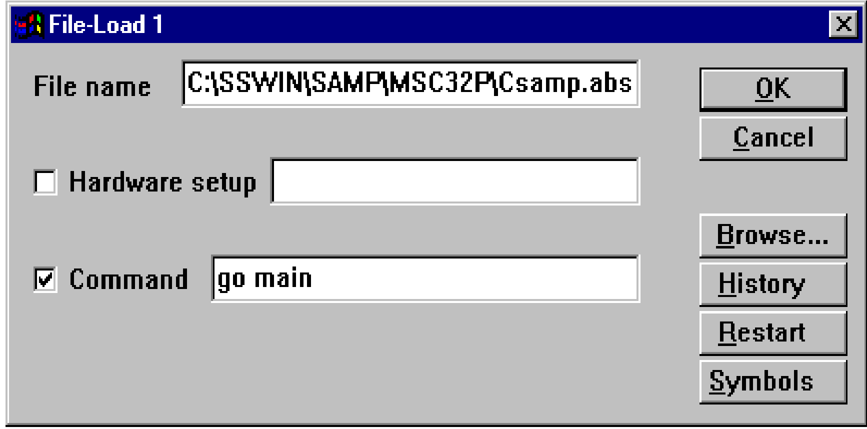


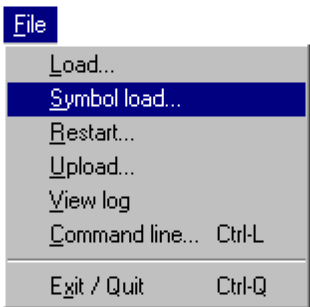

Figure 3-4: File-Load dialog box

| | |
|---|---|
| File name | Enter the file name, or choose the **Browse...** button to select a file from the directory listing. |
| Hardware setup | Enter a command to be invoked *before* your application is loaded. The check box toggles the invocation on/off. For example, invoke a macro that writes test data into memory to help you find uninitialized-variable problems. |
| Command | Enter a command for Soft-Scope to perform *after* the application has been loaded. For example, enter `go main` to cause Soft-Scope to execute the application up to the function ***main***. The check box toggles the invocation on/off. See the *Commands and Command Line* section of this chapter. |
| Browse... | Displays the most recently accessed subdirectory and its contents. |

| | |
|---|---|
| History | Reviews file loads from the previous nine Soft-Scope sessions. |
| Restart | Resets the descriptor-table registers and program counter. **Restart** will not reload the program code, data or symbols. **Restart** will not initialize data in RAM. For example, use **Restart** if you have a stack fault or step beyond your source code. |
| Symbols | Reloads symbolic information only. It does not reload the program code. |

**NOTE:** You can load a recently loaded file by choosing its name from the list of files at the bottom of the **File** pull-down menu. The file will be loaded with the same entries for Hardware Setup and Command that were used the last time it was loaded.

# Symbol Load




Select **Symbol load...** from the **File** pull-down menu to load symbolic information without disturbing your application or changing register values. This is useful if your application is already loaded or if you want to debug an application with multiple symbol sets.

**File/Symbol load...** will open the following dialog box:

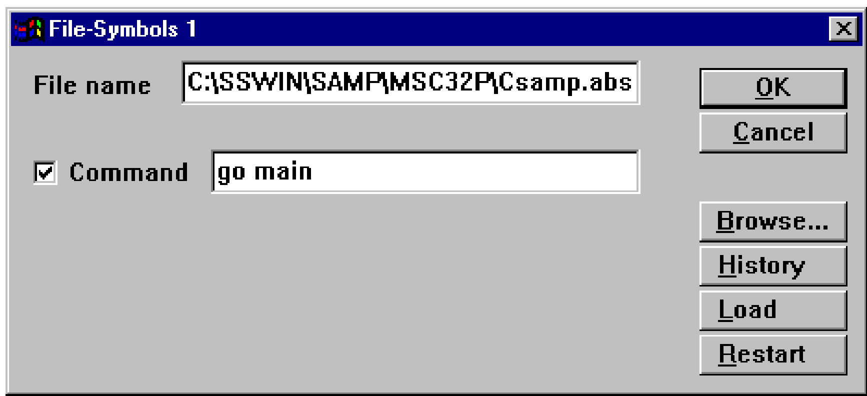
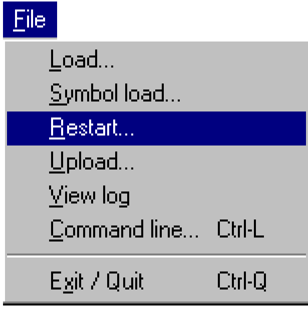



Figure 3-5: File-Symbol load dialog box

| | |
|---|---|
| File name | Enter the file name, or choose the **Browse...** button to select a file from the directory listing. |
| Command | Enter a command for Soft-Scope to perform *after* the application has been loaded. For example, enter `go main` to cause Soft-Scope to execute the application up to the function ***main***. The check box toggles the invocation on/off. See the *Commands and Command Line* section of this chapter. |
| Browse... | Displays the most recently accessed subdirectory and its contents. |
| History | Reviews file loads from the previous nine Soft-Scope sessions. |
| Load | Downloads program code, data and symbolic information. |
| Restart | Resets the descriptor-table registers and program counter. **Restart** will not reload the program code, data or symbols. **Restart** will not initialize data in RAM. For example, use **Restart** if you have a stack fault or step beyond your source code. |

3

## Restart



Select **Restart...** from the **File** pull-down menu to load symbols and set the initial register values. This is useful if the target system contains a load image in ROM or if the load image has already been loaded by some other means.

Because **Restart...** does not reload your applications data area, your application may not execute the same way as it does when you do a complete load, especially if it depends on initialized data.

**File/Restart...** will open the following dialog box:

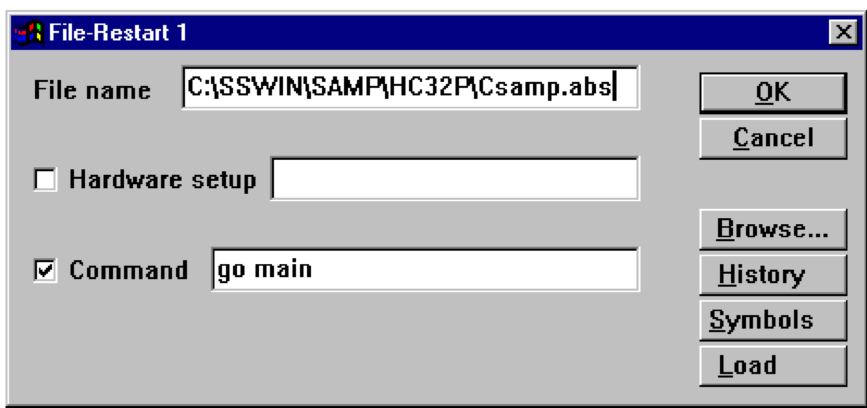



Figure 3-6: File-Restart dialog box

| | |
|---|---|
| File name | Enter the file name, or choose the **Browse...** button to select a file from the directory listing. |
| Hardware setup | Enter a command to be invoked *before* your application is loaded. The check box toggles the invocation on/off. For example, invoke a macro that writes test data into memory to help you find uninitialized-variable problems. |
| Command | Enter a command for Soft-Scope to perform *after* the application has been loaded. For example, enter `go main` to cause Soft-Scope to execute the application up to the function ***main***. The check box toggles the |

| | |
|---|---|
| | invocation on/off. See the *Commands and Command Line* section of this chapter. |
| Browse... | Displays the most recently accessed subdirectory and its contents. |
| History | Reviews file loads from the previous nine Soft-Scope sessions. |
| Symbols | Reloads symbolic information only. It does not reload the program code. |
| Load | Downloads program code, data and symbolic information. |

# After the Load

After making all of your selections in the dialog box, choose the **OK** button. The status line at the bottom of the **Soft-Scope** window will record the percentage of file loaded as your application is loading. The **Code** window will open to show your application in source mode when symbolic information is available. See figure 3-7 for an example.

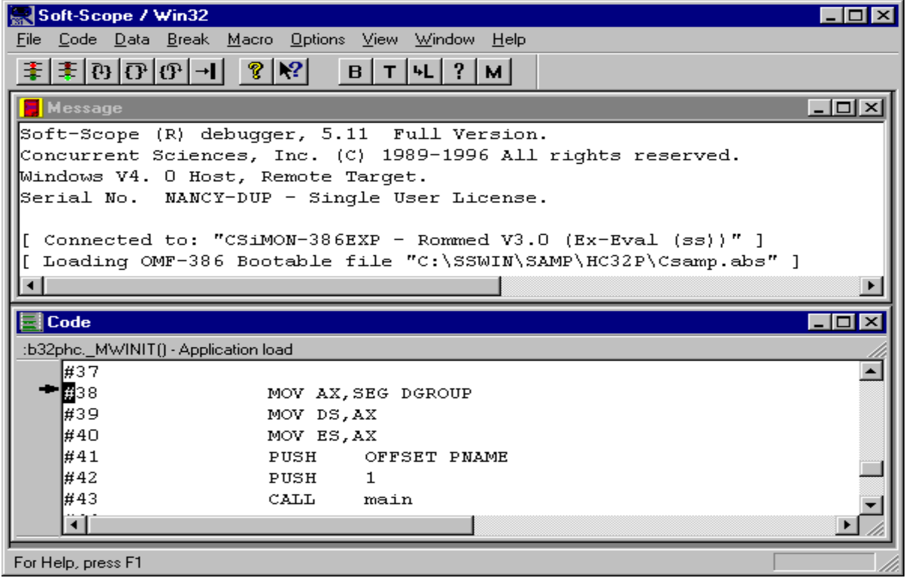



Figure 3-7: Soft-Scope after an application load

# Soft-Scope .tmp Files

When Soft-Scope loads an application, it must find or build an internal representation of the application's symbols. This information is initially read from the absolutely located **.abs** file or a **.bug** file (see the chapter *Tools that Soft-Scope Supports* for a discussion of this file type), and is placed in a temporary file ***application*.tmp**, for example, **csamp.tmp**.

This **.tmp** file is built incrementally during the execution of an application, as Soft-Scope actually makes use of the application's symbolics. For an application with a large amount of symbolic information, procedures not called until later in the execution of the program will not have their symbolics represented in the **.tmp** file until they are actually called.

When you exit Soft-Scope, the temporary file is saved in the directory that contains the application file or symbolics file that was recently loaded and from which it is derived. Soft-Scope reuses the **.tmp** file the next time you load the corresponding application, enabling a much faster load. If your disk space is limited or faster loads are not important, you can erase any and all temporary files with no harmful consequences for future Soft-Scope sessions. Soft-Scope will automatically rebuild a new **.tmp** file for any application for which it can't find one.

# Command Line

You can load applications (***LOAD***), restart applications (***LOAD RESTART***), or load just symbols (***LOAD SYMBOLS***) from the **Command line** dialog box. To invoke the dialog box, enter <Ctrl>+<L>. The syntax of the ***LOAD*** command follows:

[*count*] LOAD [[RESTART | SYMBOLS] *filename*]

# 4. Controlling Program Execution with Soft-Scope

## Chapter Contents

**Overview** .......................................................................................... **4-3**
**Controlling Program Execution** ........................................................... **4-3**
Stepping through Code .................................................................. 4-4
Single Step ..................................................................................... 4-4
Specify a Number of Steps ............................................................. 4-5
Step Command via the Command Line ........................................... 4-5
**Code Window** ........................................................................................ **4-6**
Figure 4-1: Code window in Source mode ................................... 4-7
Figure 4-2: Code reference dialog box .......................................... 4-8
Toolbar Buttons ................................................................................ 4-8
Figure 4-3: Display modes dialog box .......................................... 4-9
Figure 4-4: Code window in Assembly mode with logical addresses4-10
Code Window Execution Pointers ..................................................4-11
**Code References** .................................................................................. **4-12**
Line Numbers .................................................................................. 4-12
Symbol Names ................................................................................. 4-12
Guidelines ........................................................................................ 4-13
**Locating Code** ...................................................................................... **4-14**
**Breakpoints Window** ............................................................................ **4-16**
Figure 4-5: Breakpoints window ................................................... 4-17
Toolbar Buttons ................................................................................ 4-18
Command Line ................................................................................ 4-19
**Editing Breakpoints** .............................................................................. **4-20**
Figure 4-6: Breakpoint edit dialog box ......................................... 4-20

**Software Breakpoints** .......... **4-22**
Permanent Software Breakpoints .......... 4-22
Temporary Software Breakpoints .......... 4-23
**Hardware Breakpoints** .......... **4-24**
Data Breakpoints .......... 4-24
Command Line .......... 4-24
Debug Registers .......... 4-25
Exec Breakpoints .......... 4-26
Command Line .......... 4-26
**Executing to a Location** .......... **4-27**
Go .......... 4-27
Go to a Specific Location .......... 4-27
Return from a Procedure Call .......... 4-28
Go to a Cursor Position .......... 4-28
Stop .......... 4-28
**Procedure Call Sequence** .......... **4-30**
Calls Window .......... 4-30
Figure 4-7: Calls window .......... 4-31
Command Line .......... 4-31
**Stack Information** .......... **4-32**
**Trace Window** .......... **4-33**
Figure 4-8: Trace window displaying procedures .......... 4-33
Toolbar Buttons .......... 4-34
Figure 4-9: Assembly display modes dialog box .......... 4-35
Figure 4-10: Trace window displaying procedures and source .......... 4-36
Command Line .......... 4-37
Figure 4-11: Trace window displaying procedures, source, and assembly code .......... 4-37
Trace Buffer .......... 4-38
Trace File Size .......... 4-38

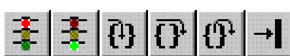
# Overview

With Soft-Scope you can monitor your application's source while executing at the source or assembly level. This chapter will describe the mechanisms that allow you to execute one source line at a time, execute to a predetermined location, set hardware and software breakpoints, view source code and trace Soft-Scope's actions.

# Controlling Program Execution

The basic target execution toolbar buttons, shown below, remain on the toolbar at all times. Whenever a window becomes active, the buttons specific to that window are added to the toolbar. When another window becomes active, the previous set of window specific buttons are replaced with a new set of buttons.





Stop • Go • Step into • Step over • Go to return • Go to cursor

| | |
|---|---|
| Stop | Stops execution without setting a breakpoint. This function works only when your target contains an interrupt-driven CSi-Mon monitor. To activate from the keyboard press <S>. |
| Go | Causes the target to execute until a breakpoint or fault is encountered. Note, no breakpoint is set. To activate from the keyboard press <G>. |
| Step into | Steps into the next procedure call. You may specify a *count,* such as `2 I`, which will step twice and into procedures if called on the lines executed. To activate from the keyboard press <I>. |

| | |
|---|---|
| Step over | Steps over the next procedure call. Accepts *counts* as described above. To activate from the keyboard press <O>. |
| Go to return | Returns to the calling procedure. You may specify a *count,* such as **`4 R`**, which will return from four calls. To activate from the keyboard press <R>. |
| Go to cursor | Execution will continue until the cursor is encountered. You may specify a *count,* such as **`10 C`**, which is handy if you need to execute a loop several times. To activate from the keyboard press <C>. |

## Stepping through Code

Stepping allows you to execute one source line at a time. You can single step or step a specified number of times. You can also step into or step over procedure calls.

## Single Step

Press <Spacebar> to execute a single line of source or assembly code, depending on the **Code** window's display mode (source or assembly). Use the **Mode** toolbar button in the **Code** window to set the default step mode (into or over).

## Specify a Number of Steps

Step a specified number of times by typing a number while the **Code** window is currently active and pressing <Spacebar>. You can specify any number from 1 to 65535. Pressing the **Stop** toolbar button will terminate execution immediately if you are using an interrupt driven CSi-Mon monitor. The following example initiates 10 steps:

```
10 <Spacebar>
```

## Step Command via the Command Line

Enter the following stepping commands, both in the **Command line** dialog box (<Ctrl>+<L>) and in the body of macro definitions. ***STEP*** automatically opens the **Code** window:

S[TEP] [INTO | OVER]

# Code Window

The **Code** window is opened by selecting **Display...** from the **Code** pull-down menu shown below. From this window, you can step through your code, set breakpoints and examine data references.

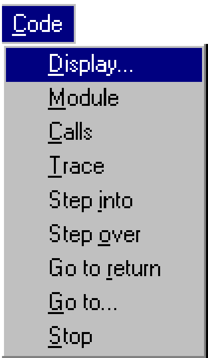



Figure 4-1 shows the **Code** window in source mode. The current module, procedure, and line number are identified on the status line at the top of the **Code** window. Key words that describe the current execution status—such as Break, Running, or General Protection Fault—are also displayed here.

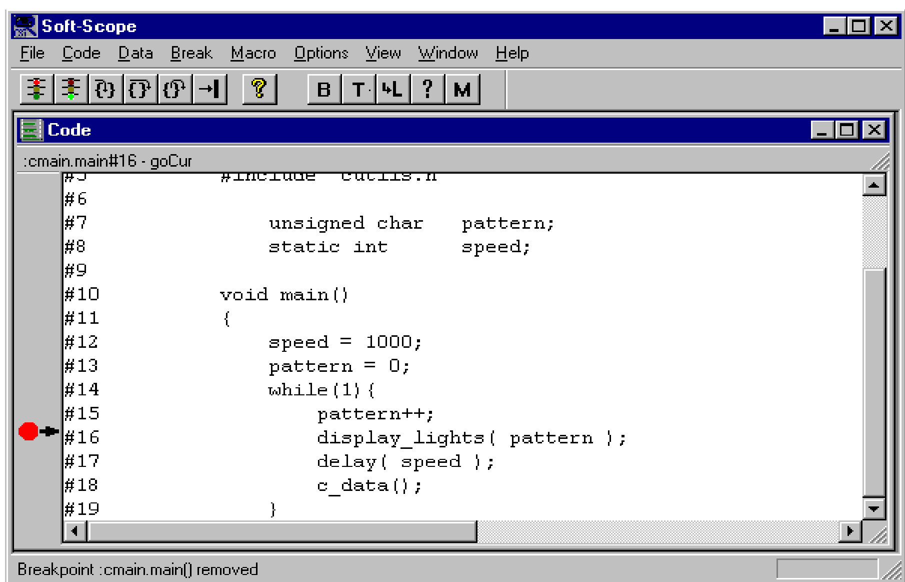



Figure 4-1: Code window in Source mode



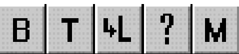
Double-clicking <Button-L> on a data reference displays information about the reference in the **Data** window. Double-clicking <Button-L> on a code reference displays the source code associated with the reference.

The **Display** command from the **Code** pull-down menu opens a dialog box prompting you for a code reference as shown in figure 4-2. This reference is used to identify the source code that will be displayed in the **Code** window. The text box in this example contains a command instructing Soft-Scope to execute up to the beginning of the function ***main***. If you press <Enter> without entering a reference, the display defaults to the current execution point.

Figure 4-2: Code reference dialog box

# Toolbar Buttons

The following toolbar buttons allow you to set breakpoints, evaluate data references, and control the contents of the **Code** window:




Break • Temp break • Locate • Evaluate • Mode

Break — Toggles a permanent software breakpoint on/off at the current cursor position. For more information, see the *Software Breakpoints* and *Hardware Breakpoints* sections of this chapter. To activate from the keyboard type <B>.

Temp break — Toggles a temporary software breakpoint on/off at the current cursor position. This type of breakpoint clears itself after the first time it is encountered. To activate from the keyboard type <T>.

Locate — Returns the cursor to the current execution point. To activate from the keyboard type <Enter> or <L>.

Evaluate (?) — Opens a dialog box where you enter a data reference to be evaluated in the **Data** window. A move convenient method is to double-click <Button-L> on a data reference in the **Code** window. To activate from the keyboard type <?>.

Mode — Opens the **Display modes** dialog box as shown in figure 4-3. To activate from the keyboard type <M>.

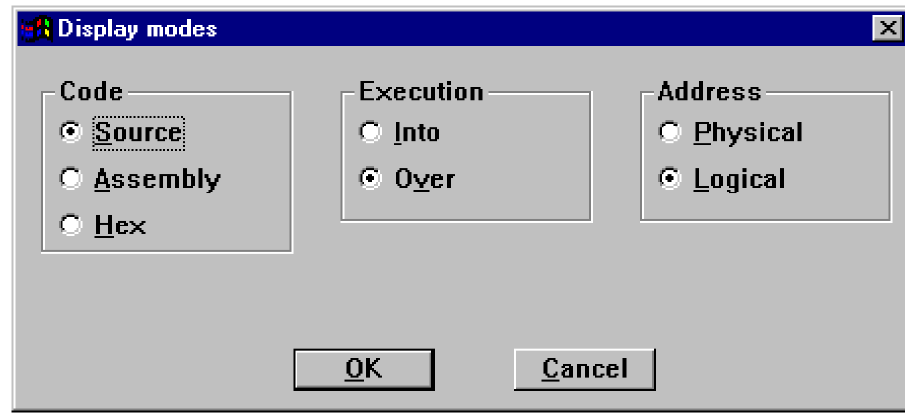



Figure 4-3: Display modes dialog box

Code — The **Code** radio buttons control the way in which source code is displayed. **Source** mode is the default option. **Assembly** mode shows both source and assembly code. **Hex** mode shows source, assembly, and hex code. See figure 4-1 for an example of source mode.

Execution — The **Execution** radio buttons set the default step type. **Into** steps into the next procedure call. **Over** steps over the next procedure call. A step is invoked by pressing <Spacebar>.

**Address** The **Address** radio buttons control whether assembly code addresses are displayed as physical (with a "P" suffix) or logical (in Segment:Offset format). See figure 4-4 for an example of assembly mode using logical addresses.

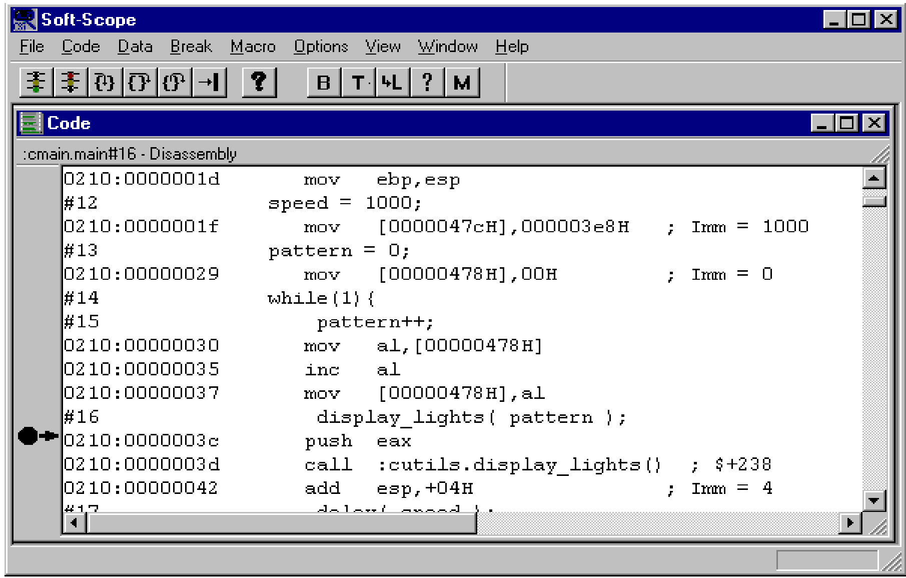



Figure 4-4: Code window in Assembly mode with logical addresses

**NOTE:** When you scroll up (backwards) in the **Code** window in Assembly mode, Soft-Scope can't always show accurate information. Approximated information is identified with a question mark (?).

# Code Window Execution Pointers

Soft-Scope uses the far left side of the **Code** window to indicate the special status of certain source lines. The symbols used are as follows:

- A solid arrow means execution is stopped at that line of code.
- An outline of an arrow indicates execution is stopped at a location inside a source-level statement.
- A solid octagon means there is a permanent breakpoint set at this line of code.
- An outline of an octagon means there is a temporary breakpoint set at this line of code.

The execution pointers may be shown in combination with the breakpoint indicators. For example, indicates that execution has halted on a line where a permanent breakpoint is set.

4

# Code References

## Line Numbers

You can access any program symbol or line of code that is currently loaded into Soft-Scope. To reference a line of code, simply use the line-number operator “#”:

Code reference: `#24`

## Symbol Names

To reference a procedure in the current module use its name:

Code reference: `c_data`

To reference a procedure in a module other than the current module, use the module operator “:” and the symbol operator “.”:

Code reference: `:cutils.c_data`

You can reference code with a logical address:

Code reference: `203:0d1`

You can reference code with a physical address:

Code reference: `20P`

## Guidelines

Use the following guidelines when referencing code elements:

- If they are in the module containing the execution pointer, use:

```
#line number
procedure name
memory address
```

- If they are in a module other than the one containing the execution pointer, use:

```
:module name#line number
:module name.procedure name#line number
memory address
```

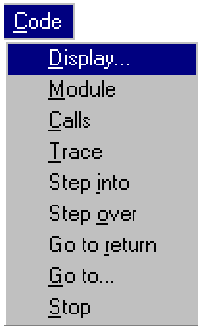
4

If you can't remember the exact name of the module, procedure, or symbol, use **Code/Module** or **Data/Symbols** pull-down menu commands to look at the possible entries.

For more information, see the *Reference Scoping* section of the *Examining Data with Soft-Scope* chapter.

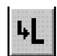
# Locating Code

Several mechanisms are provided to let you see any part of your application's source code as shown below:




**Code/Display...** prompts you for a code reference, and displays the code associated with that reference in the **Code** window. If no reference is given, Soft-Scope displays the code at the current execution point.

For example, if you entered #82 in the dialog box, the **Code** window will display the code located at line number 82.



The **Code** window's **Locate** toolbar button returns the **Code** window to the line where the execution pointer is pointing.

`LIST [[TO] lineref]`

From the command line, ***LIST*** will display source lines from *lineref* down. If ***TO*** is used, the list is from the bottom of the **Code** window up. Use ***LIST*** with no *lineref* to open the **Code** window at the current execution point.

Double-click <Button-L>

Double-click <Button-L> on the symbol you want to display. If the symbol is a code symbol, the **Code** window opens and displays the code where that symbol is located. If the symbol is a data symbol (variable), the **Data** window opens and displays the variable in normal mode.

**NOTE:** Because more than one logical address can resolve to a single physical address, Soft-Scope cannot locate specific source-code lines using a physical address. Using logical addresses with **Code/Display...** will ensure the accurate location and display of source code. This does not apply to **Code/Go to...** and other **Code** menu choices.

# Breakpoints Window

The **Breakpoints** window is opened by selecting **Display** from the **Break** pull-down menu.

The **Breakpoints** window shows how many breakpoints are set and identifies their type and location.

The **Breakpoints** window in figure 4-5 shows a variety of breakpoints. The first is a permanent software breakpoint set in ***C_DATA*** at line 80. This is followed by a hardware breakpoint set to stop execution when the address associated with ***OLDCUST*** is accessed. The third is a hardware breakpoint set to stop execution when the address associated with ***HI-BIT*** is written to. The final entry is a temporary software breakpoint set in ***CUTILS.DELAY*** at line 128.

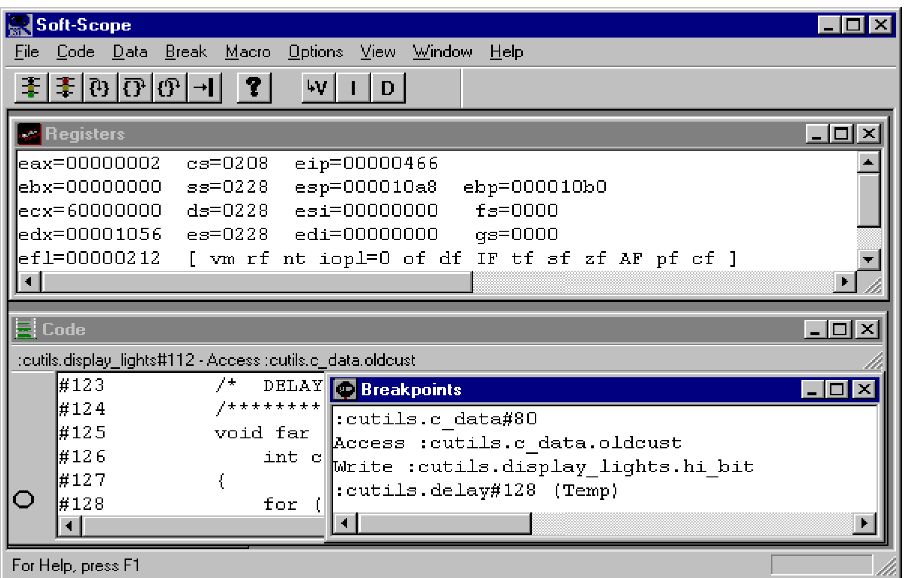



Figure 4-5: Breakpoints window

# Toolbar Buttons

Breakpoints are manipulated by placing the cursor on a breakpoint listed in the **Breakpoints** window and using one of the following toolbar buttons:

View • Insert • Delete

View Opens the **Code** window and displays the code at the location where the breakpoint is set. To activate from the keyboard press <Enter> or <V>.

Insert Opens a dialog box where you can specify a new breakpoint. For hardware breakpoints, enter a breakpoint type before the reference. To activate from the keyboard press <I>.

Delete Deletes the breakpoint the cursor is on. To activate from the keyboard press <D>.

To modify an existing breakpoint, double-click <Button-L> on it's entry in the **Breakpoints** window.

For more information, see the *Editing Breakpoints*, *Software Breakpoints*, and *Hardware Breakpoints* sections of this chapter.

# Command Line

Enter the following commands in the **Command line** dialog box (<Ctrl>+<L>) to insert and remove breakpoints, and open the **Breakpoints** window:

```
BR[EAKPT] [-] [coderef   [when-then]]

BR[EAKPT] [-] [EXEC coderef    [when-then]]

BR[EAKPT] [-] [WRITE memref  [when-then]]

BR[EAKPT] [-] [ACCESS memref  [when-then]]
```

`BR[EAKPT]` with no *`coderef`* opens the **Breakpoints** window.
`BR[EAKPT] -` with no *`coderef`* deletes all breakpoints.

The following example will set an **access** breakpoint at the hex address 0f200:

```
br access 203:0f200
```

The next example will set a conditional **write** breakpoint on the variable *`pattern`* when it's value equals 25. When the condition is met, the value of *`lights[4]`* will be set to 'x':

```
br write pattern when pattern==25 then
lights[4]=='x'
```

4

# Editing Breakpoints

To edit a breakpoint open the **Breakpoint edit** dialog box by double-clicking <Button-L> on an item in the **Breakpoints** window, or using the window's **Insert** toolbar button.

The dialog box shown in figure 4-6 allows you to define breakpoint status, conditions, and specify an action to be performed when the breakpoint is encountered.

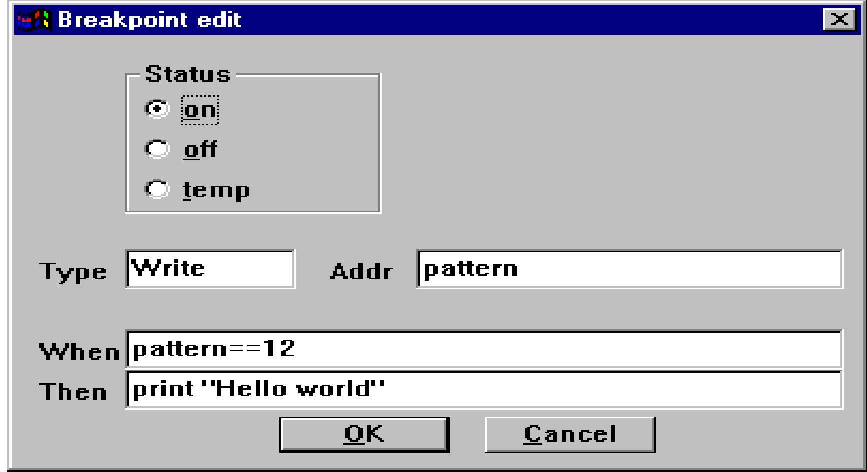



Figure 4-6: Breakpoint edit dialog box

Status **Status** allows you to turn the breakpoint on or off, or designate it as a temporary break.

Type **Type** can be any breakpoint type supported by the target processor you are using. Possible types are Write, Exec, or Access. The **Break** pull-down menu contains the types that are available to you.

The default type is **Software**, which does not display in the

edit field, and for which there is no predefined name. Leave the text box blank and the default will apply.

Addr — An **Addr** address can be any memory reference, including symbol references. See the *Code References* section of this chapter for more information.

When — **When** is the breakpoint condition. The condition is evaluated when the breakpoint is encountered. If the condition is true, the action entered in the **Then** text box is performed. Otherwise, target execution continues.

If the condition is invalid, Soft-Scope displays an error message that allows you to abort execution of the condition, provide a true/false response to the condition, or edit the breakpoint specification and try again.

The following condition stops execution when the variable `pattern` equals 5:

When: **`pattern==5`**

Use any C-based expression in the **When** text box. See the *Operator* section in the *Examining Data with Soft-Scope* chapter and appendix A for a list of valid operators.

Then — **Then** is the action taken when the condition is true. An action can be any valid Soft-Scope command or macro. The default action is to stop execution. If an error is encountered, a dialog box opens that gives you the options of aborting the action, ignoring the error, or editing the breakpoint and trying again.

An example action would be to execute a Soft-Scope macro that prints the value of a particular variable, sets a breakpoint at another code location, and then executes to that location in memory.

The example in figure 4-6 turns on a permanent breakpoint of type **write** at the location of the variable `pattern`. When the value of `pattern` is equal to 12, the string “Hello world” is displayed in the **Message** window.

# Software Breakpoints

There are two types of software breakpoints, permanent and temporary. A permanent software breakpoint persists until it is removed or you exit Soft-Scope. A temporary software breakpoint clears itself after the breakpoint is encountered. Software breakpoints stop target execution when the line of code associated with the breakpoint is executed.

Soft-Scope causes this to happen by inserting an INT3 software interrupt instruction in place of the instruction at the location where you want to break. The INT3 is later replaced by the original instruction. If you try to set a software breakpoint in ROM, Soft-Scope will use the EXEC breakpoint type discussed in the *Hardware Breakpoints* section of this chapter.

## Permanent Software Breakpoints

You can specify software breakpoints in several ways:

- Use the **Execution...** command from the **Break** pull-down menu. Enter a code reference in the dialog box. The example below will set a software breakpoint at source line number 45.

  Code reference: `#45`

- Find the code in the **Code** window where you want to stop execution. Move the cursor to the desired line of code, and choose the **Break** toolbar button. This sets a permanent breakpoint at that line.

- Use the ***BREAKPT*** command in the **Command line** dialog box (<Ctrl>+<L>) using the following syntax:

  BR[EAKPT] [-] [*coderef* [*when-then*]]

  The following example sets a software breakpoint at source line number 83 of the ***cutils*** module.

  Command: `br :cutils#83`

To delete permanent software breakpoints, do one of the following:

- From the **Code** window put the cursor on the source line where you want to remove the breakpoint and choose the **Break** toolbar button.
- From the **Breakpoints** window put the cursor on the breakpoint you want removed and choose the **Delete** toolbar button.
- Use the minus (-) parameter with the ***BREAKPT*** command line command and code reference:

  Command: `br -` ***`coderef`***

**NOTE:** When you set a breakpoint using an address, make sure that the address reference refers to the start of an instruction. Otherwise, the result is unpredictable.

## Temporary Software Breakpoints

To set a temporary software breakpoint, from the **Code** window, put the cursor on the source line where you want the breakpoint and choose the **Temp break** toolbar button.

To remove a temporary breakpoint from the **Code** window, put the cursor on the line where the breakpoint is located and choose the **Temp break** toolbar button.

Soft-Scope permits up to 32 temporary breakpoints.

# Hardware Breakpoints

Even though Soft-Scope has no hardware components, it can provide hardware type breakpoints by using the debug registers that are found on 386, 486 and Pentium processors. The debug registers make it possible to provide breakpoint conditions (access, write, instruction execution) and set a breakpoint in code that is running in ROM. Two types of hardware breakpoints are provided: Data and Exec.

## Data Breakpoints

Two conditions can be applied to the Data breakpoint, Access and Write. When the condition is met, execution is halted immediately *after* the specified memory location.

**NOTE:** If you set a data breakpoint on a stack-based variable and the contents of the stack is changed, the breakpoint is no longer valid.

Data breakpoints persist until you explicitly remove them. Removal is accomplished using the **Breakpoints** window **Delete** toolbar button or via the command line which is discussed below.

Data breakpoints are set using the **Break/Access...** or **Break/Write...** pull-down menu commands, **Breakpoints** window **Insert** toolbar button, and command line. When using the **Insert** toolbar button, enter `access` or `write` in the **Type** text box.

## Command Line

You can set data breakpoints using the following syntax in the **Command line** dialog box (<Ctrl>+L):

BR[EAKPT] [-] [WRITE *memref* [*when-then*]]
BR[EAKPT] [-] [ACCESS *memref* [*when-then*]]

The abbreviation **BR** can also be used to invoke this command. The referenced breakpoint may be deleted by using the optional minus (-) in the command.

## Debug Registers

Debug registers (DR0-DR3) are found on 386 and up processors. Because of the way the four debug registers work, one hardware breakpoint can use more than one register, which limits the number of hardware breakpoints you can set.

The number of registers used depends on the following:

1. Alignment of starting address
2. Length of variable referenced

A single register can cover any one of the following ranges:

| **Length:** | **Address:** |
|---|---|
| 1 byte | anywhere |
| 2 bytes | aligned on a 2-byte boundary (word aligned) |
| 4 bytes | aligned on a 4-byte boundary (dword aligned) |

For example, assume you have an 11-element array, *arrayx*, declared as type **char**, and that the first byte of the array begins at address 1007P.

Setting the following breakpoint would use all four registers: one for the first byte from 1007P to 1008P, one for the next four bytes, another for the next four bytes, and one for the last two bytes.

Command: `br access arrayx`

If you knew that all of **arrayx** was going to be accessed at the same time, you could do the following and use only one register:

Command: `br access byte arrayx`

# Exec Breakpoints

Exec breakpoints make use of the debug registers and the *break on instruction execution only* condition. They are provided to allow you to set a breakpoint on an instruction that resides in ROM.

When you set a software breakpoint, Soft-Scope checks the reference you entered to see if it is a RAM or ROM address. If it is a ROM address, a software breakpoint won't work because software breakpoints save the instruction that exists at the referenced location, then write over that instruction at the referenced location with an INT3 break instruction. This can't be done in ROM.

For ROM addresses, Soft-Scope automatically sets an Exec breakpoint. So, most of the time, you don't have to worry about whether the reference is in RAM or ROM.

Sometimes, however, the RAM location where a breakpoint is set might be written over by the application you are debugging. In such a case, Soft-Scope checks the reference you entered and if it corresponds to a RAM location, it sets a conventional software breakpoint. Then, when you run the application, the code at the referenced location is overwritten, removing the software breakpoint. To avoid this situation, use the Exec breakpoint instead of a software breakpoint.

# Command Line

Use the following syntax in the **Command line** dialog box (<Ctrl>+<L>) to set an Exec breakpoint:

BR[EAKPT] [-] [EXEC *coderef* [*when-then*]]

# Executing to a Location

Soft-Scope provides several methods to start target execution. Some of them will stop execution at a specific location.

## Go

Use the **Go** toolbar button or enter ***GO*** in the **Command line** dialog box (<Ctrl>+<L>) to start target execution until a breakpoint or fault is encountered.

G[O] [WRITE | ACCESS *memref*]
G[O] [[EXEC] *coderef*]
G[O] [RETURN]

If you do not specify where you want execution to stop, and there are no other breakpoints set, Soft-Scope opens a dialog box asking you to confirm that you really want to start execution.

## Go to a Specific Location

To execute to a specific location, use the **Go to...** command from the **Code** pull-down menu. To specify a code location, enter a code reference in the text box. To specify a memory location, enter a memory reference.

You can add a condition to the memory reference by entering a hardware breakpoint specifier (access, write, exec) in front of the memory reference using the following syntax:

[WRITE | ACCESS *memref*]
[[EXEC] *coderef*]

# Return from a Procedure Call

Use the **Go to return** toolbar button or the **Code/Go to return** pull-down menu command to return from a called procedure.

Soft-Scope calculates the expected return address from the stack and sets a breakpoint at that address. Target execution is started and continues until that breakpoint or some other breakpoint in the same scope is encountered.

# Go to a Cursor Position

1. From the **Code** window, move the cursor to the line where you want execution to stop.
2. Click on the **Go to cursor** toolbar button .

Soft-Scope sets a temporary breakpoint on the line containing the cursor and starts target execution. Execution continues until that breakpoint or some other breakpoint in the same scope is encountered.

# Stop

Use the **Stop** toolbar button or **Stop** command from the **Code** pull-down menu to stop execution without setting a breakpoint. This function works only when your target contains an interrupt-driven CSi-Mon monitor.

Soft-Scope can only stop an interrupt-driven monitor when interrupts are enabled. If you have long sections of critical code that disable interrupts, don't use the **Code/Stop** pull-down menu command while that code is executing.

When interrupts are disabled, Soft-Scope continues to assume that you have an interrupt-driven monitor, and receiver timeout messages may result.

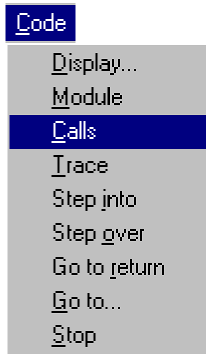
The configuration option **targ.polling** tells Soft-Scope when you are debugging code that disables the interrupts. Use the **Display** command from the **Options** pull-down menu to set this option to `on` to eliminate receiver-timeout messages:

```
targ.polling=on
```

4

# Procedure Call Sequence

To display the procedure-call sequence use the **Calls** command from the **Code** pull-down menu.




# Calls Window

Figure 4-7 shows the **Calls** window. The top entry is the current execution point. Each entry that follows, called the entry on the line above it.

To display the code for a specific call in the **Code** window, double-click <Button-L> on the call's entry or move the cursor to the desired call and press the **View** toolbar button.

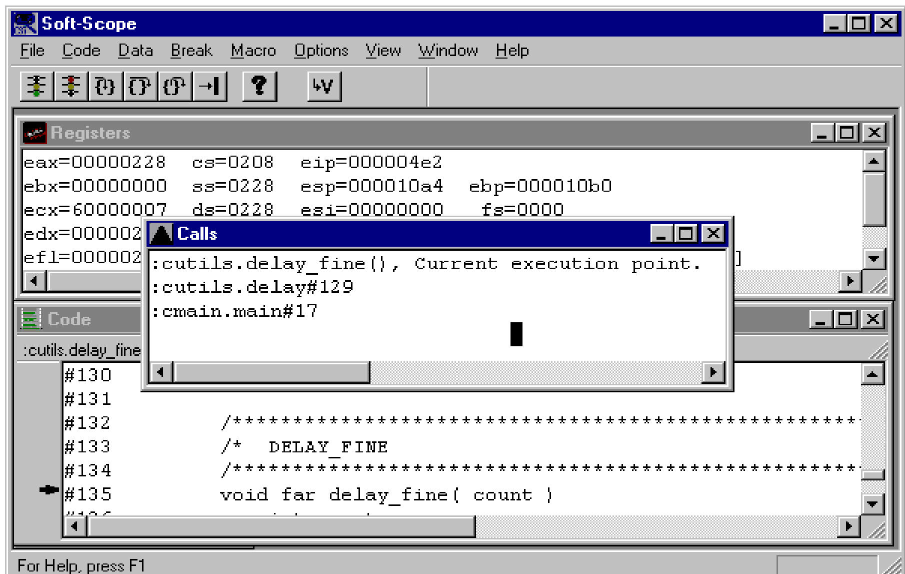



Figure 4-7: Calls window



## Command Line

To open the **Calls** window, enter ***CALLS*** in the **Command line** dialog box (<Ctrl>+<L>).

# Stack Information

If you have several nested calls, and you believe your application may be running low on stack space, enter ***STACK USAGE*** in the **Command line** dialog box to display the following information in the **Message** window:

- Range of addresses that the stack occupies
- Number of bytes free and the percentage of free stack space
- Number of bytes and corresponding percentage of stack space available when the deepest nested call was made

Enter ***STACK RESET*** in the **Command line** dialog box (<Ctrl>+<L>) to clear stack locations between the stack pointer and the bottom of the stack. If you are at a specific execution point and want to see the stack usage from this point on, use ***STACK RESET***.

After using ***STACK RESET***, run your application, then enter ***STACK USAGE*** in the **Command line** dialog box to see how close you have come to overflowing your stack area.

# Trace Window

The **Trace** window records all actions that affect execution. This information is useful in determining how your application reached a particular state. This information includes:

- Source lines executed via stepping
- Breakpoints encountered
- Program faults encountered
- Application loads and restarts
- Modification of application registers
- Modification of application memory
- I/O port access
- Breakpoints set and deleted
- Procedures not stepped over

4

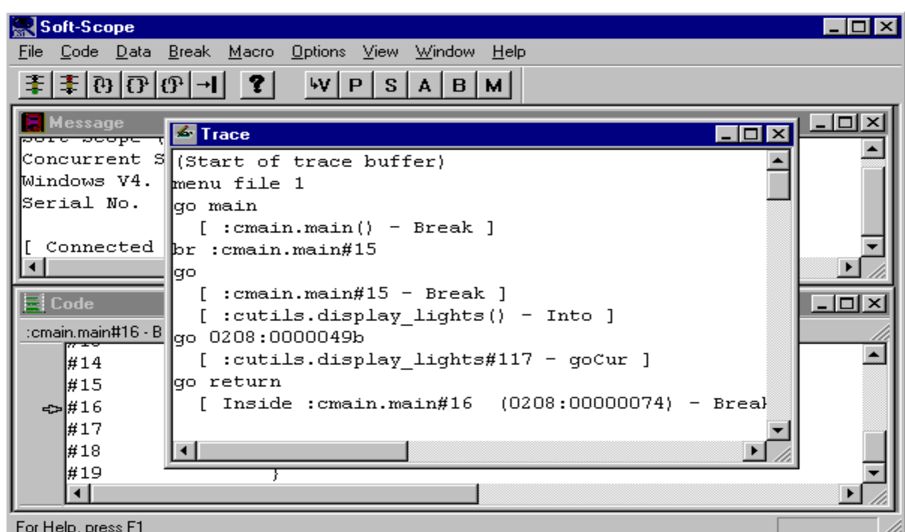
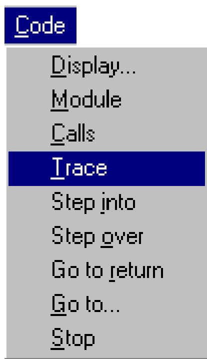



Figure 4-8: Trace window displaying procedures

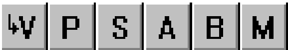
Trace information is kept in a circular buffer that is stored in a disk file. Controls for flushing the buffer and setting the file size are discussed later in this section.




Open the **Trace** window by selecting **Trace** from the **Code** pull-down menu.

## Toolbar Buttons

The **Trace** window displays information in four formats: Procedures, Source, Assembly, and Bus. Select the format using the toolbar buttons described below:




View • Procedures • Source • Assembly • Bus • Mode

View — Displays selected code, as identified by the cursor position, in the **Code** window. You can also double-click <Button-L> on the code. To activate from the keyboard press <Enter> or <V>.

Procedures — Displays procedures and execution events. See figure 4-8. To activate from the keyboard press <P>.

Source — Displays procedures, execution events, and source for each line executed. See figure 4-10. To activate from the keyboard press <S>.

Assembly — Displays procedures, execution events, source, and assembly code for executed lines. See figure 4-11. To activate from the keyboard press <A>.

Bus — Same as Assembly for this version of Soft-Scope. To activate from the keyboard press <B>.

Mode — Opens the **Assembly display modes** dialog box as shown in figure 4-9. To activate from the keyboard press <M>.



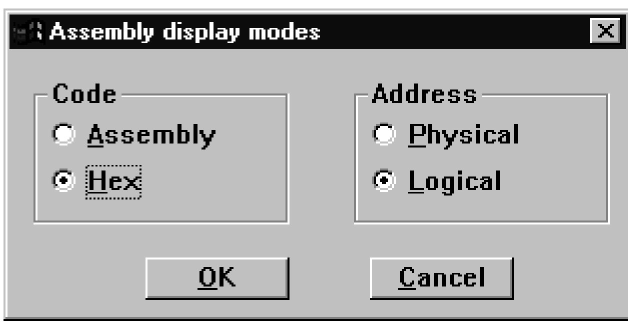



Figure 4-9: Assembly display modes dialog box

Code — The **Code** radio buttons control the way assembly source is displayed. **Assembly** mode is the default option. **Hex** mode adds opcodes in hex to the display.

Address — The **Address** radio buttons control whether assembly code addresses are displayed as physical (with a “P” suffix) or logical (in Segment:Offset format).

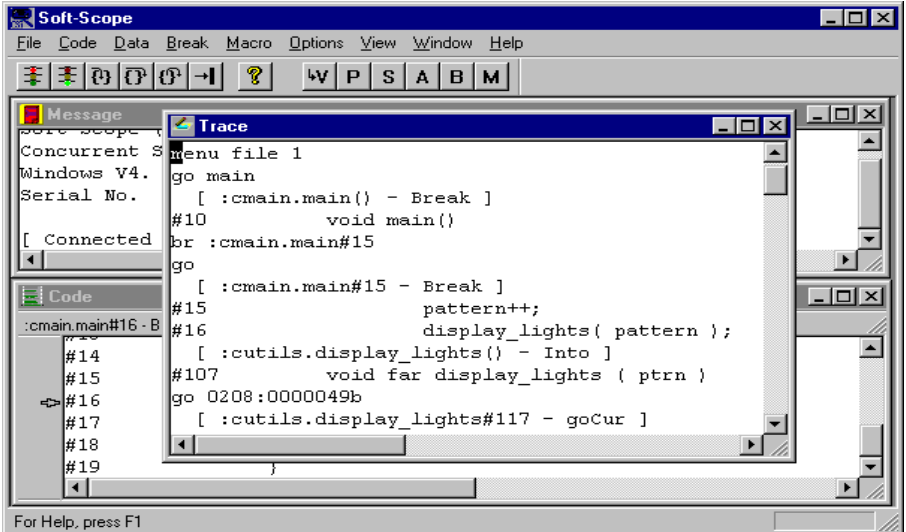



Figure 4-10: Trace window displaying procedures and source

# Command Line

Enter ***TRACE*** in the **Command line** dialog box (<Ctrl>+<L>) to open the **Trace** window.

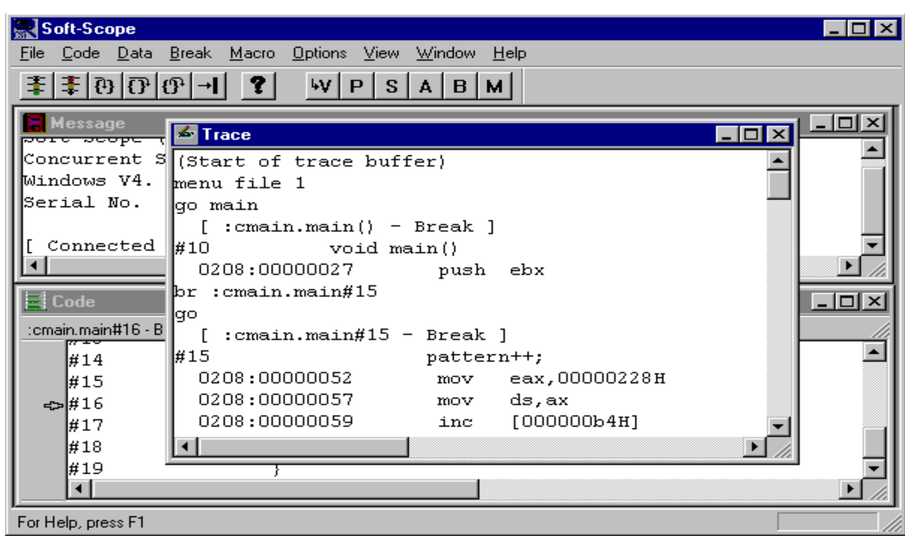



Figure 4-11: Trace window displaying procedures, source, and assembly code

# Trace Buffer

The trace buffer is a circular buffer that is flushed after each load operation. If you want to display trace information for several loads, use the following option in your configuration-options **.ini** file:

**trace.load=off** The trace buffer is flushed after each load. Off is the default.

**trace.load=on** The trace buffer is not flushed.

# Trace File Size

You can control the size of the trace file using the following option in your configuration-options **.ini** file:

**trace.filesize=128** Defaults to 128 kilobytes, but can be set from 16 to 1024 kilobytes.

# 5. Examining Data with Soft-Scope

## Chapter Contents


**Overview ........ 5-3**
**Numbers ........ 5-3**
Table 5-1: Default number bases ........ 5-5
**Operators ........ 5-6**
Table 5-2: C operators ........ 5-8
Table 5-3: Soft-Scope specific operators and functions ........ 5-9
**Strings ........ 5-10**
Table 5-4: String escape sequences ........ 5-11
**Reference Summary ........ 5-12**
Table 5-5: Reference summary ........ 5-12
**The Data Window ........ 5-14**
Figure 5-1: Data reference dialog box ........ 5-14
Figure 5-2: Display modes dialog box ........ 5-15
Figure 5-3: Data window in Eval mode ........ 5-17
Figure 5-4: Data window in expanded format ........ 5-18
**Data References ........ 5-19**
Figure 5-5: Before double-click on "->" ........ 5-23
Figure 5-6: After double-click on "->" ........ 5-23
**Referencing Memory ........ 5-25**
**Reference Scoping ........ 5-27**
Table 5-6: Reference Scoping ........ 5-28
**The Watch Window ........ 5-30**
Figure 5-7: Display modes dialog box ........ 5-31
Figure 5-8: Watch window in Normal display mode ........ 5-32

**The Symbols Window** .......... **5-34**
Figure 5-9: Symbols window in Procedures mode .......... 5-36
**Built-in Functions** .......... **5-37**
**Type Overrides** .......... **5-40**
**The Dump Window** .......... **5-46**
Figure 5-10: Dump modes dialog box .......... 5-47
Figure 5-11: Dump window in Byte mode, 8 bytes per line .......... 5-49
**Uploading Memory and Registers** .......... **5-50**
**The Registers Window** .......... **5-52**
Figure 5-12: Registers window for 80386EX target .......... 5-54
**CPU Structures** .......... **5-56**
Figure 5-13: IDT descriptors .......... 5-56
Figure 5-14: Data window in Normal mode .......... 5-58
Figure 5-15: Data window in Eval mode .......... 5-58
Table 5-7: Descriptor abbreviations .......... 5-59
**Real-Mode Structures** .......... **5-60**
Table 5-8: Peripheral Control Block .......... 5-60
Table 5-8: Peripheral Control Block (continued) .......... 5-61
**Application Input/Output** .......... **5-64**

# Overview

This chapter tells you how to reference and change data, and how to use operators, functions, and type overrides to view data in a format that will provide you with maximum information. You can reference and view static symbols anywhere your application can access them, and you can access many symbols outside the current execution context. In addition, you can reference, change, and dump memory, and access and change registers and CPU structures.

# Numbers

Soft-Scope supports the following number formats and bases:

- Binary numbers consist of the digits 0 and 1 and are designated by the suffix "Y".
- Decimal numbers are made up of the digits 0..9 and are designated by the suffix "T".
- Hexadecimal numbers can be designated by the prefix "0x", or with the suffix "H". They may contain the digits 0..9 and the letters A..F. Hex numbers must start with a digit to distinguish them from symbol names:

  **`e000ffa9H`** must be represented as **`0e000ffa9H`** or **`0xeoooffa9`**

- Floating-point numbers contain a decimal point and an optional fraction. They must begin with a digit (0..9) rather than a decimal point to differentiate them from symbol names:

  **`.132`** must be represented as **`0.132`**

- Exponential numbers use standard exponential format:

  | | |
  |---|---|
  | mantissa | may have an optional + or -<br>must start with characters 0..9 |

| | |
|---|---|
| | must contain a decimal point followed by some combination of characters 0..9 |
| exponent | must begin with an "E"<br>may have an optional + or - followed by some combination of characters 0..9 |

The following example demonstrates an exponential number:

```
-1.098567E+4
```

# Setting the Default Base

If a number does not have a suffix or prefix, its base is determined from the value of the **base** configuration option. To change the **base** option value, use the **Display** command from the **Options** pull-down and the **Modify** toolbar button.

The **base** option may be set to **10** or **16.** If the option is not set, numbers default to **base = 10**.

Some number bases are not determined by the base option. See table 5-1 for a list of number types and their default bases.

Table 5-1: Default number bases

| Number Type | Default Base |
|---|---|
| b800:04ac | Parts of a pointer always default to hex |
| #123<br>:module#123 | Line numbers are always assumed to be decimal |
| 123 <Spacebar> | Counts are always decimal |
| byte at arrayx<br>len 123 | Length counts are always assumed to be decimal |
| array[123],<br>array[2..6] | Array subscripts are always assumed to be decimal |
| 8..20 | Ranges of numbers default to decimal |
| 0x1fff>>x | Operand for shift operations (x) default to decimal |
| PORT 7f | Ports default to hex |
| PORT (9000) | Ports with expression values (defined as anything surrounded by parentheses) default to decimal |
| RETURN (12) | Return counts default to decimal |
| SELECTOROF | Selector overrides default to hex |
| -4.000000045E+5 | Exponential format defaults to decimal |

5

# Operators

Soft-Scope supports three classes of operators:

| | |
|---|---|
| Symbolic | Which provide quick access to data references |
| Arithmetic | Which provide standard arithmetic operations |
| Logical | Which provide standard, C-based true/false operations |

## Symbolic Operator Examples

Symbolic operators are used as shortcuts to access data references. Examples include pointer dereferencing, ranges, and type overrides:

```
*table_pointer

array_1[1..24], array_1[1...] and array_1[...24]

long at $ss:ebp
```

## Arithmetic Operators Return Numeric Values

Arithmetic operators are C-based arithmetic operators entered in the **Command line**, **?** (question mark), or **Data/Examine** dialog boxes. The following is an example of the increment operator and module operator:

```
++i

i % 3
```

# Logical Operator Examples

Logical operators are those used in true/false C-based operations. Examples include the *and* operator (**&&**) and the *not equal* operator (**!=**):

```
i && y
i != 1
```

Soft-Scope operator precedence is the same as C operator precedence. In table 5-2, operators on the same line have the same precedence, and rows are ordered in decreasing order of precedence.

**NOTE:** Soft-Scope does not use C's conditional operator (**?:**) and C's comma operator (**,**); Soft-Scope's SIZEOF parallels C's sizeof.

Table 5-3 lists Soft-Scope specific operators and their precedence relative to the C operators in table 5-2.

5

Table 5-2: C operators

<table>
<tr><th>Operator<br><u>Precedence</u></th><th><u>Associativity</u></th></tr>
<tr><td>( ) [ ] -> .</td><td>left to right</td></tr>
<tr><td>! ~ ++ -- + - * &</td><td>right to left</td></tr>
<tr><td>* / %</td><td>left to right</td></tr>
<tr><td>+ -</td><td>left to right</td></tr>
<tr><td><< >></td><td>left to right</td></tr>
<tr><td>< <= > >=</td><td>left to right</td></tr>
<tr><td>= = !=</td><td>left to right</td></tr>
<tr><td>&</td><td>left to right</td></tr>
<tr><td>^</td><td>left to right</td></tr>
<tr><td>|</td><td>left to right</td></tr>
<tr><td>&&</td><td>left to right</td></tr>
<tr><td>| |</td><td>left to right</td></tr>
<tr><td>= = += -= *= / = %= &= ^= | = <<= >>=</td><td>right to left</td></tr>
</table>

Table 5-3: Soft-Scope specific operators and functions

| Operator/Function | Precedence |
| --- | --- |
| # (:module#23) | same as -> |
| Type overrides | same as ++ |
| OFFSETOF | same as ++ |
| SELECTOROF | same as ++ |
| LENGTHOF | same as ++ |
| LEN[GTH] | same as ++ |
| AT | same as ++ |
| PORT | same as ++ |
| RETURN | same as ++ |
| SIZEOF | same as ++ |
| # (#123) | same as ++ |
| : (:module name) | same as ++ |
| . (.symbol name) | same as ++ |
| : (1234:5678) | between ++ and multiply |
| .. (array[1..2]) | between add and << |
| ... (array[...3]) | between add and << |
| ... (array[4...]) | between add and << |

For further information, see the *Operator* section in appendix A.

# Strings

You can enter data in string format, delimited by either single or double quotes and containing any printable ASCII character.

The only difference between the use of single and double quotes is that Soft-Scope includes a terminating null character within the string delimited by double quotes.

| If you type | Soft-Scope will create |
|---|---|
| **“frogs”** | **frogs\0** |
| **‘frog\’s’** | **frog’s** |

# Escape Sequences

An escape sequence represents the name of a character, a hex or octal number. Escape sequences start with the backslash (\).

**NOTE:** Escape sequences create a problem that Soft-Scope solves the same way C does. If you actually want a backslash in a string, you must use two of them (\\). For example, if you want to define a string that contains a subdirectory pathname, you must use the following format: “C:SUB_DIR1\\SUB_2”. This is not an issue when Soft-Scope prompts for a file name.

The escape sequences listed in table 5-4 are supported within strings, and are case-sensitive as in C.

# Where to Enter Strings

You can enter strings in the **Command line**, **?** (question mark), or **Data** dialog boxes.

Table 5-4: String escape sequences

| **Escape Sequence** | **Description** | **Hex Value** |
|---|---|---|
| \0 | Null Character | 0x00 |
| \b | Backspace | 0x08 |
| \t | Horizontal Tab | 0x09 |
| \n | Newline | 0x0a |
| \r | Carriage Return | 0x0d |
| \" | Double Quote | 0x22 |
| \' | Single Quote | 0x27 |
| \\ | Backslash | 0x5c |
| \f | Form Feed | 0x0c |
| \a | Audible Bell | 0x07 |
| \v | Vertical Tab | 0x0b |
| \xnn | nn is hex value | nn |
| \nnn | nnn is octal value | N/A |

# Reference Summary

The following table summarizes how to reference a data element or a memory address:

Table 5-5: Reference summary

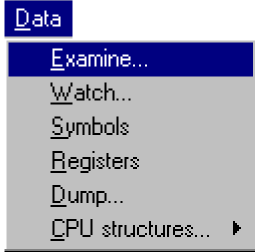
| | |
|---|---|
| *1234:0ffff* | Refers to a logical address. Note the "0" before the first "f." All numbers that start with an alpha character (A..F) must be prefaced with a "0". |
| *12345678L* | Refers to a linear address. |
| *12345678P* | Refers to a physical address. |
| *array1* | Refers to an array (unqualified). |
| *array1[1..3]* | Refers to a range of elements in an array. |
| *#linenumber* | Refers to a line in the current module. |
| :*module#linenumber* | Refers to a line in a module other than the current one. |
| *:module.procname.variable* | Refers to a variable whose scope is in another procedure in another module. |
| *:module.variable* | Refers to a variable whose scope is in another module. |
| *pointername* | Refers to the value of a pointer. |
| **pointername* | Refers to the area of memory where a pointer points (a dereferenced pointer). |

Table 5-5: Reference summary *(continued)*

| | |
|---|---|
| *pointername->elementname* | Refers to a single element of the structure where a pointer points. |
| *$register* | Refers to one of the target processor's registers. |
| *string at 200:0ffff* | Refers to a string at the given memory location. |
| *structurename* | Refers to a structure (unqualified). |
| *pointername.elementname* | Refers to a single element of a structure. |
| *structx at 200:0ffff* | Refers to the display of the contents of the given address in the format defined by the data type of *structx* . |
| *typeoverride variable* | Refers to a memory location where the variable is stored displayed as the type specified by *typeoverride* . |
| *variable* | Refers to a variable in your program. |
| *word at $ss:$esp* | Refers to the word at the top of the current stack. |

# The Data Window

The contents of memory locations associated with data are displayed in the **Data** window as shown in figure 5-3. A variable can be evaluated in the **Data** window by double-clicking <Button-L> on it in the **Code** window.

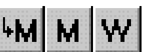



To open the **Data** window, choose **Examine...** from the **Data** pull-down menu. This opens the **Data reference** dialog box. Enter one or more data references (separated by a comma) in the text box as shown in figure 5-1.

Figure 5-1: Data reference dialog box

# Toolbar Buttons

Items in the **Data** window are manipulated by using the toolbar buttons discussed below:



Modify • Mode • Watch

Modify — Modifies the value of a data reference. Place the cursor on a data reference. Click <Button-L> on the **Modify** toolbar button and enter the new value in the dialog box. Double-clicking <Button-L> on a data reference will also open the dialog box. To activate from the keyboard press <Enter>.

Watch — Moves a data reference from the **Data** window to the **Watch** window. Place the cursor on a data reference and click <Button-L> on the **Watch** toolbar button. To activate from the keyboard type <W>.

Mode — Opens the **Display modes** dialog box as shown in figure 5-2. To activate from the keyboard type <M>.

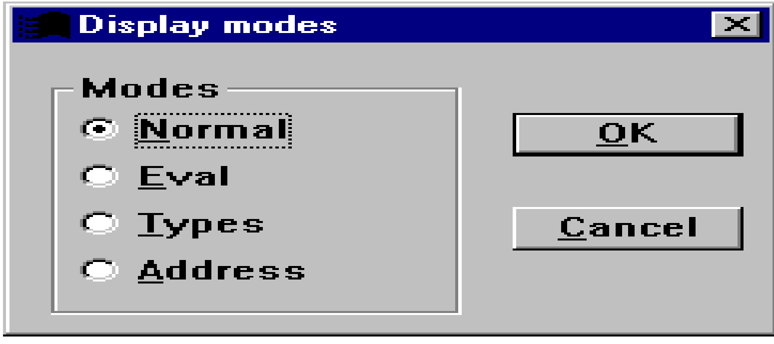



Figure 5-2: Display modes dialog box

| | |
|---|---|
| Normal | Displays variable's name and value. Display content varies depending on the variable's data type. |
| Eval | Displays additional information if available. Display content for a pointer includes the LDT or GDT, physical address, and number of bytes from the pointer location to end of segment. |
| Types | Displays reference name and type. |
| Address | Displays selector, offset, and physical addresses associated with a symbol. |

## Command Line

To display data references in the **Data** window enter the following commands in the **Command line** dialog box (<Ctrl>+<L>). EVAL opens the **Data** window in Eval mode. TYPE opens it in Type mode.

EVAL(*memref* | *coderef* ) [, (*memref* | *coderef* )]*

TYPE (*memref* | *coderef* ) [, (*memref* | *coderef* )]*

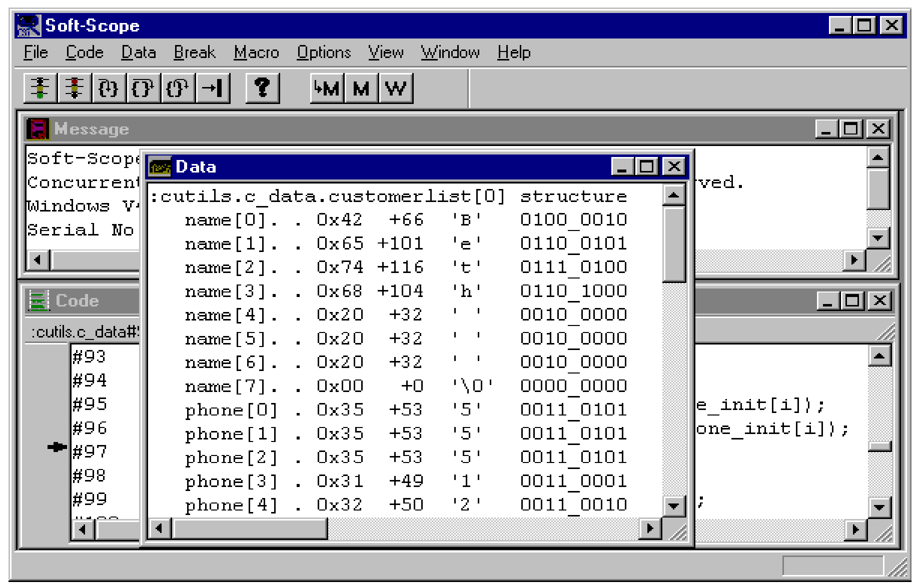



Figure 5-3: Data window in Eval mode

5

# Double-click for Quick References

In the **Data** and **Watch** windows, you can use the double-click <Button-L> function to manipulate the way you view structures, unions, and pointers. Assume the following structure:

```
struc1 structure {...}
```

Double-clicking on <Button-L> or after the word “structure” toggles the display between compressed format and expanded format shown in figure 5-4.

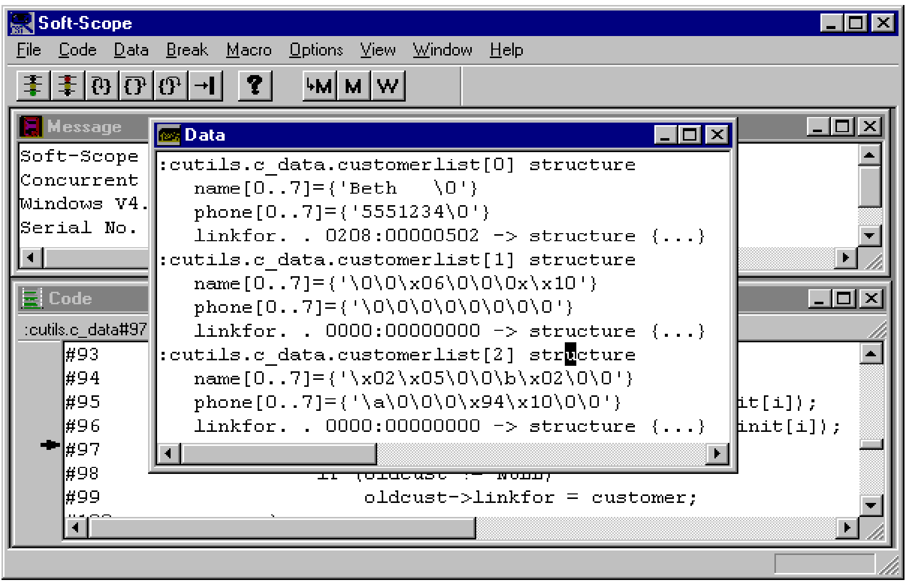



Figure 5-4: Data window in expanded format

Double-clicking <Button-L> before the word “structure” places the structure in a dialog box for modification. Use this feature to reference pointers as follows:

Double-click <Button-L> Displays the pointer in a dialog box before the “->” for modification

Double-click <Button-L> Dereferences the pointer and on the “->” displays the dereferenced data

Double-click <Button-L> Displays the indirect data in a dialog after the “->” box for modification

# Data References

Soft-Scope can reference and examine the following variable types:

- Simple variables
- Arrays
- Structures
- Pointers
- Unions
- Bit fields

If you use a Soft-Scope keyword, such as ***PORT***, ***INT***, or ***OFFSETOF*****,** as a variable, you won't be able to examine it in the data window unless you put a period in front of it to distinguish it from a symbol:

Data reference: **`.port`**

5

# Simple Variables

Reference a variable by typing the variable's name at the prompt. If the variable isn't a structure or array, Soft-Scope determines the variable's type and displays the hex and decimal values of the associated memory locations:

Data reference: **`pattern`**

```
PATTERN = 0x00000041    +65
```

# Referencing Arrays

If the variable is an array, referencing it without an index or subscript implies you mean the entire array. You can also display single elements of an array, or ranges of elements, by using the appropriate subscripts. You can even use integer variables as subscripts.

## Displaying an Entire Array

To reference an entire array, like the character array shown below, use the array name:

Data reference: **lights**

```
LIGHTS[0..7]={'**-***-*'}
```

The display is similar for numeric arrays:

Data reference: **numarray**

```
NUMARRAY[0..9]={2,0,3,1,8,6,7,3,7,4}
```

## Displaying a Single Element of an Array

To reference single elements of an array, use the array name with a subscript:

Data reference: **lights[2]**

```
LIGHTS[2]={'*'}
```

## Displaying a Selected Number of Arrays

To reference several array elements, use the array name with a subscript range:

Data reference: **lights[2..6]**

```
LIGHTS[2..6]={'-*-*'}
```

Use the open-ended operators to reference array elements from or to a specific element:

Data reference: **`lights[2...]`**

Data reference: **`lights[...6]`**

## Variables as Subscripts

You can use an integer variable as a subscript. If the value of *`i`* is 3, the following example demonstrates the reference and the resulting display:

Data reference: **`lights[i]`**

```
LIGHTS[3]={'-'}
```

If you specify an index that is outside the defined size of an array, Soft-Scope returns the value of the memory location specified, but a question mark is displayed next to the index.

```
LIGHTS[9?]= {'*'}
```

5

## Referencing Structures

Soft-Scope handles structure references similarly to arrays. To reference the entire structure named *`struc1`*, use its unqualified name:

Data reference: **`struc1`**

To reference an individual element of a structure, type a period to separate the structure's name from the member's name:

Data reference: **`struc1.xint`**

# Referencing Unions

Union-reference syntax is based on the syntax of structures; simply enter the union's name:

Data reference: **date**

```
union {
      struct {
            unsigned char  day;
            unsigned char  month;
            unsigned char  year;
      } today;
      unsigned long days_since_year_0_ad;
}     date;
```

# Referencing Bitfields

Soft-Scope also handles bitfields like structures. To reference a structure of bitfields, use the structure's name:

Data reference: **enet_pkt**

```
struct enet_pkt_type {
      unsigned int  crc       :2;
      unsigned int  data            :16;
      unsigned int  pkt_type  :3;
      unsigned int  source_addr     :4;
      unsigned int  dest_addr :4;
      unsigned int  preamble  :3;
} enet_pkt;
```

To reference a single bitfield, separate the structure name from the bitfield name with a period:

Data reference: **enet_pkt.data**

# Referencing Pointers

To reference the value of a pointer, use the pointer's name:

Data reference: **`oldcust`**

# Dereferencing Pointers

To dereference a pointer, double-click <Button-L> on the pointer operator ( -> ) in the **Data** or **Watch** window. See figures 5-5 and 5-6 for examples.

```
Data
:cutils.c_data.oldcust=(0540):0000 -> structure {...}
```

Figure 5-5: Before double-click on "->"

```
Data
*:cutils.c_data.oldcust structure
   name[0..7]={'L\0T\0\\0d\0'}
   phone[0..7]={'l\0t\0\x7f\x053\x05'}
   linkfor. . (0540):06c9 -> structure {...}
```

Figure 5-6: After double-click on "->"

Double-clicking <Button-L> before the pointer operator lets you modify the pointer, and double-clicking <Button-L> after the pointer operator lets you modify the indirect data.

When a pointer points to a structure, the pointer's name with the structure-pointer operator (->), entered in a dialog box, references a single element of the structure:

Data reference: **`oldcust->name`**

## Selector Is Not Stored in Memory

Near pointer selectors for flat and small memory model applications aren't stored in memory. The offset is stored and the selector is assumed. When a dereferenced near pointer has parentheses around its selector, Soft-Scope is telling you that the selector is not actually stored in memory.

## Making Complex Assignments

Use complex expressions in assignment statements. Expressions such as b=c or a + (b=c) can assign values to arrays, GDTs, or other complex types:

```
byte at 100P len 5= byte at 40p len 5
byte at 20P len 5 + (byte at 100P len 5= byte at 40p
len 5)
```

# Referencing Memory

You can reference memory with any address, symbol name, or expression that resolves to a memory location. You can even use data types to dictate formatting.

- Use a code reference as a memory reference, because code is stored in memory:

  ***symbolname***

- A logical address consists of a selector and an offset, separated by a colon:

  ***selector:offset***

- A linear address is an address that has not been passed through the processor paging tables. Use the following syntax:

  ***hexnumber*L**

- A physical address is the address as it appears on the data bus, and is identical to a linear address if paging is not enabled. Use the following syntax:

  ***hexnumber*P**

- Use operators and values in any combination:

  ***symbolname operator hexnumber***

- Data references aren't necessarily stored in memory, so you can't use them as memory references unless you know they resolve to memory addresses:

  ***variablename***

5

# Using the Symbols Window to Find Code References

Use a code reference when you are referencing a program symbol:

Memory reference: **`display_lights`**

Use logical references when you know the selector of the memory you want to view. The following example displays memory at offset 0f200 in the segment given by selector 203 in the format of *`structx`*:

Memory reference: **`structx at 203:0f200`**

If you know the name of the symbol you want to reference, but not the logical address, use the ADDRESSOF operator (&). For example:

Command line: **`eval & lights`**

```
0228:000000b8 gdt [69] 00005238p - 4136 bytes
```

This example displays memory at *`structy`* in the format of *`structx`*:

Memory reference: **`structx at &structy`**

Use a physical reference to view memory without regard to the segment that contains it. The reference in the example below might be used to set a hardware breakpoint on the first byte of a variable that begins at physical address 20P:

Memory reference: **`byte at 20P`**

By using an expression as a memory reference, you can define memory locations that you might not know the physical or logical address for. The expression in the example below references a location 10 hex below the base pointer register:

Memory reference: **`$ss:$ebp-0x10`**

# Reference Scoping

You can access the same variables your application can access.

You can also reference many variables outside of your current program context by using the following basic guidelines:

- Put a colon in front of the module name.
- Use periods to separate modules from procedures and procedures from variables.

## Examples

See the examples below to learn when to use the module name, procedure name, colon, and period to define a reference.

To reference a global variable or a static variable in the module where the execution pointer is currently located, use the variable's name:

Data reference: `c`

You can reference a static variable in a procedure other than the one where the execution pointer is located by separating the procedure name from the variable name with a period:

Data reference: `c_data.i`

Reference a variable declared in a module other than the one the execution pointer is located in by putting a colon in front of the module name, and a period between the module name and the variable name:

Data reference: `:cutils.i`

To reference a static variable defined in a procedure located in a module other than the one where the execution pointer is, put a period between the module and the procedure and the procedure and the variable:

Data reference: `:cutils.delay.i`

By using the rules listed in table 5-6, you can reference any variable located in any module or procedure as long as it is not a register variable or automatic (stack-based) variable.

Table 5-6: Reference Scoping

| **Where is the variable declared?** | **How should it be referenced?** |
|---|---|
| Same procedure | variablename |
| Global in scope | variablename |
| In a different procedure, but the same module, static | procname.variablename |
| In a different module, but not in a procedure | :modname.variablename |
| In a different module and in a procedure, static | :modname.procname.variablename |

# Referencing Automatic (Stacked-based) Variables

Because stack-based variables are stored on the stack, they are only accessible when the execution pointer is in the procedure where they are located. Trying to reference these variables from outside the procedure in which they are defined displays an error message.

If you try to examine a stack-based variable before it has been initialized, a value may be displayed in the **Data** window, but it will probably be the wrong value. There will be a question mark next to the reference in the display because you have to step at least once in a procedure to initialize the stack for that procedure.

Also, before you examine variables that aren't initialized until the program accesses them, you should execute to a point at least one line beyond the one that assigns a value to them.

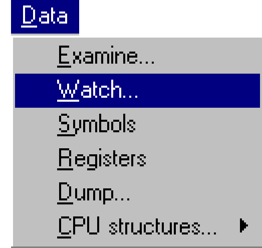
## Referencing Register Variables

Register variables aren't stored in memory, so Soft-Scope can't access the value of a register variable unless the execution pointer is in the procedure where the variable is defined. Trying to reference these variables from outside the procedure in which they are defined displays an error message.

# The Watch Window

The **Watch** window is used to monitor data references as your application executes on the target. The contents of the **Watch** window are updated after every Soft-Scope command that causes the target to execute, such as single stepping. If you are using an interrupt driven CSi-Mon monitor, the window update rate is defined by the **exec.refresh** configuration option (default value is zero seconds). See the *Soft-Scope Configuration Options* section in the chapter *Configuring Soft-Scope* for details on using this option.

One way to place a data reference into the **Watch** window is by using the **Watch...** command from the **Data** pull-down menu. Enter the data reference you would like to monitor in the dialog box. To monitor more than one reference at a time in the **Watch** window, enter multiple references in the dialog box, separated with a comma.




To place a data reference into the **Watch** window from the **Code** window, double-click <Button-L> on the data reference. This will move it to the **Data** window. Then use the **Watch** toolbar button to place the data reference into the **Watch** window.

You can also use the **Watch** toolbar button in the **Symbols**, **Data**, and **Registers** windows to place a reference in the **Watch** window.

## Toolbar Buttons

Use the following toolbar buttons to manipulate items in the **Watch** window:

Modify • Mode • Insert • Delete

| | |
|---|---|
| Modify | Assign a value to the scalar variable nearest the cursor position. To activate from the keyboard press <Enter>. |
| Insert | Insert a data reference in the **Watch** window by entering a data reference in the dialog box. To activate from the keyboard press <Ins> or <I>. |
| Delete | Delete a data reference from the **Watch** window identified by the cursor position. To activate from the keyboard press <Del> or <D>. |
| Mode | Change the **Watch** window display mode. Figure 5-7 shows the **Display modes** dialog box. To activate from the keyboard press <M>. |



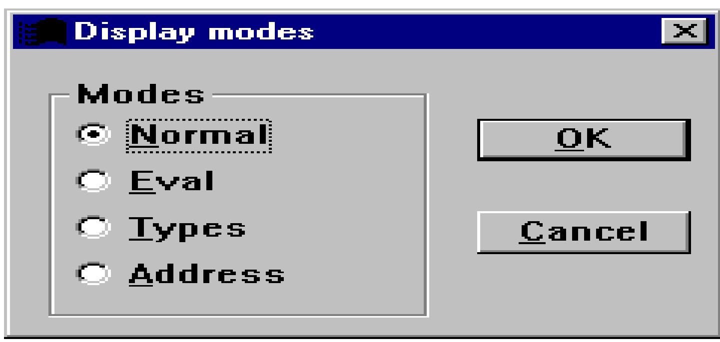



Figure 5-7: Display modes dialog box

| | |
|---|---|
| Normal | Display variable name and value. Display content varies depending on variable's data type. See figure 5-8 for an example. |
| Eval | Display additional information if available. |
| Types | Display reference name and type. |
| Address | Display selector, offset, and physical address. |

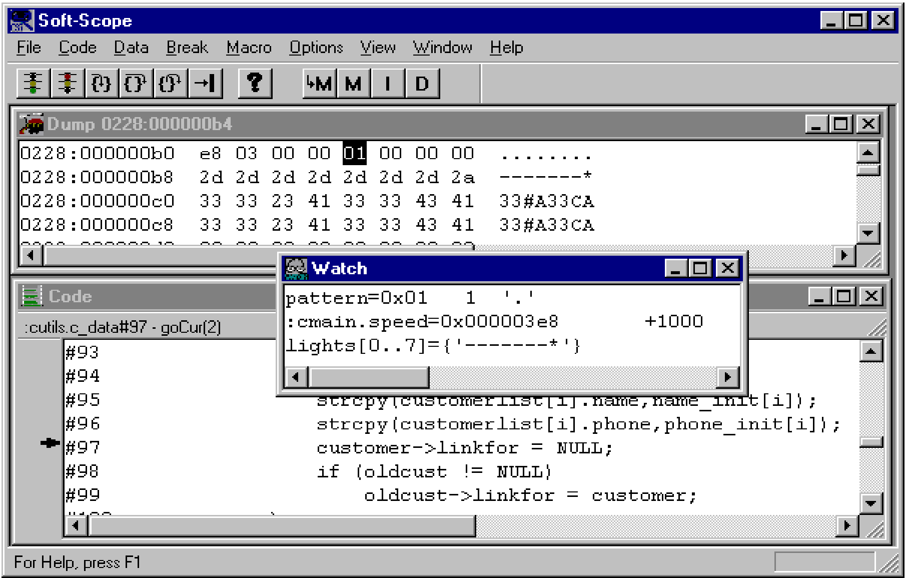



Figure 5-8: Watch window in Normal display mode

# Command Line

Use the following syntax in the **Command line** dialog box (<Ctrl>+<L>), to place a reference in the **Watch** window:

WATCH [*memref* ] [, *memref* ]*

# Watching a Pointer

When you place a pointer in the **Watch** window, the value of the pointer itself is monitored for change and not the location where the pointer is pointing. To view the data pointed to by the pointer, *dereference* the pointer by double-clicking <Button-L> on the pointer operator (->).

# Watching Memory

To watch the contents of any other memory location, use a type override. The following command line example will cause the first byte at the address specified to be displayed in the **Watch** window:

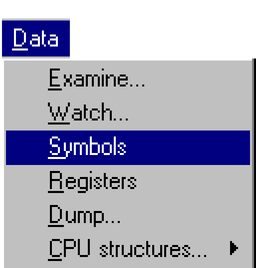
Command line: `WATCH BYTE AT 200:12`

**NOTE:** A large number of references in the **Watch** window will degrade Soft-Scope's performance because it has to fetch information from the target for each reference.

5

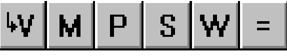
# The Symbols Window

View the symbols of your application by opening the **Symbols** window. Both **Data/Symbols** and **Code/Module** pull-down commands open this window, but the display mode is different.



## Toolbar Buttons

The toolbar buttons described below allow you to view your application's symbols:




View • Modules • Procedures • Symbols • Watch • Assign

View Places the symbol identified by the cursor in the **Data** window if it is a variable, or opens the **Code** window at the symbol's location if it is a module or procedure. When the **Code** window is open, you can press the **Locate** toolbar button to return the display to the current execution point. To activate from the keyboard press <Enter> or <V>.

| | |
|---|---|
| Modules | Displays a list of your application's modules. To activate from the keyboard press <M>. |
| Procedures | Changes the display to include your application's modules and procedures. See figure 5-9 for an example. To active from the keyboard press <P>. |
| Symbols | Changes the display to show the application's modules, procedures, and symbols (variables). To activate from the keyboard press <S>. |
| Watch | Places the symbol identified by the cursor position in the **Watch** window. To activate from the keyboard press <W>. |
| Assign (=) | Lets you assign a filename to the module identified by the cursor position. To activate from the keyboard press <=>. |

## Command Line

5

Use the following syntax in the **Command line** dialog box (<Ctrl>+<L>) to select what information is placed in the **Symbols** window:

MODULES [[TO] *:modname*]
MODULES :*modname=filename*

MODULES :*modname =filename* assigns a listing or source file to a program module. This is useful to use in macros, or to specify a pathname:

```
modules :cmain="c:\prog1\main.c"
```

PROCEDURES [[TO] *coderef*]

SYMBOLS [[TO] *coderef*]

If you use an address for a *coderef*, the window will open with the procedure nearest the address displayed.

# Displaying Global Symbols

Global symbols, including procedure names, are displayed at the top of the **Symbols** window when you select either the **Procedures** or **Symbols** toolbar buttons. When you select the **Modules** toolbar button, only the heading, *Global Symbols,* is displayed.

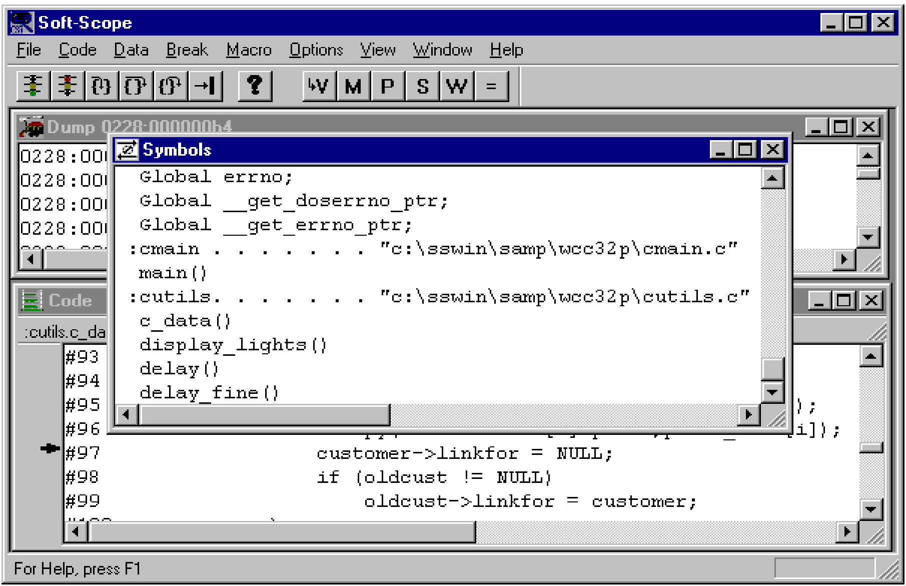



Figure 5-9: Symbols window in Procedures mode

# Built-in Functions

Soft-Scope provides six functions that allow you to perform specialized operations. They can be used in any valid expression (the parentheses around the parameters are optional):

| | |
|---|---|
| LENGTHOF (*x*) | Returns the number of array elements associated with reference *x*. |
| OFFSETOF (*x*) | Returns the offset portion of pointer *x*. |
| PORT (*x*) | Performs target-hardware I/O at port *x*. Only 8-bit, 16-bit, or 32-bit type overrides are allowed with this function. |
| RETURN or RETURN (*n*) | Returns the expected return address of the current procedure. Return(*n*), where *n* is an integer parameter, calculates the return address for the nth nested call. |
| SELECTOROF (*x*) | Returns the selector portion of pointer *x*. |
| SIZEOF (*x*) | Returns the size of *x* in bytes. |

**NOTE:** Use a period (.) in front of these reserved words if you have a variable of the same name.

## Determining Addresses

OFFSETOF, SELECTOROF, and RETURN help you determine the address of a reference. Use the ADDRESSOF operator (&) with the first two functions:

Data reference: `offsetof &lights`

5

# Using Return as a Memory Reference

RETURN can be used to find an expected return address, or in combination with other Soft-Scope commands to define a memory reference. The following example causes target execution until the expected return address of the current procedure is reached:

Command: `go return`

This is the same as choosing the **Go to return** toolbar button .

# Determining How Many Elements in an Array

LENGTHOF is useful if you need to determine how many elements are in an array. If the reference doesn't represent an array, LENGTHOF returns a '1'. The following example shows a reference to the array **lights** and the resulting display:

Data reference: `lengthof lights`

```
0x00000008        8
```

# Reading and Writing to Port Addresses

You can read from or write to I/O port addresses using PORT. Valid port addresses range from 0 to 0FFFFH.

**CAUTION:** Be careful about reading what you have just written to an I/O address. With some devices, reading from them may change their state, and may not return a value just written to them.

Also, it is important that you reference the correct number of bytes when reading to or writing from a port. For example, if you read 32 bits from a 16-bit port 3, Soft-Scope will read all of port 3 and 16 bits of port 4 (assuming port 4 is at least 16 bits).

If you write a byte to a word-length port, your target could hang while waiting for an expected second byte of data.

To view the value of a port, reference the port in the **Data** dialog box:

Data reference: `port 3`

The following example writes a byte-length value to port 3:

Data reference: `port 3 = 04H`

The next example reads 32 bits from port 2:

Data reference: `dword port 2`

5

# Type Overrides

By using a type override, you can cause a variable to be displayed *as though* it were a type other than that declared in your application. Type override does not perform a true type conversion on the variable, but merely overlays a new type at its address.

This is especially helpful for logical, linear, or physical references, since they have no types assigned to them, and for symbols that have been compiled without type information.

Type overrides have two basic forms:

***type-override variable***

***type-override* AT *address***

The following can be used to instantiate ***type-override***:

- Any C data type.
- Any data type listed in table A-1, “Data types for use in type overrides,” in appendix A.
- Any user-defined variable that is currently accessible by your application and Soft-Scope (stack-based variables must actually be on the stack).

## Applying a Type Override to a Variable

The simplest of the above forms specifies a type before a variable:

Data reference: **`long n`**

# Applying a Type Override to an Address

Use the second form to apply type overrides to addresses, including registers, selectors, and pointer (use the AT operator).

The following example displays the contents of the specified logical address in pointer format:

Data reference: **`pointer at 200:0ffff`**

The next example displays (as a double) the contents of the memory specified by a logical reference:

Data reference: **`double at $ss:$ebp`**

(Soft-Scope requires that register names begin with a '$'.)

If you had just pushed the contents of the flags register and needed to know what had been pushed, you could use the following to display the data on the stack in flag format:

Data reference: **`fltype at $ss:$esp`**

```
fltype at 0040:000000d0 = 0x03e8       1000
         [nt iopl=0 of df IF TF SF ZF af pf cf]
```

AT works with TSS overrides:

Data reference: **`TSS386 at $tr`**

You can use the ADDRESSOF operator (&) to specify an address for use with the operator AT.

5

# Using a Variable to Superimpose its Data Type over the Address of Another Variable

You can also override the address of a symbolic reference to superimpose the type of one reference over the address of another. Suppose two structures *structx* and *structy*. Use the following override to display *structy* in the format of *structx*:

Data reference: **`structx at &structy`**

Or you can use an address to designate the location you want overlaid with a new format:

Data reference: **`structx at 200:ff0f`**

# Using a User-declared Variable to Define a Type Override

A user-declared variable can also be a type override of data at a specified address. The following example displays memory at `$ss:$ebp - 0x10` in the data-type format of the variable *n*.

Data reference: **`n at ($ss:$ebp - 0x10)`**

## Changing the Amount of Memory Displayed

The operator LEN[GTH] helps you specify how much memory you want Soft-Scope to display.

This example displays 10 words beginning at the location of *n*:

Data reference: **`word n length 10`**

The next example dumps ten bytes beginning at the address specified:

Command: **`dump byte at 200:1df length 10`**

## Using Expressions in Type Overrides To Do Mathematical Operations

5

You can use expressions in type overrides.

The example below causes Soft-Scope to apply the type override to the contents of the memory location of *n*, add 2 to the value in that location, and display the result:

Data reference: **`long n + 2`**

```
0x00000004        +4
```

The next example displays one word beginning at a stack memory location 10 (hex) less than the base pointer:

Data reference: **`word at ($ss:$ebp - 0x10)`**

## Assigning Values Using Type Overrides

You can assign a value to a variable using a type override.

The following example assigns a real value of 3.0 to the memory location associated with the variable *speed*. The data type—float in this example—and the value must be of the same type:

Data reference: **float speed = 3.0**

Make complex assignments by using a type override on the right side of the equal sign:

```
200P len 10 = byte at 100P len 10
```

If you want to examine the new value in the format of the override's type, be sure to reference the variable using the appropriate basic form:

Data reference: **float speed**

Use complex expressions in assignment statements. Expressions such as *b=c* or *a + (b=c)* can assign values to arrays, GDTs, or other complex types:

```
byte at 100P len 5 = byte at 40p len 5
byte at 20P len 5 + (byte at 100P len 5 = byte at 40p
len 5)
```

## Displaying Data in its Most Useful Format

Use type overrides to manipulate the way data is displayed so you can see the information you need in a format that is easy to understand. Here are a couple of examples.

Assume a C pointer called *dev_names*, declared as pointing to *char* (that is, *char *dev_names*):

Data reference: ***dev_names**

The example above only displays a single byte, because of the declared type. If you knew that the pointer was pointing to a string of characters, you could override the default display and display the entire string:

Data reference: **string *dev_names**

```
'DISK\0' 5
```

Use a variable defined as an array along with the operator AT to display a section of memory in array format:

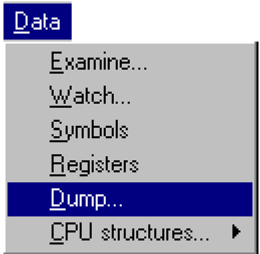
Data reference: **array1 at 400:6**

There are other ways to do the same thing. For example, if in the example above *array1* is a 3-element array of *long*, the following creates the same display:

Data reference: **long at 400:6 length 3**

# The Dump Window

The **Dump** window is used to modify the value of a target memory location and display target memory in a formatted list. The display contains three columns showing the memory address, hex representation, and ASCII representation of memory values. See figure 5-11 for an example. The display format is selected using the **Mode** toolbar button described below.




One way to open the **Dump** window is to use the **Dump...** command from the **Data** pull-down menu and entering a memory reference in the dialog box. For example, enter `byte at 203:0f` in the dialog box to open the **Dump** window and display memory starting at the logical address 203:0f with the byte highlighted.

**NOTE:** If you don't specify an address the first time you open the **Dump** window, memory will be displayed starting at physical address 00000000P.

# Toolbar Buttons

Use the following toolbar buttons to modify items or change their display format:

Modify • Mode • Shift

Modify Change the value of the memory location identified by the cursor position. Enter the new value in the dialog box. To activate from the keyboard press <Enter>.

Shift Shift the starting address of each line forward by one byte to align 16- or 32-bit fields. To activate from the keyboard press <S>.

Mode Change the **Dump** window display mode. Figure 5-10 shows the **Dump modes** dialog box. To activate from the keyboard press <M>.

5

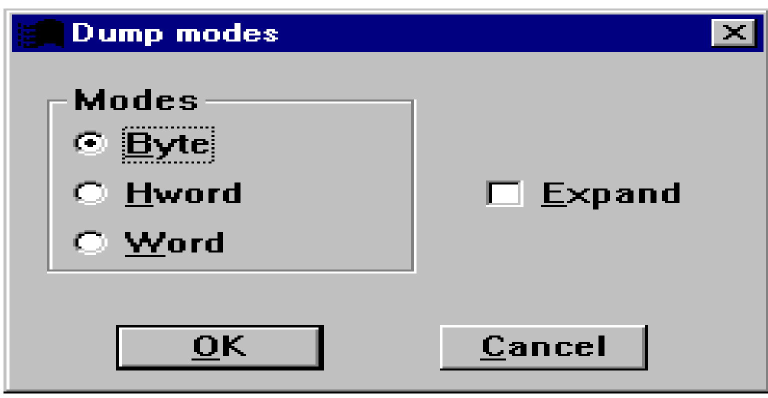



Figure 5-10: Dump modes dialog box

Byte Set hex display width to byte (8 bits).

Word Set hex display width to word. The word will be either 16- or 32-bits depending on the value of the **sym.wordsize** configuration option. The default value of **sym.wordsize** is 32. You can change its value using the **Display** command from the **Options** pull-down menu as described in the *Soft-Scope Configuration Options* section of the *Configuring Soft-Scope* chapter.

Hword Set hex display width to half-word (16 bits). This radio button appears when **sym.wordsize** equals 32.

Dword Set hex display width to dword (32 bits). This radio button appears when **sym.wordsize** equals 16.

Expand Set display width to 16 bytes. This does not effect the value of **sym.wordsize**.

**NOTE:** The address format depends on the type of memory reference you used to open the **Dump** window. For a logical reference, the address format will be logical. For a physical reference, the address format will be physical.

# Command Line

From the command line, enter a memory reference in the **Command line** dialog box (<Ctrl>+<L>) using the following syntax:

```
DUMP[[TO] memref]
```

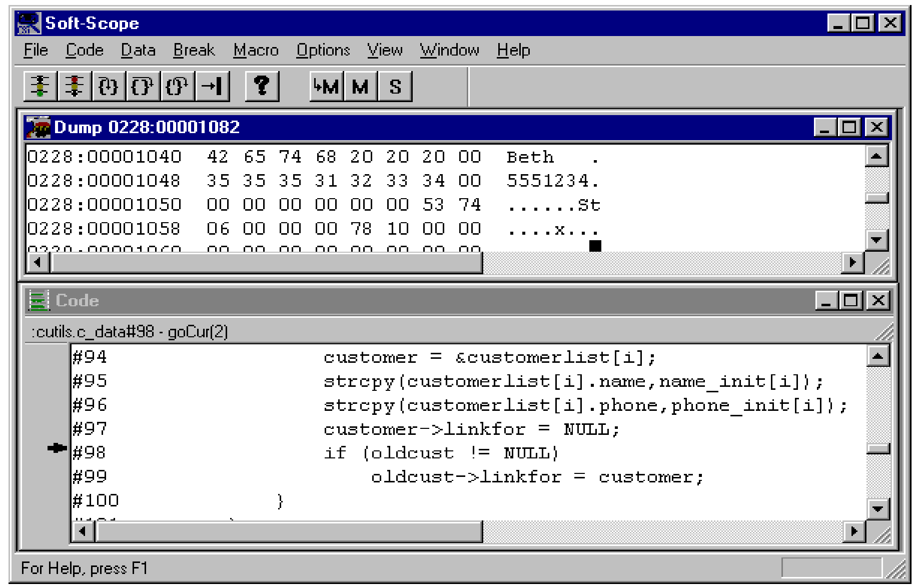
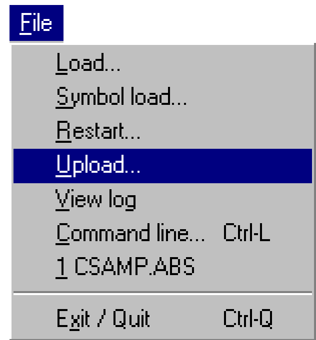



Figure 5-11: Dump window in Byte mode, 8 bytes per line

5

# Uploading Memory and Registers

To save memory and register values in a disk file, use the **Upload...** command from the **File** pull-down menu. From the disk file you can view or edit these values and reload them using the **Load...** command from the **File** pull-down menu.




Enter the starting address, length (in bytes), and the name of the disk file where you want to store this information in the **Upload** dialog box. To save the register values, enter **`registers`** after the length and before the file name. To save an entire procedure or module, enter its name in the dialog box followed by the file name.

**NOTE:** If you enter an address without a length, only the address is saved.

To store 8 bytes starting at the logical address 208:00000028 in the file **test.dat**, enter the following:

```
208:00000028 length 8 c:\temp\test.dat
```

To store the ***delay*** module in a file named **delay.dat**, enter **`delay test.dat`** in the dialog box.

# Command Line

Use the following syntax in the **Command line** dialog box (<Ctrl>+<L>):

UPLOAD *memref* [REGISTER[S]] *filename*
UPLOAD REGISTER[S] *filename*

Using the optional REGISTERS parameter causes the current register image to be saved to the Upload file.

**NOTE:** If Soft-Scope determines that the *filename* you give already exists, it asks if you want to append to the file, overwrite the file, or escape so you can try again with a new *filename*.

# Format of Upload Files

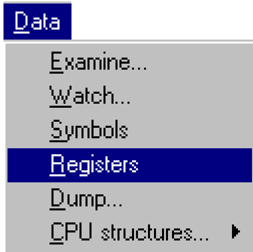
5

Upload files contain a series of one-line text records that you can edit with a text editor. The format, described below, must be maintained when you edit the file. Otherwise Soft-Scope will consider the file invalid or corrupt, and it won't be able to load it.

- Each file contains a tag record that has the time and date the file was saved.
- Each record in the file begins with a '+', which marks the record for identification and format when reloaded.
- Each register is listed as one record, in assignment format for easy reading.
- Each region of uploaded memory contains the starting address, length, and binary image. At the end of a line where binary data are displayed, there is a \r\n (carriage-return/newline sequence), which Soft-Scope uses for newline recognition.

# The Registers Window

The **Registers** window allows you to examine, modify, and monitor register values. The contents of the window varies for different members of the x86 family. Figure 5-12 shows the registers for a 80386EX target.




To open the **Registers** window, use the **Registers** command from the **Data** pull-down menu.

# Toolbar Buttons

The contents of a register can be modified or monitored using the toolbar buttons described below:

Modify • Watch

Modify — Change the contents of the register identified by the cursor position. Enter the new value in the dialog box. To access an individual register fields, enter ***`.fieldname`*** in the dialog box. For example, to change the zero flag bit in the **efl** register to 1, enter **`$efl.zf=1`** in the dialog box. To activate from the keyboard press <Enter>.

Watch Place the register identified by the cursor position in the **Watch** window. To activate from the keyboard press <W>.

## Command Line

Enter ***REG*** in the **Command line** dialog box (<Ctrl>+<L>) to open the **Registers** window.

## Accessing Registers When the Target is Running

If you are using an interrupt-driven CSi-Mon monitor and interrupts are not disabled with the option **targ.polling=on**, you can access system registers while your application is running. Question marks (?) identify registers that are not displayed.

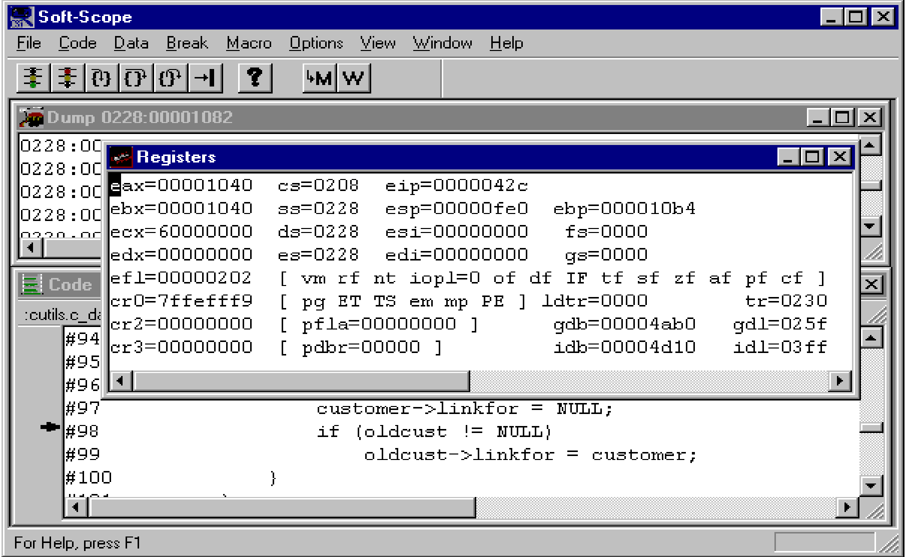



Figure 5-12: Registers window for 80386EX target

# Contents of the Registers Window

The contents of the **Registers** window display varies for different applications. For example, 32-bit 80386 applications support different registers than 16-bit 80286 applications. All register-subfield displays have certain conventions in common:

- Subfields displayed with an equal sign and a value (pri=0) are made up of more than one bit. See your processor reference manual to determine the number of bits.
- Subfields displayed in uppercase letters are in the **on** (1) state.
- Subfields displayed in lowercase letters are in the **off** (0) state.
- Subfields are displayed right-to-left, with the least significant bit (LSB) on the right and the most significant bit (MSB) on the left.
- Subfields that will not change or that do not apply to your processor are not displayed.
- Subfield names are taken from Intel reference manuals.

See the *Data Types, Operators, Registers, and Descriptors* appendix for more information.

# CPU Structures

CPU structures can be viewed and modified using the **Data** window. Figure 5-13 shows a **Data** window containing IDT descriptors for an Intel 80386EX.

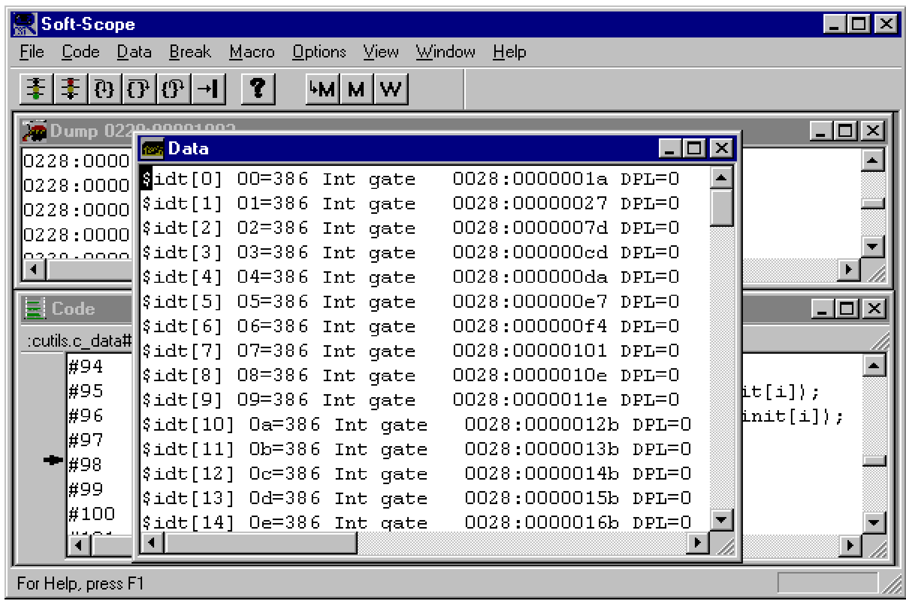
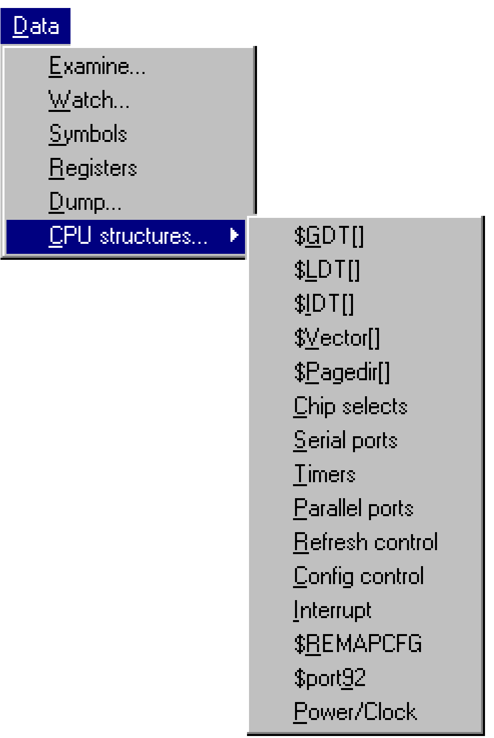


Figure 5-13: IDT descriptors

To view your target's CPU structures in the **Data** window, use the **CPU structures...** command from the **Data** pull-down menu. Select what you want to view from the companion menu. The menu example below shows the structures and peripherals for an Intel 80386EX.

**NOTE:** The contents of the **CPU structures...** will vary depending on your target.



You can access individual descriptors by treating the GDT, IDT, and LDT as if they were arrays of structures. To view the 9th GDT element, select **Data/Examine...** and enter `$gdt[8]` in the **Data** window dialog box.

**NOTE:** Put a dollar sign ($) in front of the descriptor name to convert a symbolic reference into a CPU reference.

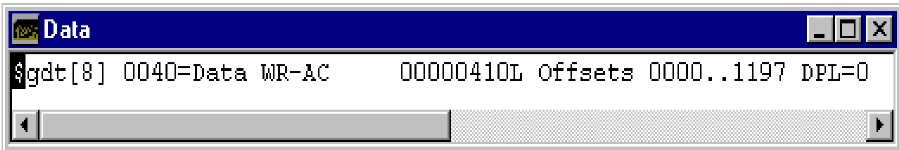



Figure 5-14: Data window in Normal mode

Figure 5-14 shows the descriptor **Data** window in Normal mode. Figure 5-15 shows the same descriptor with the **Data** window in Eval mode. See table 5-7 for a list of descriptor abbreviations used in the **Data** window.

Figure 5-15: Data window in Eval mode

# Command Line

To place a CPU structure in the **Data** window, use the following syntax in the **Command line** dialog box (<Ctrl>+<L>):

EVAL (*memref* | *coderef*) [, (*memref* | *coderef*)]*

**NOTE:** To view page tables, use the Page macro found in the macro file **sswin32.mac**.

Table 5-7: Descriptor abbreviations

| Abbreviation | Meaning |
|---|---|
| WR | Write/read |
| ED | Expand down |
| AC | Access |
| RO | Read only |

## Modifying a Descriptor Element

To modify an element from the **Data** window, complete the following steps:

1. Put the cursor on the element you want to change.
2. Select the **Modify** toolbar button.

   A dialog box containing something like the following appears:

   ```
   $gdt[2]=
   ```

3. Delete the equal sign (=).
4. Enter a period (.), subfield and an equal sign.
5. Enter the new value and press <Enter>.

   Your modified dialog box might look like the following:

   ```
   $gdt[2].limit=4fffh
   ```

**NOTE:** If you enter a new value after the equal sign (=) without the period (.), an error message will appear.

For Intel 80386EX, $SDA is the State Dump Area. $SDA is a structure defined in SMM memory that holds the machine state. $SDA is restored when entering and leaving SMI.

5

# Real-Mode Structures

The peripheral control block (PCB) is supported only for applications running on Intel's 80186/188 microprocessors. You can access this structure by choosing **Data/CPU structures**.../$PCB.

The $PCB structure members and subfields are displayed in a format similar to the format used below. Changing the **$pcb.rr.slave** bit does not change the display of the interrupt controller structure (**$pcb.pic**) immediately; you must exit the **Registers** window and reopen it to see the changed structure. Note that subfields of structure members are enclosed in brackets, and members are not.

If your code changes the contents of the relocation register, use the configuration option **targ.pcb=*16-bit number.*** The *16-bit number* is the value that the $PCB relocation register should contain. See your *Intel 80186/188, 80C186/C188 Hardware Reference Manual* (Intel order #270788-001) for more information.

Table 5-8: Peripheral Control Block

| | | |
|---|---|---|
| **$pcb.rr** | | **Relocation register**<br>**[ et slave ms base=fff ]** |
| **$pcb.timer [0..2]** | | **Timer/counter structure (three element array)** |
| | **count** | **Current value of timer/counter** |
| | **max_a** | **Max count value a** |
| | **max_b** | **Max count value b (not used on timer[2])** |
| | **control** | **Timer control word**<br>**[ en inh int riu mc rtg p ext alt cont ]** |

Table 5-8: Peripheral Control Block *(continued)*

| | | |
|---|---|---|
| **$pcb.pic** | | **Interrupt controller structure (Master mode)** |
| | **irq** | **Interrupt request register**<br>**[ i3 i2 i1 i0 dma1 dma0 tm ]** |
| | **service** | **In-service register**<br>**[ i3 i2 i1 i0 dma1 dma0 tm ]** |
| | **mask** | **Interrupt mask register**<br>**[ i3 i2 i1 i0 dma1 dma0 tm ]** |
| | **primask** | **Interrupt priority mask register**<br>**[ pri=0 ]** |
| | **status** | **Interrupt status register**<br>**[ dhlt tm2 tm1 tm0 ]** |
| | **poll** | **Poll and poll-status register contents**<br>**[ ir s4 s3 s2 s1 s0 ]** |
| | **eoi** | **End of interrupt register**<br>**[ spec s4 s3 s2 s1 s0 ]** |
| | **timers** | **Timer control register**<br>**[ msk pri=0 ]** |
| | **dma0** | **DMA channel 0 control register**<br>**[ msk pri=0 ]** |
| | **dma1** | **DMA channel 1 control register**<br>**[ msk pri=0 ]** |
| | **int0** | **Interrupt 0 control register**<br>**[ sfnm c ltm msk pri=0 ]** |

Table 5-8: Peripheral Control Block *(continued)*

| | | |
|---|---|---|
| | **int1** | **Interrupt 1 control register**<br>**[ sfnm c ltm msk pri=0 ]** |

5

| | | |
|---|---|---|
| | **int2** | **Interrupt 2 control register**<br>**[ ltm msk pri=0 ]** |
| | **int3** | **Interrupt 3 control register**<br>**[ ltm msk pri=0 ]** |
| **$pcb.pic** | | **Interrupt controller structure (Slave mode)** |
| | **irq** | **Interrupt request register**<br>**[ tm2 tm1 dma1 dma0 tm0 ]** |
| | **service** | **In-service register**<br>**[ tm2 tm1 dma1 dma0 tm0 ]** |
| | **mask** | **Interrupt mask register**<br>**[ tm2 tm1 dma1 dma0 tm0 ]** |
| | **primask** | **Interrupt priority mask register**<br>**[ pri=0 ]** |
| | **status** | **Interrupt status register**<br>**[ dhlt tm2 tm1 tm0 ]** |
| | **eoi** | **Specific end of interrupt register**<br>**[ pri=0 ]** |
| | **vector** | **Interrupt vector register** |
| | **timer0** | **Timer 0 control register**<br>**[ msk pri=0 ]** |

Table 5-8: Peripheral Control Block *(continued)*

| | | |
|---|---|---|
| | **dma0** | **DMA channel 0 control register**<br>**[ msk pri=0 ]** |
| | **dma1** | **DMA channel 1 control register**<br>**[ msk pri=0 ]** |
| | **timer1** | **Timer 1 control register**<br>**[ msk pri=0 ]** |

| | | |
|---|---|---|
| | **timer2** | **Timer 2 control register<br>[ msk pri=0 ]** |
| **$pcb.dma[0..2]** | | **DMA controller structure (two-element array)** |
| | **src_ptr** | **Source pointer** |
| | **dst_ptr** | **Destination pointer** |
| | **count** | **Transfer count** |
| | **control** | **DMA control word** |
| | **[ dm dd di sm sd si tc int syn=0 pri tm2 st=0 wd ]** | |
| **$pcb.umcs** | | **Upper memory chip select<br>[ size=0 rdy=0 ]** |
| **$pcb.lmcs** | | **Lower memory chip select<br>[ size=0 rdy=0 ]** |
| **$pcb.pacs** | | **Peripheral address chip select<br>[ base=0 rdy=0 ]** |
| **$pcb.mmcs** | | **Midrange memory chip select<br>[ base=0 rdy=0 ]** |



Table 5-8: Peripheral Control Block *(continued)*

| | |
|---|---|
| **$pcb.mpcs** | **Memory/peripheral chip select<br>[ size=0 ex ms rdy=0 ]** |
| **$pcb.pdcon** | **Power-down control register<br>[ en div=0 ]** |
| **$pcb.edram** | **DRAM control register<br>[en time=0 ]** |
| **$pcb.cdram** | **DRAM control register<br>[ count=0 ]** |
| **$pcb.mdram** | **DRAM control register<br>[ base=0 ]** |

# $VECTOR[ ] Array

$VECTOR is a built-in array that spans the real-mode interrupt vector table. It is an array of 256 32-bit far pointers, starting at linear address 00000000L. This feature is only applicable to virtual-86 and real-mode applications.

To view the vector table in the **Registers** window, use the **CPU structures** command in the **Data** pull-down menu and select **$Vector[ ]**.

# Application Input/Output

Application I/O allows you to receive output from, or send input to, the target application, using the same serial line that Soft-Scope uses to communicate with the target.

To view Application I/O displayed in the **Message** window, you can scroll up or down and left or right using the cursor keys or your mouse. You can also move or enlarge the **Message** window. However, because input goes directly to the target without being displayed in this window, you cannot enter a line of input text and then edit it.

When the **Application I/O** window is open, all cursor movements and key sequences are sent to the target, which makes it impossible to support the window manipulation functions available in other windows.

Press <F10> to toggle the **Application I/O** window open and closed.

# 6. Configuring Soft-Scope

## Chapter Contents

**Overview** ....................................................................................... **6-3**
**Options Window** .............................................................................. **6-3**
Toolbar Buttons ........................................................................ 6-4
Save and Restore Options ........................................................ 6-4
Command Line ........................................................................ 6-5
Figure 6-1: Options window showing default values ........................ 6-5
**Soft-Scope Configuration Options** .................................................... **6-6**
Table 6-1: Soft-Scope configuration options .................................. 6-6
Control Default Number Base .................................................. 6-7
Change Log File Name ............................................................ 6-7
Define Initial Command ........................................................... 6-7
Define Initial Macro File .......................................................... 6-7
Configure Host To Target Communications ..................................... 6-8
Control Screen Refresh Rate ..................................................... 6-8
Control Command Delay ........................................................... 6-8
Define Command ..................................................................... 6-9
Change Log File Size ............................................................. 6-10
Define Path To Application Files .............................................. 6-10
Define Tab Spaces ................................................................. 6-10
Define Case for Symbol Search ................................................ 6-10
Access CPU-specific Data Types ............................................... 6-11
Display LDTR register value .................................................... 6-11
Define Pointer Type Override Display ........................................ 6-12
Specify Integer Data Type Size ................................................ 6-13

6

Specify Floating Point Emulation Parameter .................................... 6-14
Control Memory Caching ............................................................ 6-14
Control Code Memory Cache Flush ............................................. 6-14
Define Host Communication Device ............................................... 6-15
Specify Where To Search For Memory Control Block ................... 6-15
Specify Where To Search for the NULL Device ............................. 6-16
Specify Size of Memory Reads ...................................................... 6-16
Tell Soft-Scope that Interrupts are Disabled .................................. 6-16
Verify Memory Writes .................................................................... 6-17
Specify temporary file location ....................................................... 6-17
Specify the Size of the Trace File .................................................... 6-17
Preserve Trace Data across Applications ........................................ 6-18

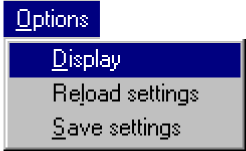
# Overview

Soft-Scope uses a Windows-type initialization file (default = **sswin32.ini**) containing a list of parameters and their values to configure many of its features. Throughout this manual these options are explained in the context of the features they control. However, for clarity and convenience, this chapter contains a description of each of the available options, and how to modify, add, or delete individual options in your **sswin32.ini** file.

# Options Window

The **Options** window displays the current Soft-Scope configuration options based on the Options section of the **sswin32.ini** file. See figure 6-1 for an example of the **Options** window.

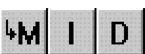
Use the **Display** command from the **Options** pull-down menu to open the **Options** window.




6

# Toolbar Buttons

The following toolbar buttons are used to modify, insert and delete configuration options:




Modify • Insert • Delete

| | |
|---|---|
| Modify | Change the value of an option identified by the cursor position. Enter the new value in the dialog box. To activate from the keyboard press <Enter>. |
| Insert | Insert a new option and value by entering them in the dialog box. To activate from the keyboard press <Ins> or <I>. |
| Delete | Delete the option identified by the cursor position. To activate from the keyboard press <Del> or <D>. |

# Save and Restore Options

To save the current options in the **sswin32.ini** file for the next Soft-Scope session, use the **Save settings** command from the **Options** pull-down menu. To restore the current option settings from the **sswin32.ini** file, use the **Reload settings** command from the **Options** pull-down menu.

# Command Line

You can use the command line to open the **Options** window, save and reload option settings, and modify an option value. Enter the following syntax in the **Command line** dialog box (<Ctrl>+<L>):

| | |
|---|---|
| SET [[TO] optionname] | open Option window |
| SET RELOAD \| SAVE | reload/save options |
| SET optionname=optionvalue | modify an option |

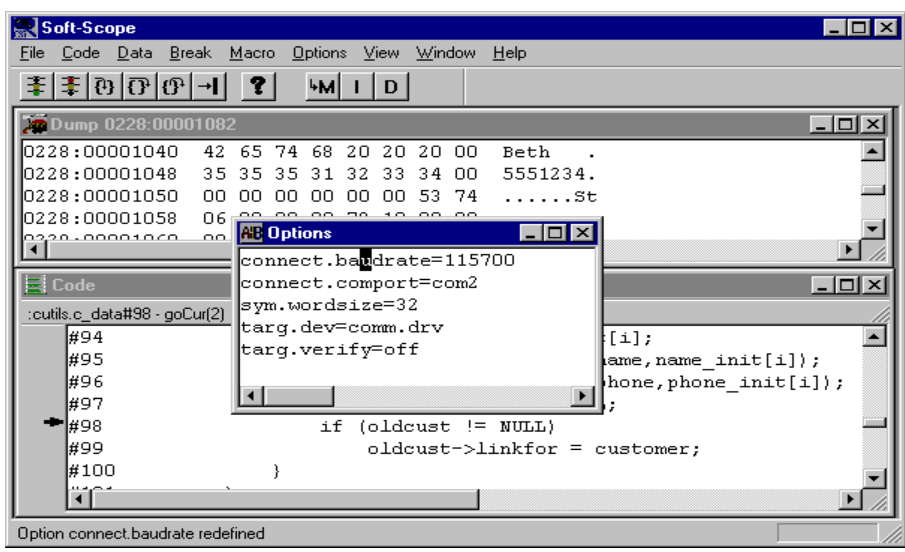



Figure 6-1: Options window showing default values

# Soft-Scope Configuration Options

Table 6-1 contains a list of the configuration options. Each option is discussed below.

**NOTE:** Spaces are not allowed before or after the "=" when assigning a value to an option.

Table 6-1: Soft-Scope configuration options

| | | |
|---|---|---|
| base | log.winsize | targ.dos_mcb_end |
| cmd.file | src.path | targ.dos_mcb_start |
| cmd.initial | src.tab | targ.dos_nul_end |
| cmd.macro | sym.case | targ.dos_nul_start |
| connect.baudrate | sym.cpu | targ.grain |
| connect.comport | sym.ldt | targ.polling |
| exec.refresh | sym.pointer | targ.verify |
| exec.wait | sym.wordsize | tmp.path |
| load.init_command | targ.87emulate | trace.filesize |
| load.init_enable | targ.cache | trace.load |
| load.setup_command | targ.code_cache | |
| load.setup_enable | targ.dev | |

**NOTE:** The [File] and [Layout] sections of the **sswin32.ini** initialization file are modified by Soft-Scope as a result of menu selections or dialog-box options. Do not modify them manually by editing the file.

## Control Default Number Base

**base=10 | 16**

Set **base** to the decimal value of the number base you want to use when inputting numbers (i.e., **base=16** sets it to hexadecimal). The choices are 10 or 16. Default: **base=10**.

## Change Log File Name

**cmd.file=*log filename***

Use this option to specify a log file name. If you change the file name while a debug session is in progress, the contents of the **Log** window are not changed unless you do a window capture (**Window/Capture**) or select **File/View log**. Default: **cmd.file=sswin32.log**.

## Define Initial Command

**cmd.initial=*command***

When Soft-Scope is invoked, it will perform the Command line command specified by *command before* it loads an application. Default: None.

## Define Initial Macro File

**cmd.macro=*macrofilename;...;macrofilename***

This option lets you define the initial macro file(s) that is loaded when Soft-Scope is first invoked. Each *macrofilename* must include a complete path, unless it's located in the current working directory. Default: **cmd.macro=sswin32.mac**.

## Configure Host To Target Communications

**connect.baudrate=*baudrate***

This option is only valid for serial connections. The value of the *baudrate* can be either 300, 1200, 2400, 9600, 19200, 38400, 57600, or 115200.

**connect.comport=*comport***

This is only valid for serial connections. The value of the *comport* can be either "com1", "com2", "com3", or "com4".

## Control Screen Refresh Rate

**exec.refresh=0 | *n***

Set this option equal to the number of seconds (*n*) you want Soft-Scope to wait before refreshing the screen while your application is running, assuming your CSi-Mon monitor is interrupt driven. Zero disables the screen-refresh function. Default: **exec.refresh=0** (seconds).

## Control Command Delay

**exec.wait=*value***

You can control the amount of time (*value* is in seconds) Soft-Scope waits before attempting to process the next command. This is useful when stepping, because if you step over a procedure that takes several seconds to execute, Soft-Scope doesn't attempt to step again until the time specified by this option expires. Press <Esc> to escape from the waiting mode. Default: **exec.wait=3** (seconds).

# Define Command

**load.init_command=*command***

This option allows you to specify a Command line command to execute *after* your application is loaded. Typically, it is used to go to the section of code being debugged, or to set initial breakpoints. Use it to call a macro that does both.

Define *command* using the Command text box found in the **File-Load**, **File-Symbols**, and **File-Restart** dialog boxes. Default: None.

**load.init_enable=on | off**

This option provides a way to toggle **load.init_command** on and off. The option is set using the Command check box found in the **File-Load**, **File-Symbols**, and **File-Restart** dialog boxes. Default: **load.init_enable=off.**

**load.setup_command=*command***

Use this option to specify a Command line command that will execute *before* your application is loaded. You could use this option to invoke a macro that writes test data into memory to help you find uninitialized-variable problems.

Define *command* using the Hardware Setup text box found in the **File-Load** and **File-Restart** dialog boxes. Default: None.

**load.setup_enable=on | off**

This option provides a way to toggle **load.setup_command** on and off. Set the option using the Hardware Setup check box found in the **File-Load** and **File-Restart** dialog boxes. Default: **load.setup_enable=off**.

# Change Log File Size

**log.winsize=*n***

The option changes the number of lines stored in the **Log** window temporary file. The value of *n* can range from 16 to 1024. Default: **log.winsize=500**.

# Define Path To Application Files

**src.path=*d:\subdir\...\subdir***.asm*;...;d:\subdir\...\subdir***.c**

Use this option to define a path to your application. If you specify a path when you load your application, Soft-Scope searches in the specified path before searching the one defined by **src.path**. Default: None.

# Define Tab Spaces

**src.tab=*n***

This option defines the number of blank characters that are used when expanding a tab character in the **Code** window. Default: Dependent on source language.

# Define Case for Symbol Search

**sym.case=on | off**

When Soft-Scope searches the symbol table it will match the case of the symbol when **sym.case=on**. Default: **sym.case=off**.

## Access CPU-specific Data Types

**sym.cpu=*cpu***

This option allows you to access different CPU-defined types. For example, if your application is for a 186 but you want to use 386EX specific register types in type overrides, use **sym.cpu=386EX**.

This option also changes the way code is disassembled. The default is the actual CPU in your target. Set this option to any of the following values:

| | | | | |
|---|---|---|---|---|
| **Pentium** | **486SX** | **486DX** | **486** | **386EX** |
| **386SX** | **386DX** | **386** | **376** | **286** |
| **188EA** | **186EA** | **188EB** | **186EB** | **188EC** |
| **186EC** | **188XL** | **186XL** | **C188** | **C186** |
| **V20** | **V30** | **V40** | **V50** | **188** |
| **186** | **88** | **86** | **Am386Elan** | |

## Display LDTR register value

**sym.ldt=on | off**

Soft-Scope has the ability to reference code and symbols whose addresses don't use the current LDTR. This option allows you to see which LDT is used. If **sym.ldt**=**on**, the address display includes the LDT selector. If this option is **off**, the LDT is not shown. Default: **sym.ldt**=**off.**

# Define Pointer Type Override Display

**sym.pointer=*value***

When you use **pointer** as a type override, Soft-Scope can interpret it in four different ways. To control this interpretation, set *value* to one of the following:

**far16** selector with 16-bit offset

**far32** selector with 32-bit offset

**near16** 16-bit offset only

**near32** 32-bit offset only

Remember that a far pointer is a 16-bit selector and a 16- or 32-bit offset. A near pointer has only an offset with a default selector. Default: **sym.pointer=far16**.

# Specify Integer Data Type Size

**sym.wordsize=16 | 32**

This option defines the size of the C integer data type **int**. If you set this option equal to **16**, you can select from the following display modes:

| | |
|---|---|
| Byte | Select byte-width (8-bits) hex display with ASCII on the right |
| Word | Select word-width (16-bits) hex display with ASCII on the right |
| Dword | Select dword-width (32-bits) hex display with ASCII on the right |

If you set this option equal to **32,** you can select from the following display modes:

| | |
|---|---|
| Byte | Select byte-width (8 bits) hex display with ASCII on the right |
| HWord | Select half-word-width (16-bits) hex display with ASCII on the right |
| Word | Select word-width (32-bits) hex display with ASCII on the right |

Default: 16

# Specify Floating Point Emulation Parameter

**targ.87emulate=*value***

This option is discussed in the *Intel Floating Point Emulation* appendix. The *value* you specify here is the value of the first interrupt that is used by the emulation library that lets Soft-Scope disassemble emulated instructions as floating-point instructions. Default: None.

# Control Memory Caching

**targ.cache=on | off**

Set **targ.cache=on** to enable Soft-Scope's normal caching of previously read memory. You might find this useful if you are actually reading from a memory-mapped I/O device instead of memory, or if some other device, e.g., a DMA device, is writing to memory. Default: **targ.cache=on**.

# Control Code Memory Cache Flush

**targ.code_cache=off | on**

If you set **targ.code_cache**=**on**, Soft-Scope does not flush memory areas that correspond to code when it executes your application. This provides an increase in performance on some machines. Default: **targ.code_cache=off.**

# Define Host Communication Device

**targ.dev=*devicename***

This option specifies the name of the device that Soft-Scope uses to communicate with the CSi-Mon monitor on the target. To change *devicename*, use the **Options** window. This option cannot be changed while Soft-Scope is running. Default: **targ.dev=comm.drv**.

# Specify Where To Search For Memory Control Block

**targ.dos_mcb_start=*address***
**targ.dos_mcb_end=*address***

These options are used to define an area in memory where Soft-Scope can search for the first DOS MCB (Memory Control Block) header file. See the appendix, *Debugging .exe Executable Files*, for more information. Defaults: **targ.dos_mcb_start=00000701L**; **targ.dos_mcb_start=000106ffL**.

# Specify Where To Search for the NULL Device

**targ.dos_nul_start=***address*
**targ.dos_nul_end=***address*

When debugging DOS device drivers, Soft-Scope must search target memory for the NULL device, which begins the device-driver chain in memory. However, the NULL device and its location in memory are not documented. If Soft-Scope cannot find the NULL device in the default range, use these options to define a new search range. See appendix, *Debugging .exe Executable Files*, for more information. Defaults: **targ.dos_nul_start=00000701L**; **targ.dos_nul_end=000106ffL**.

# Specify Size of Memory Reads

**targ.grain=1 | 2 | 4**

If your target is configured to read memory 2 or 4 bytes at a time, you can define the memory access size with this configuration option. Set this option to allow memory accesses of 1, 2, or 4 bytes. Default: **targ.grain=1**.

# Tell Soft-Scope that Interrupts are Disabled

**targ.polling=on | off**

Soft-Scope can only stop an interrupt-driven monitor when interrupts are enabled. When interrupts are disabled, Soft-Scope continues to assume you have an interrupt-driven monitor, and receiver time-out messages may result. Set this option to **on** to eliminate the receiver time-out messages. Default: None.

## Verify Memory Writes

**targ.verify=off | on**

When set to **on**, this option causes Soft-Scope to perform read-after-write verification of all memory writes. When set to **off**, no verification is performed. Default: **targ.verify=on**.

## Specify temporary file location

**tmp.path=*d:\subdir\...\subdir***

The first time you debug an application, Soft-Scope creates a temporary file called ***application.*tmp**. It is used to store initialization information needed to load the application.

The next time you invoke Soft-Scope and load the application, it searches the path defined with this option for ***application*.tmp**. If it finds it, and the application has not been modified, Soft-Scope uses it to load the application. Default: Current directory.



## Specify the Size of the Trace File

**trace.filesize=16K | ... | 1024K**

This option controls the size of the temporary file where trace information is stored. Default: **trace.filesize=128K**.

# Preserve Trace Data across Applications

**trace.load=off | on**

If you set this option to **on**, the trace buffer shows trace information across multiple loads. Use **off** to cause the trace buffer to be flushed each time you load an application. Default: **trace.load=off**.

**NOTE:** During execution, Soft-Scope creates several temporary files. The names of these files are determined by your host operating system. The location of these files are determined by the **temp** environment variable.

# 7. Creating and Using Soft-Scope Macros

## Chapter Contents

**Overview** .......... **7-3**
**Creating a Macro** .......... **7-3**
Compiled Macro Files .......... 7-4
Built-in CPU Variables .......... 7-5
**Macros Window** .......... **7-6**
Loading a Macro File .......... 7-6
Toolbar Buttons .......... 7-6
Figure 7-1: Macros window .......... 7-7
Command Line .......... 7-7
Example Use of cmd.macro and load.init_command .......... 7-8
Identify Macros in the Macros Window .......... 7-9
**Macro Parameters** .......... **7-10**
Optional Parameters .......... 7-10
Integer Type .......... 7-10
LITERAL Parameter .......... 7-11
TEXT Parameter .......... 7-12
EXPRESSION Parameter .......... 7-12
REFERENCE Parameter .......... 7-12
ADDRESS Parameter .......... 7-13
LINE Parameter .......... 7-13
MODULE and PROCEDURE Types .......... 7-13
**Local Variables** .......... **7-14**
Declaring Local Variables .......... 7-14
Defining One-dimensional Arrays .......... 7-15

7

Assigning Numeric Values to Arrays ............................................. 7-15
Assigning Pointer Values from Your Application ............................. 7-16
**Macro Statements ................................................................. 7-17**
ABORT ...................................................................................... 7-17
BREAK ...................................................................................... 7-17
IF, IF...ELSE ............................................................................. 7-17
RESPOND .................................................................................. 7-18
RETURN .................................................................................... 7-18
WHILE ....................................................................................... 7-18
MACRO SUSPEND .................................................................. 7-19
MACRO RESUME ................................................................... 7-19
Custom Commands with an Extended Monitor ............................... 7-20
**Manipulating Windows from Macros ............................................ 7-22**
WMOVE ..................................................................................... 7-23
WRESIZE .................................................................................. 7-23
WFUNCTION ........................................................................... 7-23
Examples ...................................................................................... 7-24
**Macro Print Function .............................................................. 7-25**
PRINT ........................................................................................ 7-25
Conversion Specifiers .................................................................... 7-25
Table 7-1: Conversion specifiers ................................................ 7-26
$ Parameter Prefix in Control Strings ............................................. 7-27
Escape Sequences ......................................................................... 7-27
Directed Output from Macros ....................................................... 7-27
Using Field-width Specifiers with PRINT or WPRINTF .................. 7-28
Specifying the Leading Zero Flag .................................................. 7-28

# Overview

Soft-Scope's macro facility lets you create your own macros that can:

- Rename a Soft-Scope command
- Create pseudo-command files of commands
- Create new Soft-Scope pseudo-commands

Look in the directory where you installed Soft-Scope for the example macro file **sswin32.mac**. It contains several macros that you can examine or modify to meet your special needs.

# Creating a Macro

Use an ASCII text editor to create macros. Macro source files have the following characteristics:

- You can declare an unlimited number of macros in a macro source file.
- Macro files must use the file extension **.mac**.
- Each declaration must look similar to a C-function declaration.
- The keyword ***MACRO*** should be used where the C function return type would be.
- You can use control statements and function calls within the declaration.
- The use of semicolons after source lines is optional, except after the macro header line. Placing a semicolon after the first line of the macro name and parameters results in an error message.
- The syntax shown below defines a Soft-Scope macro:

```
MACRO macroname [parameter_list]{statements}
```

7

Any of the following can be Soft-Scope macro statements:

- Any Soft-Scope command discussed in the chapter *Soft-Scope Basics* in the *Commands and Command Line* section.
- A C-type expression, including mathematical expressions, Soft-Scope type overrides, and Soft-Scope functions.
- A macro control statement.
- A macro function.
- A compound statement that is enclosed in braces. Unlike the compound statement in C, a Soft-Scope compound statement may not contain local-variable declarations.

The following example macro makes use of Soft-Scope type overrides to substitute an opcode for every byte of a source-code line:

```
macro arr_chg ($line, $value)
{
    byte at &#$line
length(sizeof#$line)=0x$value;
}
```

# Compiled Macro Files

The first time you load a macro file, ***filename*.mac**, the file is compiled and a ***filename*.mob** file is created that contains the compiled code. The next time you load the macro file, Soft-Scope looks for the corresponding **.mob** file. If it exists and if the original **.mac** file has not been edited since its creation, Soft-Scope loads the **.mob** file. Compiled **.mob** files load and execute faster than **.mac** files.

If the **.mob** file does not exist or the original **.mac** file has been edited since the compiled version was created, Soft-Scope recompiles the original and creates a new **.mob** file.

# Built-in CPU Variables

Soft-Scope contains a set of built-in CPU variables that you can use to return information to macros. For example, you can determine if your target is stopped or running, as in the following:

```
go;         /*   Stop  before   */
while  (!  $stopped);   /*   opening  Calls  window  */
calls;               /*   Open  Calls  window  */
```

The following is a list of built-in CPU variables and their possible values:

$CPU This is the processor type as reported by the CSi-Mon monitor. Possible values are 80586, 80486, 80386, etc.

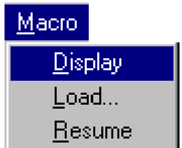
$NPX Use this to determine if a coprocessor is present. If a coprocessor is present, the value of $NPX is the coprocessor's name. If it is not present, the value is 0.

$STOPPED Reports the execution state of your application. If stopped, the value is 1. If running, the value is 0.

7

# Macros Window

The **Macros** window shows which macros are currently loaded. Use the **Display** command from the **Macro** pull-down menu to open the window. Figure 7-1 shows an example of a **Macros** window.




# Loading a Macro File

To load and compile a macro source file, use the **Load** command from the **Macro** pull-down menu.

# Toolbar Buttons

From the **Macros** window you can run or delete loaded macros using the toolbar buttons described below:

Run • Delete

Run — Runs the macro identified by the cursor position. To activate from the keyboard press <Enter> or <R>.

Delete — Deletes the macro identified by the cursor position from the current Soft-Scope session. The macro source file is not erased. To activate from the keyboard press <Del> or <D>.

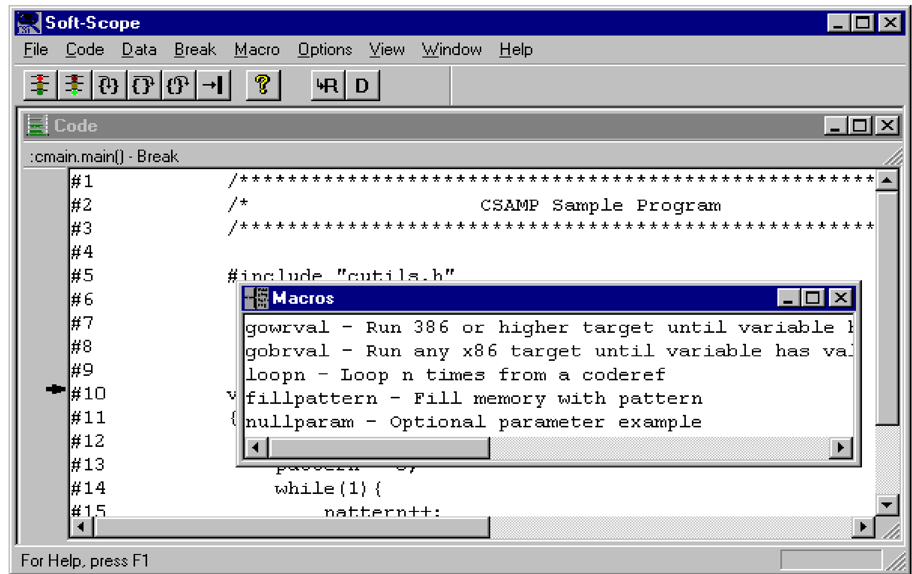



Figure 7-1: Macros window

# Command Line

Macros can also be loaded, displayed, and deleted using the following syntax in the **Command line** dialog box (<Ctrl>+<L>):

MACRO [LIST] [TO] [*macroname*]
MACRO LOAD *filename*
MACRO DELETE [*macroname*]

To run a macro that is already loaded, enter the macro name and applicable parameters in the dialog box. For example, the following would run the macro **test** discussed later in this chapter:

Command: `test 2 #26 "Go to line 26 two times"`

**NOTE:** Loaded macros are stored inside Soft-Scope. Changes to their source do not affect the macros until the source file is loaded (and compiled) again.

## Example Use of *cmd.macro* and *load.init_command*

A macro can be loaded and executed as part of the Soft-Scope invocation by using the **cmd.macro** and **load.init_command** configuration options. To do this you will need to set these configuration options in the **sswin32.ini** file as follows:

**cmd.macro=sample.mac** Loads the **sample.mac** macro source file.

**load.init_command=setbreak** Executes the **setbreak** macro found in **sample.mac**. Because macros are a subset of Soft-Scope commands, it isn't necessary to preface **setbreak** with a command such as ***RUN*** or ***LOAD***.

The example below executes a macro that sets a breakpoint, executes to it, and evaluates the symbol *s1* in the **Data** window.

```
macro setbreak ( )
{
      br :cutils.c_data#98;
      go;
      eval :cutils.strcpy.s1;
}
```

# Identify Macros in the Macros Window

Use a text string, placed inside quotes between a macro name and the first parenthesis, to identify or otherwise customize the macro name in the **Macros** window.

```
"setbreak   breaks at line number 98"
```

Quotation marks with no text between them tell Soft-Scope not to display that macro in the **Macros** window. This is helpful if you have a macro that exists only to provide data to another macro.

7

# Macro Parameters

In the macro source file, a parameter is recognized by a dollar sign ($) preceding the parameter name:

```
$length
```

You can specify macro parameters as any type (except arrays) listed in table A-1, "Data types for use in type overrides," in appendix A.

# Optional Parameters

The number of parameters passed to a macro must match the number of parameters that the macro expects, unless the parameters are specified as optional. The ***OPT*** keyword defines all parameters that follow it in the list as optional. In the following example, *`$start`* and *`$length`* are both optional.

```
macro fill (reference $memref,opt int $start,int
$length)
```

# Integer Type

You can also use ***HEX*** and ***DEC*** as keywords. ***HEX*** specifies an integer as a hex number, so you don't have to use the prefix 0x, or the suffix H when entering parameter values. A parameter with the ***DEC*** keyword won't accept a hex value:

```
macro test (hex int $memref, dec int $length)
```

# LITERAL Parameter

The LITERAL parameter type causes direct replacement during macro execution. When you enter values, the literal strings are parsed and inserted in the appropriate places.

LITERAL parameters can be used for most common macro parameters, and are most useful when a parameter need not be evaluated at the beginning of the macro.

For example, the macro below uses Soft-Scope type overrides to substitute an opcode for every byte of a source-code line:

```
macro src_chg (literal $line, literal $value)
{
   byte at &#$line length(sizeof#$line)=0x$value;
}
```

If you ran this macro by selecting it from the **Macros** window, you could enter something like the following when prompted:

```
23 90
```

The numeric value 23 would be passed as the value for *$line*, and the hex number 90 would be passed as the value for *$value*. Keep in mind that a LITERAL is a string of nonblank characters. The following would be interpreted as a single LITERAL parameter value:

```
2390
```

## TEXT Parameter

TEXT parameters are parsed and turned into string constants. No processing of the string takes place (e.g., \n is not converted to 0xa).

The following example prints a string in a window:

```
macro printstr (
      literal $win,
      text $outstr)
{
      wprintf ("$win", "%s\n", $outstr)
}
```

For example, enter the values when prompted as shown below:

printstr: `log "macro succeeded"`

## EXPRESSION Parameter

This type allows any valid Soft-Scope expression to be input as a parameter. The advantage over the 'literal' parameter type is that the expression is evaluated only once at the time the macro is invoked. Any syntax errors within the expression are trapped and reported before calling the macros.

## REFERENCE Parameter

A reference parameter is similar to an expression parameter but has an additional requirement that it must be assignable. For example, you can assign the register $ax a value (for example, $ax=00002fe8), but you cannot assign a constant a value (for example, 5=3). As with expression parameters, reference parameters are evaluated only once when the macro is invoked.

## ADDRESS Parameter

ADDRESS types have all of the characteristics of reference types, except they must have a port or memory address. For example, you can use target variable names as ADDRESS types.

## LINE Parameter

LINE types have all of the characteristics of reference types, except they must be line numbers. Any number you use is interpreted as a line number.

## MODULE and PROCEDURE Types

MODULE types must be module names. PROCEDURE types must be procedure names.

# Local Variables

Local variables may be declared as any type found in table A-1, “Data types for use in type overrides,” in appendix A.

Local variables may be used anywhere you would use a parameter within the body of a macro.

# Declaring Local Variables

Variable names may contain up to 40 characters, and the number of variables that can be declared is limited only by available memory. Once a macro is terminated, the value of the variable is lost.

Variables must be declared by inserting the keyword ***AUTO*** and a type before the variable name, immediately after the opening brace. The example below declares `$counter` as a local variable:

```
macro test (int            $value,
         reference  $coderef,
         text $str)
{
  auto int        $counter

  $counter = 0
  print (“%s”, $str)
  while (1)     {
      go $coderef
      $counter++
      if ($counter == $value)
         break
  }
}
```

**NOTE:** Because the dollar sign ($) is used to designate macro variables and is also used to specify the names of CPU structures, care should be taken when naming variables to avoid conflicts.

## Defining One-dimensional Arrays

In addition, you can define variables as one-dimensional arrays. To define an array, use an index in the declaration:

```
auto char $str[5]
```

Then simply assign a value to the character array in your macro using the equal sign and quotation marks. You can assign a character string to any array, regardless of the array's type:

```
$str="Hello"
```

## Assigning Numeric Values to Arrays

To initialize an entire array to zero, use the array name without an index. For example, the following sets all the elements of $arr[8] to 0:

```
$arr=0
```

However, to set each element of the integer array $arr[8] to a different value, you must assign the values individually:

```
$arr[0]=1
$arr[1]=2
$arr[2]=3 ...
```

You can use two declarations to define a pointer in a macro:

```
auto char *$ptr
auto far32 $ptr
```

7

# Assigning Pointer Values from Your Application

The only restriction is that you must assign the pointer an address defined in your application:

```
$ptr=name_init[0]
```

# Macro Statements

The Soft-Scope macro language supports the following control statements:

## ABORT

```
ABORT [("format" [,optional parameters])]
```

ABORT returns execution to the command line. Typically, it is used when a severe error occurs in a macro and you want to stop execution. An aborted macro cannot be resumed with **Macro/Resume.** ABORT also lets you print a comment to the screen when a macro is aborted.

```
abort ("Macro halted—value out of bounds,%d",
$value)
```

## BREAK

BREAK functions the same as the C break: it exits the current block.

## IF, IF...ELSE

```
IF (condition) statement
IF (condition) statement ELSE statement
```

The IF/ELSE control statement functions just like its C counterpart. The *condition* can be any Soft-Scope expression that evaluates to a number. If it evaluates to any number except 0, the statements after the IF are executed. If it evaluates to 0, the statements after the ELSE are executed. Enclose multiple statements in braces.

7

# RESPOND

`RESPOND ("`*`resp`*`")`

RESPOND allows you to respond to Soft-Scope questions from within a macro, so Soft-Scope doesn't need to prompt you. The following example responds to a query to append or overwrite the log file:

```
respond ("a")
```

Responses are limited to a single character, and should be placed early in the macro, before the response is needed.

# RETURN

RETURN functions like its C counterpart, returning execution to the place from which its containing block was called.

# WHILE

`WHILE (`*`condition`*`)` *`statement`*

WHILE parallels its C-language counterpart. The *condition* can be any Soft-Scope expression that evaluates to a number. If *condition* evaluates to any number except 0, *statement,* which can be a compound statement in braces, is executed. If it evaluates to 0, control passes to the next command after the loop. To create an endless loop, simply make the condition 1 (e.g., WHILE (1) {...} ).

## MACRO SUSPEND

You can suspend a macro that is running by using the MACRO SUSPEND command inside the macro.

This is handy if you want to track and possibly change the value of an application variable at different stages of macro execution, or, in combination with an IF statement, when an error condition occurs. You cannot change the value of a local variable while a macro is suspended.

This command is not available in the **Command line** dialog box (<Ctrl>+<L>) or the pull-down menus.

## MACRO RESUME

Resume a suspended macro using the **Resume** command from the **Macro** pull-down menu.

You can also resume a macro from the **Command line** dialog box (<Ctrl>+<L>) using the following syntax:

```
MACRO RESUME
```

7

# Custom Commands with an Extended Monitor

The _USER_ macro command enables direct communication with the monitor by allowing the user to send predefined and user-defined commands to it.

Output responses to any _USER_ command appear in the **Message** window. Syntax for the macro command is as follows:

_USER_ MONITOR "*Command_String*"
_USER_ MONHOLD "*Command_String*"

| | |
|---|---|
| MONITOR | Output from the monitor in response to "*Command_String*" is displayed in the **Message** window, and then the display returns immediately to the previously active Soft-Scope window. |
| MONHOLD | Output appears in the **Message** window and the window persists until you press <F9>. |

Note that the difference between the MONITOR and MONHOLD versions of this command is that MONHOLD causes the output window to remain open so you can study monitor output.

An example of how this might be applied would be to use the MONITOR version for the first several commands in a series and the MONHOLD version for the last monitor command. That way, the macro would execute until all the commands were completed before stopping to show you the results.

The "*Command_String*" can be any monitor command defined in the default configuration of the monitor or any user-defined monitor-extension command. The "*Command_String*" can also contain the following parameter specifiers:

| | |
|---|---|
| %c, %d, ..., %% | Any of the conversion specifiers listed in table 7-1, "Conversion specifiers," can be used with the exception of %p, which cannot be used with default monitor commands in _USER_. |

| | |
|---|---|
| $*parameter_name* | Passes a literal value to a macro. Can be any local variable, application variable, or macro parameter. |

The substitution specifier "%" allows you to generate specifier parameter values symbolically. For example, you can use the functions SELECTOROF and OFFSETOF inside the macro.

The following macro returns the address where the monitor is located:

```
macro whereis_carmen_csimonitor ()
{
    message
     wprintf(message, "\nThis is where the monitor
is:\n"
     _user_  monhold "E0"
}
```

7

# Manipulating Windows from Macros

You can use the following commands to manipulate windows and the cursor from within macros (the double-quotation marks are required):

```
WFUNCTION ("window_id", "key_sequence")
WMOVE ("window_id", newr, newc)
WRESIZE ("window_id", width, height)
```

where,

| | |
|---|---|
| *window_id* | Is a window name. Use any of the following names; breakpoints, calls, code, data, dump, log, macros, options, registers, symbols, task, trace, watch. |
| *newr* | Is a decimal number that specifies a row on the screen. (0-23, top to bottom). |
| *newc* | Is a decimal number that specifies a column on the screen. (0-79, left to right). |
| *width* | Is a decimal number that specifies the width of a window in character columns, not greater than 80. |
| *height* | Is a decimal number that specifies the height of a window in character rows, not greater than 24. |
| *key_sequence* | Is any sequence of keys, up to 80 characters in length, that are valid in the window specified. Window buttons, accelerator keys, and counts are valid. Use the following names for the indicated keys. Note that the left and right braces are part of the key name, and that all letters in key names are lowercase. |

| | |
|---|---|
| Enter | {enter} \| ~ \| ^M, where (^) represents <Ctrl> |
| Up / Down | {up} / {down} |
| Left / Right | {left} / {right} |

| | |
|---|---|
| Page up | {pgup} |
| Page down | {pgdn} |
| Home /End | {home} / {end} |
| Left brace | {{} |
| Right brace | {}} |

If you specify a parameter that is outside the screen or window size, the parameter will be truncated when a border is reached.

## WMOVE

WMOVE places the upper, left-hand corner of the window at *newr*, *newc*.

## WRESIZE

WRESIZE changes the size of the window relative to the upper, left-hand corner of the window, which remains in a constant position.

Any changes made to windows that are not open change the defaults, so when you do open the window the new size and location are used.

## WFUNCTION

If the window specified in a WFUNCTION command is not open, it attempts to perform the function specified by the key sequence on the current window. To ensure that the window specified is open, simply place the command that opens it before the WFUNCTION in your macro. For example, the ***DUMP*** command opens the **Dump** window.

# Examples

| | |
|---|---|
| **`wfunction("trace", "")`** | Makes the specified window the current window. |
| **`wfunction("data", "^X")`** | Closes the specified window. |

Use WFUNCTION to specify information for dialog boxes. For example, the following changes the code window mode to Assembly:

```
wfunction("code", "MA")
```

The following example uses a count to move the cursor down 10 lines:

```
wfunction ("options", "10{down}")
```

The following example uses the brace keys to modify the first configuration option:

```
wfunction ("options", "{enter};{{};{}};{enter}")
```

# Macro Print Function

## PRINT

```
PRINT ("control_string" [,optional parameters])
```

This command functions much like C's formatted print command, allowing you to print formatted output to the **Message** window. Unless there are conversion specifiers embedded within the control string, everything inside the quotes is printed to the screen.

# Conversion Specifiers

Conversion specifiers have the following format:

```
%CHAR
```

CHAR can be any one of the conversion characters listed in table 7-1.

If you use a conversion specifier in a control string, Soft-Scope expects a substitute value to be in the parameter list following the string. Just as in C, you should separate parameters with commas.

The first parameter in the list is substituted for the first conversion specifier in the control string, the second parameter for the second specifier, and so on. If the parameter and the conversion specifier have different types, Soft-Scope applies a type override to the parameter.

Table 7-1: Conversion specifiers

| Conversion Character | Description |
|---|---|
| c | Specifies a character. One byte will be displayed for this operator. If the character is non-printable, its value will be displayed in hex. |
| d | Specifies a signed integer. Leading zeros are suppressed. |
| f | Specifies a double (64-bit double-precision floating-point number). Leading zeros are suppressed. |
| p | Specifies an address: logical, linear, or physical. Suffix L or P, specifying a linear or a physical address, is not removed. The colon is not removed from logical addresses. |
| s | Specifies a character string. The entire character string will be displayed, with non-printable characters displayed in hex. |
| u | Specifies an unsigned integer. Leading zeros are suppressed. |
| x | Specifies an unsigned hexadecimal integer. Leading zeros are suppressed. |
| % | Escapes the percent sign that starts the format specifier. |

# $ Parameter Prefix in Control Strings

When "$" occurs as a parameter prefix in a control string, a direct substitution is performed. If this is not what you want, the best solution is to rename the conflicting parameter.

# Escape Sequences

The control string can also contain C-type escape sequences, which are listed in table 5-4.

When specifying an octal number, use one, two, or three octal digits (0-7). An error occurs if the octal number is greater than 377 (255 in decimal).

You can specify a hex number that contains one or two hex digits by using the "\x" escape sequence. Errors are generated if the first number after the x is not a hex digit, (i.e., 0-9, a-f or A-F).

# Directed Output from Macros

To direct formatted output from macros to a selected destination, use the following syntax:

WPRINTF ("*destination*", "*control string*"[,*optional parameters*])

All of the following destinations are Soft-Scope windows except Status, which is the Status line at the bottom of the Soft-Scope display. Place quotation marks around *destination*:

Log Message Status Trace

# Using Field-width Specifiers with PRINT or WPRINTF

Use decimal integer constants as field-width specifiers in control strings as part of the format specification. For example, the following specifies a field width of 8 characters:

```
wprintf("message", "%8d", $value)
```

If `$value` was equal to 45522601, the following would be printed in the **Message** window:

```
45522601
     601
```

If `$value` does not contain eight characters, padding is inserted in front of the needed spaces. If `$value` was equal to 601, the above example shows how it would be displayed.

The first five spaces are left blank as padding. If `$value` is more than eight characters, the field width is expanded to display as many characters as needed.

You can also specify field width, as in C, with the asterisk "*", which causes a type *int* argument to be substituted for the field width. In the following example, the field width is 5:

```
wprintf("trace", "unit=%*d", 5, 12)
```

# Specifying the Leading Zero Flag

A leading minimum field-width specifier as defined above is required to specify the leading zero flag. Use this only with integer-type conversion specifiers, that is, %d, %u, and %x:

```
print( "%04x", 3)
```

Prints the following:

```
0003
```

# 8. Tools that Soft-Scope Supports

This chapter provides information to help you insure that your application is fully compatible with Soft-Scope. If you need to learn more about the tools that Soft-Scope supports, consult the appropriate development-tool reference guide.

## Chapter Contents

**Tool Summary** .......... **8-2**
Table 8-1: Supported tools .......... 8-2
Sample Files .......... 8-4
**Linking Your Application** .......... **8-5**
CSi-Link™ .......... 8-5
**Generating Symbolic Information** .......... **8-6**
SSBUG .......... 8-6
**Tool Directives** .......... **8-7**
Borland .......... 8-7
Intel .......... 8-7
MetaWare .......... 8-10
Microsoft .......... 8-10
Phar Lap .......... 8-11
Watcom .......... 8-12

8

# Tool Summary

Below is a list of tools that can be used to build applications that can be debugged using Soft-Scope. New versions of these tools are constantly being released. See the **readme.wri** file on distribution disk one or in the directory where you installed Soft-Scope, for the most current list.

**NOTE:** For information on Microsoft, Borland, Watcom, and MetaWare compiler and assembler controls, see the *CSi-Link User's Guide*.

Table 8-1: Supported tools

| Supported Tools | 16-bit Real Mode | 16-bit Protected Mode | 32-bit Protected Mode Flat | 32-bit Protected Mode Segmented |
|---|---|---|---|---|
| Borland C++ | X | X | X | |
| Borland TASM | X | X | X | X |
| CSi-Link | X | X | X | X |
| Intel ASM86 | X | | | |
| Intel ASM286 | | X | | |
| Intel ASM386 | | | X | X |
| Intel BND286-BLD286 | | X | | |
| Intel BND386-BLD386 | | | X | X |
| Intel iC-86 | X | | | |
| Intel iC-286 | | X | | |
| Intel iC-386 | | | X | X |

Table 8-1: Supported tools *(continued)*

| **Supported Tools** | **16-bit Real Mode** | **16-bit Protected Mode** | **32-bit Protected Mode Flat** | **32-bit Protected Mode Segmented** |
|---|---|---|---|---|
| Intel LINK86/LOC86 | X | | | |
| Intel PLM86 | X | | | |
| Intel PLM286 | | X | | |
| Intel PLM386 | | | X | |
| MetaWare High C/C++ | | X | | X |
| Microsoft MASM | X | X | X | X |
| Microsoft Visual C/C++ version 1x | X | X | | |
| Microsoft Visual C/C++ version 2x or greater | | | X | |
| Phar Lap 386/ASM | X | X | X | X |
| Phar Lap LinkLoc Linker | X | X | X | X |
| Watcom C/C++ | X | X | X | X |
| Watcom WASM | X | X | X | X |

# Sample Files

We have included real- and protected-mode sample applications for each of the supported tools in the **/samp** subdirectory. Within **/samp** is a series of subdirectories for each compiler. The subdirectory name includes the compiler vendor, number of bits, and mode. For example, **msc16pf** contains a Microsoft C/C++, 16-bit, protected-mode, flat-model sample. The Borland C/C++ compiler appears as **bcc**, Watcom C/C++ compiler as **wcc**, and MetaWare's High C compiler as **hc**.

The **mapi32p** subdirectory includes a sample that demonstrates application I/O using Soft-Scope's **Message** window. The subdirectories **bccexe** and **mscexe** include a sample in the form of a DOS executable which can be debugged with Soft-Scope.

Included with each sample is its source, makefile, CSi-Link command file, map file, listing and debug file. The samples were run on Intel 386EX, AMD 186EM/ES evaluation boards and a target PC.

# Linking Your Application

## CSi-Link™

We created the CSi-Link linker/locator so you could use the popular C/C++ compilers from Microsoft, Borland, Watcom, and MetaWare to develop an embedded application. CSi-Link creates an **.abs** debug file that Soft-Scope uses to download your application to your target board or a **.hex** or **.bin** binary file so you can burn it into ROM.

CSi-Link creates an absolute image of your application by linking your object and library files and locating the segments, classes and groups that make up an x86 program. It builds 16- and 32-bit protected mode CPU structures, including the GDT, IDT, LDT, gates, page tables, and TSSs. CSi-Link supports multiple-mode or mixed-mode applications.

For more information, see the *CSi-Link User's Guide*.

# Generating Symbolic Information

## SSBUG

The SSBUG utility makes it possible to use Soft-Scope for debugging real-mode applications built with CSi-Link, Phar Lap's LinkLoc, Paradigm's LOCATE and Intel's LINK86/LOC86. SSBUG produces a **.bug** file that includes symbolic information for Soft-Scope.

Debugging an **.exe** application running on a target PC with Soft-Scope is also made possible by SSBUG.

To invoke SSBUG from the DOS prompt, enter the following:

```
ssbug  filename.abs
```

# Tool Directives

This section lists the tool directives to use when building an application for debugging.

## Borland

Consult the *CSi-Link User's Guide* for tool directive information.

## Intel

**NOTE:** Since Intel tools are no longer on the market, our support is on an "as is" basis.

### ASM86, ASM286 and ASM386

Use with Intel's linker/locator or builder/binder, whichever is appropriate.

| Use these controls: | Don't use these controls: |
| --- | --- |
| **type** | **nodebug** |
| **debug** | **notype** |
| | **noobject** |
| | **nolist** |
| | **noprint** |
| | **optimize(2) or optimize(3)** |

8

Use these directives in your assembly modules:

**name**
**proc/endp**

Example invocation:

```
asm386  init.a38  type  debug
```

## BND286/386 and BLD286/386

| Use this control: | Don't use this control: |
| --- | --- |
| **noload** | **purge** |

Controls for BLD286 and BLD386 are shown in the **readme.wri** file.

Example invocation:

```
bnd386  cmain.obj  name(csamp)  oj(csamp.lnk)
noload
bld386  csamp.lnk  bf(csamp.bld)  oj(csamp.abs)
```

## Intel iC-86, iC-286 and iC-386

Use with the appropriate Intel linker/locator or binder/builder.

Use this control:

**debug**

Don't use these controls:

**nodebug**
**notype**
**optimize(2) or optimize(3) type**
**noprint**
**noobject**
**nolist**

Example invocation:

```
ic86  cmain.c  debug
```

## Intel LINK86/LOC86

Use with Intel iC, PL/M, ASM, and FORTRAN-386 compilers.

Don't use this control:

**purge**

We used a filter file to specify controls to the locator when preparing the sample programs. To view a sample filter file, see the **readme.wri** file.

8

Example invocation:

```
link86  cmain.obj  to  csamp.lnk
loc86  csamp.lnk  <  csamp.flt
```

### Intel PL/M-86, PL/M-286 and PL/M-386

Use with the appropriate Intel linker/locator or binder/builder.

| Use this control: | Don't use these controls: |
| --- | --- |
| **debug** | **nodebug** |
| | **noobject** |
| | **nolist** |
| | **notype** |
| | **noprint** |
| | **optimize(2) or optimize(3)** |

Example invocation:

```
plm386 pmain.p38 debug optimize(0) large
```

## MetaWare

Consult the *CSi-Link User's Guide* for tool directive information.

## Microsoft

Consult the *CSi-Link User's Guide* for tool directive information.

# Phar Lap

## Phar Lap LinkLoc

Use with MetaWare High C/C++.

Use these controls:

| | |
|---|---|
| **-symbols** | Include symbol table |
| **-compat softscope** | Using Soft-Scope |
| **-omfboot** | Produce OMF boot-loadable file |

To build the sample programs, we used a command file that usually has the extension **.lnk** to specify controls.

When preparing 386/486 protected-mode applications, it is important to specify correct controls to ensure that the proper symbolics are generated and the GDTs and IDTs are set up properly. To see a 386 command file look at **csamp.lnk** in the **/samp/hcxxx** subdirectories where you installed Soft-Scope.

Example invocation:

```
linkloc @csamp
```

### Phar Lap 386/ASM

Use these controls:

Use the **-386P**, **-386**, **-286P**, **286**, or **-86** switch on the command line to specify the instruction format you want.

**-cv** Debug information

Example invocation:

```
386asm -386P preamble.asm
```

**NOTE:** Version 2.2d does not produce line-number information.

## Watcom

Consult the *CSi-Link User's Guide* for tool directive information.

# A. Data Types, Operators, Registers, and Descriptors

This appendix contains a table of data types for use in type overrides, operators, descriptors, subfields, and figures of registers for the Intel386, Intel486™, and Pentium® processors. For more information on using or accessing these items in Soft-Scope, see the chapter, *Examining Data with Soft-Scope*. For more detailed information about the registers, see the Intel *Programmer's Reference Manual* for the processor you are using.

## Chapter Contents


**Data Types** ........ A-2
**Operators** ........ A-8
**General-Purpose Registers** ........ A-10
**NPX Registers** ........ A-13
**Protected-Mode Registers** ........ A-14
**Descriptors and Subfields** ........ A-15

# Data Types

The following table lists data types that can be used with Soft-Scope type overrides. Some of the types have subfields that can be identified in the register tables later in this appendix. [NOTE: Pentium is listed as 586.]

Table A-1: Data types for use in type overrides

| Data Type | Description | CPU/NPX |
|---|---|---|
| BCD | NPX data type, 10-byte BCD integer | 87/187/287/387/486/586 |
| BIT0 - BIT31 | These overrides provide access to individual bits in the specified reference | All |
| BOOLEAN | 1-byte boolean (00H=false, otherwise true) | All |
| BYTE | 8-bit unsigned integer | All |
| CHAR | 8-bit signed character | All |
| CR0TYPE | 32-bit $cr0 image | 376/386/486/586 |
| CR2TYPE | 32-bit $cr2 image | 386/486/586 |
| CR3TYPE | 32-bit $cr3 image | 386/486/586 |
| CWTYPE | NPX80x87 control word | 87/187/287/387/486/586 |

Table continued on next page.

Table A-1: Data types for use in type overrides *(continued)*

| Data Type | Description | CPU/NPX |
|---|---|---|
| DESC | Protected-mode descriptor | 286/376/386/486/586 |
| DOUBLE | 64-bit real | All |
| DWORD | Double-length unsigned integer, bit length sym.wordsize *2 | All |
| EFLTYPE | 32-bit $efl image | 376/386/486/586 |
| EXTINT | 64-bit signed integer | All |
| FAR16 | Far 16-bit offset pointer | All |
| FAR32 | Far 32-bit offset pointer | All |
| FLOAT | 32-bit real | All |
| FLTYPE | 16-bit $FL image | All |
| GDBTYPE | GDT base address type | 286/376/386/486/586 |
| HWORD | Half length unsigned integer, bit length is sym.wordsize/2 | All |
| IDBTYPE | IDT base address type | 286/376/386/486/586 |
| INT | Signed integer, bit length is sym.wordsize | All |

Table continued on next page.

Table A-1: Data types for use in type overrides *(continued)*

| Data Type | Description | CPU/NPX |
|---|---|---|
| LDTRTYPE | LDT table selector image | 286/376/386/486/586 |
| LINEAR | 32-bit linear address | 376/386/486/586 |
| LONG | 32-bit signed integer | All |
| MSWTYPE | 16-bit $msw image | 286/376/386/486/586 |
| NEAR16 | Near 16-bit offset pointer | All |
| NEAR32 | Near 32-bit offset pointer | 376/386/486/586 |
| NPX16R | 16-bit real-mode NPX save image | 87/187/287/387 486/586 |
| NPX16P | 16-bit protected-mode NPX save image | 287/387/486/586 |
| NPX32R | 32-bit real-mode NPX save image | 387/486/586 |
| NPX32P | 32-bit protected-mode NPX save image | 387/486/586 |
| NPX | Displays the NPX save image | As defined in option file |
| PAGEDIRTYPE | 32-bit page directory image | 386/486/586 |

Table continued on next page.

Table A-1: Data types for use in type overrides *(continued)*

| Data Type | Description | CPU/NPX |
|---|---|---|
| PAGETABLETYPE | 32-bit page table image | 386/486/586 |
| PCBTYPE | Peripheral control block | 186/188 |
| PHYSICAL | 32-bit physical address | 376/386/486/586 |
| POINTER | This type is set by setting sym.pointer to: NEAR16, NEAR32, FAR16, or FAR32 | All |
| QWORD | Unsigned integer, bit length is sym.wordsize*4. Not valid if sym.wordsize=32 | All |
| SELECTOR | 16-bit selector | All |
| SHORT | 16-bit signed integer | All |
| SIGNED | Signed integer, bit length is sym.wordsize | All |
| SIGNED BYTE | 8-bit signed integer | All |
| SIGNED DWORD | Signed integer, bit length is sym.wordsize*2 | All |

Table continued on next page.

Table A-1: Data types for use in type overrides *(continued)*

| Data Type | Description | CPU/NPX |
|---|---|---|
| SIGNED QWORD | Signed integer, bit length is sym.wordsize*4. Not valid if sym.wordsize=32 | All |
| SIGNED WORD | Signed integer, bit length is sym.wordsize | All |
| STRING | Zero-terminated string (max 255 characters) | All |
| SWTYPE | NPX 80x87 status word | 87/187/287/387/486 |
| TEMPREAL | 80-bit real | All |
| TR3TYPE | Test register 3 | 486 |
| TR4TYPE | Test register 4 | 486 |
| TR5TYPE | Test register 5 | 486 |
| TR6TYPE | Test register 6 | 376/386/486 |
| TR7TYPE | Test register 7 | 376/386/486 |

Table continued on next page.

Table A-1: Data types for use in type overrides *(continued)*

| Data Type | Description | CPU/NPX |
|---|---|---|
| TSS286 | 286 task state segment | 286/376/386/486/586 |
| TSS386 | 386 task state segment | 376/386/486/586 |
| TWTYPE | NPX80x87 tag word | 87/187/287/387/486/586 |
| UNSIGNED | Unsigned integer, bit length is sym.wordsize | All |
| UNSIGNED CHAR | 8-bit unsigned | All |
| UNSIGNED EXTINT | 64-bit unsigned | All |
| UNSIGNED INT | Unsigned integer, bit length is sym.wordsize | All |
| UNSIGNED LONG | 32-bit unsigned integer | All |
| UNSIGNED SHORT | 16-bit unsigned integer | All |
| WORD | Unsigned integer, bit length sym.wordsize | All |

# Operators

In addition to the Soft-Scope operators described in the table below, you can use all C operators except the ternary conditional operator (?:) and the comma operator(,).

Table A-2: Soft-Scope operators

| Operator | Description | Example |
|---|---|---|
| * | Displays the symbolic reference pointed to by the pointer | *xyz |
| -> | Displays a single element of the structure pointed to by the pointer | structname-> |
| .. | Creates a numeric range for accessing arrays | array[1..9] |
| ... | Specifies a range, starting at the first address of the array | array[...5]<br>array[5...] |
| & | Obtains the address of a symbolic reference | &xyz |
| : | Identifies a module name | :xyz |
| : | Constructs a pointer from a selector value and a 16- or 32-bit offset | 1234:1234 |

Table continued on next page.

Table A-2: Soft-Scope operators *(continued)*

| Operator | Description | Example |
|---|---|---|
| . | Prefixes a program symbol name to prevent confusion with Soft-Scope commands. For example, a variable named load | .load |
| . | Separates module names from variable names | :abc.xyz |
| . | Accesses members of a structure or variables within a procedure (or named block) | abc.xyz |
| length | Specifies how much memory beyond the referenced location to include in an operation | byte at 1234:456 length 5 |
| # | Converts an unsigned integer to a line-number | #89 ::xyz#89 |
| at | Converts an address into a null-type symbolic reference | at 0000:0000 |
| at | Dereferences the following address | byte at &abc |
| $ | Identifies register and CPU structure names | $GDT |
| $ | Designates macro symbols and parameters | $y |

# General-Purpose Registers

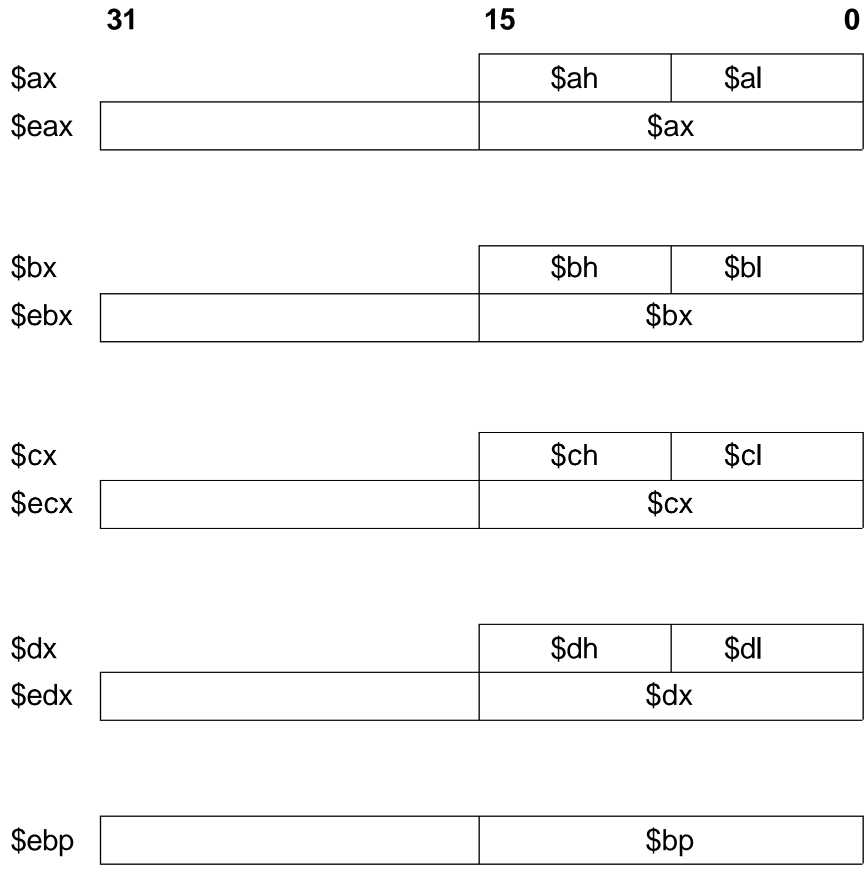



Figure A-1: General-purpose registers

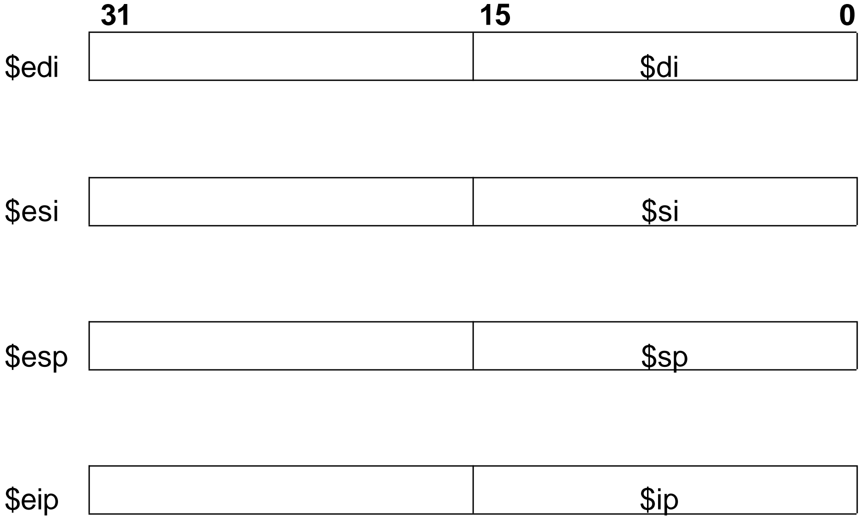



Figure A-1: General-purpose registers *(continued)*

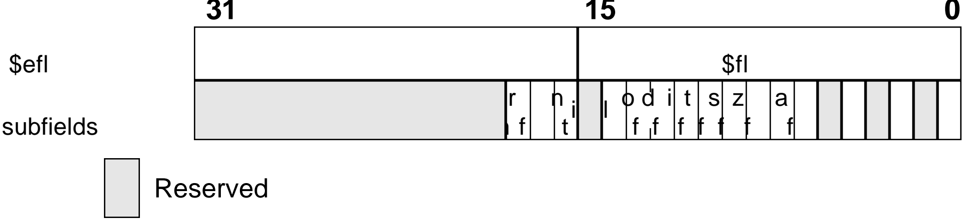



Figure A-2: Flags register

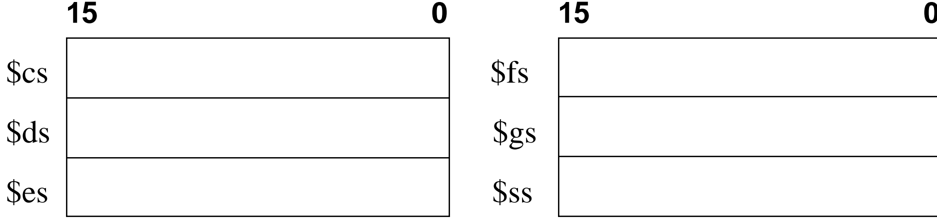



Figure A-3: Segment registers

# NPX Registers

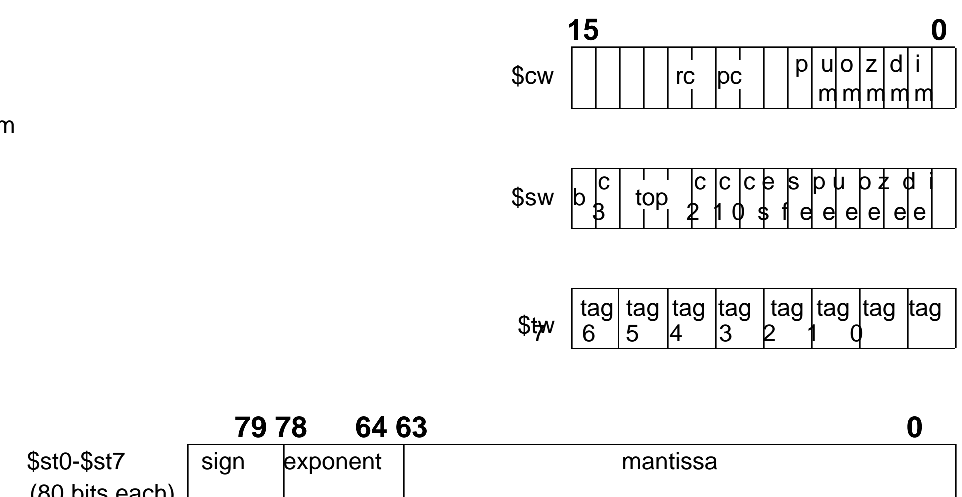



Figure A-4: NPX registers

# Protected-Mode Registers

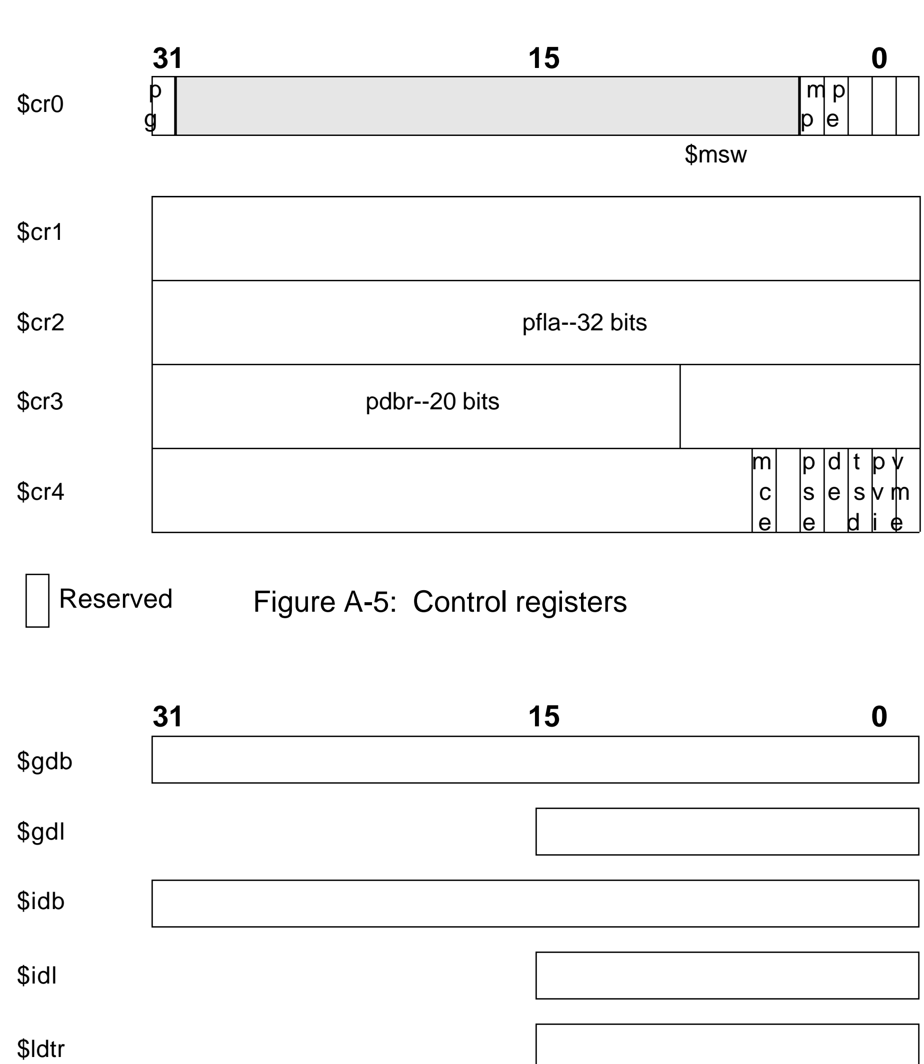



Figure A-5: Control registers



Figure A-6: Protected-mode registers

# Descriptors and Subfields

Table A-3: 386 protected-mode variables

| Structure | Description |
|---|---|
| $GDT | An array reference that spans the current global descriptor table. Use $GDT[n] to access a specific element. Use $GDT[n] to view the G, B, P, and AV bits and the actual limit value in the descriptor. |
| $IDT | An array reference that spans the current interrupt descriptor table. It can be referenced the same way as GDT. |
| $LDT | An array reference that spans the current local descriptor table. |
| $PAGEDIR | An array representation of the current 386 page table directory. |

Table A-4: Page table entries

| Name | Description | Starting Bit | Size (bits) | CPU |
|---|---|---|---|---|
| frame | Page frame address | 12 | 20 | 386/486/Pentium |
| avail | Available for use | 9 | 3 | 386/486/Pentium |
| d | Dirty | 6 | 1 | 386/486/Pentium |
| a | Accessed | 5 | 1 | 386/486/Pentium |
| pcd | Page cache disable | 4 | 1 | 486/Pentium |
| pwt | Page write transparent | 3 | 1 | 486/Pentium |
| us | User/Supervisor | 2 | 1 | 386/486/Pentium |
| rw | Read/Write | 1 | 1 | 386/486/Pentium |
| p | Present | 0 | 1 | 386/486/Pentium |

Table A-5: Descriptor subfields

| Name | Description | Starting Bit | Size (bits) |
|---|---|---|---|
| base | Segment base | 56,16 | 8,24 |
| g | Granularity | 55 | 1 |
| b | Big | 54 | 1 |
| d | Default | 54 | 1 |
| av | Available | 52 | 1 |
| lim | Segment limit | 48,0 | 4,16 |
| limit | Segment limit | 48,0 | 4,16 |
| offset | Offset in segment | 48,0 | 16,16 |
| p | Present | 47 | 1 |
| dpl | Descriptor privelege level | 45 | 2 |
| type | Segment type | 40 | 5 |
| dt | Descriptor type | 44 | 1 |
| code or data | Code or data | 43 | 1 |
| ed or cfm | Expand down or conforming | 42 | 1 |
| wr or rd | Write or read | 41 | 1 |
| ac | Accessed | 40 | 1 |
| count | Dword count | 32 | 5 |
| seg | Segment selector | 16 | 16 |
| sel | Segment selector | 16 | 16 |

Table A-6: TSS386 subfields

| **Name** | **Description** | **Starting Bit** | **Size (bits)** |
|---|---|---|---|
| io_map | I/O map offet from start of TSS | 102 | 16 |
| idtr | Register image | 96 | 16 |
| gs | Register image | 92 | 16 |
| fs | Register image | 88 | 16 |
| ds | Register image | 84 | 16 |
| ss | Register image | 80 | 16 |
| cs | Register image | 76 | 16 |
| es | Register image | 72 | 16 |
| edi | Register image | 68 | 32 |
| esi | Register image | 64 | 32 |
| ebp | Register image | 60 | 32 |
| esp | Register image | 56 | 32 |
| ebx | Register image | 52 | 32 |
| edx | Register image | 48 | 32 |
| ecx | Register image | 44 | 32 |
| eax | Register image | 40 | 32 |
| efl | Register image | 36 | 32 |
| eip | Register image | 32 | 32 |
| cr3 | Register image | 28 | 32 |

Table continued on next page.

Table A-6: TSS386 subfields *(continued)*

| Name | Description | Starting Bit | Size (bits) |
|---|---|---|---|
| ss2 | Level 2 stack segment | 24 | 16 |
| esp2 | Level 2 stack pointer | 20 | 32 |
| ss1 | Level 1 stack segment | 16 | 16 |
| esp1 | Level 1 stack pointer | 12 | 32 |
| ss0 | Level 0 stack segment | 8 | 16 |
| esp0 | Level 0 stack pointer | 4 | 32 |
| link | Backlink | 0 | 16 |

# B. Error Messages

B

## Chapter Contents

**Overview** ........................................................................................ **B-2**
**Address Error Messages** ..................................................................... **B-3**
Example Address Error Message ...................................................... B-3
Explanation ........................................................................................ B-3
How To Interpret Address Errors ..................................................... B-4
Table B-1: Conversion entry codes ................................................ B-4
Table B-2: Address error messages ............................................... B-5
**Error Messages** ...................................................................................... **B-7**

# Overview

Soft-Scope generates an error message when it cannot execute a command. Many of the error messages are displayed with a line of carets (“^^^^”) displayed beneath some part of the problematic command. The carets show where in the command Soft-Scope ran aground. Some of the messages in this chapter are warning messages and are identified as such in the message text.

When possible, error messages are discussed in the following format:

1. `< error message >`
2. Explanation describing why the error message was displayed
3. What to do to eliminate the error message or avoid it in the future

Error messages are presented in alphanumerical order.

# Address Error Messages

B

Because memory management, especially in protected-mode applications and applications using the processor's paging tables, is complex and does not allow descriptions of all possible memory errors, we have provided information to help you interpret address error messages.

All address error messages have the following format:

```
<Address - cvt ... - cvt - error message>
```

The variable *cvt* is an address where a conversion was attempted, and how the conversion was done. In all cases, the final *cvt* is where the error occurred.

# Example Address Error Message

The following example describes an extremely complicated error message. Most of the error messages you see won't be this complicated:

```
Reference: mysymbol

   struct {
         first . . . . . . . . . .128
         next  . . . . . . . . . .

<Address - ffff:12345678 fffb[8191] -
fffb:0000fff8 gdt[8191] - fffff123L
Page[1023][1023] - 00000123P - Page not present >
```

# Explanation

- Logical address ffff:12345678 with noncurrent LDT, fffb, required the LDT entry in LDT fffb.
- To read the descriptor, gdt[8191] had to be read.

- Reading the GDT entry at linear address fffff123L caused the page table entry for page[1023][1023] to be read from physical address 00000123P.
- Page[1023][1023] is missing or corrupted.

## How To Interpret Address Errors

Table B-1 below lists all possible values for *cvt*. If you get an address error message, compare the *cvt* entries with the table and determine what conversion was taking place when the error occurred. Then look in table B-2 on the next page to determine what the error part of the message means. You should be able to identify what happened to cause the error.

Table B-1: Conversion entry codes

| Conversion entry (cvt) | Description |
|---|---|
| logical_addr gdt[0] | 286/386 gdt[ ] |
| logical_addr ldt[0] | 286/386 current ldt[ ] |
| logical_addr 0000[0] | 286/386 noncurrent ldt[ ] |
| logical_addr cs_desc | CPU segment register cache |
| logical_addr v86 | virtual 86 logical to linear mode translation |
| logical_addr real | real mode |
| linear_addr page[0] | page directory |
| linear_addr page[0][0] | page table entry |

Table B-2: Address error messages

| Message | Description |
|---|---|
| Address wrap | Address attempted to wrap a segment, linear memory, or physical memory |
| GDT limit exceeded | Soft-Scope trapped a reference outside the bounds defined by the GDL register (the GDT limit) |
| LDT via LDT selector | The LTR register caontains a LDT selector |
| Memory bounds exceeded | The memory location that Soft-Scope is trying to access is out of range |
| Non-addressable segment type | The descriptor specified does not have an addressable segment associated with it |
| Not code segment | Soft-Scope requires code memory for the attempted operation |
| Not data segment | Soft-Scope requires data memory for the attempted operation |
| Not IDT segment | The selector given for the IDT indicated a GDT slot that did not specify an IDT segment |
| Not LDT segment | The selector given for the LDT indicated a GDT slot that did not specify an LDT segment |
| NULL selector | Zero is not a valid selector for protected mode |
| Page not present | Either the page-table page or the final memory page was missing |

Table continued on next page.

Table B-2: Address error messages *(continued)*

| Message | Description |
|---|---|
| Physical limit exceeded | The address specified is outside the limit of physical memory |
| Segment limit exceeded | The address specified is outside the segment limit or an attempt was made to use an offset greater than 0xffff in real mode |
| Segment not addressable | The descriptor associated with address specifies a segment type that is not addressable |
| Segment not present | The descriptor specified is not present in memory |
| Stack frame not set up | The referenced symbol is in a procedure that is not in the current scope and doesn't have an address. You can still inspect its type |
| Target limit exceeded | The address specified is beyond the memory range available in the computer |
| Write fail | Either no memory exists at the location specified or it is in ROM |

# Error Messages

< *filename linenum/column* - msg >

Soft-Scope encountered an invalid configuration-option specification while executing the specified line of file *filename*. The message is Soft-Scope's explanation of what caused the error.

Check your configuration-options file, usually **sswin32.ini**, for options that are not defined as specified in the chapter, *Configuring Soft-Scope*.

< (*selector*) not found in load object >

The *selector* given in this message is not part of the expected load object.

Make sure that the load object you specify in the **File-Load** dialog box is a valid load object for your current application.

< :name... not found in "*filename*" >

Soft-Scope can't find the module you have referenced.

Check to see that the module name is entered in the dialog box correctly. If it is, and you are sure the module exists in *filename*, make sure the application is built according to the instructions in the chapter, *Tools that Soft-Scope Supports*, and that the file is not corrupted.

< Application task running >

The function or command that you are attempting to execute requires target execution to be stopped.

If you have an interrupt-driven monitor, use the **Code/Stop** command to halt target execution.

```
< Application $gdt[0..x] can’t hold CSiMon $gdt[0..y]
>
```

The application was built without enough reserved GDT slots for CSi-Mon.

Rebuild the application, reserving GDT slots 0 thru 63.

```
< Application $idt[0..x] can’t hold CSiMon $idt[0..y]
>
```

The application was built without enough reserved IDT slots for CSi-Mon.

Rebuild the application, reserving IDT slots 0 thru 39.

```
< Attempted division by zero >
```

The specified expression resolves to a division by zero.

```
< Bad type for increment/decrement >
```

An increment or decrement operator (i.e., `i++, –i`) exists with a variable that has an invalid data type for that operation. For example: “GDT[5]++”.

Increment and decrement operators only work on scalar variables.

```
< Break is only valid inside while -"token" at line ###, col ### >
```

The macro compiler has encountered a ***BREAK*** statement outside of a loop.

Use ***BREAK*** statements only within ***WHILE*** statements to terminate loops.

< Breakpoint already set >

The referenced memory location or data area already has a breakpoint set for it.

Check in the **Breakpoints** window to see what breakpoints are set. Perhaps you will have to delete one and replace it with a new type. For example, delete a hardware breakpoint so you can replace it with an execution breakpoint.

< Breakpoint has not been set >

There is no breakpoint at the referenced location.

Check in the **Breakpoints** window for a list of existing breakpoints.

< Can’t increment constant or expression >

Increment or decrement operators (i.e., *i++, –i*) are set on a constant expression or array.

Increment and decrement operators work only on scalar variables.

< Can’t turn this into an array >

The operand above the carats is not a memory reference.

Use the length operator ***LEN[GTH]*** only with memory references.

< Cannot display local variables in data window >

The value of a local variable in a macro is not saved when the macro is terminated. Therefore, a local variable cannot be placed in the **Watch** or **Data** windows.

Use the macro ***PRINT*** command to display the values of local variables (see the *Macro Print Function* section in the chapter, *Creating and Using Soft-Scope Macros*).

`< Cannot unzoom window >`

Soft-Scope was attempting to unzoom a window but could not allocate enough memory to do so.

You may need to quit other Windows applications to create more memory.

`< Count not in range 1..9999 >`

Soft-Scope does not support counts outside the given range for the ***LIST*** command.

`< CSi-Mon — description >`

The target is reporting an error to Soft-Scope that *description* explains.

`< Ctrl-C break >`

<Ctrl>+<C> aborted the executing command.

`< Cursor not on execution breakpoint >`

The breakpoint under the cursor is a data breakpoint. Soft-Scope can only display source code for execution breakpoints.

The source code for some data breakpoints can be viewed by locating the reference in your code using the **Symbols** window for variables or the **Code** window in Assembly mode for addresses. Once you have located the address or variable, switch the **Code** window to Source mode.

`< Cursor not on line with address >`

Soft-Scope can't set a breakpoint on the line the execution pointer is on.

Possibly the current module has no line numbers or you are on a line beyond the end of the module.

B

```
< Expected "=" >
```

Soft-Scope found an assignment operator missing in the initialization file.

Modify your configuration **sswin32.ini** file and verify that for each option there is an equal sign between the option and its value.

```
< Expected "macro" keyword - "token" at line ###, col
### >
```

While compiling a macro file, the macro compiler was expecting the start of a macro but got *token* instead.

Examine your macro file and make sure the macro at the given line number has the keyword ***MACRO*** as the first word on the header line, and that all of the braces are in the right places.

```
< Expected %s - "token" at line ###, col ### >
```

While scanning the format string of a print statement, the macro compiler expected a string-format specifier and didn't find one.

Examine your macro file and make sure the print statement on the given line conforms to the specifications in the chapter, *Creating and Using Soft-Scope Macros.*

```
< Expected closing paren - "token" at line ###, col ###
>
```

While parsing the current macro's arguments or a ***WHILE*** or ***PRINT*** statement, the macro compiler expected a closing parenthesis but got *token* instead.

Examine the macro at the given line number. Make sure the parentheses are used according to the specifications in the chapter, *Creating and Using Soft-Scope Macros.*

< Expected comma - "*token*" at line ###, col ### >

Instead of a comma, which delimits arguments in a list, the compiler found *token*.

Check to make sure there is not a typographical error at the given line number.

< Expected format string - "*token*" at line ###, col ### >

While parsing a ***PRINT*** statement, the macro compiler expected a string indicating the format but got *token* instead.

Examine the macro at the given line number. Make sure the ***PRINT*** statement meets the specifications laid out in the *Creating and Using Soft-Scope Macros* chapter.

< Expected identifier - "*token*" at line ###, col ### >

The macro compiler has found *token* instead of a parameter or local variable.

Look in your macro file and make sure the identified line has no typographical errors and meets the specifications in the *Creating and Using Soft-Scope Macros* chapter.

< Expected macro name - "*token*" at line ###, col ### >

Instead of a macro name after the ***MACRO*** keyword, the macro compiler found *token.*

This is usually a typographical error at the given line number.

< Expected opening brace - "*token*" line ###, col ### >

The macro compiler expected a brace ("{") to start the macro but got *token* instead.

Look in the chapter, *Creating and Using Soft-Scope Macros*, for rules defining braces in macros.

< Expected opening paren - "*token*" at line ###, col ### >

While parsing a new ***MACRO***, ***PRINT***, or ***WHILE*** statement, the compiler found *token* instead of an opening parenthesis.

Examine the macro at the given line number. Make sure the parentheses are used according to the specifications in the chapter, *Creating and Using Soft-Scope Macros.*

< Expected parameter or variable - "*token*" at line ###, col ### >

Instead of a parameter or variable, the compiler found *token.*

Look at the given line number in your macro file for typographical errors. You may want to review the rules for using parameters and variables in the chapter, *Creating and Using Soft-Scope Macros.*

< Expected quoted string >

Soft-Scope expected a quoted string to be entered.

Please check that you are using the correct syntax for the macro ***PRINT*** command. The ***PRINT*** command is discussed in the *Macro Print Functions* section in the chapter, *Creating and Using Soft-Scope Macros.*

< Expression is too complex >

You are attempting to evaluate an expression that has more than 10 pending operators, for example, A+(B+(C+(D+...))).

Simplify the expression.

<< Fatal exception: msg >>

Soft-Scope encountered a severe error (*msg*), and it aborted execution.

Please restart Soft-Scope and reload your application.

`< Hardware breakpoint already set at this address` *`address`* `>`

There is already a data breakpoint set on *address*.

Look in the **Breakpoints** window for a complete list of currently set breakpoints.

`< Help not available >`

The topic you have entered does not have any help associated with it.

To see a list of help topics, select **Help/Index** from the menu, or press <F1>.

`< Identifier already defined - "`*`token`*`" at line ###,col ### >`

The macro compiler encountered a duplicate symbol declaration in the source file.

Check the macro file at the line number identified by the message, and correct any errors according to the specifications given in the chapter, *Creating and Using Soft-Scope Macros.*

`< Initial task register is an ldt selector >`

The initial task register was defined so that a selector in a local descriptor table was selected (warning only).

`< Initial task register is outside gdt limit >`

The initial task register was defined outside the limits of the initial GDT (warning only).

`< Initial TR->non-TSS type descriptor >`

The GDT entry that the initial task register pointed to is not an Intel386 task- state segment descriptor (warning only).

```
<< Insufficient memory >>
```

Soft-Scope was unable to allocate memory for window data structures.

You may need remove some TSR (Terminate but Stay Resident) utilities or device drivers to free memory for Soft-Scope.

```
< Insufficient memory to store option >
```

Soft-Scope was attempting to allocate memory to store a configuration option but could not allocate enough memory.

You may need to quit other Windows applications to create more memory.

```
< Internal error [- message] >
```

Soft-Scope has encountered either data or a situation that was thought to never occur but has in this particular case.

Please report this error to us (see title page for contact information), along with as much information as possible on why this error might have occurred.

```
< Invalid field near ########: "filename" >
```

A bad symbolic record was found in the load file *filename* at offset ########.

You may have to recompile and rebuild your application. This usually means the application file is corrupted.

```
< Invalid file: filename >
```

The load file specified is not recognized by Soft-Scope.

This could mean the file is corrupted, or not prepared according to the specifications in the chapter, *Configuring CSi-Mon* of the *CSi-Mon Monitor User's Guide.*

```
< Invalid file: filename >
```

The load file specified is not recognized by Soft-Scope.

This could mean the file is corrupted, or the file was not prepared according to the specifications in the chapter, *Tools that Soft-Scope Supports*.

```
< Invalid macro compiler version >
```

Your macro object file contains a version number that does not match the version Soft-Scope was expecting.

Erase ***filename*.mob** so Soft-Scope will recompile your macros.

```
< Invalid macro object file >
```

The macro compiler produced bad object code, or some other process corrupted its output.

Try erasing the file ***filename*.mob** so the macro compiler recompiles your macros.

```
< Invalid macro opcode >
```

While executing a macro, Soft-Scope has encountered an unknown macro command in the macro object file.

Look in your macro file for typographical errors. If you can't find any mistakes, you might want to review the macro commands given in the chapter, *Creating and Using Soft-Scope Macros.*

```
< Invalid number format >
```

Soft-Scope can't understand the specified number (e.g., **X = 1234Q5H**).

This usually means the number or variable has an invalid base attribute. Valid bases are as follows: T = base 10, H = base 16. See the *Numbers* section in the chapter, *Examining Data with Soft-Scope*.

B

```
< Invalid override >
```

Either the attempted override is a bit*xx* override of a reference that does not contain bit*xx* (e.g., bit20 $al), or the override contains two data types that do not produce a meaningful type (e.g., **swtype tss386** is not meaningful, but **signed byte** is).

See appendix A for list of data types usable in type overrides.

```
< Invalid override for processor type >
```

The specified type is not valid for your processor.

See appendix A for supported data types and their descriptions.

```
< Invalid Range >
```

The range specified has a starting value greater than its ending value.

Please retype the range.

```
< Invalid size for I/O port >
```

Overrides for the I/O port must be 8-bit, 16-bit, or 32-bits long. The specified type doesn't match the processor port sizes (e.g., **tempreal port 0**, which attempts to specify a 10-byte type to port 0).

See appendix A for supported data types and their descriptions.

```
< Invalid value for parameter >
```

You have specified an invalid parameter with the function ***RETURN***.

Please specify an integer value.

```
< Line number out of range (### to ###) >
```

The line number specified isn't within the range of line numbers for the module or procedure you're currently in or for the module/procedure specified.

< Line too long: “*filename*”>

The given text file contains a line that is too long to be processed.

Edit the file and shorten the line.

< Listing file invalid: Improper listing end >

Soft-Scope doesn’t recognize a file you’ve specified as a listing file. Possibly the file isn’t a listing file, or at least some character within the file isn’t recognized by Soft-Scope (e.g., you’re using a version of some language that Soft-Scope doesn’t yet understand).

Please review the information in the chapter, *Tools that Soft-Scope Supports,* to see what tools Soft-Scope supports.

< Listing file invalid: Improper listing header >

Soft-Scope doesn’t recognize a file specified as a listing file. Possibly the file isn’t a listing file, or at least some character within the file isn’t recognized by Soft-Scope (i.e., it was prepared using a version of some language that Soft-Scope doesn’t yet understand).

Please review the information in the chapter, *Tools that Soft-Scope Supports,* to see what versions of tools Soft-Scope supports.

< Macro Abort >

A macro executed an abort command.

This happens when a macro contains an***ABORT*** statement.

< Macro execution halted - current macro has been deleted
>

A macro deleted the macro that called it, making it impossible to return.

The original macro file stored on your disk is not erased when this happens. Edit the called macro so it doesn’t delete the calling macro, reload the macro file into Soft-Scope, and try again.

```
< Macro name expected >
```

Soft-Scope is expecting a valid macro name.

Use **Macro/Display** to see which macros are loaded.

```
< Macro nesting too deep >
```

Macro execution has executed too many nested macros.

Only ten macros may be nested.

```
< Minimum field width specifier required with 0 padding
>
```

Soft-Scope doesn't know how many padding zeros to use in the output of your macro ***PRINT*** or ***WPRINTF*** statement.

See the *Macro Print Functions* section in the chapter, *Creating and Using Soft-Scope Macros*.

```
< Mismatched ()'s >
```

Soft-Scope is expecting another right parenthesis.

Make sure your macro has the correct number of right and left parentheses.

```
< Mismatched [ ]'s >
```

You've forgotten a right bracket ("]") or have used too many left brackets ("[").

Make sure your macro has the correct number of right and left brackets.

```
< modname contains no lines >
```

The module *modname* that you've specified or are currently in doesn't contain line numbers. Possibly the module is empty, has not been built with line numbers, or is an assembly-language module.

Look in the chapter, *Tools that Soft-Scope Supports,* to see how to build your application with line numbers.

```
< Module not found >
```

Soft-Scope cannot find the specified module name.

Make sure the module name doesn't contain typographical errors. If it doesn't, make sure it is located in the current application.

```
< More parameters given than the macro defined >
```

You've tried to invoke a macro, and specified more parameters than the macro needs.

Retype the macro invocation. You may have to shell out to a text editor to examine the macro file and refresh your memory.

```
< No address associated with reference >
```

The expression entered has no address associated with it.

This is usually a typographical error. If you can't find an error, look in the **Symbols** window to refresh your memory of symbol spellings.

```
< No breakpoint to edit >
```

User has tried to edit a breakpoint, but there is no breakpoint displayed under the cursor.

```
< No initial TSS is defined >
```

During loading, the TR (Task Register) value was set to 0 (warning only).

```
< No macro currently running >
```

You have attempted to suspend a macro, but no macro is currently running.

```
< No macro currently suspended >
```

You have attempted to resume a macro but no macro is currently suspended.

B

```
< No macros defined >
```

A macro file (with one or more macros) has not been loaded into Soft-Scope.

Use **Macro/Load** to load a macro file, and then select **Macro/Display** to list the macros it contains in the **Macros** window.

```
< No modules loaded >
```

The given command requires a default module, and there are no modules found in Soft-Scope's symbolic database.

Use **File/Load** to load an application.

```
< No modules loaded and application task running >
```

Soft-Scope cannot find symbolics to display in the **Code** window and the current task is running.

Use **File/Symbol load** to load symbolic information for the running application.

```
< No modules loaded and no target attached >
```

The connection to your target has failed.

Check your cable to make sure it is properly connected. If the problem persists, see the section on troubleshooting in the *CSi-Mon Monitor User's Guide*.

```
< No return address available >
```

The specified return address isn't resolvable (i.e., `RETURN( )`). Review the specifications given in the chapter, *Tools that Soft-Scope Supports,* to make sure your application is properly built.

```
< No source available for address >
```

Soft-Scope cannot display the source for the *address* shown.

Perhaps the address is in an assembly module and doesn't have source, or it wasn't prepared with debug information. See the chapter, *Tools that Soft-Scope Supports,* to see how to prepare an application with debug information.

`< No symbolic information loaded >` **or**
`< No symbols loaded >`

Soft-Scope can't find any symbols loaded.

Possibly you haven't yet loaded an application, or your application is loaded but not built for debugging. See the chapter, *Tools that Soft-Scope Supports,* for information on how to build an application for debugging with Soft-Scope.

```
< No target attached >
```

Soft-Scope can't communicate with the target.

See the section on troubleshooting in the *CSi-Mon Monitor User's Guide*.

```
< Not 286/386 absolute file >
```

The file you are trying to debug is not compiled in the right format, or it is not a 286/386 **.abs** file.

See the chapter *Tools that Soft-Scope Supports* for information describing how to use specific compilers and other tools.

B

```
< Not valid for processor >
```

The CPU doesn’t contain the register you’ve specified.

See appendix A for applicable registers.

```
< Number too large >
```

The specified floating-point value is too large to be converted to a valid floating-point number.

The NPX register supports any floating point number with a 15-bit exponent and a 64-bit mantissa. See appendix A.

```
< Only 64 options permitted >
```

Soft-Scope allows up to 64 configuration options to be specified at one time.

Perhaps you have some options you can delete because you are using the default values.

```
< Option “src.tab” - Must be 1 to 16 >
```

In your initialization file, the entry for tab stops is set to something other than one of the integers between 1 and 16.

Open the **Options** window to see what **src.tab** is set to. Click the **Modify** toolbar button to change the setting. See the chapter *Configuring Soft-Scope.*

```
< Option “sym.case” - Must be ON or OFF >
```

In your initialization file, the entry **sym.case** is set to something other than **on** or **off.**

Open the **Options** window to see what **sym.case** is set to. Click the **Modify** toolbar button to change the setting. See the chapter *Configuring Soft-Scope*.

< Option “sym.pointer” - Must be FAR16, FAR32, NEAR16, NEAR32 >

The **sym.pointer** option must be set to one of the specified values.

Open the **Options** window to see what **sym.pointer** is set to. Click the **Modify** toolbar button to change the setting. See the chapter, *Configuring Soft-Scope*.

< Option name expected >

The ***SET TO*** command requires that an option name be specified.

See the ***SET*** command syntax in the chapter, *Soft-Scope Basics*.

< Option not defined >

You are attempting to use ***SET*** and Soft-Scope did not find the specified option name.

The available configurations are in the chapter, *Soft-Scope Basics.*

< Option *optionname* - Must be defined >

The option *optionname* isn’t defined in your initialization file, and is required for the operation you’ve just attempted.

See the available options in the chapter, *Configuring Soft-Scope*. Use the **Insert** toolbar button of the **Options** window to put the needed option in your initialization file **sswin32.ini**.

< Out of hardware breaks >

The processor debug registers are full. Either too many data breakpoints are set or the reference you gave includes too much memory.

For information explaining the registers and how to use them efficiently, see the chapter, *Controlling Program Execution with Soft-Scope.*

```
< Out of memory for trace buffer >
```

Your host machine doesn't have enough memory for a trace buffer.

Try setting the option **trace.filesize** to a lower value.

```
< Out of symbol space >
```

The macro compiler has exceeded its limit of 100 symbols (including keywords) in a macro.

Try breaking the macro into one or more smaller macros.

```
< Override not permitted on non byte-aligned bitfield
>
```

Soft-Scope trapped an attempted bitfield type override.

Possibly the override is not a supported data type, or there is a typographical error in the specification.

```
< Port addresses must be 0 to 0ffffH >
```

The specified port address is not between 0 and 0ffffH.

Retype the specification with an acceptable port address.

```
< Read-only register GDB >
```

The GDB (GDT base register) can only be changed by an application load.

```
< Read-only register IDB >
```

The IDB (IDT base register) can only be changed by an application load.

```
< Received fatal error trying to reset >
```

Soft-Scope received a fatal error while trying to initialize your target.

You may need to manually reset your target.

```
< Register doesn't contain this flag >
```

The register specified doesn't contain the flag specified.

See the **Registers** window for a display of all registers and their flags.

```
< Return (#) address unknown >
```

Soft-Scope was unable to calculate a return address for the current procedure or for the #th nested call.

Perhaps # is too large, if specified. This error may also appear if your application is built incorrectly.

```
< Serial error — Data overrun >
```

Your host machine can't interpret data from the target as fast as it is arriving.

Lower the baud rate as specified in the *Installing Soft-Scope on the Host* section in the chapter, *Getting Started with Soft-Scope*.

```
< Size of override exceeds size of non-memory operand
>
```

Soft-Scope can't override a smaller variable with a larger type. (e.g., overriding a WORD register with a DWORD type).

See appendix A for a list of data types that can be used as type overrides in Soft-Scope.

```
< Stack location is not known >
```

Soft-Scope was unable to locate the stack.

Perhaps you are using the SMALL memory model where stack and data are in the same segment.

```
< String too long >
```

The string type override was applied to memory starting at the specified address, but Soft-Scope didn't find a terminating null character (\0) within the first 255 characters.

Use the char type override and specify the number of bytes to view as characters using the length operator LEN[GTH]. For example:

```
char at 1000p length 5
```

```
< Subscript ranges on pointers are not supported >
```

Soft-Scope only recognizes a single reference (e.g., PTR[5]) for pointers.

Soft-Scope does support array-subscript ranges (e.g., array1[5..20]).

```
< Subscripts must be integers or ranges of integers >
```

The specified subscript or range is invalid.

Possibly the subscript isn't an integer, or there is a typographical error in the range operator. See the *Data References* section in the chapter, *Examining Data with Soft-Scope*.

```
< Symbol not found >
```

Soft-Scope has no record of the specified symbol.

Make sure the symbol is in the module you are currently executing in, that you have specified the correct module with a colon (:), as described in the chapter, *Examining Data with Soft-Scope*, or that the symbol is public.

```
< Symbol without base –Invalid field >
```

There is an invalid field in the OMF file.

Please verify that you have correctly built your application using the information presented in the chapter, *Tools that Soft-Scope Supports*. Contact us (see title page for contact information) if you cannot eliminate this problem.

```
< Symbolic name expected >
```

The parameter above the carets is not a symbolic name.

Look in the **Symbols** window for a list of application symbols.

```
< Syntax error >
```

The specified command is an invalid command or an invalid form of a valid command. Complete command syntax can be found in the chapter, *Soft-Scope Basics*.

```
< System - filename too long >
```

The *filename* (including *pathname*) is longer than 66 characters.

Shorten the *filename* or *pathname*.

```
< System - not enough memory >
```

Soft-Scope was attempting to allocate memory and was unable to do so. You may need to quit other Windows applications to create more memory.

```
< Target not responding >
```

Soft-Scope tried to open communication with the target, but could not synchronize I/O.

Check your cable connections and baud rate, reset the target, and try again.

```
< Target still not responding >
```

Soft-Scope cannot communicate with the target.

Check your cable connections and baud rate, reset the target, and try again. If the problem persists, see the troubleshooting section in the *CSi-Mon Monitor User's Guide*.

`< Target timeout on read >`

Soft-Scope has asked the target for specific information, but the information wasn't available or the target couldn't transmit.

This could be a serial communication problem, such as a data overrun. Try resetting the target, and re-invoking Soft-Scope. If the problem persists, we recommend you install a FIFO UART.



`< These addresses are not compatible >`

Soft-Scope cannot perform the specified operation because the addresses given have different types.

When Soft-Scope attempts an operation on two addresses, it expects them to be of the same type (logical, linear, or physical), and it expects logical addresses to have the same selector.

`< These are in the wrong order >`

The two parameters above the carets are in the wrong order.

Try the command again, switching the placement of these two parameters.

`< These are not comparable >`

The two parameters above the carets are of incomparable data types.

`< These operands are not compatible >`

The addition or subtraction operation uses two operands that are not compatible.

`< These types are not compatible >`

The types of the variables above the carats are not compatible for assignments.

For example, structures can only be assigned to structures of the same type.

```
< This address has no associated symbols >
```

The specified module contains no symbolic information.

Use **Code/Module** to see which modules are available.

```
< This is not a code reference >
```

The parameter above the carets does not refer to executable code, and the command you attempted expected this parameter to reference executable code.

Look in the **Symbols** window for a list of application symbols.

```
< This is not a logical address expression >
```

The parameter above the carets must evaluate to a logical address.

See the chapter, *Examining Data with Soft-Scope.*

```
< This is not a memory reference >
```

The parameter above the carets must evaluate to a memory location or address.

See the chapter, *Examining Data with Soft-Scope.*

```
< This is not a module reference >
```

The parameter above the carets must evaluate to a module.

Possibly you've misspelled the module name, or forgotten to preface the name with a colon (e.g. :**cmain)**. It might help to refresh your memory if you open the **Symbols** window and examine the list of application symbols. See the chapter, *Examining Data with Soft-Scope.*

B

```
< This is not a numeric expression >
```

Soft-Scope is expecting a number, and the parameter above the carets doesn't resolve to one. See the chapter, *Examining Data with Soft-Scope*.

```
< This is not a pointer >
```

The parameter above the carets is not a pointer.

Find out the type of the variable by placing it in the **Data** window and switching to Types mode.

```
< This is not a pointer or address >
```

The parameter above the carets is not a pointer or a memory address.

Find the variable's type by placing it in the **Data** window and changing to Types mode.

```
< This is not a structure or union pointer >
```

The dereferenced variable is not a pointer to a structure or union.

Find the variable's type by placing it in the **Data** window and changing to Types mode.

```
< This is not a symbolic reference >
```

The reference is not a symbol or variable.

Soft-Scope defines a symbolic reference as something you can assign a value to. For example, **i** is a symbolic reference, while **5** is not.

```
< This is not an array or pointer >
```

The parameter above the carets is not an array.

Perhaps you have provided subscripts on a variable that does not require subscripts. See the chapter, *Examining Data with Soft-Scope*.

```
< This is not an integer expression >
```

The given expression does not evaluate to an integer.

Try checking the types of variables in the expression by placing them in the **Data** window and changing to Types mode.

```
< This module was not compiled for debugging >
```

The module name above the carets does not contain debugging information, and Soft-Scope only knows that it's a module without debug information.

Make sure the application was prepared using the specifications given in the chapter, *Tools that Soft-Scope Supports.*

```
< This reference contains no lines >
```

The referenced source file contains no source lines.

Make sure the application was prepared using the specifications given in the chapter, *Tools that Soft-Scope Supports.*

```
< This subscript indexes to before the array >
```

The subscript above the carets evaluates to a number less than the first element in that array.

Try examining in the **Data** window any variables you have used in the index to make sure their values are what you thought they were.

```
< This type cannot have members >
```

The specified type doesn't support subfields. See the *Data References* section in the chapter, *Examining Data with Soft-Scope*.

```
< Too many breakpoints are set >
```

Soft-Scope supports up to 32 execution breakpoints.

You cannot set another breakpoint without removing an already set breakpoint.

B

`< Too many jump targets >` **or** `< Too many jumps >`

The macro compiler has exceeded its internal limit of 100 jumps per macro. Here are two examples of these "jumps": the compiled code for a while-statement contains one jump and so does the code for an if-statement

Try rewriting the macro as two or more macros.

`< Too many parameters >`

The specified function doesn't require as many parameters as were supplied.

`< Too many subscripts >`

The reference specifies more subscripts than there are array dimensions.

Examine the array in the **Data** window to see the array size.

`< Undefined identifier - "token" at line ###, col ### >`

The macro compiler has parsed an identifier that it can't find in its symbol table.

Define the identifier in the macro it is in. See the chapter, *Creating and Using Soft-Scope Macros,* for defining macros.

```
< Unexpected end of file - "token" at line ###, col ###
>
```

The macro compiler has unexpectedly encountered the end of file while parsing for *token*.

Check to see if the macro file is corrupted, or if an opening comment delimiter is not matched with a closing one.

```
< Unexpected end of line >
```

The macro compiler has unexpectedly encountered the end of a line while parsing for a token.

Edit the macro file and make sure the macro is written according to the specifications given in the chapter, *Creating and Using Soft-Scope Macros.*

```
< Unexpected end-of-file >
```

Soft-Scope was attempting to read data from a file and encountered the end-of-file before reading all expected data.

Perhaps the file is corrupted.

```
< Unknown macro name >
```

Soft-Scope was unable to find the macro you specified.

Use **Macro/Display** to see which macros are loaded.

```
< Unknown member >
```

The member above the carets doesn't exist for that structure, union, or register.

Possibly a misspelled member name or a reference to the wrong structure. Examine the structure in the **Data** window to see what members it contains.

B

```
< Unknown window name >
```

The name specified in a ***WMOVE***, ***WRESIZE***, or ***WFUNCTION*** macro command does not refer to a Soft-Scope window.

See the chapter, *Creating and Using Soft-Scope Macros*, for a list of Soft-Scope window names usable with these functions.

```
< Unsupported assignment operation >
```

The parameter above the carets cannot be assigned to the value attempted (e.g., GDT[5]=GDT[0] or $ax=”abcde”).

```
< Warning: Lines ### to ### are missing for modname >
```

A line number record has been generated in your object module *modname* for which there is no line number in your source or listing file. This may indicate a problem with your compiler.

```
< Warning: Not connected to CSi-Mon or CodeTAP >
```

Soft-Scope cannot read the CSi-Mon version string to determine what your target processor is.

The CSi-Mon licensing agreement forbids modification of the version string.

```
< Warning: Target processor assumed to be 8086 >
```

The CSi-Mon version string has been modified and Soft-Scope cannot read it to determine what your target processor is.

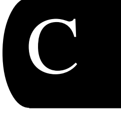
The CSi-Mon licensing agreement forbids modification of the version string.

```
< WFUNCTION output buffer too large, maximum=128
characters >
```

The ***WFUNCTION*** macro command output requires more memory than is allocated to the buffer.

Each ***WFUNCTION*** command is assigned an individual buffer. Use more than one ***WFUNCTION*** command to perform the desired operations.

# C. Debugging .exe Executable Files




## Chapter Contents

**Overview** .......... **C-2**
**Debugging .exe Files** .......... **C-2**
Preparing Your Application .......... C-2
Using the Special Monitor .......... C-3
Loading an .exe Application .......... C-3

# Overview

This appendix discusses how to debug an **.exe** executable file using the SSBUG utility and a special version of CSi-Mon.

# Debugging .exe Files

**NOTE:** Soft-Scope cannot debug **.exe** applications for which memory assignments may change during execution, such as applications designed to run under Microsoft Windows or Quarterdeck DESQview.

## Preparing Your Application

To debug an **.exe** or **.exp** file, Soft-Scope needs access to the application's symbolic information and CSi-Mon needs to take control when the application starts executing. This is accomplished using the SSBUG utility and linking an assembly routine to your application as described below:

1. Use the SSBUG utility, which is described in the chapter *Tools that Soft-Scope Supports*, to create a **.bug** file from your **.exe** or **.exp** file. The **.bug** file contains your application's symbolic information.

2. Link the assembly routine **ss_brkexe**, located in the file **\samp\mscexe\brkexe.asm**, to your application. Call **ss_brkexe** to invoke a special interrupt that will cause CSi-Mon to stop your application and take control so you can begin debugging. Make sure to call **ss_brkexe** before you reach the area of your application that you wish to debug.

# Using the Special Monitor

A special version of the CSi-Mon monitor, **exedbg.exe**, is required to debug an **.exe** file. See the chapter, *Configuring CSi-Mon*, in the *CSi-Mon Monitor User's Guide* for details on installing the monitor on your target PC.

The monitor can be invoked from the DOS prompt or as a device driver. From the DOS prompt, type `exedbg`. To install the monitor as a device driver, add the following line to your target PCs **config.sys** file:

```
device=drivename:\pathname\exedbg.exe
```

# Loading an .exe Application

When you boot the target PC with the device-driver specification shown above in its **config.sys** file, or invoke the monitor as an **.exe** file, CSi-Mon is installed on the target machine as a Terminate but Stay Resident (TSR) program. To debug your application, do the following:

1. Run the application from the DOS prompt on the target. When it hits the breakpoint set by **ss_brkexe**, it will stop.
2. Invoke Soft-Scope on your host computer. Do not include an application load on the invocation line.
3. Choose **File/Symbol load...** and enter the *pathname* and *filename* of the **.bug** file created by the SSBUG utility.

To load symbols from the **Command line** dialog box (<Ctrl>+<L>), use the following syntax:

```
FILENAME.BUG [filename]
FILENAME.BUG :device | f:
FILENAME.BUG (SEGMENT | JOB) relocationseg
```

*FILENAME*.BUG — A **.bug** file name associated with a relocatable DOS program, including a path to the file

| | |
|---|---|
| *filename* | Executable file whose name differs from *FILENAME* |
| *:device* | The name of a character device driver. The ( **:** ) differentiates the device driver from an ordinary DOS application |
| *f:* | The name of a block device driver |
| JOB | DOS job handle (PSP segment) |
| *relocationseg* | A segment specified in hex |

*FILENAME.BUG* [*filename* ] is used to attach symbolics to a relocatable DOS executable program (e.g., an **.exe** file). If *filename* is not given, the *filename* part of ***filename*.bug** will be used to search for a matching **.exe** file. If a *filename* is given, Soft-Scope will search the current directory on the target for an exact match.

Soft-Scope searches target memory for DOS's memory control-block (MCB) chain. Since the structure of DOS MCBs and the probable starting location are undocumented, Soft-Scope searches target memory between linear addresses 701 and 106ffl for the first MCB header, which contains the bytes 4d 08 00.

If Soft-Scope does not find the first MCB header, you can specify another memory range to search using the following options in your **sswin32.ini** initialization file:

```
targ.dos_mcb_start=0xstart
targ.dos_mcb_end=0xend
```

Where *start* is the linear address where you want Soft-Scope to begin the search and *end* is the linear address where the search is to stop.

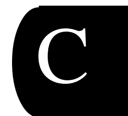
*FILENAME.*BUG *:device | f:* is used to attach symbolics to a DOS device driver. *device* is used for character device drivers and *f:* is used for block device drivers.

This command requires Soft-Scope to search target memory between the linear addresses 701 and 106ff for the NUL device, which begins the

device-driver chain. However, because the NUL device and its location are undocumented and the search may fail, you can redefine the search range by using the following options in your **sswin32.ini** file:

```
targ.dos_nul_start=0xstart
targ.dos_nul_end=0xend
```

Where *start* is the linear address where you want Soft-Scope to begin the search and *end* is the linear address where the search is to stop.



*FILENAME.*BUG JOB *relocationseg* is used to attach symbolics using a DOS job handle (PSP segment).

4. Now you can control the DOS application through Soft-Scope, using breakpoints and the various commands and functions to stop and start execution.

(This page blank)

# D. Helpful Hints

## Chapter Contents

**Overview** ........................................................................................ **D-2**
**Helpful Hints** .................................................................................. **D-3**
Changing the Execution Point ................................................. D-3
Source Line Address ................................................................ D-3
Changing an Executable Instruction ........................................ D-4
Bypassing Start-up Code .......................................................... D-5
Copying Memory ...................................................................... D-5
Receiver Timeouts .................................................................... D-6
Segment Limit Exceeded .......................................................... D-6

# Overview

This appendix explains features that the experienced user might find useful in special circumstances. It is important that you read the introductory paragraph for each topic because it may contain warnings or limitations that you should be aware of.

*If you discover an undocumented or unusual way to use Soft-Scope, and would like to share your discovery with other users, call (208) 882-0445, fax (208) 882-9774, or email tech@consci.com and tell us about it. If possible, we will include your new idea in this appendix the next time we update the manual.*

# Helpful Hints

## Changing the Execution Point

By modifying the value of the **$eip** register, you can change the execution point. However, you must be sure that the stack and registers are set up properly for the new execution point. *You should not use this feature to move to a function you are not currently in.* If you do, the program stack will be incorrect for that function and the results will be unpredictable.

D

To change the execution point, do the following:

1. Choose the **Registers** command from the **Data** pull-down menu
2. Move the cursor to the **$eip** register
3. Click the **Modify** toolbar button
4. Enter the new value in the dialog box

Be sure to specify hex by placing **0x** in front of the value:

```
$eip = 0x123
```

## Source Line Address

You can determine the address of a source line by using the line number with the ADDRESSOF (&) operator in the **Data/Examine** dialog box:

Data reference: `-`

You can set the **$eip** to the address of a source line in one step using the OFFSETOF operator in an expression like the one below, which sets **$eip** to the offset of line number 45:

```
$eip = offsetof(&#45)
```

# Changing an Executable Instruction

It is possible to change an executable instruction. The following statement assigns the value 90H (NOP), the no-operation opcode, to every byte of source line 99:

```
byte at &#99 length (sizeof #99) = 90H
```

This command is made up of the following subexpressions:

| | |
|---|---|
| byte | in one-byte increments |
| at c | at the address of the beginning of line 99 |
| length | for the length of... |
| (sizeof #99) | the number of bytes that make up line 99 |
| = 90H | assign the value 90H |

The macro shown below can be used to substitute an opcode for every byte of a source-code line.

```
macro arr_chg (line $line, hex int $value)
{
    byte at &#$line length(sizeof #$line)=$value;
}
```

## Bypassing Start-up Code

If the compiler you are using places a preamble module of assembly start-up code at the beginning of your application, Soft-Scope will always display that module when you load.

You can use an initial macro to go to main. In your **sswin32.ini** configuration file, make the following assignments:

```
cmd.macro=sswin.mac
load.init_command=go main
```

You can set the option **load.init_command** to a command or a macro name.

Alternatively, you can use the **Command** text box in the **File-Load** dialog box to go to *main*.

## Copying Memory

You can copy a block of memory from one location to another while in Soft-Scope. Use type-override syntax with an equal sign. The types must be compatible, as in the following example:

```
byte at 200P len 10 = byte at 100P len 10
```

## Receiver Timeouts

If you experience receiver timeouts, try adding **targ.debug=*filename*** to your **sswin32.ini** configuration options file. This option will create a log file that contains the communication stream between Soft-Scope and CSi-Mon. This log was designed for internal use and the format is very cryptic, however you may find it helpful when trying to understand where the receiver timeout occurs.

**NOTE:** The log file can become very large, so be sure and remove the option from **sswin32.ini** when it is no longer needed.

## Segment Limit Exceeded

If you encounter the error message "Segment limit exceeded for GDT[*xx*]" during an application load, insure that the TSS for the application matches the CSi-Mon/target you are using. It is possible that an application built for a 286TSS is being downloaded to a 386 or higher monitor/target. When Soft-Scope updates the IP portion of the EIP register, the EIP may have contained a value greater than 64K. This value may be larger than the application's GDT entry, since 286 descriptors are limited to 64K.

# E. Add Ons

## Chapter Contents

**Real-Time Operating Systems Support ............................................. E-2**
Kernel Objects ...................................................................................... E-3
Figure E-1: SuperTask! kernel objects dialog box .......................... E-3
Task List .............................................................................................. E-4
Figure E-2: SuperTask! task list dialog box ................................... E-4
Current Task ........................................................................................ E-4
Figure E-3: SuperTask! current task dialog box ............................. E-4

# Real-Time Operating Systems Support

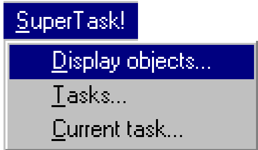
Debugging an application that contains a real-time operating system (RTOS) can be a very big challenge. To overcome this challenge, it is important to be able to view the state of various kernel objects such as events, resources and tasks.

From the perspective of a debugger, it is difficult to assist in this debugging process because every RTOS is different, not only in implementation but also in nomenclature. Some RTOSs, for example, have tasks whereas others have jobs, some have resources whereas others have semaphores. Furthermore, different RTOSs may even use completely different paradigms. Because of these differences, we have adopted an industry standard for providing kernel awareness for your RTOS to Soft-Scope.

If your RTOS vendor supports the Soft-Scope Kernel Awareness Standard, then they can provide you with software for adding support to Soft-Scope for their RTOS. If you wrote your own, then we can provide you with the necessary documentation so you can develop kernel awareness support yourself.

After you have obtained the kernel awareness software from your RTOS vendor, you will need to install it according to their instructions. Once the software has been installed, when you start Soft-Scope, a new kernel awareness menu item will appear on the menu bar. The new item is usually the name of the RTOS you are using.

The following examples were created using the SuperTask! RTOS from US Software.




The kernel awareness pull-down menu provides three commands (**Display objects...**, **Tasks...**, and **Current task...**). Each command opens a dialog box that contains information about the RTOS. The contents of the dialog box varies, depending on how the RTOS vendor chose to implement the Soft-Scope Kernel Awareness Standard.

# Kernel Objects

This dialog box contains details about tasks and other kernel objects such as resources, semaphores, events and mailboxes.

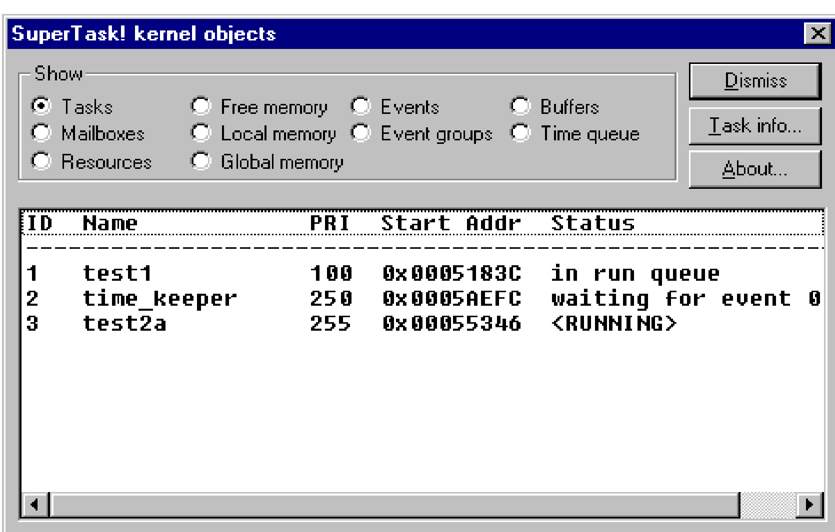



Figure E-1: SuperTask! kernel objects dialog box

# Task List

This dialog box lists the currently existent tasks in your application and their status.

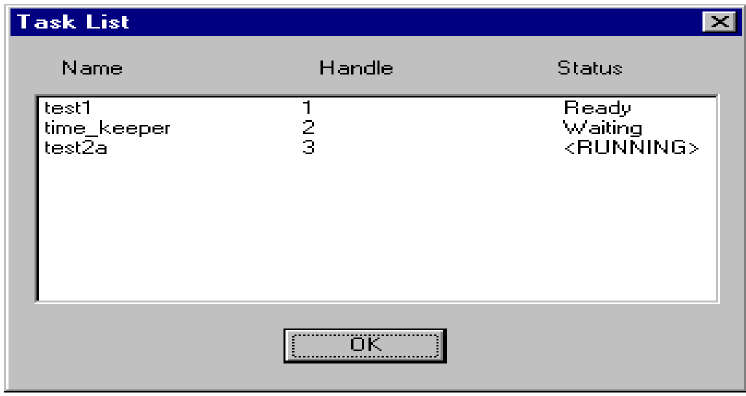



Figure E-2: SuperTask! task list dialog box

# Current Task

This dialog box shows the name and handle of the current task.

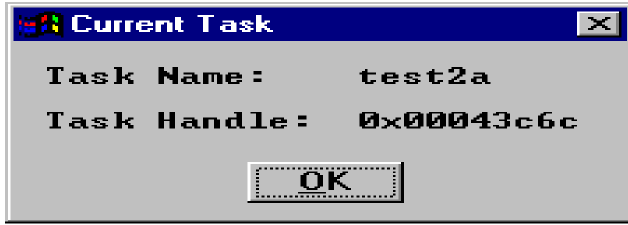



Figure E-3: SuperTask! current task dialog box

For the latest list of supported RTOSs, contact technical sales at (800) 897-3001, (208) 882-0445, or info@consci.com. To obtain a copy of the Kernel Awareness Specification, contact technical support at (208) 882-0445 or tech@consci.com.

(This page blank)

# F. Intel Floating-Point Emulation

## Chapter Contents

**Overview** ........................................................................................... **F-2**
**Intel Floating-Point Emulation** ............................................................ **F-2**

# Overview

This appendix describes how you can configure Soft-Scope to interpret Intel 8087 floating-point emulation. This information is only applicable if you are using Intel tools to build your application.

# Intel Floating-Point Emulation

When you step, Soft-Scope inserts a temporary breakpoint at the end of the instruction to be executed and internally issues a ***GO*** command. Soft-Scope then executes your application at full-speed until it finds a breakpoint, which it does as soon as it prepares to execute the next instruction.

When the target application is linked with a floating-point emulation library instead of an 8087 chip, the linker modifies the numeric instructions in such a way that Soft-Scope cannot determine the end of an instruction, and cannot step correctly.

When the compiler sees a line of code that requires a floating-point instruction, it inserts the 8087 opcode at that point and marks this location for a fixup. A fixup is a compiler-allocated area in the object code where the linker/locator can replace the existing code with other values.

In the case of floating-point emulation, the fixup inserts an interrupt into the op-code of the floating-point instruction. The interrupt number points to a dedicated vector for handling this floating-point instruction.

Floating-point emulation interrupt vectors are a range, one vector per floating-point instruction. Floating-point emulation libraries are linked into the load module. Each floating-point interrupt vector points to the library entry for that floating-point routine. From the interrupt vector given, the emulator can determine the length of this instruction and can execute correctly, but Soft-Scope will need to be informed that numeric emulations are being performed.

For Soft-Scope to function correctly in this environment, if your application contains floating point instructions and uses Intel 8087 instruction emulation (by linking to the libraries **e8087.lib** and **e8087**), you must set **targ.87emulate** to the value of the first interrupt vector. These libraries default to 20 decimal.

If you are using DOS 8087 instruction emulation (by linking to the libraries **de8087.lib** and **de8087**), you must set **targ.87emulate** to the first interrupt vector. These libraries default to 212 decimal.

You may set **targ.87emulate** in either **sswin32.ini** or in the **Command line** dialog box using the ***SET*** command. The former method sets the option every time you load an application.

Using floating-point emulation instead of an 8087 numeric coprocessor causes stepping of floating-point instructions to be slow.

(This page blank)

# Index

## Symbols


# [operator: line-number] 4-12, 5-12
$ [Command-line and text-box syntax:
CPU-structure name prefix] 5-57, A-9
register-name prefix] 5-13, 5-52, A-9
$ [macros: parameter-name prefix and local-variable prefix] 7-10, 7-27
$CPU (CPU variable) 7-5
$cr0-$cr4 registers A-14
$eip register D-3
$NPX (CPU variable) 7-5
$STOPPED (CPU variable) 7-5
& [memory-reference operator: addressof operator] 5-26, 5-37, 5-41, A-8, D-3
() [command metasymbol: alternative or required] 3-17
() [Data/Watch window operator: selector not stored in memory] 5-24
* [data reference operator: pointer-dereference operator] 5-12
+ [upload-file symbol: begins each file record] 5-51
- [command parameter (BREAKPT): delete breakpoint] 4-23, 4-25
-> [Data/Watch window operator pointer] 5-13, 5-18, 5-24, A-8
. [data reference operator:
keyword prefix in commands] 5-19, A-9
structure member selector] 5-61, A-9
. [operator: symbol] 4-12, 5-12, 5-13, 5-27
.. [subscript range operators] 5-20
... [data reference operator: open-ended and closed range of array] 5-21, A-8
... [Data/Watch window symbol: compressed format] 5-17
...x [open-ended operators] 5-21
.abs absolute file 3-21
.exe executable file 3-21
how to debug C-2
.fieldname (command syntax element) 5-52
.ini configuration options file 4-38
.mac macro source file 7-4
.mob compiled macro file 7-4
.mob file. See under extensions.
.omf file 3-21
.tmp files. See temporary files.
: [address operator: selector-offset separator] 5-12, 5-13, 5-25, A-8
: [operator: module] 4-12, 5-12, 5-27, A-8
<Alt>+<B> [opens Break pull-down menu] 3-12
<Alt>+<C> [opens Code pull-down menu] 3-12
<Alt>+<D> [opens Data pull-down menu] 3-12
<Alt>+<F> [opens File pull-down menu] 3-12
<Alt>+<H> [opens Help pull-down menu] 3-12
<Alt>+<M> [opens Macro pull-down menu] 3-12
<Alt>+<O> [opens Options pull-down menu] 3-12
<Alt>+<W> [opens Window pull-down menu] 3-12
<Ctrl>+<A> [captures current window to log file] 3-9, 3-12

<Ctrl>+<C> [cancels current operation] 3-12
<Ctrl>+<End> [displays last page of current window] 3-12
<Ctrl>+<F> [opens Find dialog box] 3-12
<Ctrl>+<Home> [displays first page of current window] 3-12
<Ctrl>+<L> [opens Command line dialog box] 3-12, 7-19
<Ctrl>+<PgDn> [pages down half of current window] 3-13
<Ctrl>+<PgUp> [pages up half of current window] 3-13
<Ctrl>+<Q> [exits Soft-Scope] 3-12
<Ctrl>+<Shift>+<Tab> [moves to previous window in queue] 3-13
<Ctrl>+<Tab> [moves to next window in queue] 3-13
<Ctrl>+<X> [closes active window] 3-12
<F10> [toggles Application I/O window open/closed] 5-64
<Spacebar> [steps once in Code window] 4-4, 4-9
= [Configuration-option assignment operator] 6-6
= [Registers window symbol: subfield used with, has more than one bit] 5-55
= [Symbols window toolbar button: filename assignment for module] 5-35
? [Code window symbol indicates approximated information] 4-10
? [Data/Watch window symbol: indicates uninitialized stack variable] 5-28
? [in displays] 5-21
? [Registers window symbol: register not displayed] 5-53
[] [command metasymbol: optional entries] 3-17
[File] (configuration-file section) 6-6
[Layout] (configuration-layout section) 6-6
\r\n [upload-file symbols: ends each line of binary data] 5-51
{ } [macro WFUNCTION: keyboard-key names] 7-22
| [command metasymbol: alternatives] 3-17
00000000P (default starting address for Dump window data) 5-46

## A

ABORT (macro statement) 7-17
absolute file (.abs) 3-21
accelerator keys
  list of 3-12
ACCESS (keyword) 4-19, 4-25, 4-27
action at a breakpoint 4-21
address
  as code references 4-12
  as macro parameters 7-13
  command syntax element 3-19
  determining 5-37
  format of 4-10, 5-25
  if in RAM or ROM checked for breakpoints 4-26
  LDT as part of 6-11
  linear 5-25
  logical 4-10, 4-12, 4-15, 5-25
    as memory references 5-25
    in Dump window 5-48
    with type overrides 5-40
  mode (display mode) 5-16, 5-32
  of symbolic references and type overrides 5-42
  physical 4-10, 4-12, 4-15, 5-25, 5-26
    in Dump window 5-46

radio buttons
Logical 4-10
Physical 4-10
selector:offset format 5-25
type overrides useful with 5-40, 5-42
ADDRESS (parameter type with macros) 7-13
ADDRESSOF operator (&) 5-26, 5-37, D-3
with type overrides 5-41
application
after load, example of 3-28
full compatibility with Soft-Scope, how to assure,
I/O 5-64
I/O (input/output) window
description of 5-64
loading
confirmation of 2-9, 3-28
how to 3-21
path to, defining 6-10
arithmetic operators 5-6
arr_chg (macro example) 7-4
array
built-in
$VECTOR 5-64
data references to 5-20
entire array 5-20
range of elements of 5-20
single element of 5-20
variable subscripts with 5-21
indexes out of range 5-21
local variables in macros 7-14
number of elements returned by LENGTHOF 5-38
subscript, number base (default) 5-5
arrow (graphic)
outline of (in Code window) 4-11
solid (in Code window) 4-11
assembly
display modes (dialog box)
example of 4-35
radio buttons, Address 4-35
radio buttons, Code 4-35
mode (of Code window) 4-9
example of 4-10
assignment statements
assigning values to complex types 5-24
AT (operator)
examples of use 5-45
use with type overrides 5-41
AUTO (keyword, local-variable declarations, macros) 7-14


## B


backslash (\)
escape character 5-10
base
configuration option 5-4, 6-7
number (default) 5-4
baud rates 2-4, 6-8
binary numbers (with suffix Y) 5-3
bitfield
data references to 5-22
single, data references to 5-22
block of memory, copying from one location to another D-5
Borland 8-2, 8-7
sample applications 8-4
BR[EAKPT] (command) 4-19, 4-23, 4-25
braces ({ }) (used in WFUNCTION for key names) 7-22
BREAK (macro statement) 7-17


I

breakpoint
  access (see also ACCESS) 4-19
  action at 4-21
  addresses used to specify 4-23
  break pull-down menu options
    Access 4-24
    Display 4-16
    Execution... 4-22
    Write... 4-24
  break toolbar button 4-22
  conditions 4-20
  data 4-24
    limitations 4-24
  defining new 4-20
  deleting 4-18, 4-23, 4-25
    hardware 4-24
    software 4-23
    temporary 4-23
  edit (dialog box)
    Addr (options) 4-21
    example of 4-20
    how to open 4-20
    Status (options) 4-20
    Status radio buttons 4-20
    Then (options) 4-21
    Type (options) 4-20
    When (options) 4-21
  editing 4-20–4-21
  exec (see also EXEC) 4-26
  execution 4-22–4-23
    deleting, how to 4-23
    exec breakpoint as special form of 4-26
    if address in RAM or ROM 4-26
    RAM (not all in) 4-26
    specifying, how to 4-22–4-23
  hardware 4-24–4-26
    debug registers 4-25
    example of 4-16
    resources of, used by exec
    breakpoints 4-26
    ROM addresses 4-26
    specifiers 4-27
    two kinds (Data and Exec) 4-24
  how many are set 4-16
  indicators 4-11
  insert 4-18
  limitations 4-25
  modify 4-18
  processor debug registers 4-25
  software
    permanent 4-22
    temporary 4-23
  software (see also breakpoint, execution), 4-22–4-23
  status 4-20
  temporary 4-28
  temporary software 4-8
    how to set and remove 4-23
  then condition 4-21
  types 4-20
  variables, stack-based 4-24
  view 4-18
  when condition 4-21
  window
    adding breakpoints in 4-19
    deleting breakpoints in 4-19, 4-23
    double-click function in 4-18, 4-20
    example of 4-17
    open breakpoint edit dialog box from 4-20
    toolbar button, Delete 4-18, 4-24
    toolbar button, Insert 4-18, 4-20, 4-24
    toolbar button, View 4-18

write (see also WRITE) 4-19
built-in CPU variables
$CPU 7-5
$NPX 7-5
$STOPPED 7-5
buttons. See under toolbar buttons.


## C


C (language)
data types and type overrides 5-40
escape sequences 7-27
expressions as breakpoint conditions 4-21
formatted output function 7-25
operators (see also arithmetic/logical operators)
table of 5-8
Soft-Scope expressions and operators 5-6
statements
break 7-17
if-else 7-17
return 7-18
while 7-18
caching memory, Soft-Scope's 6-14
call sequence 4-30
CALLS (command) 4-31
calls window
double-clicking to display code 4-30
example of 4-31
how to open 4-31
opening (see Code pull-down menu options, Calls) 4-30
capture command
window contents to a file 3-9
case of symbols 6-10
chapter summaries 1-5–1-7
cmd.file (configuration option) 3-9, 6-7
cmd.initial (configuration option) 6-7
cmd.macro (configuration option) 6-7, 7-8
code
caching 6-14
disassembly and CPU type 6-11
location 4-14
pull-down menu options
Calls 4-30
Display 4-7, 4-14, 4-15
Go to... 2-10, 4-15, 4-27
Module 4-13, 5-34
Stop 4-28
Trace 4-34
references 4-12–4-13
as memory references 5-25
double-click to examine 4-7
examples 4-12, 4-22
symbols 4-14
how to find 4-13
window 4-6–4-11
arrow (graphic) 2-10, 4-11
dialog box (see under display modes) 4-9
double-clicking in 4-7
execution pointer in, how to return to 4-14
in assembly mode (example) 4-10
in source mode (example) 4-7
modes of 4-9
octagon (graphic) 4-11
opening from Trace window 4-34
opening with LIST command 4-14
pointer 4-11
scrolling up (backwards) in 4-10
symbols used in 4-11
toolbar button, Break 4-8, 4-22, 4-23
toolbar button, Evaluate (?) 4-9
toolbar button, Go to return 5-38

toolbar button, Locate 4-9, 4-14
toolbar button, Mode 4-4, 4-9
toolbar button, Temp break 4-8, 4-23
coderef (command syntax element) 3-19
codesym (command syntax element) 3-19
com port 2-4, 6-8
comma operator (,) (C language) 5-7
command
examples 4-23, 4-25, 5-35, 5-38, 5-43, 5-52
execute after load 6-9
execute before load 6-7, 6-9
line
commands, list of 3-17
commands, syntax 3-17
dialog box, example 3-8
metasymbols used in syntax statements of 3-17–3-19
Soft-Scope, complete list of 3-17–3-19
syntax elements of 3-19–3-21
communication
device 6-15
host to target 6-8
parameters, serial
how to change 2-4
with monitor 7-20
compiling macros 7-4
compressed format data representation
toggles between, and expanded format 5-17
Concurrent Sciences, Inc.
email address D-2
fax number D-2
phone number D-2
conditional operator (?:) (C language) 5-7
configuration
options (see also individual options)
file (sswin32.ini) 6-3
how to modify a value 6-5
how to save and reload 6-4
list of all 6-6
Windows-type .ini file for 4-38
connect.baudrate (configuration option) 6-8
connect.comport (configuration option) 6-8
control strings (in formatted output macros) 7-25
conventions. See documentation conventions.
conversion specifiers (for formatted output) 7-25
table of 7-26
coprocessor 7-5
copy memory from one location to another D-5
copyright information, Soft-Scope's 2-6
count
number base (default) 5-5
CPU
data types 6-11
structures 5-56–5-59
chip selects 5-56
config control 5-56
descriptor abbreviations 5-59
displayed in Data window 5-56
how to display or view 5-56
how to modify 5-59
interrupt 5-56
page dir 5-56
parallel ports 5-56
power/clock 5-56
refresh control 5-56
serial ports 5-56

timers 5-56
vector 5-56
variables 7-5
CSi-Link 8-5
current task E-4

## D

data
breakpoints. See under breakpoints.
bus
how addresses appear on 5-25
display of in most useful format 5-44
pull-down menu options
CPU structures 5-56
CPU structures (to view vector table) 5-64
Dump 5-46
Examine 5-14, 5-57
Registers 5-52, D-3
Symbols 4-13, 5-34
Watch 5-30
reference 5-19–5-24
double-clicking to examine 4-7
examine with ? command/toolbar button 4-9
example uses of 5-19, 5-20, 5-21, 5-22, 5-24, 5-27, 5-37, 5-38, 5-40, 5-41, 5-42, 5-43, 5-44, D-3
not necessarily memory references 5-25
summary of 5-12
symbols 4-14
types A-2–A-8
for type overrides, subfields used in A-2
for type overrides, table of A204--A209
window 5-14–5-18
CPU structures displayed in 5-56
double-click function in 5-17, 5-23
open display modes (dialog box) [Code window] 4-9
pointer dereferencing in 5-23
question mark in 5-28
toolbar button, Mode 5-15
toolbar button, Modify 5-15, 5-59, D-3
toolbar button, Watch 5-15
dataref (command syntax element) 3-19
datasym (command syntax element) 3-19
debug
.exe executable file C-2
registers 4-25
and hardware breakpoints 4-25
efficient use of 4-24
example usage of 4-25
number of registers used 4-25
DEC (keyword with macro parameters) 7-10
decimal numbers (with suffix T) 5-3
DELETE (keyword) 7-7
deleting
breakpoints 4-23, 4-24
descriptor subfields, table of A-16
device driver, serial 2-4
dialog boxes
Assembly display modes 4-35
Breakpoint edit 4-20
Code reference 4-7
Command line 3-17
Data reference 5-14
Display modes 5-15, 5-31
Dump modes 5-47
File-Load 3-21
File-Restart 3-26
File-Symbols 3-24


I

Find 3-8
specify information for, using macro
WFUNCTION 7-24
direct memory access (DMA) 6-14
disassembling code and CPU type 6-11
display
format of data, with type overrides 5-44
modes (dialog box) [Code window] 4-7
example of 4-9
radio buttons, Address 4-10
radio buttons, Code 4-9
radio buttons, Execution 4-9
modes (dialog box) [Data window]
example of 5-15
opened from Watch window 5-31
radio buttons, Modes 5-16
distribution disks
Soft-Scope 2-3
documentation conventions iv
double-click function 3-14
on data references 3-14, 4-7
to modify existing breakpoint 4-18
to reference pointers 3-15
use in Breakpoints window 4-18, 4-20
use in Code window 3-14, 4-7, 4-14
use in Data window 4-14, 5-17, 5-23
use in Watch window 5-17, 5-23, 5-33
view code for specific call 4-30
dump
modes (dialog box)
example of 5-47
Expand check box 5-48
Modes radio buttons (Byte) 5-48
Modes radio buttons (Dword) 5-48
Modes radio buttons (Hword) 5-48
Modes radio buttons (Word) 5-48
window 5-46–5-49
address format used to open 5-46
default starting address of dump 5-46
example of 5-49
how to open 5-46
modes of 5-47
toolbar buttons, Mode 5-47
toolbar buttons, Modify 5-47
toolbar buttons, Shift 5-47
DUMP (command) 5-49

## E

E (prefix of exponent) 5-4
emulation, floating-point
targ.87emulate F-2, F-3
environment variable 6-18
equal sign (=) toolbar button
(Symbols window) 5-35
error
condition
action at breakpoint 4-21
messages
address B-3
general B-7
escape sequences (for strings) 5-10
table of 5-11
use in (formatted output) control strings
7-27
EVAL (command) 5-16, 5-58
Eval mode (display mode) 5-16, 5-32
examining data 5-83--5-146 5-1
EXEC (keyword) (see also breakpoint, exec)
4-19, 4-26, 4-27
exec.refresh (configuration option) 6-8
exec.wait (configuration option) 6-8
executable
file (.exe) 3-21
instruction, changing D-4

execution
  breakpoint. See under breakpoint.
  events
    displayed in Trace window 4-33, 4-35
  point, current 4-6, 4-9, 4-14
    changing D-3
  pointer 4-13
    how to return Code window to 4-14
  radio buttons
    Into 4-9
    Over 4-9
expanded format data representation (...)
    toggles between, and compressed format 5-17
exponential numbers 5-3
exponents (of floating-point numbers)
  number base (default) 5-5
EXPRESSION (parameter type with macros) 7-12
expressions
  as memory references 5-26
  assignable, as macro parameters 7-12
  complex, used in assignment statements 5-24
  in type overrides 5-43

## F

far16 (value for sym.pointer configuration option) 6-12
far32 (value for sym.pointer configuration option) 6-12
field-width specifiers (in print macros) 7-28
file
  extensions
    .abs 3-21, 8-5
    .bug 8-6, C-2
    .exe 3-21, 8-6
    .exe C-257--C262
    .exp C-2
    .ini 4-38
    .mac 7-4
    .mob 7-4
  names
    backslashes in strings 5-10
  pull-down menu options
    Load 2-8, 5-50
    Restart 3-26
    Upload 5-50
    View log 3-10, 6-7
file-load (dialog box)
  example of 2-9, 3-22
  parts of
    Browse... 3-22
    Command 3-22
    File name 3-22
    Hardware setup 3-22
    History 3-23
    Restart 3-23
    Symbols 3-23
file-restart (dialog box)
  example of 3-26
  parts of
    Browse... 3-27
    Command 3-26
    File name 3-26
    Hardware setup 3-26
    History 3-27
    Load 3-27
    Symbols 3-27
file-symbol load (dialog box)

I

example of 3-24
parts of
Browse... 3-25
Command 3-25
File name 3-25
History 3-25
Load 3-25
Restart 3-25
filename (command syntax element) 3-19
filename.bug (command syntax element) 3-19
files
sample 8-4
temporary. See temporary files.
find (dialog box)
example of 3-8
parts of
Cancel 3-8
Direction 3-8
Find next 3-8
Match case 3-8
Match whole word only 3-8
Find string 3-7
flag format (fltype) 5-41
flags register 5-41
modifying 5-52
floating-point
emulation 6-14
targ.87emulate F-2, F-3
numbers 5-3
fltype (data type in type overrides) 5-41
formatted output (print macros) 7-25
frequently asked questions 1-3–1-4
functions, built-in Soft-Scope 5-37
table of 5-9


## G


global
descriptor table
displaying 5-57
example display 5-58
subfields, table of A-16
variables A-15
symbols
in Symbols window 5-36
go
toolbar button 4-3
until cursor position 4-4
GO (command) 4-27


## H


H (suffix) for hexadecimal numbers 5-3
hardware breakpoint. See under breakpoint. 4-24
HELP (command) 3-16
Help pull-down menu options
About Soft-Scope 3-16
Index 3-16
Using help 3-16
helpful hints for power user D263--D266
HEX (keyword with macro parameters) 7-10
hexadecimal numbers
0x (prefix for) 5-3
H (suffix for) 5-3
hexnumber16 (command syntax element) 3-19
host system requirements 2-3

## I


I/O
  device, memory-mapped 6-14
  port
    reading/writing use PORT 5-38
    restrictions on reading from 5-38
IF (macro statement 7-17
IF...ELSE (macro statement) 7-17
indexes, array 5-20
installation instructions
  host system requirements 2-3
  Soft-Scope for Windows 95/NT 2-4
instruction
  changing D-4
  pointer 2-10
integer data type, size of 6-13
Intel
  ASM86, ASM286 and ASM386 8-7
  BLD286/386 8-8
  BND286/386 8-8
  iC-86, iC-286 and iC-386 8-9
  LINK86/LOC86 8-9
  PL/M-86, PL/M-286 and PL/M-386 8-10
  register subfield names used by
    Soft-Scope come from 5-55
interrupt
  descriptor table
    displaying 5-57
    subfields, table of A-16
    variables A-15
  disable 6-16
  driven CSi-Mon 4-28
INTO (keyword) 4-5
into, stepping 4-3, 4-4
invoking Soft-Scope 2-6–2-7


## K


kernel
  awareness E-3
  objects E-3
key_sequence (with macro function
    WFUNCTION) 7-22
keyboard keys
  activate pull-down menus using 3-12
  concurrent presses of iv
  open dialog boxes using 3-12
  selecting menu commands using 3-12
keyword
  (command parameters)
    as variables in data references 5-19
    use of period (.) with 5-19
  (see also individual keyword)
keyword (command syntax element) 3-19


## L


L (suffix for linear address) 5-12, 5-25
LDTR register 6-11
leading zero flag (in conversion specifiers)
    7-28
LEN[GTH] (operator) 5-43, 5-50
  number base (default) 5-5
LENGTHOF (function) 5-37, 5-38
LINE
  (command) 4-9
  (parameter type with macros) 7-13
line number
  as macro parameter 7-13
  how to find address of D-3
  number base (default) 5-5
  operator (#) 4-12
linenum (command syntax element) 3-19
lineref (command syntax element) 3-19
linker (CSi-Link) 8-5
linking your application 8-5

LinkLoc (Phar Lap linker locator) 8-11
LIST
  (command) 4-14
  (keyword) 7-7
LITERAL (parameter type with macros) 7-11
LOAD
  (command) 3-30
  (keyword) 7-7
load.init_command (configuration option) 6-9, 7-8
  command executed after load 6-9
  example use of D-5
load.init_enable (configuration option) 6-9
load.setup_command (configuration option) 6-9
  command executed before load 6-9
load.setup_enable (configuration option) 6-9
loading
  .exe application C-3–C-5
  application (see application loading)
  memory and registers 5-50
  symbolics only 3-24
local
  descriptor table
    and address displays 6-11
    subfields, table of A-16
    variables A-15
  variables, macro. See under macros.
locator 8-5
log
  file (see also Window menu option, Capture)
    capturing a window to 3-9
    default name 3-9
    specifying name 3-9, 6-7
  window
    how to open 3-10
    log-file name change 6-7
    specify size 3-10, 6-10
    toolbar button, Clear 3-10
log.winsize (configuration option) 3-10, 6-10
logical operators 5-7


## M


MACRO
  (command) 7-7
  (keyword in macro definitions) 7-3
  RESUME (macro statement) 7-19
  SUSPEND (macro statement) 7-19
macro
  assign
    array value 7-15
    pointer value 7-16
  compiling to .mob file 7-4
  CPU variables 7-5
  creating 7-3
  currently loaded 7-6
  deleting 7-6, 7-7
  displaying name in Macros window 7-9
  examples
    arr_chg 7-4, D-4
    printstr 7-12
    src_chg 7-11
    test 7-10, 7-14
  file of, loading 7-6
  files. See sswin32.mac
  functions
    PRINT 7-25, 7-28
    WFUNCTION 7-22, 7-23
    WMOVE 7-22, 7-23
    WPRINTF 7-28
    WRESIZE 7-22, 7-23

initial macro file load 6-7
invoking/running
  from Command line 7-7
  from Macros window 7-6
local variables 7-14–7-16
  any type in table A-1 7-14
  AUTO required in declaration 7-14
  can be arrays (one-dimensional) 7-15
  how to declare 7-14
  names begin with $ 7-15
  pointers 7-15
name customizing for Macros window 7-9
output 7-25, 7-27
parameter types
  ADDRESS 7-13
  EXPRESSION 7-12
  LINE 7-13
  LITERAL 7-11
  MODULE 7-13
  PROCEDURE 7-13
  REFERENCE 7-12
  TEXT 7-12
parameters 7-10–7-13
  any type in table A-1 except arrays 7-10
  names begin with $ 7-10
print macros 7-25–7-28
  control strings 7-25
  conversion characters 7-25
  escape sequences 7-27
  field-width specifiers 7-28
  leading zero flag 7-28
  wprintf 7-28
pull-down menu options
  Display 7-6
  Load 7-6
  Resume 7-19
running 7-6, 7-7
source files of 7-3
statements 7-17–7-21
  _USER_ MONHOLD 7-20
  _USER_ MONITOR 7-20
  ABORT 7-17
  BREAK 7-17
  IF 7-17
  IF...ELSE 7-17
  MACRO RESUME 7-19
  MACRO SUSPEND 7-19
  MONHOLD 7-20
  MONITOR 7-20
  RESPOND 7-18
  RETURN 7-18
  WHILE 7-18
syntax of 7-3
window 7-6–7-9
  customizing names for 7-9
  example of 7-7
  how to open 7-6
  not displaying selected names in 7-9
  toolbar button, Delete 7-6
  toolbar button, Run 7-6
macroname (command syntax element) 3-19
manipulating windows from macros 7-22–7-24
memory
  access size 6-16
  cache flush 6-14
  caching 6-14
  control block search 6-15
  copying a block of from one location to another D-5
  modify 5-46
  reference
    examples of 5-26
  references 5-25–5-26
    summary of 5-12


I

using expressions for 5-26
save in disk file 5-50
write verification 6-17
writes, read-after-write verification of 6-17
memory-mapped I/O device 6-14
memref (command syntax element) 3-19
menu map, Soft-Scope's pull-down
table of 3-4–3-6
message
window
description of 2-6
example of 2-7
MESSAGE (command) 2-6
messages, error
address B-3
general B-7
MetaWare 8-2, 8-10
sample applications 8-4
Microsoft 8-2, 8-10
sample applications 8-4
modname (command syntax element) 3-20
module
names
as macro parameters 7-13
how to assign file to 5-35
how to find 4-13
operator (:) 4-12
MODULE (parameter type with macros) 7-13
MODULES (command) 5-35


## N


near16 (value for sym.pointer
configuration option) 6-12
near32 (value for sym.pointer
configuration option) 6-12
nested calls
return addresses of 5-37
stack usage 4-32
NOP (assembly-language instruction--no operation)
changing instructions to D-4
Normal mode (display mode) 5-16, 5-32
null
character (\0) 5-10
device search 6-16
null-modem configuration 2-11
numbers
base changing 6-7
base of (default) 5-4
formats/bases supported by Soft-Scope 5-3–5-5


## O


octagon (graphic)
outline of (in Code window) 4-11
solid (in Code window) 4-11
octal number, specified with escape sequences 7-27
OFFSETOF (function) 5-37, D-3
OMF file 3-21
open-ended operator (with arrays) (...) 5-21
Operation codes, modify D-4
operators
built-in Soft-Scope 5-6–5-9
precedence of 5-7
Soft-Scope specific, table of A-8–A-9
table of 5-9
three classes of 5-6
offsetof D-3
OPT (keyword with macro parameters) 7-10
optionname (command syntax element) 3-

20
options
configuration file 6-3
pull-down menu options
Display 2-4, 4-29, 6-3
Reload settings 6-4
Save settings 6-4
window
example of 6-5
toolbar button, Delete 6-4
toolbar button, Insert 6-4
toolbar button, Modify 5-4, 6-4
optionvalue (command syntax element) 3-20
out-of-range array indexes 5-21
OVER (keyword) 4-5
over, stepping 4-4

## P

P (suffix for physical address) 4-10, 5-12, 5-25
page
directory
macros for displaying tables of 5-58
variables A-15
table entries, table of A-15
paging
when disabled, linear address equals physical address 5-25
path to application, defining 6-10
PC target CSi-Mon monitor C-3
peripheral control block (PCB)
table of 5-60–5-63
permanent software breakpoints 4-22
Phar Lap
386/ASM 8-12
LinkLoc 8-11
pointers
as type overrides
how to interpret 6-12
data references to 5-17, 5-23
dereferencing 5-23, 5-33
far 6-12
macro local variables as 7-15
near 5-24, 6-12
number base (default) 5-5
PORT (function) 5-37
number base (default) 5-5
port I/O using PORT (command) 5-37
power user, helpful hints for D263--D266
printstr (macro example) 7-12
procedure
call, return from 4-28
call sequence 4-30
names
as macro parameters 7-13
how to find 4-13
referencing in current module 4-12
referencing in different module 4-12
PROCEDURE (parameter type with macros) 7-13
PROCEDURES (command) 5-35
processor type 7-5
pull-down menu map
table of 3-4–3-6

## Q

questions, frequently asked 1-3–1-4
quick contents (table of) iii
quotation marks
in macro functions 7-22
single or double 5-10

I

## R

RAM address
  and breakpoints 4-26
ranges, number base (default) 5-5
read-after-write verification 6-17
real mode
  interrupt vector table 5-64
  structures 5-60–5-63
real-time operating systems support E-2
receiver timeouts D-6
reference
  scoping 5-27–5-28
    rules for 5-28
  summary 5-12
REFERENCE (parameter type with macros) 7-12
referencing structures
  arrays of structures 5-21
  individual elements 5-21
REG (command) 5-53
register
  $cr0-$cr4 A-14
  $eip, changing D-3
  access to 5-52, 5-53
  control 5-52, A-14
  flags A-11
  general-purpose A-10–A-11
  modifying 5-52
  names begin with $ 5-52
  NPX A-13
  protected mode A-14
  save in disk file 5-50
  segment A-12
  subfield
    displays 5-55
    names 5-55
  systems, list of 5-55
  window 5-52–5-55
    display different for different applications 5-55
    example of 5-54
    toolbar button, Modify 5-52
    toolbar button, Watch 5-53
REGISTER[S] (keyword) 5-51
RELOAD (keyword) 6-5
RESPOND (macro statement) 7-18
restart application 3-26
RETURN
  (function) 5-37, 5-38
    number base (default) 5-5
  (keyword) 4-27
  (macro statement) 7-18
return
  from procedure call 4-28
  to calling procedure 4-4
ROM address
  and breakpoints 4-26
RTOS, supported
  list of E-5

## S

S[TEP] (command) 4-5
sample
  applications, directory located in 8-4
  files 8-4
  programs disk 2-3
SAVE (keyword) 6-5
scoping, references 5-27–5-28
screen refresh 6-8
segment limit exceeded D-6
segment:offset format (addresses) 4-10

selector
  and offset 5-24, 5-25
  in parentheses 5-24
  near pointer 5-24
SELECTOROF (function) 5-37
  number base (default) 5-5
serial
  communication parameters
    how to change 2-4
  device driver 2-4
  number, Soft-Scope's 2-6
SET (command) 6-5
setbreak (macro)
  as initial command 7-8
shift-operators (>>, <<), operand of
  number base (default) 5-5
single step 4-4
SIZEOF (function) 5-37
small memory model applications 5-24
Soft-Scope
  commands, table of 3-17–3-19
  copyright information for 2-6, 3-16
  how to invoke 2-6–2-7
  Kernel Awareness Standard E-2
  operators, table of A-8–A-9
  pull-down menu map
    table of 3-4–3-6
  serial number 2-6
  version information for 2-6, 3-16
software breakpoint. See under breakpoint.
source
  code
    displaying sections of 4-15
  files
    and modules 5-35
    path to 6-10
  mode (of Code window) 4-7
specify a number of steps 4-4
src.path (configuration option) 6-10
src.tab (configuration option) 6-10
src_chg (macro example) 7-11
SSBUG (conversion utility) 8-6, C-2
sswin (directory) 2-3
sswin32.exe (executable file) 2-3
sswin32.ini (default configuration file) 6-4
sswin32.log (log file) 3-9, 6-7
sswin32.mac (macro file) 5-58, 6-7, 7-3
STACK (command)
  reset 4-32
  usage 4-32
start target execution 4-27
start-up code
  how to jump over when debugging D-5
status line
  tells when application is loading 2-9, 3-28
step 4-4
  default mode of 4-4
  into 4-4
  into next procedure call 4-3
  over 4-4
  over next procedure call 4-4
  single 4-4
  specify number of steps 4-5
STEP (command) 4-4
STOP (command) 4-28
stop execution toolbar button 4-3, 4-28
strings 5-10–5-11
  and escape sequences 5-10
  where to enter 5-11
structures
  data references to 5-17
  members of, data references to 5-21
subscripts used in data references
  to array elements 5-20

I

superimposing types with type overrides 5-42
SuperTask! E-2
sym.case (configuration option) 6-10
sym.cpu (configuration option) 6-11
sym.ldt (configuration option) 6-11
sym.pointer (configuration option) 6-12
sym.wordsize (configuration option) 5-48, 6-13, A-3, A-5, A-6, A-7
symbol
  case 6-10
  loading 3-24
  names, how to find 4-13
  operator (.) 4-12
  window 5-34–5-36
    display mode described 5-34
    display modes exemplified 5-36
    examples of 5-36
    opening, how to 5-34
    toolbar button, Assign (=) 5-35
    toolbar button, Modules 5-35
    toolbar button, Procedures 5-35
    toolbar button, Symbols 5-35
    toolbar button, View 5-34
    toolbar button, Watch 5-35
    use, to find code references 5-26
symbolic
  and uploading memory and registers 5-51
  information 3-21, 8-6
  loading 3-24
  operators 5-6
SYMBOLS (command) 5-35
syntax
  command 3-17
  elements of commands 3-19
system
  register access 5-53
  requirements 2-3

# T

T (suffix) for decimal numbers 5-3
tab character 6-10
targ.87emulate (configuration option) 6-14, F-2, F-3
targ.cache (configuration option) 6-14
targ.code_cache (configuration option) 6-14
targ.dev (configuration option) 6-15
targ.dos_mcb_end (configuration option) 6-15
targ.dos_mcb_start (configuration option) 6-15
targ.dos_nul_end (configuration option) 6-16
targ.dos_nul_start (configuration option) 6-16
targ.grain (configuration option) 6-16
targ.pcb (configuration option) 5-60
targ.polling (configuration option) 4-29, 6-16
targ.verify (configuration option) 6-17
target execution
  toolbar buttons
    Go 4-3, 4-27
    Go to cursor 4-4
    Go to return 4-4
    Step into 4-3
    Step over 4-4
    Stop 4-3
task list E-4
temp (environment variable) 6-18
temporary breakpoint. See under breakpoint.
temporary files
  location 6-17
  Soft-Scope's 3-29, 6-18

Terminate but Stay Resident (TSR) program C-3
test (macro example) 7-10, 7-14
TEXT (parameter type with macros) 7-12
tmp.path (configuration option) 6-17
TO (keyword) 3-20, 6-5
  environments of use 5-35, 5-49, 7-7
tool
  directives 8-197--8-202
  summary 8-2
toolbar buttons
  Assembly 4-35
  Assign (=) 5-35
  Break 4-8
  Bus 4-35
  Delete 4-18, 5-31, 6-4, 7-6
  Evaluate (?) 4-9
  Go 4-3, 4-27
  Go to cursor 4-4, 4-28
  Go to return 4-4, 4-28
  Insert 4-18, 4-24, 5-31, 6-4
  Locate 4-9, 4-14
  Mode 4-9, 4-35, 5-15, 5-31, 5-47
  Modify 5-15, 5-31, 5-47, 5-52, 6-4, D-3
  Modules 5-35
  Procedures 4-35, 5-35
  Run 7-6
  Shift 5-47
  Source 4-35
  Step into 4-3
  Step over 4-4
  Stop 4-3, 4-28
  Symbols 5-35
  Temp break 4-8, 4-23
  View 4-18, 4-30, 4-34, 5-34
  Watch 5-15, 5-35, 5-53
tools supported by Soft-Scope 8-191--8-202
  table of 8-2
trace
  buffer size limited 4-38
  data access multiple loads 6-18
  file size 6-17
  information across multiple loads 6-18
  temporary file where trace information stored 6-17
  window 4-33–4-38
    example of 4-33, 4-36, 4-37
    how to open 4-34
    toolbar button, Assembly 4-35
    toolbar button, Bus 4-35
    toolbar button, Mode 4-35
    toolbar button, Procedures 4-35
    toolbar button, Source 4-35
    toolbar button, View 4-34
TRACE (command) 4-37
trace.filesize (configuration option) 4-38, 6-17
trace.load (configuration option) 4-38, 6-18
troubleshooting 2-11–2-12
TSR (Terminate but Stay Resident) program C-3
TSS386
  subfields, table of A-17–A-18
TYPE (command) 5-16
type overrides 5-40–5-45
  and Watch window data 5-33
  data types for
    complete list of A204--A209
  expressions in 5-43
  for displaying data in most useful format 5-44
  for memory copying D-5
  not true type conversion 5-40

permissible types with 5-40
complete list of A204--A209 A-2
using pointers in
how to interpret 6-12
Types mode (display mode) 5-16, 5-32
typographical conventions. See
documentation conventions.


## U


unions
data references to 5-22
upload
dialog box
example of entry 5-50
file
format of 5-51
UPLOAD (command) 5-51
uploading memory and registers 5-50–5-51
reloading uploaded memory and
registers 5-50


## V


variables (see also data symbols)
and type overrides 5-40
assigning values to,
using type overrides 5-44
automatic (nonstatic) 5-28
CPU 7-5
data references 5-19
global, referencing 5-27
keywords as, in data references 5-19
protected-mode, table of A-15
referencing, outside current
program context 5-27
register
how to reference 5-29
scalar 5-31
stack-based
and type overrides 5-40
referencing 5-28
uninitialized, with ? in Data window 5-28
static, referencing 5-27
use of subscript in array references 5-21
user-declared as type overrides 5-42
vector table
how to view in Registers window 5-64
real-mode interrupt 5-64
VERSION (command) 3-16
version information, Soft-Scope's 2-6


## W


wait to execute next command 6-8
watch
memory 5-33
window 5-30–5-33
adding references, how to 5-30
and type overrides 5-33
description of 5-17
display modes described 5-31
double-click function in 5-17, 5-23
example of 5-32
how many references can be watched 5-33
opened with Data/Watch command 5-30
pointer dereferencing in 5-23, 5-33
references in, how to add 5-30
toolbar button, Delete 5-31
toolbar button, Insert 5-31
toolbar button, Mode 5-31
toolbar button, Modify 5-31
update frequency 5-30
WATCH (command) 5-33
Watcom 8-2, 8-12
sample applications 8-4

WFUNCTION (macro function) 7-22, 7-23
examples of 7-24
key_sequence in 7-22
WHILE (macro statement) 7-18
window
name (for macro WFUNCTION) 7-22
pull-down menu options
Capture 3-9
Find string 3-7
Layout save 3-11
Soft-Scope's
manipulating, from macros 7-22–7-24
windows
Application I/O 5-64
Breakpoints 4-16
Calls 4-30
Code 4-6
Data 5-14
Dump 5-46
Log 3-10
Macro 7-6
Message 2-6
Options 6-3
Registers 5-52
Symbols 5-34
Trace 4-33
Watch 5-30
WMOVE (macro function) 7-22, 7-23
WPRINTF (macro)
output from 7-27
sending output to log file from 3-9
WRESIZE (macro function) 7-22, 7-23
WRITE (keyword) 4-19, 4-25, 4-27

## Y

Y (suffix) for binary numbers 5-3