The RadiSys logo is a blue rectangular box with the word "RadiSys." in white serif font. A thin black line extends from the right side of the box, connecting to a small circle at the top of a vertical line that runs down the page.

RadiSys.

# Soft-Scope<sup>®</sup> Debugger User's Guide

RadiSys Corporation  
5445 NE Dawson Creek Drive  
Hillsboro, OR 97124  
(503) 615-1100  
FAX: (503) 615-1150  
[www.radisys.com](http://www.radisys.com)  
07-0823-01  
December 1999

EPC, iRMX, INtime, Inside Advantage, and RadiSys are registered trademarks of RadiSys Corporation. Spirit, DAI, DAQ, ASM, Brahma, and SAIB are trademarks of RadiSys Corporation.

Soft-Scope is a registered trademark of Concurrent Sciences, Inc.

Microsoft and MS-DOS are registered trademarks of Microsoft Corporation and Windows 95 is a trademark of Microsoft Corporation.

IBM and PC/AT are registered trademarks of International Business Machines Corporation.

Intel is a registered trademark of Intel Corporation.

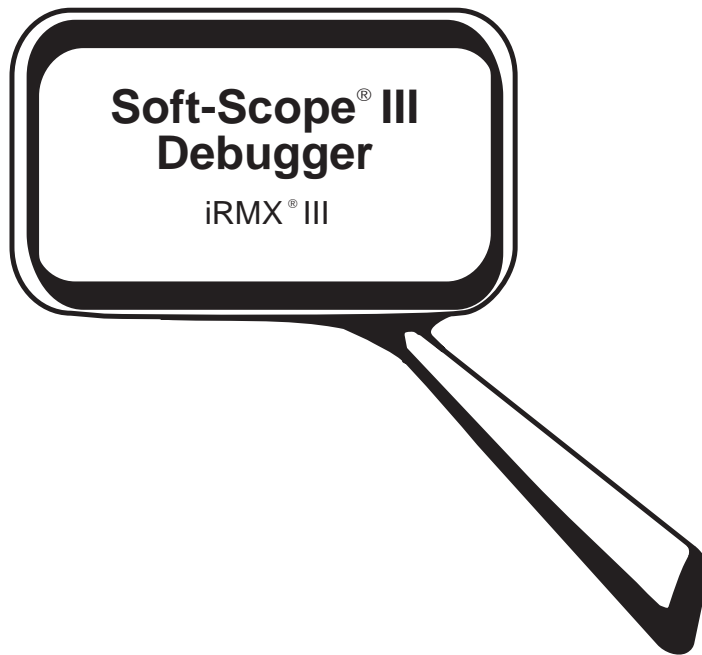
All other trademarks, registered trademarks, service marks, and trade names are property of their respective owners.

December 1999

Copyright © 1999 by RadiSys Corporation

All rights reserved.

**Target Microprocessors**  
**Intel386 and Intel486**



Concurrent Sciences, inc.  
P.O. Box 9666  
Moscow, Idaho 83843  
Phone: (208) 882-0445  
FAX: (208) 882-9774

© 1990 Concurrent Sciences, inc.  
All rights reserved. Third revision, August, 1993.  
Printed in the United States of America.

No part of this document may be copied or reproduced in any form without the prior written consent of Concurrent Sciences, inc.

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 assumes no responsibility for any errors that may appear in this document and makes no commitment to update or keep current the information contained in this document.

The Soft-Scope III debugger runs on any IBM or compatible PC with an Intel386 or Intel486 processor and at least 4 MB of memory and a hard disk.

Soft-Scope III debugs protected-mode, single task loadable (STL) files built with RadiSys tools.

Concurrent Sciences' software products are copyrighted by and shall remain the property of Concurrent Sciences. Use, duplication, or disclosure is subject to restrictions stated in Concurrent Sciences' software license.

Soft-Scope is a registered trademark of Concurrent Sciences, inc.

IBM is a registered trademark of International Business Machines Corporation.

MS-DOS, Microsoft, and Codeview are registered trademarks of Microsoft Corporation.

iRMX is a registered trademarks of RadiSys Corporation.

Intel386 and Intel486 are trademarks of Intel Corporation.

---

# Contents

<b>1 - Introduction .....</b>	<b>1</b>
Special Features .....	2
Manual Organization .....	4
Conventions .....	6
<b>2 - Getting Started .....</b>	<b>7</b>
Installation .....	8
Invoking Soft-Scope III .....	10
Load Applications .....	12
Editing Functions .....	16
Soft-Scope III Help .....	17
Log to a File .....	18
Miscellaneous Commands .....	20
System Debug Commands .....	22
Command Syntax .....	24
Troubleshooting .....	28
<b>3 - Controlling Execution .....</b>	<b>29</b>
Display Code .....	30
Examine a Line of Code .....	34
Stepping .....	36
GO .....	40
Assign Files to Module Names .....	43
Breakpoints .....	44
Disassemble Code .....	48
Stack Information .....	50
Task Manipulation .....	52
Trapping iRMX Exceptions .....	54
Suspend & Resume Tasks .....	57

---

<b>4 - Examining Data .....</b>	<b>59</b>
Data References .....	60
Evaluate Data .....	64
Display Type Information .....	65
Reference Scoping .....	66
Memory References .....	68
Type Overrides .....	70
Dump .....	74
Registers and CPU Structures .....	76
Built-in Functions .....	78
Numbers .....	80
Operators .....	82
Strings.....	84
Reference Summary .....	86
<b>5 - Soft-Scope III Configuration.....</b>	<b>87</b>
Setting Options .....	88
Soft-Scope III Options.....	90
<b>6 - Soft-Scope III Macros .....</b>	<b>93</b>
Create Macros .....	94
Macros Control Statements .....	96
Macro Functions .....	97
Macro Parameters .....	98
Macro Examples .....	99
<b>7 - Tools .....</b>	<b>101</b>
Tools Information.....	102
ASM286 and ASM386.....	103
BND286 and BND386 .....	104
BLD386 .....	105
iC-286 and iC-386 .....	106
PL/M-286 and PL/M-386 .....	107

---

FORTRAN-386 .....	109
<b>Appendix A Tables .....</b>	<b>111</b>
Data Types .....	112
Operators .....	114
General-Purpose Registers .....	115
Flags Register .....	116
Segment Registers .....	116
NPX Registers .....	116
Control Registers .....	117
Protected-Mode Registers .....	117
Descriptors and Subfields .....	118
<b>Appendix B Error Messages .....</b>	<b>123</b>
<b>Appendix C SSKERNEL .....</b>	<b>145</b>
<b>Appendix D C Sample Session .....</b>	<b>149</b>
<b>Index</b>	



# *Introduction*

# 1

The Soft-Scope III debugger is an interactive, source-level, symbolic debugging tool designed to accelerate software development. It is a true dynamic debugger for multi-tasking applications. Soft-Scope III allows you to debug first-level jobs, device drivers, multiple tasks, and jobs created and loaded by your application. You can set software or hardware breakpoints on multiple tasks, and you can monitor which tasks are at break and which tasks are not. Multi-user systems can run up to seven Soft-Scope III sessions at the same time.

## **Table of Contents**

Special Features .....	2
Manual Organization .....	4
Conventions .....	6

# Special Features

---

Soft-Scope III provides a complete array of functions that provide the following features and more:

- Step through and display high-level source statements or assembly-level instructions
- Set software and hardware breakpoints
- Access and modify all application symbols
- Debug multiple file types
- Full support for iRMX tasking capabilities
- View iRMX system data structures
- Examine and modify CPU registers and 286, 386 and 486 protected-mode structures
- No limit on size of files or number of symbols
- Create custom commands
- Trap faults in iRMX subsystems as well as in your application

## ***Assembly***

All breakpoint and high-level stepping operations display the original source code corresponding to the next statement to be executed, and Soft-Scope III's built-in disassembler lets you examine a target program at the assembly as well as the source level. You can execute a program one source or assembly instruction at a time.

## ***Breakpoints***

You can set up to 32 software breakpoints and 4 debug register breakpoints by reference to a symbol name, line number, or absolute address. Software breakpoints halt execution of your program when the address they reference is reached. Debug-register breakpoints halt execution when a given memory location is written to or accessed.

## ***Display and modify Application Symbols***

Symbols include arrays, structures, static variables, based variables, and stack-based variables. Symbols are accessible by their name as declared in your program. You can display the type and scope of each symbol. You can also display memory contents with absolute references or register-relative references.

In addition to STL files, you can debug device drivers and files loaded by your application through RQALOAD.

You can set breakpoints in multiple tasks, determine the status of tasks other than the currently active task, and switch between tasks. You can suspend and resume tasks. You can have multiple users operating separate copies of Soft-Scope III simultaneously on the same CPU. This is possible with multiple terminals or multiple windows on a windowing display.

Use the System Debug Commands (SDB) to view iRMX data structures, such as mailboxes, tasks, jobs, semaphores, segments and regions.

Use Soft-Scope III's REG, and EVAL commands, to examine and modify registers and CPU structures, including the IDT, GDT and LDT.

There are no limits to the size of the listing files, number of lines, or number of symbols in your program.

Soft-Scope III's macro facility lets you combine SSIII commands and expressions to create custom commands specific to your needs. Macros use C-like syntax and declarations, so you needn't learn a special language to use them.

Even when you aren't using SSIII to debug an application, the Soft-Scope kernel will monitor your system and trap faults so they don't interfere with system operation. This is especially handy for multi-user systems, where a General Protection fault can cause the entire system to stop.

***Debug multiple file types***

***Support for tasking capabilities of iRMX***

***View iRMX structures***

***Registers and CPU structures***

***No size limits***

***Soft-Scope III macros***

***Trap faults***

Your Soft-Scope III manual contains the following chapters:

## ***Introduction***

This chapter describes the features of SSIII and provides basic information that will help you use this manual.

## ***Getting Started***

Read this chapter to learn how to install Soft-Scope III. It also contains a description of the Load command so you can load your first application, and a troubleshooting section that you can refer to if you have problems getting SSIII to work. In addition, this chapter contains general descriptions of Soft-Scope III's editing functions and command syntax.

## ***Controlling Execution***

This is a reference chapter describing how to view and execute your application. It describes, in detail, how to reference the source code, single step, step to a specified location, use break-points, and examine procedure call nesting. It also describes how Soft-Scope III lets you debug an application with multiple tasks.

## ***Examining Data***

Read this chapter and learn how to access data, as well as how to use some of the more advanced features of Soft-Scope III. 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 SSIII's built in functions.

## ***Configuring Soft-Scope III***

Soft-Scope III allows you to configure many of its functions and commands to best fit your needs. This chapter provides information about using options and a detailed description of each option provided.

## ***Macros***

This chapter describes SSIII's macro language, which is provided so users can customize the debugger to their specific needs.

## ***Tools***

You might want to refer to this chapter before you start debugging an application. It is a tool-by-tool explanation of what to watch for when preparing applications for debugging.

## ***Appendices***

The appendices provide specific technical information about several topics related to the use of SSIII. Appendix A is a list of error messages and what they mean. Appendix B contains tables of supported data types, registers, and CPU structures. Appendix C describes the Soft-Scope III Kernel, and Appendix D is a C sample session that guides you through debugging an actual application.

# Conventions

---

The following format conventions have been adopted to help you read and understand this manual.

<b>LOAD</b>	Command line commands and key words are all caps.
<b>exec.wait</b>	Setfile options are bold.
<b>SS.SET</b>	Files and pathnames are shown as small capitals in the same font as the rest of the text.
<b>Data reference:</b>	Expressions, Soft-Scope III error messages, and examples are shown in this font and are bold.
<Ctrl>, <Up>, <Left>, <Right>	Keyboard keys are shown enclosed in arrows.
<i>Italics</i>	References to parts of this manual or other publications, and parameters.

Italics are also used for comments, summaries, emphasis, and when the actual content of an expression is unknown. For example, you might see something like the following:

***FILENAME.TMP***

Where filename refers to the name of a file you need or are referencing.

This chapter describes the Soft-Scope III installation, loading, and invocation processes, and what to look for if your SSIII doesn't operate as described. It also explains the SSIII command line, and contains a comprehensive list of all SSIII commands and their syntax.

Using SSIII requires the Soft-Scope III kernel, SSKERNEL, so be sure to read *Invoking Soft-Scope III* before trying to debug.

After you have installed the software, and before you start debugging, we encourage you run the C sample session included on the distribution disks. For instructions describing how to load and run the sample session, see *Appendix D, Sample Session*.

## Table of Contents

Installation .....	8
Invoking Soft-Scope III .....	10
Load Applications .....	12
Editing Functions .....	16
Soft-Scope III Help .....	17
Log to a File .....	18
Miscellaneous Commands .....	20
System Debug Commands .....	22
Command Syntax .....	24
Troubleshooting .....	28

# Installation

## **System requirements**



To install and run Soft-Scope III properly your computer must have a hard drive with at least 2 megabytes of free disk space.

If you purchased iRMX complete with a development kit, Soft-Scope III is installed automatically for you when you install iRMX. Simply choose the following product selection option from the iRMX installation program:

**iRMX for Windows Product with Development Tools**

Then select the line below from the Development Tool Selection Screen:

**Soft-Scope III for iRMX III**

If you purchased Soft-Scope III independently of iRMX, follow the installation directions on the next page.

## **Soft-Scope III files**

The Soft-Scope III installation program is on disk 1. When this program is invoked, it copies the executable and support files into directories iRMX has already created. The Soft-Scope III executable and support files are placed in /UTIL386.

The sample program are copied to /RMX386/DEMO/SSCOPE in the root directory of whatever device your current default working directory is located on.

The information on the distribution disks may be installed with the standard iRMX INSTALL utility.

1. Log on as SUPER for proper access rights.
2. For each of the disks, insert the disk and type:

INSTALL *devicename*

where *devicename* is a physical device. [See *Table 2-1* below for the correct physical device name.]

3. Log off.

## ***Installation procedure***

System type	Device name	Diskette size	Density	Format
MBI and MBII	WDF0	5.25 inch	Low	iRMX uniform
	WQF0	5.25 inch	High	iRMX uniform
PC-BUS systems	A	5.25-inch	High	iRMX uniform
	A	3.5-inch	High	iRMX uniform
	B	5.25-inch	Low	iRMX uniform

***Table 2-1 Physical device names for installation disk drives***

Notice that the physical device name of the low density disk drive is different from the name of the device for iRMXI and iRMXII versions of the Soft-Scope III debugger.

# Invoking Soft-Scope III

## Invocation steps

### SSKERNEL

The steps listed below describe how to invoke Soft-Scope III:

1. Before invoking Soft-Scope III, SSKERNEL must be running as a background job. It can be loaded when you invoke iRMX III by using a SYSLOAD command in your LOADINFO file after the line that loads SDB job:

```
Soft-Scope kernel job
sysload /util386/sskernel
```

Or, load it manually before you invoke Soft-Scope III. At the iRMX prompt, enter the following:

```
BK SSKERNEL >:BB:
```

If you don't type ">:BB:", you will be prompted for a log file.

2. Invoke SSIII using the syntax shown below:

```
SS [filename]
SS [SYMBOLS filename]
SS [optionsfile.set]
```

3. If, for some reason, you want to restart with a fresh SSKERNEL, you will need to kill the old one using the utility SSABORT, not the iRMX KILL command.



Using a wild card kill such as KILL \*, will kill all background jobs, not just jobs associated with Soft-Scope III.

4. SSKERNEL creates a file called SSKERNEL.LOG in the directory which was the default directory when SSKERNEL was invoked. This exists for diagnostic purposes only, but should not be deleted while either SSKERNEL or Soft-Scope III are active.
5. Your display should look similar to *Figure 2-1* on the next page. If it doesn't, read the section, *Troubleshooting*, in this chapter.
6. To quit Soft-Scope III, type EXIT and press <Enter>.

Use the SS command to load your application at the same time you invoke Soft-Scope III:

```
-ss /rmx386/demo/sscope/csamp  
  
Soft-Scope III (tm) debugger v1.0  
Concurrent Sciences, inc. (C) 1989, 1990 All rights reserved  
iRMXIII Version  
Serial no. xxxx  
  
[Connected to "Soft-Scope kernel v1.0 -- session #1"]  
  
[Loading OMF-386 STL file, csamp]
```

The example in *Figure 2-1* shows the message Soft-Scope III displays confirming your application load.

If you enter an OPTIONFILE.SET file on the invocation line, the file you specify is loaded after the default set file, SS.SET, and the options values in the file you specify override the values in the default file. This is handy if you need to use different options for different applications:

```
-ss appl.set
```

*Load your application when you invoke SSIII*

*Figure 2-1 The Soft-Scope III initial display*

*Load an options file*

# Load Applications



***Applications written to run under the Human interface***

***First-level jobs and device drivers***

Soft-Scope III can debug applications loaded in three different ways:

- Applications written to be run under the Human interface (STL files).
- First-level jobs and device drivers in bootable files.
- Programs loaded by your application through RQALOAD().

Use the LOAD command. LOAD syntax is shown below:

```
LOAD filename  
LOAD [SYMBOLS filename]
```

*Filename* is the name of the application you want to debug, including the path.

The following example loads the C sample program (STL file) provided with this software:

```
ss> load /rmx386/demo/sscope/csamp
```

The LOAD command loads symbols, code and data into iRMX III free space, through the Application Loader. Including an application name when you invoke Soft-Scope III initiates this version of the LOAD command.

Use LOAD SYMBOLS to load symbolic information for bootable files, such as first-level jobs or device drivers embedded in the iRMX III boot file:

1. Prepare your application using the iRMX Interactive Configuration Utility (ICU) and BLD386. (See *Chapter 7, BLD386*).
2. Modify the ICU builder file to include symbolics by replacing the NODEBUG option with DEBUG and removing the NOTYPE option.

3. Boot using the the new file, usually from the iSDM prompt after a reset. For example, if the new boot file is BOOT32/RMXTEST.386, type the following:

```
B boot32/rmxtest.386
```

Refer to your iRMX reference materials for boot instructions specific to your system.

4. After the system is booted, invoke Soft-Scope III.

```
-SS
```

5. At the SSIII prompt, load symbols using the syntax shown here:

```
ss> load symbols /boot32/rmxtest.386
```

The following example shows the LOAD SYMBOLS command used for our example program, RMXTEST.386, and SSIII's confirmation of the load:

```
ss>load symbols /boot32/rmxtest.386
```

```
[Attaching OMF-386 bootable file "/boot32/rmxtest.386", Symbols only ]
```

If you issue the LOAD command a second time in a single SSIII session without the SYMBOLS qualifier, the results are unpredictable.

When you need to reload your application, including data and registers, exit Soft-Scope III, reinvoke, and re-load your application.



# Load Applications

## *Device driver example*

If RMXTEST.386 contained a device driver that you wanted to test, you could do the following:

1. Write a Human Interface program that exercises the device driver, e.g., DDTEST.
2. Follow the directions to build, boot, and load the device driver symbols
3. Set a breakpoint in the device driver in an area you wish to debug.
3. Load your Human Interface test program:  

```
ss> load ddtest
```
4. Perform execution with STEP or GO.
5. When the breakpoint is hit, return to the context of the device driver by either using the TASK macro or loading RMXTEST.386 symbols again.
6. Likewise to re-examine the context of the Human Interface program DDTEST, use the TASK macro or load DDTEST's symbols again.

Alternately, if you have two terminals available, do this:

1. Invoke `ss ddtest` on one terminal.
2. Invoke `ss symbols boot32\rmxtest.386` on the other.

This way, you wouldn't have to switch back and forth with the TASK macro.

Use the macro `LOADSEGS` to access and debug files loaded by your application through the iRMX system call `RQALOAD()`. This macro also loads the symbolic information for the file specified.

`LOADSEGS [segtoken jobtoken filename]`

See *Table 2-2 Syntax Elements* for a description of command parameters.

To determine what *segtoken* should be, examine the number returned to `RQALOAD()` via a mailbox. *Jobtoken* is the number returned directly by the `RQECREATEIJOB()` system call, and *filename* is the file passed to `RQALOAD`.

The following example prepares CSAMP for debugging:

```
LOADSEGS 5C18 4F00 :HOME:TEST/CSAMP
```

```
ss> loadsegs 5c18 4f00 :home:test/camp
[ New definition added ]
[ Loading OMF-386 STL file ":home:csamp", Symbols ]
```

*Segtoken* and *jobtoken* are saved by `SSKERNEL`, and, unless you reboot your system, you can access the file a second time by typing `LOAD SYMBOLS filename`, or by using the `TASK` macro to select a task in that application.

`LOADSEGS` with no parameters lists those files which have previously been loaded with `LOADSEGS` or which have been loaded by Soft-Scope III (i.e., those files for which `SSKERNEL` is storing the loader result segment).

***Files loaded by your application***



***Example***

***Just change tasks to load a file the second time***

# Editing Functions

## The command line

Enter Soft-Scope III commands one line at a time. Each line is buffered (up to 80 characters) until you hit <Enter>. When you press <Enter>, if the command is not syntactically correct, Soft-Scope III generates an error message. An explanation of each Soft-Scope III error message is found in *Appendix A*.

The Soft-Scope III command line supports the editing functions listed below.



Many of the following function keys perform specific functions when you are using the LIST command. See the *Display Code* section of *Chapter 3, Controlling Execution* for more information.

## Deleting Text

<code>^F</code>	delete character under the cursor.
<code>&lt;Back Space&gt;</code>	delete character left of cursor.
<code>^A</code>	delete portion of line after cursor.
<code>^X</code>	delete portion of line before cursor.
<code>&lt;Escape&gt;</code>	delete an entire line.

## Moving the Cursor

<code>&lt;Left Arrow&gt;</code>	left one character.
<code>&lt;Right Arrow&gt;</code>	right one character.

## Command History

<code>&lt;Up Arrow&gt;</code>	Pressed once: recalls history entry. Pressed repetitively: scans back through history.
<code>&lt;Down Arrow&gt;</code>	Pressed once: recalls history entry. Pressed repetitively: scans forward through history.

HELP provides on-line assistance with Soft-Scope III syntax and usage. Each Soft-Scope III command has a HELP entry associated with it.

## **HELP** [*topic*]

HELP with no parameters displays the command syntax summary, as well as a list of other topics for which help text is available.

Soft-Scope III finds the help information in the file SS.HLP, which must be in the same directory as Soft-Scope III. If the information to be displayed is more than will fit on one screen, you are prompted with the following prompt:

```
[ More(sp, cr, 1..9) Quit ]
```

Your possible responses are:

<b>Key</b>	<b>Function</b>
<spacebar>	Display another full screen.
<carriage return>	Display exactly one more line.
<Q>	Return you to the Soft-Scope III prompt.
<1> - <9>	Display that many more lines.



## Log to a File

SS>  
LOG

### Turn logging on and off

Use LOG *filename* to create or open a file and begin copying most Soft-Scope I/O to that file.

```
LOG [devicename | filename]  
LOG ON | OFF
```

Begin logging Soft-Scope III commands and displays to the file SS\_TRACE:

```
ss> log ss_trace  
[ Log file "ss_trace" is on ]
```

If the file already exists, Soft-Scope warns you and asks if you really want to overwrite the file or append output to the end of it:

```
ss> log xxx  
[ File exists - Append Overwrite Quit ]
```

Use LOG with no parameters to see if you are logging:

```
ss> log  
[ Log file "ss_trace" is on ]
```

Once a log file is established, you can turn logging ON and OFF by issuing LOG ON and LOG OFF commands:

```
ss> log off  
[ Log file "ss_trace" is off ]
```

Resume logging to the current log device:

```
ss> log on  
[ Log file "ss_trace" is on ]
```

LOG ON restarts logging to the previously-declared log device.

LOG will not record the prompts or messages that disappear when the next command is issued. The exceptions to this are the log prompts and messages:

- [ Log file "logfile" is on]
- [ Log file "logfile" is off ]

When you are using the LIST command, only the last screenful of listing generated by any specific LIST command will be sent to the log file.

To type a comment for the log file from the SS> prompt in a debugging session, begin the comment with /\* and end it with \*/:

```
SS> /* This will appear in the current log file */
```

### **LOG limitations**



**Put comments in your log file**

See also: *Display Code, Chapter 3*  
*Stepping, Chapter 3*

## Miscellaneous Commands



Use the CONSOLE command to redirect Soft-Scope III output to a second terminal.

**CONSOLE** [*devicename* [*termtype*]]

*Termtype* must be defined in the file :CONFIG:TERMCAP. Also, the terminal identified by *devicename* must not be listed as one of the user terminals in :CONFIG:TERMINALS. See your iRMX documentation.

If no *termtype* is specified, Soft-Scope III assumes the second terminal is the same type as the first.

CONSOLE with no parameters directs output back to the original terminal.

The example below redirects Soft-Scope III output to device t1. You are directed to press <Enter> on the t1 device to check the connection:

```
ss>console t1
[ press return on "t1" within 60 seconds ]
[ Other terminal ("t1") now active ]
```



SYSTEM directs commands to the operating system. When a given command is completed, the Soft-Scope III prompt is displayed.

**SYSTEM** *program*

The example below displays the contents of the current directory:

```
ss>system dir
cmain.c      csamp.bnd   csamp.h     cutils.c
```

The VERSION command displays Soft-Scope III's version number and information about the host operating system. Use the syntax below:

### VERSION

The following example demonstrates the VERSION command display:

```
ss>version
Soft-Scope III (tm) debugger, v1.0
Concurrent Sciences, inc. (C) 1989, 1990 All rights reserved
iRMX III Version
Serial No. xxxx
```

You can exit Soft-Scope III with either the QUIT or EXIT commands. They both return you to the system command level. Syntax is shown below:

EXIT  
QUIT

When you exit Soft-Scope III, all of the debugger work files are deleted except for a work file containing symbolic information about the application you were debugging.

This file has the same file name as your application, but has the extension, TMP. When you invoke Soft-Scope III it looks for this initialization information. The first time you load an application, this information is not available, so Soft-Scope initializes the data and builds the file. Because of this, subsequent loads are faster than the first one.

If you want, and disk space is a consideration, you can erase the initialization file after every session. The only penalty is slower loads.



# System Debug Commands

***SDB commands make the Macro window an additional command menu***

iRMX III includes SDB commands that make it possible to view iRMX system information, including jobs, and tasks mailboxes, and other iRMX objects. Soft-Scope III supports SDB commands through its macro feature.

The set option, **cmd.macro**, loads the SDB commands into the Macro window by default when you invoke Soft-Scope III, along with some generic macros we include as examples for you to examine and use.

The generic macros are in the file, SS.MAC, and the SDB commands are in the file SDB.MAC. In your Options window, look for the following assignment:

```
cmd.macro=ss.mac;sdb.mac
```

If you want to write your own macros, you can either add them to SS.MAC, or you can start your own macros file and include it as part of the options assignment. Simply separate file names with a semi-colon.

If you would like, you can use the MACRO command and load your macro file after SSIII is up and running. See *Chapter 6, Soft-Scope III Macros* for more information about writing and running macros.

***What do the SDB commands do?***

Syntax and usage for each SDB command is included in the iRMX Operating System documentation and is the last manual in this book.

To give you an idea which commands you should look up in the reference manual when you need to, and to help you learn what the SDB commands do, we have compiled *Table 2-2*, on the next page.

# System Debug Commands

Command	Description
<b>vb</b>	Displays DUIB information for a physical device in the system configuration. For example, the WMF0 device
<b>vc</b>	Displays system call information
<b>vd</b>	Displays a job's object directory
<b>vf</b>	Displays number of Global Descriptor Table (GDB) slots available
<b>vh</b>	Lists the SDB commands with their parameters and short descriptions
<b>vj</b>	Displays the tokens in the job tree hierarchy beginning with a specified job token
<b>vk</b>	Displays the tokens for ready and sleeping tasks
<b>vmf</b>	Disables or enables timeout
<b>vmi</b>	Displays input messages
<b>vmo</b>	Displays output messages
<b>vo</b>	Displays information about the objects in a job
<b>vr</b>	Displays an IORS segment, which contains information about the last I/O operation
<b>vs</b>	Displays the number of stack elements or all of the stack contents
<b>vt</b>	Displays object information, depending on the type of the object:
Job	Including tasks associated with the job and memory usage
Task	Both interrupt and non-interrupt tasks
Mailbox	With no queue, with task queue, with object queue, and with data queue
Semaphore	For semaphores with no queue and semaphores with task queue
Region	For regions with no queue and regions with task queue
Segment	Including segment size and containing job
Extension	Including type, containing job, and deletion mailbox
Composite	Non-BIOS, BIOS user object, BIOS physical file connection, BIOS stream file connection BIOS named file connection, BIOS remote file connection, BIOS EDOS file connection, Signal Protocol Port, and Data Transport Protocol Port
<b>vu</b>	Displays system calls in a task's stack

**Table 2-2 SDB command functions**

# Command Syntax

## ***Entering Commands***

Soft-Scope III ignores tabs and extra spaces, so you can use them freely in commands.

Although, for emphasis, command names and keywords are shown in this manual in upper case, Soft-Scope III recognizes commands entered in either lower-case or upper-case characters.

If a syntax parameter is surrounded in square brackets (“[]”), it is optional. The vertical bar (|) signifies that a command can include one of two options, but not both.

For example, the syntax for the DISASM (Disassemble) command is:

```
[count] DISASM [ALL] [NOLINES] [coderef] [TO coderef]
```

DISASM is the command word. ALL, NOLINES, and TO are keywords. *Coderef* and *count* are optional parameters.

Soft-Scope III syntax elements are shown in *Table 2-3*.

<i>TO</i>	Functions execute from the reference back. For example, LIST <i>TO reference</i> places the reference at the bottom of the display and fills the upper part of the screen with what comes before the reference
<i>count</i>	An integer in the range 1 to 32,767, function is command specific
<i>dataref</i>	<i>coderef</i> <i>memref</i> <i>lineref</i>
<i>coderef</i>	<i>address</i> [: <i>modname</i> ]# <i>linenum</i> [: <i>modname</i> .] <i>codesym</i>
<i>memref</i>	<i>address</i> <i>lineref</i> [: <i>modname</i> .][ <i>codesym</i> .] <i>datasym</i>
<i>lineref</i>	: <i>modname</i> [: <i>modname</i> ]# <i>linenum</i> [: <i>modname</i> .] <i>codesym</i>
<i>address</i>	A logical, physical, or linear address
<i>modname</i>	A module name
<i>linenum</i>	A line number found in the current module or in <i>modname</i>
<i>codesym</i>	The name of a procedure or label
<i>datasym</i>	The name of a symbol
<i>devicename</i>	A module name
<i>filename</i>	A system-dependent identifier for a disk file
<i>macroname</i>	The name of a macro from the currently loaded macros
<i>tasktoken</i>	A task identifier, <i>hexnumber16</i> <i>dataref</i>
<i>Segtoken</i>	A 16-bit hexadecimal number
<i>Jobtoken</i>	A 16-bit hexadecimal number
<i>token</i>	<i>dataref</i>
<i>termtype</i>	The physical type of a terminal
<i>loadresult</i>	A 16-bit selector. Load structure is found at <i>loadresult:0000</i>
<i>optionname</i>	The name of a Soft-Scope III set option
<i>optionvalue</i>	The value of a Soft-Scope III set option
<i>keyword</i>	A word to use for a Help search
<i>hexnumber16</i>	A 16-bit hexadecimal number
<i>decnumber</i>	A 32-bit unsigned integer

**Table 2-3 Syntax elements**

# Command Syntax

**Table 2-4 Command Syntax Summary**

Use the syntax shown in *Table 2-4* on the command line and in macros. For your convenience, we have included a page number where you can read more about each command.

Command	Page
BPSCOPE [ TASK   JOB   GLOBAL ]	44
BPTIMEOUT [ <i>decnumber32</i> ]	103
BREAKPT [-] [ <i>coderef</i> ] [ TASK   JOB   GLOBAL ]	44
BREAKPT [-] WRITE ACCESS <i>memref</i> [ TASK   JOB   GLOBAL ]	
CONSOLE [ <i>devicename</i> [ <i>termtype</i> ]]	20
[ <i>count</i> ] DISASM [ALL] [NOLINES] [ <i>coderef</i> ] [TO <i>coderef</i> ]	48
[ <i>count</i> ] DUMP [ BYTE   WORD   DWORD ] [ <i>memref</i> ]	74
DUMP [ BYTE   WORD   DWORD ] <i>memref</i> [TO <i>memref</i> ]	
EVAL [ <i>memref</i>   <i>coderef</i> ]	64
EXIT	21
GO [WRITE   ACCESS] <i>memref</i>	40
GO <i>coderef</i>	
GO RETURN	
HELP [ <i>topic</i> ]	17
LINE [ <i>coderef</i> ]	34
[ <i>count</i> ] LIST [ <i>lineref</i>   TO <i>lineref</i> ]	30
LIST <i>lineref</i> TO <i>lineref</i>	
LOAD <i>filename</i>	12
LOAD [SYMBOLS <i>filename</i> ]	
LOADSEGS <i>segtoken jobtoken filename</i>	15
LOG [ <i>devicename</i>   <i>filename</i> ]	18
LOG ON   OFF	

---

<b>Command</b>	<b>Page</b>
MACRO [LIST]	94
MACRO LOAD <i>filename</i>	
MACRO DELETE [ <i>macroname</i> ]	
MACRO STEP [ <i>macroname</i> ]	
MODULE [: <i>modname</i> = <i>filename</i> ]	43
QUIT	21
REG [ALL ]	76
RESUME	57
SET [ <i>optionname</i> [= <i>optionvalue</i> ]]	88
[ <i>count</i> ] STACK [TRACE] [LINES]	50
STACK USAGE   RESET	
[ <i>count</i> ] STEP [ASM] [INTO]	36
SUSPEND <i>tasktoken</i>	58
SYSTEM <i>program</i>	20
TASK [ <i>tasktoken</i> ]   [ALL]	52
TYPE [ <i>memref</i>   <i>coderef</i> ]	65
VERSION	21

# Troubleshooting

## ***Installation problems***

You need to read this section if:

- ❑ You can't run Soft-Scope III.
- ❑ Soft-Scope III runs, but you can't load an application.

**Symptom:** When you tried to load your application Soft-Scope III wasn't invoked and you did not get the Soft-Scope III prompt (ss>).

**Probable Causes:** Your system is not pathed to Soft-Scope III or your application. Try typing the path name with the invocation.

Soft-Scope III wasn't installed properly. Make sure the file, SS, is in /UTIL386. If you can't locate the problem, call the RadiSys technical support number supplied with your iRMX development set.

## ***Memory problems***

**Symptom:** Soft-Scope III is unable to load itself (or the application) and errors are generated.

**Probable Causes:** Soft-Scope III requires almost 690k even before the application is loaded, 530k for Soft-Scope III itself, and 160k for the Soft-Scope III kernel (SSKERNEL).

The "Unable to communicate with SSKERNEL" message may mean your stack size is too small. Try increasing the stack size with the SEGSIZE control.

SSKERNEL will fail if you try to load an 80286 application compiled in SMALL model.

# *Controlling Execution*

# 3

With Soft-Scope III you can monitor your code while executing at the source or assembly level. You can execute one line at a time, or to a pre-determined location, and you can set hardware and software breakpoints on a single task, a group of tasks in one or more jobs, or globally.

## Table of Contents

Display Code .....	30
Examine a Line of Code .....	34
Stepping .....	36
GO .....	40
Assign Files to Module Names .....	43
Breakpoints .....	44
Disassemble Code .....	48
Stack Information .....	50
Task Manipulation .....	52
Trapping iRMX Exceptions .....	54
Suspend & Resume tasks .....	57

## Display Code

**SS>**  
**LIST**

Use LIST to display source lines from a module's listing, or find a specified string in a source file. Soft-Scope III uses the lines from the compiler-generated listing file, or in the case of iC-286 the source file. Every source line has a line number associated with it that can be used as a lineref for a LIST command.

```
[count] LIST [lineref | TO lineref]
        LIST lineref TO lineref
```

### List specified lines

The abbreviation for LIST is "L".

Specify a starting and ending line reference to list all lines between two points:

```
ss> list #18 to #25
```

### List to a reference

LIST TO *lineref* displays as much of the source listing as can fit on one screen. The line specified by *lineref* is the last line listed:

```
ss> list to :cmain.count_task#2807
```

### Specify the number of lines to list

Use the *count* parameter to tell Soft-Scope III to list a specified number of lines:

```
ss> 10 list
```

When you use a form of the LIST command that doesn't use up the entire display, the area of the screen used for the listing display becomes a listing window. You can scroll within this window.

### The list command line

Any form of the LIST command puts you into an interactive list mode. After the last line listed, you will see a command line in the following format:

```
[ location :modname ( keys ) Mode -Find quit ]
```

LIST functions are described in *Table 3-1*.

If source file lines are greater than 80 columns wide, Soft-Scope III truncates the displayed listing. You see the first 79 columns, and an exclamation point (!) in column 80, signifying that Soft-Scope III truncated the rest of the line. The Find facility will, however, search for strings that go past column 79.

If you are logging output to a file or device when you use the LIST command, only the last screen display will be recorded. If you do want to log multiple screens of listing, issue multiple LIST commands, exiting after each full display.



# Display Code

**Table 3-1**  
**Interactive LIST**  
**function**  
**descriptions**

<b>Field</b>	<b>Possible</b>	<b>Description</b>
<i>location</i>	Top of	You are at the top of the module, and the module contains more lines than can be displayed. You can list only downward.
	End of	You are at the bottom of the module, and the module contains more lines than can be displayed. You can list only upward.
	All of	The entire module is displayed.
	Module:	You are in the middle of the module, and the module contains more lines than can be displayed. You can list upward or downward.
<i>:modname</i>		This is the name of the module you are listing. To list another module, exit this LIST command and type "LIST : <i>module</i> " using the new module name.
<i>keys</i>	cr	Press <Enter> to display the next listing line.
	1..9	Display one to nine more lines of the listing by pressing 1 through 9.
	sp	Press <Spacebar> to display downward one screen's worth of listing.
	+	This key is active when the listing window is smaller than the screen size. Pressing + expands the window by one line and displays downward the next listing line.

*Table 3-1 continued*

<b>Field</b>	<b>Possible Displays</b>	<b>Description</b>
<i>Mode</i>		Press M to display the mode that the FIND and -FIND options use for string searches, Case or NOCASE. Case differentiates between upper- and lower-case letters. Nocase ignores this difference. The option shown in all capital letters is the current default. Press <i>&lt;Spacebar&gt;</i> or <i>&lt;Esc&gt;</i> to keep the existing value, C to change from Nocase to Case, and N to change from Case to Nocase.
Find		This is the search option. Pressing F places a highlight bar over the first listing line displayed and prompts: [ String “”]. Enter the string you are searching for. The last 10 search strings issued are buffered. The last search string used is pre-stuffed into the string field. Press <i>&lt;Up Arrow&gt;</i> to scroll back through previous search strings. Press <i>&lt;Esc&gt;</i> or <i>&lt;Down Arrow&gt;</i> to clear the search string. Press <i>&lt;Enter&gt;</i> to start the forward (downward) search through the module.
-Find		Pressing - places a highlight bar over the last listing line displayed, and searches backward (upward) through the module. Except for direction, forward and backward searches function identically.
<i>quit</i>	Quit	Pressing Q exits the interactive LIST mode and returns you to the Soft-Scope III prompt.
	Quit(sp)	You are at the end of a module, or the entire module is displayed. Press Q or <i>&lt;Spacebar&gt;</i> to exit interactive LIST mode and return to the Soft-Scope III prompt.

## Examine a Line of Code



**Display the current execution point**

**Use LINE with an address**

**Use LINE with a line number**

LINE directs Soft-Scope III to display as much information as it can about the code you reference. The display includes some or all of the following:

- line number
- module name
- procedure name
- source line or assembly instruction

LINE [*coderef*]

LINE is the default command. You can invoke it by just pressing <Enter>. This is equivalent to LINE \$CS:\$EIP, and displays the current execution point.

If you don't specify a *coderef*, Soft-Scope III opens the module that contains the current execution point and displays that line:

```
ss> line
[ Inside :CUTILS.sample#88 ]
#88          return &cmd_buffers [cmd_count++];
```

LINE with a code reference displays the line associated with the reference. The following displays the line identified by the current return address:

```
ss> return
000c:000e
ss> line 0c:0e
[ :CMAIN.MAIN#17 ]
#16          do {
#17          c = sample ();
```

If you specify a module that hasn't been opened, Soft-Scope III opens it.

If you know the line you want to see, use LINE with a line number:

```
ss> line :cutils.sample#88
[ Inside :CUTILS.sample#88 ]
#88          return &cmd_buffers [cmd_count++];
```

The output of the LINE command depends entirely on what Soft-Scope III can determine about the execution address. You may see any one of the following types of displays :

- ❑ If the address is exactly at the beginning of a line, LINE displays the module name, the procedure name (if within a procedure), and the listing line:

```
[ :TESTIO.LINEOUT#135 ]  
135: CALL CO(lower_block); /* Block of name. */
```

- ❑ If the address is in the middle of a line, LINE displays the message “Inside,” and the module name the procedure name (if within a procedure), and the listing line.

```
[ Inside :TESTIO.LINEOUT#135 ]  
135: CALL CO(lower_block); /* Block of name. */
```

- ❑ If Soft-Scope III cannot find debug information for the execution point or reference, only the address and the disassembly are displayed:

```
[In Unknown Module]  
5471:0134 mov ax,bx
```

If Soft-Scope III’s current context (given by the TASK command) is a task which is not at a breakpoint, the values of all the registers are given as 0, since registers cannot be determined for a task not at break. A side effect of this is that the following is displayed if the LINE command is given:

```
[ In unknown module (0000:00000000)  
< Address 0000 -> GDT[0] - Segment not present >
```

This message does not indicate any problem with Soft-Scope III or the application. It simply means that the task you are examining is not at a breakpoint.

### *LINE command output*

### *The LINE command and running tasks*

# Stepping

SS>  
STEP

## Initiate STEPPING mode

The STEP command executes source code one or more lines at a time. If execution doesn't start at the beginning of a line, the "[ Inside ]" prompt displays, telling you that the first step began in the middle of the assembly code generated for that line.

[count] STEP [ASM] [INTO]

Abbreviation: **S**

STEP	Steps over all calls
STEP INTO	Steps into all calls
STEP ASM	steps in assembly-language increments and displays disassembled instructions.

Use *count* STEP to execute a specific number of steps. This form of STEP is useful in macros. For example, the following steps 10 times.

```
10 step
```

If Soft-Scope III encounters a breakpoint set with the BREAKPT command before count steps are executed, execution stops.

When you issue STEP you initiate the default stepping mode. The next line to be executed and a menu bar display on your monitor. The menu bar gives you the options shown on the bottom of the following example:

```
ss> step
#30      main ()
#31      {

[ Auto Into OVER(sp,1..9) Mode Quit Return ]
```

The string "(sp, 1..9)" next to the INTO or OVER prompt, and either INTO or OVER displayed in all capital letters indicates the stepping mode.

Table 3-2 contains a description of each element in the menu bar.

Option	Function
AUTO	Press A to step continuously until any key is pressed.
INTO	Press I to execute the current line or instruction, stepping into encountered procedure calls.
OVER	Press O to execute the current line, stepping over all calls in the line.
sp	Press <spacebar> to step once.
1..9	Press 1 through 9 to step through that number of statements.
MODE	Press M to change stepping modes.
QUIT	Press Q to exit to the command line.
RETURN	Press R to go until execution returns from the current procedure.

**Table 3-2 Step menu options**

Soft-Scope III sets a temporary breakpoint each time you step OVER a procedure call. If you step over a procedure call that causes the current task to be suspended or be put to sleep (such as RQSLEEP()), or that takes a long time to execute, you may exceed the value of BPTIMEOUT. When this occurs Soft-Scope III reports the following message and prints a running prompt:

```
<TASK running>
!ss>
```

You cannot continue stepping in the current task until the task hits the breakpoint set by the STEP command.

Press M to change the default mode. Then press S or A to step in source or assembly, and I or O to step into or over.

**Change the default stepping mode**

```
[ Auto INTO(sp,1..9) Over Mode Quit Return ] "M"
steps [Source ASSEMBLY] calls [INTO Over] "S"
#31     {
#32         struct cmd *c;
#33         for (;;) {
#34             init ();
```

# Stepping

## *Override the default stepping mode*

The INTO and OVER options allow you to override the default stepping mode. Selecting INTO when the default is over steps into any procedures called by the line executed. Selecting OVER when the default is into works the same way.

```
ss> step into asm
#32          struct cmd *c;
#33          for (;;) {
#34              init ();
              0200:00000006 call  :CUTILS.init();    $+21
[ Auto INTO(sp,1..9) Over Mode Quit Return ] "O"
[ Returning inside :CMAIN.main#35 ]
#34          do {
#35              c = sample ();
              0200:00000020 push  ebp
```

Soft-Scope III displays the first assembly instruction, a call to the procedure **init** in the module **cutils**. Pressing O overrides the default of stepping into all calls. The program executes until the return to **cmain** is reached.

## *Specify the defaults when you invoke*

When you invoke STEP, the default is OVER and SOURCE unless you specify otherwise using the keywords:

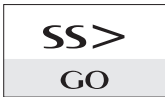
```
ss> step into asm
#32          struct cmd *c;
#33          for (;;) {
```

The following example uses the INTO keyword, and demonstrates the spacebar and return features:

```
ss> step into
#30     main ()
#31     {
[ Auto INTO(sp,1..9) Over Mode Quit Return ] "spacebar"
#32         struct cmd *c;
#33         for (;;) {
#34             init ();
[ Auto INTO(sp,1..9) Over Mode Quit Return ] "spacebar"
[ Entering :CUTILS.init() ]
#43     void init ()
#44     {
[ Auto INTO(sp,1..9) Over Mode Quit Return ] "R"
[ Returning inside :CMAIN.main#35 ]
#45     do {
#46         c = sample ();
[ Auto INTO(sp,1..9) Over Mode Quit Return ] "Q"
```

Pressing <Spacebar> executes line 31 (the entry to the procedure **main**). Pressing it again executes line 34, the call into procedure **init**. Pressing R caused the rest of the procedure to execute until the return to **main**.

*Step in source mode, INTO all calls*



***GO [reference] sets a temporary breakpoint and executes to it***

***Go with no breaks can go indefinitely***

The GO command tells Soft-Scope III to transfer execution to your application and run until it is stopped by a breakpoint. The program starts executing at the current execution point. GO syntax is as follows:

```
GO [WRITE | ACCESS memref]
GO [coderef]
GO [RETURN]
```

Use the abbreviation G to initiate the GO command.

GO with a code or memory reference sets a temporary breakpoint at the desired reference. The temporary breakpoint is assigned the scope currently reported by BPSCOPE.

Execution then proceeds at full speed until that (or any other) breakpoint is hit.

If the breakpoint stops execution at the start of a source line, the line is displayed. If your breakpoint was set at an absolute address, the breakpoint might stop at some point within a line of code. When this happens, you will see something like the following. To look at the line, use the DISASM command:

```
[ Inside ] :CMAIN.main#2679
```

If you specify GO without parameters, and there are no breakpoints set with scope in the task you are executing, Soft-Scope III displays the following message asking you if you want to go anyway:

```
ss> go
[ No breakpoints are set, go anyway? (y/n) ] "N"
```

If you go without breakpoints that will stop your task a running prompt (!ss>) is displayed when the BPTIMEOUT value is reached. If you want to stop the task, you will need to set a breakpoint at a location it will hit. By now, it may have executed much farther than you intended!

If you use a code reference with the `go` command, execution stops when it reaches the referenced location:

```
ss> go fill
[ Break at :CUTILS.fill() (0288:000001b8) ]
#91
#92     static void fill (buf, val, count)
#93         unsigned char buf [];
```

Use a line number and execution stops on the specified line. Since the code reference specified isn't in the currently open module, you must give the module name along with the line number:

```
ss> go :cutils#61
[ Break at :CUTILS.init#61 (0288:00000080) ]
#61         c->dev_code = i % 5;
```

Start execution at the current execution point. Stop execution when an instruction attempts to write to `cmd_count`.

```
ss> go write cmd_count
< Write break >
[ Break inside :CUTILS.sample#88 (0288:000001ab) ]
#88         return &cmd_buffers [cmd_count++];
```

Soft-Scope III attempts to cover the whole length of `cmd_count`. This means that if `cmd_count` is four bytes long, a write to the last byte will trigger a breakpoint. See the section on Breakpoints in this chapter for information about hardware breakpoints and the 80386 debug registers.

In this example, execution stops when `sleep100` is accessed.

```
ss> go access sleep100
< Access break >
[ Break inside :CUTILS.1st (025f:00000fff) ]
#14     for (; sleep100 >= 0; sleep100--)
```

*Go to a specified location in the code*

*Go until a memory location is written to or accessed*

***Return from procedure calls***

The next example starts execution at the current address of the CS and EIP registers, and stops when the execution pointer is pointing to the next to the latest return address found on the stack (the “Return” seen in the STACK display below).

```
ss> stack
[ :CUTILS.fill, current execution point. ]
[ Return 1 - :CUTILS.init#68 called fill ]
[ Return 2 - :CMAIN.main#33 called init ]
[ Return 3 - Unknown module called main ]
[ No return address available ]

ss> go return
[ Break at :CMAIN.main#35 (0288:0000000b) ]
[ Module :CMAIN initializing, using "cmain.lst" ]
#34         do {
#35         c = sample ();
```

Notice the STACK command before the GO RETURN command. Before, execution was nested three levels deep. The GO RETURN(2) command caused the target to execute until it returned from **:cutils.fill** and **:cutils.init**.

In some special cases, GO RETURN may not have the desired effect. For instance, if somewhere in the executed code an instruction jumps to the same address as the instruction following the calling address (as might happen in a recursive algorithm), execution will stop.



When you specify an absolute address with any execution breakpoint, including GO, make sure that the address resolves to the first byte of an instruction or the results will be unpredictable.

The `MODULE` command displays and makes listing file assignments.

```
MODULE [:modname = filename]
```

When Soft-Scope III creates listing file names, it appends the following extension:

- Intel ASM 386 files            .A38
- Intel iC-386, files           .LST
- Intel PL/M files             .LST
- Intel iC-286 files           .C

You can tell Soft-Scope III not to assign a file to a module by assigning the module to the null device (specifying “” or pressing <Enter> when prompted for the filename).

The debugger doesn't initialize a listing file until a breakpoint in that module is encountered or you issue a command that displays the source lines in that module. If at that time Soft-Scope III finds that the assigned *filename* is invalid (i.e., not a listing file), it will generate an error.

To display a list of your current listing file assignments, type `MODULE` with no parameters.

Display the current file assignments.

```
ss> module
      Name-----Assigned File
[Default] :CMAIN . . . . . "\"./cmain.lst"
          :CUTILS. . . . . "\"./cutils.lst"
```

The module `cmain` is the current default module.

To perform a list of assignments every time you load Soft-Scope III, create a macro to automate the process (see *Macros, Chapter 6*).

Assign the file `CMAIN.LST` in subdirectory `CSAMP` as the listing file for the module `cmain`.

```
ss> module :cmain = csamp/cmain.lst
```



*Display the current file assignments*



*Make a file assignment*

# Breakpoints



## Display a list of breakpoints

## Breakpoint scope



Soft-Scope III supports two kinds of breakpoints:

- Execution breakpoints
- Hardware breakpoints

Execution breakpoints stop application execution at the beginning of the line the breakpoint is set on—just before the line is executed.

Hardware breakpoints can be set to stop execution when a memory location is written to or accessed.

```
BREAKPT [-] [[WRITE memref | ACCESS memref] [TASK | JOB | GLOBAL]]  
BREAKPT [-] [[coderef][TASK | JOB | GLOBAL]]
```

The abbreviation, BR, invokes this command. Breakpoints remain until you delete them.

The breakpoint command without keywords or parameters prints a list of all current breakpoints. The "\*" means the breakpoint's scope includes the current task:

```
ss> br  
global * Access (0015:ffffffa8)  
global * :CUTILS.fil#2051  
task Write (0006:000fff78)
```

Breakpoint scope determines which tasks the breakpoint will stop. You have the following choices:

- |        |   |
|--------|---|
| TASK   | Halts execution of only the task that is current when the breakpoint is set |
| JOB    | Halts execution of all tasks in the current job                             |
| GLOBAL | Halts execution of all tasks, in any job or session in the system           |

By default, breakpoint scope is TASK. By using the syntax below for the BPSCOPE command, or the keywords, TASK, JOB, or GLOBAL with the BREAKPT command, you can specify breakpoint scope:

```
BPSCOPE [TASK | JOB | GLOBAL]
```

The BPSCOPE command specifies the scope of all breakpoints set after the command is issued.

The following example displays the current breakpoint scope:

```
ss> bpscope
Current breakpoint scope = task
```

The next example sets a breakpoint with scope JOB. If you specify scope when setting a breakpoint, the scope applies only to that breakpoint:

```
ss> br :cmain#18 job
job      *   :CMAIN#18 [Breakpoint added]
```

BPSCOPE is useful for debugging tasks which are created from the same code, or which share common procedures with other tasks.

For example a utility function that is called by several other tasks, but you want to follow the execution path of just one of them.

Set the scope to task, activate the task you want to follow and set a breakpoint in the utility function. Only the task that was current when the break was set will stop on that breakpoint.

The BPTIMEOUT command allows you to specify how long you want to wait for the target to reach a breakpoint before you get a Soft-Scope III prompt.

When the time specified by this command has elapsed, if no breakpoint is hit, Soft-Scope III prints a "<Task running>" message and displays the running prompt:

```
!ss>
```

You may enter commands at this prompt just as you do at the regular prompt, but commands such as LINE or REG, which require current register information, won't give accurate data until the breakpoint is hit and the normal prompt returns.



# Breakpoints

## *Execution breakpoints*

Use Execution breakpoints when you want to stop execution at a certain point in the program. The following example sets a breakpoint with scope JOB at line 2051:

```
ss> br #2051 job
job      *   :CUTILS.fill#2051 [Breakpoint added]
```

This example sets an execution breakpoint on the procedure `c_data`. Execution stops before the first executable line of the procedure:

```
ss> br :cutils.c_data
task     *   :CUTILS.C_DATA [Breakpoint added]
```

## *Hardware breakpoints*

Use hardware breakpoints when you want to stop execution when a certain memory location or variable is accessed or written to.

Write breakpoints stop execution when a memory location is written to. The following example sets a write breakpoint with GLOBAL scope on the variable `sleep`:

```
ss> br write sleep
global  *   write (0015:ffffffa8) [Breakpoint added]
```

Access breakpoints stop execution when a memory location is written to or read from:

```
ss> br access tally[3]
global  *   write (0015:ffffffa8) [Breakpoint added]
```

## *Delete breakpoints*

Breakpoints remain set until you delete them. The optional hyphen deletes the referenced breakpoint--without a reference it deletes all breakpoints:

```
ss> br -
[All breakpoints removed]
```

Hardware breakpoints make use of the four 80386 debug registers. Because of the way these 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)

Breakpoints set on variables or memory that do not conform to these conditions will use more than one register.

Assume you have an 11 element array, `arrayx`, declared as type `char`, and that the first byte of the array begins at address `1007P`:

```
ss> br access arrayx
```

Setting the above 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 two end bytes.

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:

```
ss> br access byte arrayx
```

Hardware breakpoints are not enabled until a `GO` or `STEP` command is issued and they are disabled when any breakpoint is hit. It is possible for breakpoints to be "missed."

*The 80386, 80486  
debug registers*



## Disassemble Code



DISASM disassembles the instructions found at the specified address, and if the corresponding high-level lines can be determined, displays them. If you aren't in a source line, DISASM with no parameters disassembles the code at the current execution point. The parameter *coderef* can be a source line number, address, code symbol, or module name defining where disassembly should start.

DISASM [ALL] [NOLINES] [*coderef*] [TO *coderef*]

The first time you use DISASM in a Soft-Scope III session, if you don't specify a *coderef*, DISASM starts at the location found in the CS:EIP register pair.

If you are already in or at a source line, DISASM without a count disassembles the current source line. Subsequent DISASM commands start from the location of the last instruction displayed by a previous DISASM command, unless you execute the target program or issue a line command.

You can disassemble an area of code by specifying a starting and ending code reference. In the following example, NOLINES tells Soft-Scope III not to include source lines in the display:

```
ss> disasm noline 56 to 57
028b:00000030  mov    [ebp-04H], 00000000H    ; Imm=0
028b:00000037  cmp    [ebp-04H], +30         ; Imm=3
028b:0000003b  jge    #58                    ; $+19
028b:0000003d  mov    eax,[ebp-04H]
028b:00000040  mov    [fffffcch=4*eax],00000000H ; Imm=0
028b:0000004b  inc    [ebp-04H]              ; Dword
028b:0000004e  jmp    00000037H              ; $-25
```

The ALL keyword tells SSIII to show op-codes:

```
ss> disasm all
#91
#92  Static void fill (buf, cal, count)
#93      unsigned char buf [];
      028b:000001b8 55 push    ebp
      028b:00000169 89 e5 mov  ebp,esp
      028b:000001bb 83 ec 04 sub  esp,+04H;Imm=0
#94      int val;
#95      int count;
#96  {
#97      int i;
#98      for (i=0; i < count; i++)
028b:000001be c7 45 fc 00 00 00 00 mov  [ebp-04H, 00000000H ;Imm=0
```

**SS>**  
**STACK**

Use the STACK command to display procedure call nesting, stack usage, and information about which source lines made calls.

```
[count]  STACK [LINES]
          STACK USAGE | RESET
```

STACK with no keywords or parameters displays a trace of procedure call nesting. It tells you what procedure called what procedure, starting at the your current execution point and proceeding backwards:

```
ss> stack

[Inside :CUTILS.sample#88, current execution point.]
[Return 1 -- :CMAIN.main#35 called sample]
[Return 2 -- Unknown module called main]
```

STACK LINES displays a trace of procedure calls along with the source line that made each call:

```
ss> stack lines

[Inside :CUTILS.sample#88, current execution point.]
[Return 1 -- :CMAIN.main#35 called sample]
#34     do {
#35     c = sample (); /*Get sample until no more */
[Return 2 -- Unknown module called main]
```

**Display current information about the stack.**

STACK USAGE displays the following information:

- The defined stack area addresses
- The number of bytes free and percentage of the stack free at the current stack pointer location
- The number of bytes free and percentage of the stack free at the deepest stack level yet reached

The following is an example STACK USAGE display:

```
ss> stack usage

Stack size & address : 4097 bytes. 0014:efff to 0014:ffff
current level : 4092 bytes free, 0% used.
lowest level : 3062 bytes free, 32% used.
```

STACK RESET clears the stack between the pointer and the bottom of the stack--the currently unused portions of the stack.

To determine available stack resources, give the STACK RESET command first. This makes it possible for STACK USAGE to determine how much of the stack is used.

The following example shows a STACK USAGE display before STACK RESET:

```
ss> stack usage
Stack size & address: 4096 bytes c450:00000000 to c450:00000fff
Current level:      4024 bytes free, 1% used
Lowest level:      0 byte free, 100% used
```

The next example shows the same display after STACK RESET is invoked. Notice the difference in the lowest level resources:

```
ss> stack usage
Stack size & address: 4096 bytes c450:00000000 to c450:00000fff
Current level:      4024 bytes free, 1% used
Lowest level:      4024 byte free, 1% used
```

# Task Manipulation



**Display tasks that are at break**

**Change to a different task**



Use the TASK command to view the source code in tasks, change the task context, and determine the status of tasks.

TASK [*tasktoken*] | [ALL]

ALL = All tasks from all Soft-Scope III sessions currently running

TASK reports information on which tasks are at a breakpoint, and prints source-level information about the breakpoint, if possible. The task whose context Soft-Scope III is currently using is denoted by the asterisk ("\*") in the left-most column:

```
ss> task
* 27e0 :CUTILS.display_lights()
 2788 :CMAIN.process_task()
```

To change the task context in which Soft-Scope III is operating, use TASK with the task token as an argument:

```
ss> task 2788
Current context: task= 2788 job= 4500
[ :CMAIN.process_task() ]
#2433
#2434 void far process_task ()
#2435 {
```

Soft-Scope III displays source information for the new task if it is available.

Since iRMX tokens are dynamically allocated, their numbers will change from one invocation of Soft-Scope III to another.

If you want to see what tasks are at break in other Soft-Scope III sessions that are running, use the ALL keyword. Tasks from other sessions have a question mark in the left-hand column:

```
ss> task all
* 27e0 :CUTILS.display_lights()
 2788 :CMAIN.process_task()
? 4780 in :home:test/csamp (4500:00000055)
```

Tasks 27E0 and 2788 belong to our own current Soft-Scope III session for which we have access to symbolic information. The question mark (?) in front of task 4780 tells us it belongs to another Soft-Scope III session and that its symbolic information is not currently available: All we know is that it is inside of the file :HOME:TEST/CSAMP.

Access the symbolic information for task 4780 with the command TASK 4780:

```
ss> task 4780
[ Warning another session set this break ]
[ Loading OMF-386 STL file ":home:test/csamp", Symbols ]
[ In :CUTILS.delay#2670 ]
#2670      call delay_fine;
```

The symbolics for the file :HOME:TEST/CSAMP are loaded, the load segment information for this file is retrieved from SSKERNEL, and task 4780 is detached from another active Soft-Scope III session and becomes your current task.

Before the task is detached, you are warned that this task belongs to another Soft-Scope III session user. The other Soft-Scope III user is *not* warned.

In some cases, a task will be listed by TASK ALL with a question mark (?), and there is no other Soft-Scope III session active. This can happen if some other application encounters some kind of a fault, such as a General Protection or Stack fault. It is possible to change execution context to that task by using TASK *tasktoken*.

**Examine tasks from other SSIII sessions**



# Trapping iRMX Exceptions

## ***The Soft-Scope III exception handler***

During initialization and loading, Soft-Scope III defines a job-wide default exception handler designed to trap any iRMX exception encountered by your application, and to return control to Soft-Scope III. This handler is set for mode 3. Both environmental (0 -7FFFH) and programmer (8000H- FFFFH) exceptions are trapped.

When the debugger detects an exception, it takes control away from your executing application before it returns from the exception system call. When this happens, an iRMX Exception Handler message displays. See *Figure 4-7*.

If you type "Help Exception" Soft-Scope III Help displays an on-line version of the following:

## ***Find the code that caused an exception***

Soft-Scope III's exception handler has trapped an iRMX exception caused by your application. If you already know what line or module caused the exception, and you are sure that continuing will not immediately cause more exceptions, you may continue program execution from this point with GO or STEP.

If you don't know what line or module caused the exception, follow the directions below to get back to your own source code:

1. If a task other than the current one caused the exception, switch to the task displayed in the Exception Handler message.
2. Step through the exception handler assembly code using the STEP command until you find yourself in familiar code. This typically requires 20 steps and about four return instructions.
3. The line that caused the exception is the one just before the line you return to in your code. Use the LIST command to display that line.

```
Your application has encountered an iRMX exception:
  0006: E$EXIST (parameter #2),
  Exception occurred in task bc80
[For more information - enter 'Help Exception']
```

Exception Handler message contents:

Line 1	Notice that your application has caused an iRMX Exception
Line 2	iRMX error message describing the error that caused the exception
Line 3	Task that contains the code that caused the exception

For a given model of compilation, the number of steps to return to your source is always the same. If you know what that number is from previous experience, use a *count* to return to your code with a single command:

```
20 step
```

If you want to disable this feature completely, so Soft-Scope III doesn't handle exceptions, set the option `rmxload.excep=off`, and all exception handling will be in-line. The option defaults to `on`.

See *Define your own exception handler* on the next page to learn how you can selectively handle iRMX exceptions.

**Figure 3-1,**  
**Exception Handler**  
**message**

**Continued from**  
**previous page**

**Turn exception**  
**handling OFF**

# Trapping iRMX Exceptions

## ***Define your own exception handler***

Because Soft-Scope III saves your application's current exception handler at each and every breakpoint, and restores it just before beginning program execution again, you have the option of redefining the exception handler to one of your own.

The example in *Figure 4-8* shown below illustrates how you can disable Soft-Scope III's exception handler temporarily so you can handle exceptions in-line.

Keep in mind that the RQSETEXCEPTIONHANDLER system call sets the exception handler for *only* the *calling task*. If you want more than one task to have this new exception handler, you will need to have each task set the exception handler for itself.

***Figure 4-8, Disable Soft-Scope III's exception handler***

```
EXCEPTIONSTRUCT eh_handler;
    .
    .
    /* Soft-Scope handler active */
    .
    rqgetexceptionhandler(&eh_handler,&status);
    eh_handler.mode = 0;
    rqsetexceptionhandler(&eh_handler,&status);
    .
    /* handle exceptions in-line */
    .
    eh_handler.mode= 3;
    rqsetexceptionhandler(&eh_handler,&status);
    .
    /* Soft-Scope handler active */
```

More information on RQSETEXCEPTIONHANDLER and RQGETEXCEPTIONHANDLER can be found in the *iRMX System Call Reference*.

Use the `SUSPEND` and `RESUME` commands to suspend a task and then restart it. This is useful when you are trying to debug a task and its interaction with another task prevents you from determining the problem. Suspend the second task while you find the bug.

```
SUSPEND tasktoken
RESUME tasktoken
```

Suspend corresponds exactly to an `iRMX RQSUSPEND()` system call.

Resume is the same as the `iRMX RQRESUME()` system call, with the following exception:

If you issue the resume command and specify the token of a task that is not suspended, but is at break, Soft-Scope III removes the task from the task at break list and begins execution.

If the task is at break and suspended, you will need to issue two `RESUME` commands, one to take it off break and one to resume it.

Exert special care not to put Soft-Scope III in the context of a suspended task, since Soft-Scope III executes all of its tasking commands *in the context* of the current task. This may cause Soft-Scope III to hang waiting for a response from the task.

```
ss> resume 4600
[Suspend successful]
```

If `task1` is the name of a variable that contains a *tasktoken*, use `task1` to specify the task:

```
ss> suspend task1
[Suspend successful]
```



### Resume Example

**Use a data symbol to specify which task to suspend or resume**



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.

## Table of Contents

Data References.....	60
Evaluate Data .....	64
Display Type Information .....	65
Reference Scoping .....	66
Memory References .....	68
Type Overrides .....	70
Dump .....	74
Registers and CPU Structures.....	76
Built-in Functions .....	78
Numbers .....	80
Operators .....	82
Strings.....	84
Reference Summary .....	86

## Data References

With Soft-Scope III you can reference any variable your application can access. Some examples are listed below:

- simple variables
- arrays
- structures
- pointers
- unions
- bit fields

### *Simple variables*

You can reference a variable by typing the variable's name at the prompt. If the variable isn't a structure or array, SSIII determines the variable's type and displays the hex and decimal values of the associated memory locations:

```
ss> pattern
PATTERN = 0x00000041 +65
```

### *Array references*

If the variable is an array, referencing it without qualification—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 subscript. You can even use integer variables as subscripts.

### *Display an entire array*

To reference an entire array, use the array name:

```
ss> lights
LIGHTS[0]=0x2a          42    \'\'
LIGHTS[1]=0x2d          45    \'\'
LIGHTS[2]=0x2d          45    \'\'
LIGHTS[3]=0x2a          42    \'\'
LIGHTS[4]=0x2d          45    \'\'
LIGHTS[5]=0x2d          45    \'\'
LIGHTS[6]=0x2a          42    \'\'
LIGHTS[7]=0x2a          42    \'\'
```

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

```
ss> lights[2]
LIGHTS[2]=0x2d          45    \_'
```

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

```
ss> lights[2..6]
LIGHTS[2]=0x2d          45    \_ '
LIGHTS[3]=0x2a          42    \* '
LIGHTS[4]=0x2d          45    \_ '
LIGHTS[5]=0x2d          45    \_ '
LIGHTS[6]=0x2a          42    \* '
```

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

```
ss> lights[2...]
ss> lights[...6]
```

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:

```
ss> lights[i]
LIGHTS[3]=0x2a          42    \*'
```

(Continued on next page)

***Display a single element of an array***

***Display a selected number of array elements***

***Variables as subscripts***

## Data References

### **Structure references**

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

```
ss> struc1
```

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

```
ss> struc1.xint
```

### **Reference unions**

Union reference syntax is based on structures, simply enter the union's name:

```
ss> date
Union {
  Struct {
    unsigned char day;
    unsigned char month;
    unsigned int year;
  } today;
  unsigned long days_since_year_0_ad;
} date;
```

### **Reference bitfields**

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

```
ss> 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:

```
ss> enet_pkt.data
```

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

```
ss> oldcust
```

The pointer dereference operator (\*) dereferences a pointer and displays the type and values it points to:

```
ss> *oldcust
*:CUTILS.c_data.oldcust structure
  name[0]. . 0x42   +66   'B'
  name[1]. . 0x65  +101  'e'
  name[2]. . 0x74  +116  't'
  name[3]. . 0x68  +104  'h'
```

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

```
ss> oldcust->name
```

*Pointer references*

*Dereference a pointer*

*Display a single element of a structure a pointer points to*

# Evaluate Data

***EVAL provides more information about some kinds of references***



Use the EVAL command to display more specific information about the following reference types:

- Procedures** Displays the module name, line number, starting and ending address, and length
- GDT[x]** Displays the descriptor base and limit, and the access rights byte
- Pointers** Displays the physical address, the pointer's value, the GDT associated with the pointer, the requested privilege level, and the segment type and access privileges

EVAL [*memref* | *coderef*]

Evaluate the procedure, **fill**:

```
ss> eval fill
Module :CUTILS (#93 to #101)
Code 028b:000001b8 to 028b:000001de (39 bytes)
```

When using EVAL to examine a pointer, it isn't necessary to specify hexadecimal format, because hex is the default format for pointers. The following example demonstrates the pointer display, which is detailed in the paragraph below:

```
ss> eval c
ffffed2cH gdt[65] rpl=0 Data ED-R/W-AC (0001e2a0P)
```

- ffffed2cH** pointer's value
- gdt[65]** the GDT
- rpl=0** requested privilege level. The privilege level is reported as the privilege level of the current module's code segment
- ED-R/W-AC** expand-down segment, with read, write and access privileges
- 0001e2a0P** physical address

TYPE displays a variable's data type, scope, and storage class. You can look at the composition of large, complex data structures without having to sort through their contents.

TYPE [*memref* | *coderef*]

Display type information to determine if a variable is stack-based and only reachable from within the procedure where it is declared.

The following example displays type information about the structure, `cmd_buffers`:

```
ss>type cmd_buffers
array[0..3] of structure
  board_id ..... long
  dev_code ..... long
  dev_name ..... pointer ->char
  cmd_code ..... bitfield :2
  cmd_buf ..... pointer ->union
```



## Reference Scoping

---

### *Examples*

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.

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 current module, use the variable's name:

```
ss> c
```

You can reference a static variable in a procedure other than the current one by separating the procedure name from the variable name with a period:

```
ss> c_data.i
```

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

```
ss> :cutils.i
```

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

```
ss> :cutils.delay.i
```

By using the rules listed in *Table 4-1* below, you can reference any variable, located in any module or procedure, that is not stack-based.

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

**Table 4-1 Reference Scoping**

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 they are defined in results in the error message:

```
< No address associated with reference >
```

If you try to examine a stack-based variable before it has been initialized, a value may be displayed, 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.

See also: *Memory References, Chapter 4*  
*Reference Summary, Chapter 4*

**Reference stack-based variables**

# Memory References

## ***Memory references are more than just addresses***

With Soft-Scope III, 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.

- You can 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 80386 paging tables. Use the syntax below:

*hexnumberL*

- 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:

*hexnumberP*

- You can 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 a memory address:

*variablename*

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

```
ss> 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**:

```
ss> 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 (&). This example displays memory at **structy** in the format of **structx**:

```
ss> structx at &structy
```

Use a physical reference to view memory without regard to the segment that contains it. In the example below, we have set a hardware breakpoint on the first byte of a variable that begins at physical address 20P:

```
ss> br write 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:

```
ss> $ss:$ebp-0x10
```

See also: *Type Overrides, Chapter 4*  
*Data Types, Appendix A*

*Code references*

*Most of the time,  
you can use logical  
addresses*

*Physical references  
work well if you're  
not sure what  
selector to use*

*Use operators and  
numbers to create  
an expression*

# Type Overrides

## **Definition of type overrides**

Using type overrides, you can cause a variable to be displayed as *though* it were a type other than that declared in your application. Type overrides don't perform a conversion on the variable, they merely overlay a new type at the variable's 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 as types for overrides:

- ❑ Any data type listed in *Table 1, Data types for use in type overrides*, in *Appendix A* at the back of this book.
- ❑ Any user-defined variable that is currently accessible by your application and Soft-Scope III (stack-based variables must be on the stack).

## **Apply a type override to a variable**

The simplest of the above forms is to specify a type before a variable:

```
ss> long n
```

## **Apply a type override to an address**

Use the second form to apply type overrides to addresses, including registers, selectors, and pointers. Remember to use the **at** operator. Here are a few examples.

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

```
ss> pointer at 200:0ffff
```

The next example displays the contents of the memory specified by the logical reference in the format of a double:

```
ss> double at $ss:$ebp
```

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

```
ss> 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:

```
ss> TSS386 at $tr
```

You can use the addressof operator (&) to specify an address for use with the **at** operator.

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

```
ss> structx at &structy
```

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

```
ss> structx at 200:ff0f
```

Using the **at** operator and an address, user-declared variables can be type overrides. The following example displays memory at **\$ss:\$ebp - 0x10** in the data-type format of the variable **n**.

```
ss> n at ($ss:$ebp - 0x10)
```

*Use a variable to superimpose its data type over the address of another variable*

*User declared variables can be used to define a type override*

# Type Overrides

***How much memory do you want to display?***

The length operator will help you specify how much memory you want SSIII to display.

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

```
ss> word n length 10
```

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

```
ss> dump byte at 200:ldf length 10
```

***Expressions in type overrides do mathematical operations***

You can use expressions in type overrides.

The example below causes SSIII 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:

```
ss> long n + 2
```

```
0x00000004          +4
```

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

```
ss> word at ($ss:$ebp - 0x10)
```

***Assign 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:

```
ss> float speed = 3.0
```

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:

```
ss> float speed
```

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` (e.g., `char *dev_names`):

```
ss> *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:

```
ss> string *dev_names
'DISK\0' 5
```

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

```
ss> array1 at 400:6
```

There are other ways to do the same thing. For example, if `array1` in the example above is a 3 element array of longs, the following will create the same display:

```
ss> long at 400:6 length 3
```

See also: *Dump, Chapter 4*  
*Table 1, Appendix A*

***Display data in its most useful format***

# Dump



Use DUMP to display target memory in a formatted list. You can specify any data reference or memory location.

The first time you use DUMP, if you don't specify an address, SSIII assumes you want to start dumping at physical address 00000000P. The next time you DUMP, if you don't give parameters, Soft-Scope III assumes you want to start where the last DUMP left off.

```
DUMP [BYTE | WORD | DWORD] memref [TO memref ]
```

The default is BYTE.

BYTE	Displays in byte format, byte order=1,2,3,4
WORD	Displays in word format, byte order=2,1,4,3
DWORD	Displays in dword format, byte order=4,3,2,1

DUMP displays blocks of memory with a hexadecimal display on the left and the corresponding ASCII field on the right. WORD and DWORD formats are displayed in reverse format with the high byte first.

Soft-Scope III automatically determines the correct size for dumps that you specify using a variable reference. For example, if you issued DUMP ARRAYX, Soft-Scope III would compute the size of ARRAYX and dump that many bytes of memory.



Some processor boards hang if you attempt to access non-existent memory. This depends on how your processor board is jumpered.

The following example demonstrates the byte order when dumping in byte format:

```
ss> 5 dump word name_init[1]
      0 1 2 3 4 5 6 7 8 9 a b c e 0123456789abcde
33e0:00000076          9d 00 00 00 e0
```

The next example shows the byte order when NAME\_INIT is dumped in word format:

```
ss> 5 dump word name_init[1]
      0      2      4      6      8      a      c      e
33e0:00000076          009d 0000 33e0 00a6 0000
```

The last example shows the same dump in dword format:

```
ss> 5 dump word name_init[1]
      2          6          a          e
33e0:00000076          0000009d 00a633e0 33e00000
33e0:00000082 000000af 00b733e0
```

# Registers and CPU Structures



Display registers and CPU structures using the REG or EVAL commands.

Use the following syntax:

```
REG [ALL | FLOAT]
EVAL memref | coderef
```

ALL                    Show system registers  
FLOAT                 Show NPX registers



You can access individual descriptors by treating the GDT, IDT, and LDT as if they were arrays of structures without using the EVAL command. The following example accesses the ninth entry of the global descriptor table:

```
ss> gdt[8]
Code RD-AC            Offsets00000000..00001ca200ff    DPL=0
```

However, if you use EVAL, the display includes descriptor base and limit and selected fields of the segment descriptor:

```
ss> eval gdt[8]
Code RD-AC            ffff4a58L    LIM=01ca2h            DPL=0    gbP av
```

***EVAL shows the descriptor table element's base and limit***

REG displays the contents of the CPU registers:

```
ss> reg
eax=00000005          cs=2201          eip=000002f0
ebx=fffffff4          ss=0015          esp=ffffffac          ebp=fffffff8
ecx=00000000          ds=2209          edi=001aabf8          gs=0000
edx=001b32dc          es=2209          edi=001aabf8          gs=0000
efl=00000246 [vm rf nt iopl=0 of df if tf SF zf af pf cf]
```

***Display registers***

When running in protected mode, REG ALL displays user and system registers:

```
ss>reg all

eax=00000005  cs=2201      eip=000002f0
ebx=fffffff4  ss=0015      esp=ffffffac  ebp=fffffff8
ecx=00000000  ds=2209      edi=001aabb8  gs=0000
edx=001b32dc  es=2209      edi=001aabb8  gs=0000
efl=00000246  [vm rf nt iopl=0 of df if tf SF zf af pf cf]
cro=7fffffff  [pg ET TS em MP PE]  ldr=2778 tr=0228
cr2=00000000  [pfla=00000000]      gdb=00100000  gdl=ded7
cr3=00000000  [pbd=000000]        idb=0010ded8  idl=03ff
```

To access a single register, put a dollar sign in front of the register name:

```
ss>$eax=2H

[ was ] 0000001H
```

The Registers display is different 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 how many bits.
- Subfields displayed in upper-case letters are in the **on** (1) state.
- Subfields displayed in lower-case letters are in the **off** (0) state.
- Subfields are displayed right-to-left, with the lower-most bit on the right and the upper-most bit 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 also: *Appendix A: Tables*

***The REG ALL display includes system registers***

***Reference or change single registers***

***Registers display description***

## Built-in Functions

### ***Function descriptions***

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

<b>LENGTHOF (x)</b>	Returns the number of array elements associated with a reference.
<b>OFFSETOF (x)</b>	Returns the offset portion of a pointer.
<b>PORT (x)</b>	Performs target hardware I/O. Only Byte-, Word-, or Dword-sized type overrides are allowed with this function.
<b>RETURN or RETURN (n)</b>	Returns the expected return address of the current procedure. Return( <i>n</i> ), where <i>n</i> is an integer parameter, will calculate the return address for the <i>n</i> th nested call.
<b>SELECTOROF (x)</b>	Returns the selector portion of a pointer.
<b>SIZEOF (x)</b>	Returns the parameter size in bytes.

### ***Determine addresses***

Offsetof, Selectorof, and Return all help you determine addresses. Use the addressof operator (&) with the first two functions:

```
ss> offsetof &lights
```

### ***Use Return as a memory reference***

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

```
ss> go return
```

Using the return function with GO is exactly the same as selecting the return button on the bottom of the Code window.

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

```
ss>lengthof lights
0x00000008      8
```

You can read from or write to I/O port addresses. Valid port addresses are from 0 to 0ffffH.

Be careful about inspecting what you have just written to an I/O address by reading from it. With some devices doing a read may change the state of the device, and may not return the value written.

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 III 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:

```
ss> port 3
```

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

```
ss> port 3 = 04H
```

The next example reads 32 bits from port number 2:

```
ss> dword port 2
```

***Determine how many elements make up an array***

***Read and write to Port addresses***



# Numbers

## ***Supported number bases and formats***

Soft-Scope III 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 characters A..F. Hex numbers must start with a digit to distinguish them from symbol names:

**e00ffa9H** must be represented as, **0e00ffa9H**

- ❑ **Floating-point numbers** contain a decimal point and an optional fraction. They must begin with a digit (0..9) rather than a decimal 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 - 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 may have an optional + or - followed by some combination of characters 0..9
----------	--

The following example demonstrates an exponential number:

**-1.098567E+4**

## ***Set the default base***

If a number does not have a suffix or prefix, its base is determined from the **base** option, which can be added to your set file or edited using the SET command.

This option may be set to 2, 8, 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 4-3* for a list of number types and their default bases.

Number type	Default Base
b800:04ac	Parts of a pointer always default to hex
#123 :module#123	Line numbers are always assumed to be decimal
123 <Spacebar>	Counts are always decimal
byte at arrayx length 123	Length counts are always assumed to be decimal
array[123], 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
return (12)	Return counts default to decimal
selectorof	Selector overrides default to hex
-4.000000045E+5	Exponential format defaults to decimal

**Table 4-3 Default number bases**

See also: *Soft-Scope III Options, Chapter 5*

# Operators

---

<b>Operator types</b>	<p>With Soft-Scope III, you can make use of three classes of operators:</p>
	<p>Symbolic operators      provide quick access to data references</p> <p>Arithmetic operators    provide standardized arithmetic expressions</p> <p>Logical operators        provide standard, C-based true/false operations</p>
<b>Symbolic operator examples</b>	<p>Symbolic operators are used as short cuts to access data references. Examples include pointer dereferencing, ranges, and address overrides:</p> <pre>*table_pointer array_1[1..24], array_1[1...] and array_1[...24] ss&gt; long at \$ss:ebp</pre>
<b>Arithmetic operators will return a solution</b>	<p>Arithmetic operators are C-based arithmetic statements, such as the increment operator and the modulus operator:</p> <pre>++i i % 3</pre>
<b>Logical operator examples</b>	<p>Logical operators are those used in true/false C-based operations. Examples include the logical <b>and</b> operator and the <b>not equal</b> operators:</p> <pre>i &amp;&amp; y i != 1</pre>
<b>Operator precedence</b>	<p>Soft-Scope III operator precedence is the same as C operator precedence. In <i>Table 4-4</i> operators on the same line have the same precedence, and rows are in decreasing order of precedence.</p> <p><i>Table 4-5</i> lists Soft-Scope III specific operators and their relationship to the 'C' operators in <i>Table 4-4</i>.</p>

Operators	Associativity
() [] -> .	left to right
! ~ ++ -- + - * &	right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= > >=	left to right
== !=	left to right
&	left to right
^	left to right
	left to right
&&	left to right
	left to right
= = + = - = * = / = % = & = ^ =   = << = >> =	right to left

**Table 5-4 C operators**

Operators and Functions	Relationship
# (:module#23)	same as ->
Type overrides	same as ++
offsetof	same as ++
selectorof	same as ++
lengthof	same as ++
length	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 <<

**Table 5-5 Soft-Scope III specific operators and functions**

See also: *Operators, 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 single and double quotes is that Soft-Scope III includes a terminating null character within the double-quoted string.

<b>If you type</b>	<b>SSIII will create</b>
<b>"frogs"</b>	<b>frogs\0</b>
<b>'frog\'s'</b>	<b>frog's</b>

## *Escape sequences*



Escape sequences in strings perform pre-defined functions. Escape sequences start with the escape sequence delimiter, backslash (\).

Escape sequences create a problem which we solve 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 DOS subdirectory pathname, you must use the following format: "C:SUB\_DIR1\\SUB\_2". This is not an issue when Soft-Scope III prompts for a file name.

The escape sequences listed in *Table 4-6* are supported within strings. They are case-sensitive.

Escape Sequence	Description	Hex Value
\0	Null character	0x00
\b	Backspace	0x08
\t	Tab	0x09
\n	Newline	0x0a
\r	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	
\nnn	nnn is octal value	

*Table 4-6 String escape sequences*

## Reference Summary

The following list is a summary of some of the possible ways to reference a data element or memory address:

<i>variable</i>	refers to a variable in your program
<i>typeoverride variable</i>	refers to a memory location where variable is stored displayed as the type specified by <i>typeoverride</i>
<i>:module.variable</i>	refers to a variable whose scope is in another module
<i>:module.proc.variable</i>	refers to a variable whose scope is in another procedure in another module
<i>#linenumber</i>	refers to a line number in the current module
<i>:module#linenumber</i>	refers to a line in a module other than the current one
<i>array1</i>	refers to an unqualified array
<i>array1[1..3]</i>	refers to a range of elements in an array
<i>structurename</i>	refers to an unqualified structure reference
<i>structurename.elementname</i>	refers to a single element of a structure
<i>pointername</i>	refers to the value of a pointer
<i>*pointername</i>	refers to the area of memory where a pointer points
<i>pointername-&gt;elementname</i>	refers to a single element of the structure where a pointer points
<i>string at 200:0fff</i>	refers to a string at the given memory location
<i>structx at 200:0fff</i>	refers to the display of the contents of the given address in the format defined by the data type of <i>structx</i>
<i>\$register</i>	refers to one of the target processor's registers
<i>word at \$ss:\$esp</i>	refers to the word at the top of the current stack
<i>1234:0fff</i>	refers to a logical address. Note the "0" before the first "f." All numbers that start with a character other than a numeric digit must be prefaced with a "0"
<i>12345678L</i>	refers to a linear address
<i>12345678P</i>	refers to a physical address

# ***Soft-Scope III Configuration***

# **5**

Soft-Scope III uses an options file 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 *FILENAME.SET* file.

## **Table of Contents**

Setting Options .....	88
Soft-Scope III Options .....	90

# Setting Options

## *Soft-Scope III configuration options*



Soft-Scope III maintains a list of options for the Soft-Scope III environment and their associated values. Soft-Scope III loads SS.SET found in the same directory as the Soft-Scope III executable, by default.

You may also specify a set file (*FILENAME.SET*) on the command line when you execute SSIII. This set file, for example, could be used to set options specific to an application. While you are using Soft-Scope III, you can use the SET command to set or override any option:

SET [*optionname* [= *optionvalue*]]

Soft-Scope III uses these options for specific operations but only looks at a value when it is needed, so it's possible to specify an invalid option and not generate an error until that option is used by some other Soft-Scope III command.

The following options are available:

<b>Option</b>	<b>Description</b>
<b>base</b>	Define the number base
<b>cmd.history</b>	Number of commands available to recall.
<b>cmd.initial</b>	Initial command or macro to execute when Soft-Scope III is invoked.
<b>cmd.macro</b>	Initial macro file(s) to load.
<b>cmd.prompt</b>	Soft-Scope III prompt to use.
<b>cmd.silent</b>	Disable the bell.
<b>rmxload.excep</b>	Turn exception handling off
<b>src.path.ext</b>	Pathname for source-file searches.
<b>src.path</b>	Pathname for all file searches.
<b>src.tab</b>	Tab equivalence for any files.
<b>sym.case</b>	Consider case in searches.
<b>sym.descriptor</b>	Descriptor Type Override type.
<b>sym.pointer</b>	Type of FAR pointer to use.
<b>tmp.path</b>	Define the directory where temporary files are stored.

These options are explained in detail in the next pages.

```
[ Options currently set ]
sym.case. . . . . "off"
sym.pointer . . . . "ptr32"
sym.descriptor. . . "desc386"
src.path.a38. . . . ":%:"
src.path.lst. . . . ":%:";/src/lst/"
src.tab . . . . . "4"
cmd.history . . . . "10"
cmd.macro . . . . . "ss.mac;your.mac"
cmd.prompt. . . . . "ss>"
cmd.initial . . . . "init"
```

### *Example option values*

These settings were established by the SS.SET file, and define the following:

- Symbol searches ignore the case of the symbol.
- Pointer type overrides are 48-bit Far pointers.
- Descriptor type overrides are in 386 format.
- ASM386 source files are found in the current directory.
- Listing files are found in the current directory or in /src/lst.
- Tabs for listing files are set at every 4 characters.
- The history keys store the last 10 commands.
- The initially loaded macro files are SS.MAC and YOUR.MAC.
- The Soft-Scope III prompt appears as "ss>".
- The macro INIT, defined in SS.MAC or YOUR.MAC, automatically executes when you invoke Soft-Scope III.

## Soft-Scope III Options

---

### ***Control the default number base***

#### **base=10 | 16**

Set the SSIII option **base** to the decimal value of the number base you want to use when inputting numbers (i.e., **base=16** sets it to hexadecimal). Your choices are 10 or 16 and the default value is decimal (**base=10**).

See *Numbers* in **Chapter 4, Examining Data**

### ***Set the number of history entries to store***

#### **cmd.history = 0..255**

This option sets the number of previous commands that can be scrolled through using the command history keys *<Up Arrow>* and *<Down Arrow>*. This must be set to a decimal value between 0 and 255.

### ***Define an initial command***

#### **cmd.initial=command**

This option can be set to any Soft-Scope III command.

When SSIII is invoked it automatically loads the initial macro file as defined by the option **cmd.macro**. Then it performs the command specified by this option.

### ***Run an initial macro***

#### **cmd.macro=macro filename;filename2;...**

This option lets you define the initial macro file SSIII loads as described above. The *macro filename* must include a complete path. If it doesn't, the macro file must be in the current working directory. By default, this option loads the RadiSys SDB commands.

### ***Change the Soft-Scope III prompt***

#### **cmd.prompt = string**

This option sets the Soft-Scope III prompt. To change the prompt from SS> to Soft-Scope, for example, type this:

```
cmd.prompt=Soft-Scope
```

### ***Turn off the bell***

#### **cmd.silent = ON | OFF**

This options disables the bell when it is turned on (except at the end of a load). The default is OFF.

### **rmxload.excep=on | off**

Use this option to tell Soft-Scope III not to trap iRMX exceptions. When set to **on**, any exceptions your application causes will stop execution. On is the default. See *Trapping iRMX Exceptions*, in *Chapter 4, Controlling Execution*.

### **src.tab=value**

This changes the number of blank characters for each tab character in a source file. When SSIII encounters a tab character in the source file, it inserts *value* blanks at that point. SSIII determines the default for this number from the language of your application.

### **src.path = d:/directory/...**

### **src.path.ext = d:/directory/./ext**

These options control searches for listing/source files. The *.ext* extension is replaced with the extension of the file type you will be searching for. For example, **src.path.lst** is used to search for Intel iC-386 or PL/M listing files. **src.path.c** is used for iC-286 source files.

If all of your C files use the extension *.C00* and are in the directory */CSRC*, **src.path.c = /csrc/.c00** would both set the default directory and default filename extension.

If **src.path.ext** isn't defined, **src.path** is next in priority. The path is composed of a series of specifiers that modify the known filename. In the simplest case it works like the search path defined in an environment except that a semicolon (;) is used as a delimiter between specifiers:

```
src.path=/dir1/;/dir2/
```

The path given when you invoke Soft-Scope is appended to the existing **src.path** entries. If your set file doesn't contain a **src.path** entry, it is created and the path is assigned to it.

*Turn iRMX  
exception handling  
off*

*Define Tabs*

*Define a path to  
your application  
files*

## Soft-Scope III Options

### ***Make SSIII case sensitive***

#### **sym.case=on | off**

Setting this option to **on** causes SSIII to use the case of symbols you type on the command line when searching for a symbol in the symbol table. By default, SSIII is case-insensitive (**sym.case=off**).

### ***Define Pointer type-override display***

#### **sym.pointer=value**

When you use **pointer** as a type override, SSIII can interpret it four different ways. This must be set if you use a pointer override. To control this interpretation, set **sym.pointer** to one of four possible values:

**OFF16**      16-bit offset

**OFF32**      32-bit offset

**PTR16**      32-bit pointer

**PTR32**      48-bit pointer

### ***Define the descriptor type override***

#### **sym.descriptor = desc286 | desc386**

When you use a descriptor type override, Soft-Scope III uses this option to define it. If you use a descriptor type override when this option isn't set, or if it is set to anything but the two values shown below, Soft-Scope III prints an error message.

### ***Tell Soft-Scope III where to store temporary files***

#### **tmp.path=D:\directory\subdirectory\...**

The first time you debug an application, SSIII creates a temporary file that contains the initialization information needed to load that application. This option defines where that temporary file is stored.

The file has the same filename as the application, except the extension is TMP, and the next time you invoke SSIII and ask it to load the application, it looks for the temporary file containing the already-built data. If it finds the temporary file, and the application hasn't been modified, SSIII uses it to load the application instead of rebuilding initialization information.

This option defaults to the directory where the application you are trying to debug is located.

Soft-Scope III's macro facility lets you create your own macros. You can create macros to:

- Rename a SSIII command
- Create pseudo-command files of commands
- Create new SSIII pseudo-commands

## Table of Contents

Create Macros .....	94
Macros Control Statements .....	96
Macro Functions .....	97
Macro Parameters .....	98
Macro Examples .....	99

# Create Macros



Soft-Scope III's macro facility lets you create your own macros. You can create macros to:

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

```
MACRO [LIST]
MACRO LOAD filename
MACRO DELETE [macroname]
MACRO STEP [macroname]
```

Adding a macro to Soft-Scope III consists of the following steps:

1. Create a file containing the macro using a program editor, or use the program editor to add the macro to an existing macro file.



Although you can add macros to the macro file SS.MAC supplied with Soft-Scope III, it is not advised. You run the risk of corrupting macros that Soft-Scope III needs to fully function.

You can load multiple initial macro files by specifying them with the **cmd.macro** option. For example the following loads the macro file SS.MAC, then YOURS.MAC:

```
set cmd.macro=ss.mac;yours.mac
```

2. Load this macro file into Soft-Scope III using the MACRO LOAD command.
3. Invoke the macro like a regular Soft-Scope III command.



Loaded macros are stored internally, and are not affected by changes to their source until that source file is loaded again.

There can be any number of macro definitions in a macro source file. Each declaration must look similar to a C-procedure declaration, except that the keyword “macro” should be used where the C-procedure return type would be, and there can be no type declarations for the parameters. You can use control statements and function calls to built-in procedures within the declaration.

Almost any Soft-Scope III command or expression can be used in a macro. Two exceptions are the MACRO STEP command and the built-in function RETURN used by itself. GO RETURN(2) will work, while RETURN on a line by itself to display the current return address won't work because RETURN by itself is a Soft-Scope macro control statement.

When using the Soft-Scope III STEP command (not the MACRO STEP command), remember to use the form *count* STEP if you don't want to go into interactive STEP mode (i.e., 2 STEP to step twice then continue macro execution).

### ***Macro Source Files***

### ***Soft-Scope III Commands in Macros***

# Macro Control Statements

## Control Statements

The macro language supports the following control statements:

- abort
- break
- if/else
- return
- while

Each statement is explained below:

Abort	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.
Break	Break functions the same as the C-language break: it exits the current block.
If ( <i>condition</i> ) {...} Else {...}	The if/else control statement functions similar to its C-language counterpart. <i>Condition</i> can be any Soft-Scope III 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.
Return	Return functions like its C-language counterpart, returning execution to the place where it was called.
While ( <i>condition</i> ) {...}	While functions similarly to its C-language counterpart. <i>Condition</i> can be any Soft-Scope III expression that evaluates to a number. If <i>condition</i> evaluates to any number except 0, the statements within the brackets will be 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) {...})

There are two built-in functions: echo and print.

## ***Echo ON/ OFF***

Echo ON is a switch informing the Soft-Scope III command interpreter to send all command output to the console. Echo OFF is a switch that does the reverse. All command output is executed quietly, not echoing to the console.

The default echo condition is Echo ON.

## ***Print (string)***

Print sends a string to the Soft-Scope III output device. The string must be enclosed in parenthesis, and can contain the following escape characters:

<b>Escape Character</b>	<b>Function</b>
\a	ring bell
\n	new line
\r	carriage return
\”	double quote

## ***Macro Functions***

# Macro Parameters

## *Parameters*

Macro parameters on the command line are parsed as literals. In the macro source file, a parameter is recognized by a percent-sign preceding the parameter:

```
this is a %parameter1 reference
```

When you invoke a macro, include the string you want substituted. The literal strings are parsed and inserted in the appropriate places. For example, if the macro used in the above example is named **macrox**, the invocation below would cause the statement to be interpreted as “this is a text reference”:

```
macrox text
```

Keep in mind that literals are strings of non-blank characters. The statement “MACROX 3+4” passes the single string “3+4” to MACROX, while “MACROX 3 + 4” passes three strings: “3”, “+”, and “4”.

If you want to pass blank characters in a parameter, place the parameter in quotation marks:

```
"      "
```

The number of parameters passed to the macro can't exceed the number of parameters that the macro expects. If a macro expects one parameter, the macro can be invoked with no parameters, and will pass a null string, but it can't be invoked with two or more parameters.

## Example Macros

```

/*****/
/* gowrval */
/* SYNTAX: gowrval dataref value */
/* EXAMPLE: gowrval xbyte 10h */
/* OVERVIEW: Runs target until dataref is */
/* written with the given value. */
/* */
/* Each time the target stops the value of the */
/* dataref is printed out. */
/*****/
macro gowrval (dataref, value)
{
    echo off
    br write %dataref
    while (1) {
        go
        echo on
        %dataref
        echo off
        if (%dataref == %value){
            break
        }
    }
    br - write %dataref
    print ("%s = %s", %dataref, %value)
}

/*****/
/* lsym */
/* SYNTAX: lsym filename */
/* EXAMPLE: lsym /fs/hello */
/* OVERVIEW: Loads symbols for a specified */
/* filename. This macro could have */
/* explicitly specified a filename */
/* to decrease the amount of */
/* typing(of course there would */
/* have to be */
/* a different macro for each */
/* filename loaded). */
/*****/
macro lsym (filename)
{
    load symbols %filename
}

```



This chapter provides minimal information designed to help ensure Soft-Scope III compatibility with your application. If you need to learn more about the tools SSIII supports, consult the appropriate reference guide.

## Table of Contents

Tools Information .....	102
ASM286 and ASM386 .....	103
BND286 and BND386 .....	104
BLD386 .....	105
iC286 and iC386 .....	106
PL/M 286 and PL/M 386 .....	107
FORTTRAN-386 .....	109

## Tools Information

### ***Example files***

You can use any of the tools listed in *Table 7-1* to build your applications.

To make sure you have all of the information you need, we have included complete samples of batch and make files with the sample program on the distribution disks.

The sample program is located under this directory:

/RMX386/DEMO/SSCOPE

***Table 7-1  
Supported Tools***

<b>386 Tools</b>	<b>286 Tools</b>
ASM386	ASM286
BND386	BND286
FORTTRAN-386	iC-286
iC-386	PL/M-286
PL/M-386	

At least the following controls are required:

- DEBUG
- TYPE
- NAME program linkage directive (inside assembly source files)
- PROC & ENDP directives (inside assembly source files)

Soft-Scope III reads source information from the original source file for ASM286, usually *filename.A28*. ASM386 creates a listing file, *filename.LST*, which provides symbolic information to Soft-Scope III.

ASM286 does not supply sufficient symbolic information to support source-level stepping in Soft-Scope III.

Version 3.0 of ASM386 does not support source-level stepping. However, versions 4.0 and later do.

```
asm386 asmsamp.a38 debug type
```

**Controls**

**Source information**

**Notes**

**Example invocation**

## BND286 and BND386

---

### ***Controls***

Use at least the following controls:

- RCONFIGURE

### ***Notes***

The RCONFIGURE control makes your application loadable by the Human Interface and sets lower and upper memory boundaries.

### ***Example invocation***

We use a control file to specify bind information, so our invocation line specifies only the control file:

```
bnd386 controlfile(csamp.bnd)
```

For an up-to-date copy of CSAMP.BND, look in the directory specified in the *Tools Information* section of this chapter.

The controls for BLD386 are specified in the .CF file created by ICU386.

In this file, replace the NODEBUG option with DEBUG, and remove the NOTYPE directive *each time* new files are generated through ICU386.

This is an example created from a user definition file, TESTRMX.DEF. The TESTRMX.CF file generated by ICU386 is used for the BLD386 invocation as the final step in regenerating the operating system:

```

NUCLS.LNK                &
, /RMX386/SDM/DASM.LNK   &
, M3.LNK                  &
, SDB.LNK                 &
, IOS.LNK                 &
, ETOS.LNK                &
, LOADR.LNK               &
, HI.LNK                  &
, CLI.LNK                 &
, UDI.LNK                 &
, XNETINT.LNK             &
, RMXNET.LNK              &
, USERJOB.LNK            &
OBJECT (/BOOT32/TESTRMX.386 ) DEBUG &
BUILDFILE(TESTRMX.BLD)

```

This example is for a Multibus I system. Consult your iRMX manuals for boot directions on different systems.



### Example .CF file



# iC-286 and iC-386

## **Controls**

Use at least the DEBUG control.

Do not use the following controls:

- NODEBUG
- NOTYPE
- NOOBJECT
- OPTIMIZE(2) or OPTIMIZE(3)
- NOLIST
- NOPRINT

Version 4.5 of iC-386 has a new switch, NOSRCLINES, that reduces the size of the load file. It removes the source lines section of the OMF, which Soft-Scope III doesn't use.

## **Source information**

Soft-Scope III uses source files for source information with iC-286. The default extension is .C.

For iC-386 source information, SSIII uses the listing file produced by the compiler. The default extension is .LST.

## **Notes**

Because it does not support an OMF386 **register** type, iC-286 produces no symbolic records for register variables.

Soft-Scope III generates a message warning you that there is a missing line in your source file. This is because iC-386 creates a line record for the EOF marker. You can safely ignore this message.

## **Sample invocation**

```
ic386 cmain.c compact debug
```

Use at least the following control:

DEBUG

Do not use these controls:

NODEBUG

NOTYPE

NOOBJECT

OPTIMIZE(2) or OPTIMIZE(3)

NOLIST

NOPRINT

Soft-Scope III takes source information from the listing file produced by the compiler. These files default to *SOURCEFILE.LST*.

```
plm386 putils.p38 debug optimize(0) large
```

PL/M pointers do not indicate what variable type they point to, so Soft-Scope III displays a byte value when you dereference them. To work around this, preface pointer references with a type override to display them in pointer format. See *Type Overrides*, in *Chapter 4, Examining Data* for more information.

SSIII does understand based variables. If, for example, the following symbols were declared in your target software,

```
declare customer_ptr pointer,
customer based customer_ptr structure(
name (5) byte,
linkfor pointer);
```

### ***Controls***

### ***Source information***

### ***Example invocation***

### ***Useful tip***

The example below shows what using the pointer dereferencing operator (\*), in the Data dialog box or the Command line dialog box would display:

Data reference: \*customer\_ptr

```
*customer_ptr = 0x53 83 's'
```

The next example fully dereferences **customer\_ptr** and displays the entire structure:

Data reference: customer

```
CUSTOMER structure
NAME[0] . . .0x53      83      's'
NAME[1] . . .0x74      116     't'
NAME[2] . . .0x65      101     'e'
NAME[3] . . .0x76      118     'v'
NAME[4] . . .0x65      101     'e'
LINKFOR . . .0290:000186e9
```

To reference a pointer that doesn't have a based declaration, use a type override. Given the following declaration for a linked list:

```
declare      customer_entry structure (
              name(5)      byte,
              linkfor      pointer);
```

If you wanted to see what the pointer **customer\_entry.linkfor** pointed to you could use the example below, which displays the structure shown in greyscale:

customer\_entry at customer\_entry.linkfor

```
CUSTOMER_ENTRY structure
NAME[0] . . .0x53      83      's'
NAME[1] . . .0x74      116     't'
NAME[2] . . .0x65      101     'e'
NAME[3] . . .0x76      118     'v'
NAME[4] . . .0x65      101     'e'
LINKFOR . . .0290:000186e9
```

Use at least the following controls:

- DEBUG
- TYPE

Do not use these controls:

- NODEBUG
- NOTYPE
- NOOBJECT
- OPTIMIZE(2) or OPTIMIZE(3)
- NOPRINT
- NOLIST

Soft-Scope III uses the listing file created by the compiler for source information. The default extension for these files is *SOURCEFILE.LST*.

```
ftn386 ftnsamp debug
```

Unlike C, FORTRAN does not have a one-to-one correspondence between source files and executable program modules. Each subroutine, function, block data, or program declaration creates a different module.

However, after one FORTRAN block in a listing file is opened, SSIII automatically initializes all the other blocks declared in that listing.

Because Soft-Scope III uses the listing, it is important that you do not specify the NOLIST switch in any of the block declarations.

### ***Controls***

### ***Source information***

### ***Example invocation***

When you open a FORTRAN module, LIST only shows one subroutine or function at a time, making it appear as if you had compiled each subroutine and function as a separate source module.

To make sure SSIII has the information it needs, give names to program and block data declarations. The following declarations in a FORTRAN source file generate four modules; **fmain**, **sub1**, **sub2**, **functa**:

```
$large debug
  program fmain
  .
  .
  .
  subroutine sub1
  .
  .
  subroutine sub2
  .
  .
  function functa
  .
  .
  .
```

# Appendix A Tables

# A

This appendix contains tables of data types for use in type overrides, a table of SSIII operators with short descriptions, and figures of registers for the Intel386 and Intel486 processors.

For more information describing how to use or access these items in Soft-Scope III, see *Chapter 4, Examining Data*.

For more detailed information about the registers, see your Intel386 or Intel486 *Programmer's Reference Manual*.

## Table of Contents

Data Types .....	112
Operators .....	114
General-Purpose Registers .....	115
Flags Register .....	116
Segment Registers .....	116
NPX Registers .....	116
Control Registers .....	117
Protected-Mode Registers .....	117
Descriptors and Subfields .....	118

# Data Types

---

The following table lists data types that can be used with Soft-Scope III type overrides. Some of the types have subfields, which can be identified in the Registers tables later in this appendix.

**Table 1 Data types for use in type overrides**

<b>Data Type</b>	<b>Description</b>	<b>CPU/NPX</b>
BCD	NPX data type, 10-byte BCD integer	87/187/287/387/486
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
DESC286	286 descriptor (6 bytes)	286/376/386/486
DESC386	386 descriptor (8 bytes)	286/376/386/486
DESCRIPTOR	286/386 descriptor (determined by set file option, sym.descriptor)	286/376/386/486
DOUBLE	64-bit real	All
DWORD	Double-length unsigned integer, bit length sym.wordsize * 2	All
EXTINT	64-bit signed integer	All
FLOAT	32-bit real	All
INT	Signed integer, bit length is sym.wordsize	All
LONG	32-bit signed integer	All

Table 1 Data types for use in type overrides (continued)

Data Type	Description	CPU/NPX
OFF16	Near 16-bit offset pointer	386/486
OFF32	Near 32-bit offset pointer	386/486
POINTER	Far pointer. Real mode: offset = 16 bits. Protected mode: offset can = 16 or 32 bits. Defaults to 32 bits. See the set file option, sym.pointer.	286/376/386/486
PTR16	Far 16-bit offset pointer	All
PTR32	Far 32-bit offset pointer	386/376/486
SELECTOR	16-bit selector	All
SHORT	8-bit signed integer	All
STRING	Zero-terminated string (max 255 characters)	All
TEMPREAL	80-bit real	All
TSS286	286 Task State Segment	286/386/486
TSS386	386 Task State Segment	376/386/486
WORD	16-bit unsigned integer	All

# Operators

---

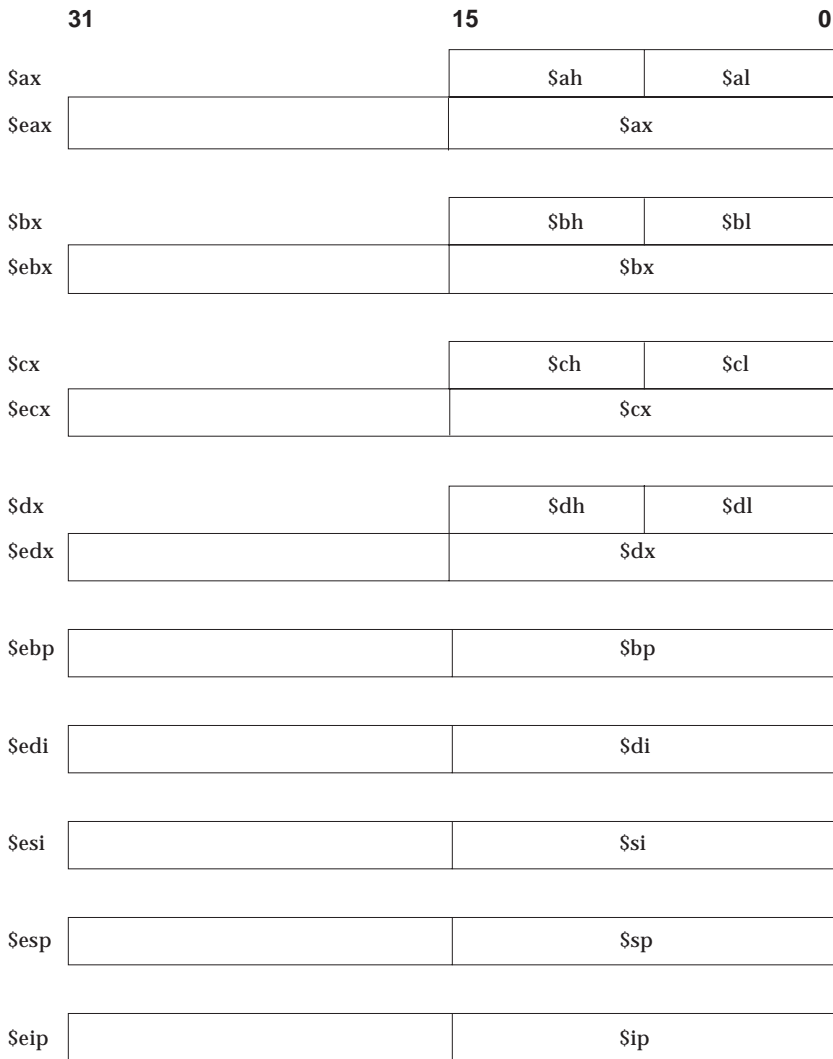
In addition to the Soft-Scope III operators described in the table below, you can use all C operators except the conditional expression operators (`?`, `:`).

**Table 2 Soft-Scope III 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] 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
.	Prefixes a program symbol name to prevent confusion with SSIII 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

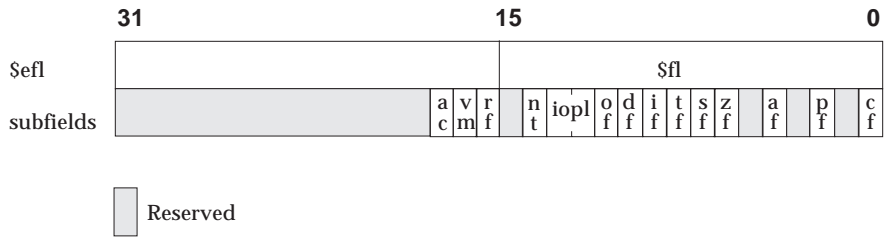
See also: *Operators, Chapter 5*  
*Your C reference manual*

# General-Purpose Registers

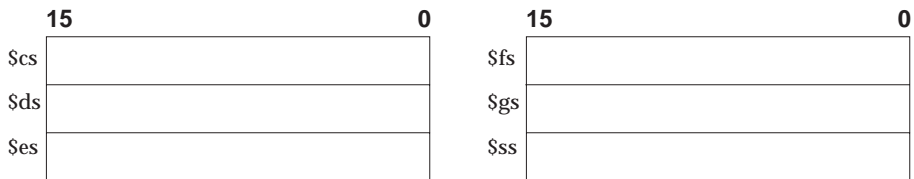


**Figure A-1 General-purpose registers**

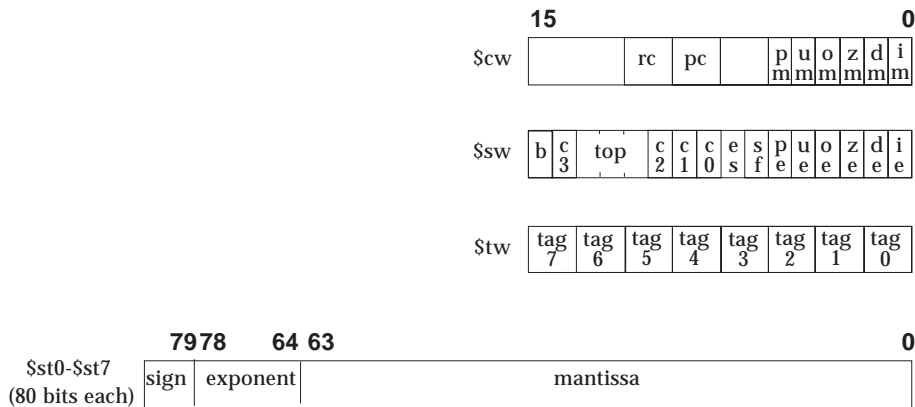
# NPX Registers



**Figure A-2 Flags register**



**Figure A-3 Segment registers**



**Figure A-4 NPX registers**



## Descriptors and Subfields

---

**Table 3 386 protected-mode variables**

<b>Structure</b>	<b>Description</b>
\$GDT	An array reference that spans the current global descriptor table. Use \$GDT[n] to access a specific element. Use \$GDT[n..m] to specify a range of elements. Use EVAL \$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 4 Page/Table directory subfields**

<b>Name</b>	<b>Description</b>	<b>Starting Bit</b>	<b>Size(bits)</b>	<b>CPU</b>
frame	Page frame address	12	20	386/486
avail	Available for use	9	3	386/486
d	Dirty	6	1	386/486
a	Accessed	5	1	386/486
pcd	Page cache disable	4	1	486
pwt	Page write transparent	3	1	486
us	User/Supervisor	2	1	386/486
rw	Read/Write	1	1	386/486
p	Present	0	1	386/486

**Table 5 Descriptor subfields**

<b>Name</b>	<b>Description</b>	<b>Starting Bit</b>	<b>Size(bits)</b>
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

## Descriptors and Subfields

---

**Table 6 TSS386 subfields**

<b>Name</b>	<b>Description</b>	<b>Starting Byte</b>	<b>Size (Bits)</b>
io_map	I/O map offset from start of TSS	102	16
ldtr	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
ss2	Level 2 stack segment	24	16
esp2	Level 2 stack pointer	20	32

**Table 7 TSS386 subfields--continued**

<b>Name</b>	<b>Description</b>	<b>Starting Byte</b>	<b>Size (Bits)</b>
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



## Appendix B Error Messages

# B

The Soft-Scope III debugger generates exception messages when it cannot execute a command.

Many of the error messages are displayed in conjunction with a line of carets (^^^^ ) displayed beneath some part of the command you issued. These carets show the field of the command that SSIII ran into problems executing.

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

For your convenience, address error messages are arranged in alphanumerical order in *Table B-1*.

## Error Messages

Address error messages take the following form:

< *Address - Message* >

<b>Message</b>	<b>Description</b>
GDT limit violation	SSIII trapped a reference outside the bounds defined by the GDL register (the GDT limit)
Non-addressable segment type	The descriptor specified does not have an addressable segment associated with it
Segment is not present	The descriptor specified in Address indicates the segment is not present in memory
Initial TSS not in initial GDT	The loader has found an initial TSS that is not defined within the initial GDT
Initial TSS selector invalid	The initial TSS selector is invalid (Warning only)
Selector outside of GDT limits	The address specified is outside the defined GDT limits (Warning only)
Segment limit exceeded	The address specified is outside the segment limit
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
Invalid descriptor type	The descriptor shown is invalid for the desired operation
Self-defined LDT	For some reason the LDTR contains an LDT selector
Not LDT descriptor	The LDTR register points to a non-LDT segment
Memory bounds exceeded	The memory location which Soft-Scope is trying to access is out of range

**Table B-1, Address Error Messages**

`< filename linenum/column - msg >`

Soft-Scope III encountered an invalid set option specification while executing the specified line of file, *filename*. The message is SSIH's explanation of what caused the error.

Check your options file, usually SS.SET, for options that are not defined as specified in *Chapter 5, Soft-Scope III Configuration*.

`< Array is too complex >`

Soft-Scope III supports up to 10 dimensions in an array.

`< 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 integer variables.

`< Breakpoint already set >`

The referenced memory location or data area already has a breakpoint set for it.

Use the BREAKPT command to see a list of existing breakpoints. 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.

Use the BREAKPT command to see a list of existing breakpoints.

## Error Messages

`< Can't step in current environment >`

There is no task current.

Attach to a task using the TASK command, as described in the section titled *Task Manipulation*.

`< Command name expected >`

Soft-Scope III doesn't recognize the first string on the command line.

Check to make sure the string is a count or a valid command. A list of commands and their syntax can be found in *Chapter 2, Getting Started*.

`< modname contains no lines >`

The specified module contains no line numbers.

Possibly the module is empty or is an assembly-language module.

`< Corrupted help file: "filename" >`

Soft-Scope III attempted to access its help file and found unreadable data.

Copy the file SS.HLP from distribution disk 1 to the directory, /UTIL386.

`< Ctrl-C break > or < Ctrl/C Abort >`

`< Ctrl-C >` aborted the executing command.

`< Expected quoted string >`

SSIII was expecting 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 *Chapter 6, Macros*.

`< Expression is too complex >`

You are attempting to evaluate an expression which has more than 10 pending operators. For example, A+(B+(C+(D+...))).

Simplify the expression.

`< Failed to remove breakpoint >`

Soft-Scope III attempted to remove the breakpoint you specified but couldn't. Make sure you have correctly referenced the breakpoint.

`<< Fatal error: Serial number >>`

The serial number has been corrupted.

Re-install Soft-Scope III.

`<< Fatal exception: msg >>`

SSIII encountered a severe error and aborted execution.

Please restart Soft-Scope III and reload your application.

`< Fatal exception - Abort to O.S. >`

Soft-Scope III encountered a severe, unrecoverable error, and aborted to the OS.

Please restart Soft-Scope III and reload your application.

`< Hardware breakpoint already set at this address address >`

There is already a data breakpoint set on *address*.

Use the BREAKPT command to see a complete list of currently set breakpoints.

`< Initial task register is an ldt selector >`

The initial task register was defined so that a selector in a local descriptor table was selected.

Redefine the initial task register.

`< Initial task register is outside gdt limit >`

The initial task register was defined outside the limits of the initial GDT.

Redefine the initial task register.

## Error Messages

`< Initial TR->non-TSS type descriptor >`

The GDT entry that the initial task register pointed to is not a 286 or 386 task state segment descriptor.

`< Initial TSS is busy >`

The initial task state segment is flagged as busy in the GDT.

`< Internal error - filename_linenumber [- message] >`

SSIII has encountered either data or a situation which was thought to never occur but has in this particular case.

Please report this error to Concurrent Sciences, inc. along with as much information as possible on why this error might have occurred.

`< Invalid character in option setting >`

The entered character cannot be used within a set option.

Verify that you do not have a control character within the option value.

`< Invalid escape character - line ###, col ### >`

While parsing a string, Soft-Scope III has detected an invalid escape sequence.

See *Strings* in *Chapter 4* of this manual for a list of valid escape characters.

`< 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 macro compiler version >`

The macro compiler and Soft-Scope III you are using are not the same version.

Copy the file, SSMACRO from distribution disk 1 to the directory /UTIL386.

### < Invalid macro object file >

The macro compiler produced bad object code, or some other process corrupted its output.

Try erasing the *FILENAME.MOB* file so the macro compiler recompiles your macros.

### < Invalid macro opcode >

While executing a macro, SSIII 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 *Chapter 6, Macros*.

### < Invalid number format >

Soft-Scope III can't understand the specified number (i.e., X = 1234Q5H).

This usually means the number or variable has an invalid base attribute. Valid bases are T = base 10, H = base 16. See *Numbers*, in *Chapter 4*.

### < Invalid override >

Either the attempted override is a bitxx override of a reference that does not contain bitxx (i.e., bit20 \$al), or the override contains two data types that do not produce a meaningful type (i.e., swtype tss386 is not meaningful, but signed byte is).

Look in *Appendix A, Tables*, for a list of data types supported for use in type overrides.

### < Invalid scope >

The breakpoint scope which you have specified with the BPSCOPE command is not valid.

Valid breakpoint scopes are TASK, JOB, and GLOBAL.

## Error Messages

`< Invalid size for I/O port >`

Overrides for the I/O port must be 1-, 2-, or 4-bytes long. The specified type doesn't match the processor port sizes (i.e., TEMPREAL PORT 0, which attempts to specify a 10-byte type to port 0).

Look in *Appendix A* for a table of supported data types and their descriptions.

`< Invalid value for parameter >`

You are using the built-in function **RETURN** and are specifying an invalid parameter.

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 (i.e., LIST 55 when the module ends at line number 40).

`< Line number out of range (### to ###) >`

The line number you've specified isn't within the range of line numbers for the module or procedure you're currently in or for the module/procedure you've specified (i.e., LIST 55 when the module ends at line number 40).

`< Listing file invalid: Improper listing end >`

SSIII 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 SSIII (i.e., you're using a version of some language that SSIII doesn't yet understand).

Please review the information in *Chapter 7, Tools* to see what versions of tools Soft-Scope III supports.

`< Listing file invalid: Improper listing header >`

SSIII 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 SSIII (i.e., it was prepared using a version of some language that SSIII doesn't yet understand).

Please review the information in *Chapter 7, Tools* to see what versions of tools Soft-Scope III supports.

`< 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 SSIII, and try again.

`< Macro name expected >`

SSIII is expecting a valid macro name.

Use the MACRO command to see which macros are loaded.

`< Macro nesting too deep >`

Macro execution has executed too many nested macros.

Only ten macros may be nested.

`< Macro terminated abnormally >`

An abort was encountered in a macro.

This happens when a macro contains an ABORT statement, or you pressed <Q> while performing a MACRO STEP command.

`< Mismatched ()'s >`

SSIII is expecting another right parenthesis.

Make sure your macro has the correct number of right and left parentheses.

## Error Messages

`< 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.

`< Module not found >`

SSIII 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, Use the LIST command to view your source and refresh your memory of symbol spellings.

`< No initial TSS is defined >`

During loading, the TR (Task Register) value was set to 0 (Warning only).

`< No modules loaded >`

The given command requires a default module, and there are no modules found in SSIII's symbolic database.

Use the LOAD command to load an application.

`< No return address available >`

The specified return address isn't resolvable (i.e., **RETURN()**).

Review the specifications given in *Chapter 7, Tools*, to make sure your application is properly built.

`< No symbolic information loaded >` or `< No symbols loaded >`

SSIII 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 *Chapter 7, Tools* for information on building an application for debugging.

`< Not valid for processor >`

The CPU doesn't contain the register you've specified.

See *Appendix A, Tables* for applicable registers.

`< Option name expected >`

The SET TO command requires that an option name be specified.

See SET command syntax in *Chapter 5, Soft-Scope III Configuration*.

`< Option opt_name - Must be defined >`

The option *opt\_name* isn't defined in your SS.SET file, and is required for the operation you've just attempted.

See *Chapter 5, Soft-Scope III Configuration* for a description of the needed option.

`< Option "src.tab" - Must be 1 to 16 >`

In your SS.SET file, the entry for tab stops is set to something other than the integers 1 through 16.

See *Chapter 5, Soft-Scope III Configuration*.

## Error Messages

< Option "sym.case" - Must be ON or OFF >

In your SS.SET file, the entry **sym.case** is set to something other than on or off.

See *Chapter 5, Soft-Scope III Configuration*.

< Option "sym.descriptor" - Must be DESC286 or DESC386 >

In your SS.SET file, The DESCRIPTOR type override must be set to one of the specified values.

Use the SET command to redefine this option.

<Option "sym.pointer"- Must be PTR16, PTR32, OFF16, or OFF32>

In your SS.SET file, The **sym.pointer** option must be set to one of the specified values.

Use the SET command to redefine this option.

< Options "cmd.history" - Must be 0 to 255 >

The **cmd.history** option must be set to a value between 0 and 255.

< Options - Out of storage space >

Too many options are defined. The total length of all option names and values cannot exceed 1024 bytes.

Perhaps there are some options you can eliminate from your set file. See *Chapter 5, Configuring Soft-Scope III* for descriptions of all the available options.

< Out of hardware breaks >

The 386 debug registers are full.

For information explaining the registers and how to use them efficiently, see *Hardware Breakpoints*, in *Chapter 3*.

`< Override not permitted on non byte-aligned bitfield >`

SSIII 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/IDB >`

The GDB and IDB can only be changed as a result of an application load.

`< Received '??' not '\r' >`

Soft-Scope has not received a carriage return from the second terminal within 60 seconds. The terminal is dead, the serial line is dead, or the key you think is the carriage return key isn't.

`< Register doesn't contain this flag >`

The register specified doesn't contain the flag specified.

See *Appendix A, Tables*, to see what register flags SSIII supports.

`< sskernel error: description >`

SSKERNEL has reported the error given in *description*.

`< String too long >`

The string type override was applied to memory starting at the specified address. Soft-Scope III 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. For example, char at 1000p length 5.

## Error Messages

`< Subscript ranges on pointers are not supported >`

SSIII only recognizes a single reference (i.e., PTR[5]) for pointers.

You can do this with arrays (i.e., 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 *Data References in Chapter 4*.

`< Symbol not found >`

Soft-Scope III 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 *Reference Scoping in Chapter 4*, or 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 *Chapter 7, Tools*. Contact Concurrent Sciences, inc., if you cannot eliminate this problem.

`< Symbolic name exceeds 40 characters >`

You've specified a symbol with more than 40 characters in its name, and OMF can only recognize up to 40 characters.

`< Symbolic name expected >`

The parameter above the carets is not a symbolic name.

Use the LIST command to view your source and see what the symbol names are.

`< Syntax error >`

The specified command is an invalid command or an invalid form of a valid command.

A complete list of command syntax is located in *Chapter 2, Soft-Scope III Basics*.

`< target/task running >`

A task is executing, and you cannot execute any Soft-Scope command that assumes the task is stopped.

`< These addresses are not compatible >`

Soft-Scope III cannot perform the specified operation because the addresses given have different types.

When subtracting two addresses, they must be of the same type (logical, linear, or physical), and logical addresses must 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 are not in the same module >`

You've attempted to list across modules.

`< This command does not support repeat counts >`

You've tried to use a count field on a command that doesn't use repeat counts.

## Error Messages

`< This does not evaluate to an address >`

You've specified an invalid address.

Possibly you're trying to use a symbolic reference, but the symbol specified can't be evaluated as an address (i.e., `BYTE AT X` where `X` is a string instead of a pointer).

`< 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.

Use the `LIST` command to see application symbols. Also, see *Code References*, in *Chapter 3, Controlling Execution*.

`< This is not a logical address expression >`

The parameter above the carets must evaluate to a logical address.

See *Memory References* in *Chapter 4, Examining Data*.

`< This is not a memory reference >`

The parameter above the carets must evaluate to a memory location or address.

See *Memory References* in *Chapter 4, Examining Data*.

`< 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 (i.e., `:cmain`). Also, see *Reference Scoping*, in *Chapter 4, Examining Data*.

`< This is not a numeric expression >`

SSIII is expecting a number, and the parameter above the carets doesn't resolve to one.

See *Numbers*, in *Chapter 4, Examining Data*.

`< 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 using the TYPE command.

`< This is not a register name >`

You've tried to reference a variable as a register name (by typing a \$ before the name).

See *Appendix A* for tables of supported registers.

`< This is not a symbolic reference >`

The reference is not a symbol or variable.

Soft-Scope III 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 fixed-length scalar type >`

The length override you're specifying isn't an integer value.

`< This is not an array or pointer >`

The parameter above the carets is not an array.

Perhaps you have provided subscripts on a variable which does not require subscripts. See *Data References*, in *Chapter 4, Examining Data*.

`< 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 using the TYPE command.

## Error Messages

`< This module was not compiled for debugging >`

The module name above the carets contained no debugging information, and SSIII only knows that it's a module and it has no debug information.

Make sure the application was prepared using the specifications given in *Chapter 7, Tools*.

`< This reference contains no lines >`

The referenced source file contains no source lines.

Make sure the application was prepared using the specifications given in *Chapter 7, Tools*.

`< 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 using the EVAL command to view 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 *Data References* in *Chapter 4, Examining Data*.

`< Too many breakpoints are set >`

You have too many breakpoints set at this time.

If possible, delete some of your breakpoints.

`< Too many parameters >`

You're trying to specify a nested RETURN value, and the value is too large (i.e., RETURN(55) where returns aren't nested 55 deep).

`< Unable to run macro compiler >`

Soft-Scope attempted to run the macro compiler, but the compiler didn't run or ran and produced no output.

`< Unexpected response from sskernel: description >`

Soft-Scope III has received an unexpected response from SSKERNEL. The *description* explains what Soft-Scope III had expected to receive from the kernel. This information is provided so that it may be reported to Concurrent Sciences, inc. Technical Support in the event that you should receive this error message.

`< Unknown member of record >`

The member above the carets doesn't exist for that structure. Possibly you've misspelled the member name or you're referencing the wrong structure.

`< Unsupported assignment operation >`

The parameter above the carets cannot be assigned to the value you've attempted to give it (i.e., `GDT[5]=GDT[0]` or `$ax="abcde"`).

`< Use [n][m] for multiple subscripts >`

You're trying to specify too many subscripts in an array. Soft-Scope III only recognizes single subscripts within a set of brackets. Use `ARRAY[1][2]`, not `ARRAY[1,2]`.

`< :name... not found in "filename" >`

The specified module name (*:name*) was not found in the filename specified.

Use the `MODULE` command to examine module name assignments.

## Macro Compiler Error Messages

`<Break is only valid inside while -"token" at line ###, col ###>`

The macro compiler has encountered a break statement outside of a loop.

`< Expected %s - "token" at line ###, col ### >`

While scanning the format string of a print statement, the macro compiler was expecting a string format specifier and didn't find one.

`< Expected closing paren - "token" at line ###, col ### >`

While parsing the current macro's arguments or a WHILE or PRINT statement, the compiler expected a closing parenthesis but got "token" instead.

`< Expected comma - "token" at line ###, col ### >`

Instead of a comma which delimits arguments in a list, the compiler found "token."

`< Expected format string - "token" at line ###, col ### >`

While parsing a PRINT statement, the compiler expected a string indication the format but got "token" instead.

`< Expected identifier - "token" at line ###, col ### >`

The compiler has found "token" instead of identifier.

`< Expected "macro" keyword - "token" at line ###, col ### >`

While compiling the macro file, the compiler was expecting the start of a macro but got "token" instead.

`< Expected macro name - "token" at line ###, col ### >`

Instead of a macro name after the MACRO keyword, the compiler found "token."

`< Expected ON or OFF - "token" at line ###, col ### >`

The token listed was encountered, instead of ON or OFF, while parsing an ECHO statement.

# Macro Compiler Error Messages

< Expected opening brace - "token" at line ###, col ### >

The compiler expected a brace to start the macro but got "token" instead.

< 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.

< Expected parameter - "token" at line ###, col ### >

Instead of a parameter, the compiler found "token."

< Identifier already defined - "token" at line ###, col ### >

The macro compiler encountered a duplicate symbol declaration in the source file.

< Out of symbol space >

The macro compiler has exceeded its limit of 100 symbols (including keywords) in a macro.

< Too many jump targets > or < Too many jumps >

The macro compiler has exceeded its internal limit of 100 jumps per macro.

< Undefined identifier - "token" at line ###, col ### >

The macro compiler has parsed an identifier that it can't find in its symbol table.

< Unexpected end of line >

The macro compiler has unexpectedly encountered the end of a line while parsing for a token.

< Unexpected end of file - "token" at line ###, col ### >

The macro compiler has unexpectedly encountered the end of file while parsing for a token.



## ***Appendix C*** ***SSKERNEL***

# **C**

The Soft-Scope III Kernel, SSKERNEL, runs as an iRMX job and operates as a server for the SS III program itself, SS. The kernel makes it possible for you to control and access your application, and to debug multiple tasks.

All operations which cause program execution, require access to CPU registers, or examine task states, are performed through communication to and from SSKERNEL.

This communication is managed via a private set of mailboxes between Soft-Scope III and SSKERNEL. SSKERNEL manages all breakpoints and protection faults, sending internal messages to Soft-Scope III when they occur. By invoking SSKERNEL as a background job, it's functions are made available to any user on the system, including the initial user.

# The Soft-Scope III Kernel

## *Trapping Faults with “SSKERNEL ON”*

In its default operation, SSKERNEL traps protection faults such as the General Protection Fault (INT 13), and the Stack Fault (INT 12), only if there is an active Soft-Scope III session. When there is no active Soft-Scope III session, SSKERNEL, in its default mode, is inactive.

When a protection fault occurs, the iSDM monitor displays a message indicating the type of fault, followed by the ‘..’ prompt. This can be troublesome for multi-user systems, because if a user encounters a General Protection Fault, and there is no Soft-Scope III session active, all the users in the system come to a screeching halt when the ‘..’ prompt appears at the system terminal.

We created a special mode for SSKERNEL to solve this problem. By adding the keyword, ON, to the SSKERNEL invocation, SSKERNEL will remain actively handling all protection faults, even if no Soft-Scope III session is active:

```
bk sskernel on
```

The advantage is that when a protection fault occurs, the operating system continues to function, and users other than the user encountering the fault are not stopped. When a fault occurs the kernel will print a message to the terminal on which SSKERNEL was invoked. The message is non-interactive, but will give information about the fault.

The message will list the type of the fault and the task token for the task containing the fault. It will also report if the fault occurred in an iRMX subsystem, such as the Nucleus, BIOS, EIOS, Application Loader, or Human Interface.

For example, if a General Protection fault were encountered in a task not being debugged by a Soft-Scope III session, a message similar to *Figure C-1* would be output to the terminal that originally invoked SSKERNEL.

If you have a second terminal, you can invoke SSIII and debug the task that caused the fault. You won't have symbolics, but you can step at the assembly level and issue all other SSIII commands.

```
[Unsolicited break in task 4020: (INT 13) General Protection ]
*****
* NOTE: This is a non-interactive message generated by sskernel **
* If you have a free terminal, invoke Soft-Scope, enter 'task xxx" **
* where xxx is the task token you see in the break message above **
* Other-wise, you may be able to abort the errant application with **
* Control/C. **
*****
```

**Figure C-1** Fault message example

If you use SSKERNEL ON, you must be sure not to use the iRMX KILL command to abort the kernel. You must use the utility, SSABORT (see below).

If you are using SSKERNEL as a fault handler (SSKERNEL ON), and you need to remove SSKERNEL from the system, always use SSABORT. SSKERNEL will clean-up its work files and restore SDB as the fault handler.

Using the iRMX KILL command to kill the background SSKERNEL job will delete the SSKERNEL job, but the interrupt descriptors identifying SSKERNEL as the handler will remain in place, causing severe problems if a fault should occur.

Attempting to KILL other background jobs with a wild card, as in KILL\*, will have the same effect.

If a protection fault occurs within either SSKERNEL or Soft-Scope III itself, SSKERNEL reports a message on the iRMX side similar to that given when you use the kernel as a fault handler outside of Soft-Scope III. It will list the address and type of fault. If the fault is in Soft-Scope III, the message will look like the one shown in *Figure C-2*, or it will look like the error message shown below:

```
<< Internal break in sskernel >>
```



## Using SSABORT



## Internal faults in Soft-Scope III and SSKERNEL

```
<<Internal break in Soft-Scope session #1 (7250:00002145)>>
[ Unsolicited break in task 4020: (INT 13) General Protection ]
*****
* NOTE: This is a non-interactive message generated by sskernel.      **
* If you have a free terminal, invoke Soft-Scope, then enter          **
* 'task xxx' where xxx is the task token you see in the break message **
* above.  Other wise, you may be able to abort the errant application **
* with Control/C.                                                    **
*****
```

**Figure C-2 Fault in SSIII example**

If you see the messages above, use cut and paste or a pencil and paper to record the information listed below:

- Task *token*, where token is taken from the break message
- Contents of the Task window
- Disassembled view of the code that caused the fault (approximately 10 assembly lines beginning with the line the execution pointer is on).
- Contents of the Registers window
- Output of vt \$cs
- Output of vt \$ds
- Output of vt \$ss

This information will help us find the problem when you call technical support.



SSKERNEL will trap Soft-Scope III faults whether the kernel is invoked with BK SSKERNEL ON or OFF.

## ***Appendix D Sample Session***

# **D**

This section provides direction and commentary for the sample session you can run with the Soft-Scope III debugger.

# Sample Session

This session consists of an introduction followed by a series of pages showing the screen-image on the left page and commentary on the right page. If more than one example is on a page, horizontal lines will separate them.

<p><b>ss&gt; command 1</b></p> <hr/> <hr/> <hr/> <hr/> <hr/>	<hr/> <hr/> <hr/> <hr/> <hr/>
<p><b>ss&gt; command 2</b></p> <hr/> <hr/> <hr/> <hr/> <hr/>	<hr/> <hr/> <hr/> <hr/> <hr/>

Page 2

Page 3

**STEP 1.** Attach to the directory /RMX386/DEMO/SSCOPE to make it the default directory. If you haven't done so yet, load SSKERNEL as a background job by typing BK SSKERNEL > :BB:.

**STEP 2.** Verify you have the following files:

<b>csamp</b>	Loadable sample program file
<b>cmain.c</b>	C source file for main module
<b>cutils.c</b>	C source file for procedure module
<b>cmain.lst</b>	C listing file for main module
<b>cutils.lst</b>	C listing file for procedure module

**STEP 3.** Begin the session.

Feel free to stop and start the session at your convenience. The program is written to loop endlessly, and the sample session only takes you through a few iterations.

The program consists of three tasks. A main task creates two other tasks (PROCESS\_TASK and COUNT\_TASK), which send information back and forth via mailboxes.

The screen image is on the left and commentary is on the right.

At times Soft-Scope III will display a prompt showing you possible responses. Once you respond, the prompt disappears from the screen. To help you follow the sample sessions, the prompts are shaded. The desired response to each prompt follows in quotation marks.

Be sure you don't miss the last section of this session. It contains the most pertinent and complete discussion of data references for C.

# Sample Session

## - ss csamp

```
Soft-Scope III (tm) debugger, v1.0
Concurrent Sciences, inc. (C) 1989, 1990 All rights reserved
iRMX III Version
Serial No. xxxx
[ Connected to "Soft-Scope kernel vX.Y - session #1" ]
load csamp
[ Loading OMF-386 STL file, "csamp", Symbols ]

[ Loading macro file "ss.mac" ]
ss>
```

---

## ss> list

```
#1  /*****
#2  /*                                CSAMP Sample Program                                */
#3  /*****
#4
#5      #include <stdarg.h>
#31     #include <stdlib.h>
#158    #include <stdio.h>
#417    #include <string.h>
#521    #include <ctype.h>
#586    #include <rmxc.h>
#2636   #include "cutils.h"
#2639
#2640   #define UINT_8      unsigned char
#2641   #define UINT_16    unsigned short
#2642   #define UINT_32    unsigned long
#2643   #define FALSE      0
#2644   #define TRUE       1
#2645
#2646   #define COUNT_PRIORITY 200 /* Priority of count task !
#2647   #define PROCESS_PRIORITY 200 /* Priority of process task !
#2648   #define BUFF_LEN 125 /* Length of buffer in MSG_S!
[ Top of :CMAIN (cr 1..9 sp) Mode -Find Quit ]      "Q"
```

## Sample Session

The Soft-Scope III debugger is loaded by the operating system, displaying its sign-on, which includes the version number and your serial number.

Soft-Scope III reads options from the initial environmental options file SS.SET and sets the initial options.

Next, Scope-Scope III loads your application, then the macro file, SS.MAC. (See *Macros, Chapter 6* for more information on creating and using macros.)

You are prompted to enter a command.

---

The LIST command lists one screenful of source code from the module CMAIN then displays a prompt listing your possible responses:

- F      Initiates a forward search for a string
- Initiates a backwards search
- M      Toggles the search between case-sensitive and caseless
- Q      Returns you to Scope-Scope III prompt.

Soft-Scope III automatically opened the first module (CMAIN) it finds in your application.

Try out the Find option forward and backwards if you like. Scope-Scope III will highlight the line containing the searched-for string.

When you are done, press “Q” to return you to the Scope-Scope III prompt.

**Note:** Do not type the quotation marks.

# Sample Session

## ss> breakpt main

```
global * :CMAIN.main() [ Breakpoint added ]
```

---

## ss> br process\_task

```
global * :CMAIN.process_task() [ Breakpoint added ]
```

---

## ss> br count\_task

```
global * :CMAIN.count_task() [ Breakpoint added ]
```

---

## ss> go

```
[ Break at :CMAIN.main() ]
#2655     typedef struct {
#2656         short          count;
#2657         unsigned char   fillchar;
#2658         unsigned char   buffer[BUFF_LEN];
#2659     } MSG_STRUC;
#2660
#2661     TOKEN      count_token;          /* Token for count task      !
#2662     TOKEN      process_token;        /* Token for process task    !
#2663     int         counter;              /* count_task's counter     !
#2664
#2665     void far main()
#2666     {
```

Set a breakpoint on the start of MAIN. The breakpoint is assigned a scope of GLOBAL.

---

Set a breakpoint on the start of PROCESS\_TASK. The abbreviation for BREAKPT is BR.

---

Set a breakpoint at the start of COUNT\_TASK.

---

Begin execution. Execution stops when the breakpoint set at the entrance to MAIN is hit.

# Sample Session

## ss> task

```
* 6980 :CMAIN.main()
```

---

## ss> go #2721

```
[ Break at :CMAIN.main#2722 ]  
#2721      while(1) {  
#2722          msg.fillchar = '*';
```

---

## ss> step

```
#2721      while(1) {  
#2722          msg.fillchar = '*';  
[ Auto Into OVER(sp,1..9) Mode Quit Return ]      " "  
#2723          if (msg_count >= MAX_COUNT)  
[ Auto Into OVER(sp,1..9) Mode Quit Return ]      "Q"
```

---

## ss> task

```
* 6980 :CMAIN.main#2723  
69b8 :CMAIN.count_task()  
5630 :CMAIN.process_task()
```

---

## ss> task 69b8

```
Current context: task = 69b8 job = 7620  
[ Break at :CMAIN.count_task() ]  
#2799  
#2800 void far count_task ()  
#2801 {
```

Scope-Scope III reports that MAIN task is at break, and that its task token is 6980. The task token you see will be different because tokens are dynamically assigned.

---

This command produces the same results as the combination of BR #2721 and GO commands. However, the breakpoint set at line #2721 is removed after it is encountered.

---

STEP displays all lines of source code that have the current address, and then the prompt “[ Auto Into OVER(sp,1..9) Mode Quit Return ]”, listing your possible responses.

Press the *<spacebar>* to execute a source line in the procedure MAIN and display the next executable line.

Press “Q” to stop stepping.

---

All three tasks are at break. The asterisk denotes the current task context. Only tasks that are at break are displayed.

---

Change the context to COUNT\_TASK by specifying its token. The current execution point in this task is reported.

**Note:** Type the token for COUNT\_TASK as reported by the TASK command. It will be unique to this debugging session.

## Sample Session

**ss> step**

```
#2799
#2800     void far count_task ()
#2801     {
[ Auto Into OVER(sp,1..9) Mode Quit Return ]      " "
#2802     WORD     exception; /* Status code returned from system !
#2803
#2804     counter = 0;
[ Auto Into OVER(sp,1..9) Mode Quit Return ]      " "
#2805
#2806     while (1) {
#2807         counter++;
[ Auto Into OVER(sp,1..9) Mode Quit Return ]      " "
#2808         delay (1000);
[ Auto Into OVER(sp,1..9) Mode Quit Return ]      "I"
[ Entering :CUTILS.delay() ]
[ Module :CUTILS initializing, using "cutils.lst" ]
#2718
#2719
#2720     void delay (msecs)
#2721     int msecs;
#2722     {
[ Auto Into OVER(sp,1..9) Mode Quit Return ]      "Q"
```

Begin stepping again. Press the *<spacebar>* three times until the call to the procedure DELAY. At the next prompt press “I” to step into DELAY.

Press “Q” to quit stepping.

## Sample Session

**ss> list**

```
#2722 {
#2723     int     sleep100;    /* Each unit of "sleep100" is 100 m!
#2724
#2725         sleep100 = msec / 100;
#2726
#2727     for (; sleep100 >= 0; sleep100--)
#2728         delay_fine (10);    /* delay_fine(10) delays for 100 ms!
#2729     }
#2730
#2731
#2732     static void delay_fine (count)
#2733     int count;
#2734     {
#2735         WORD     exception;
#2736
#2737         if (count != 0) {
#2738             rgsleep (1, &exception); /* One rgsleep unit is 10 msec!
#2739             delay_fine (--count ); /* Call self recursively!
#2740         }
#2741     }
```

[ Module :CUTILS (cr 1..9 sp) Mode -Find Quit ] **"Q"**

## Sample Session

List one screenful of the module CUTILS. You can scroll through the listing by using the following keys:

P	Display a screenful, moving up the module
<Up Arrow>	Display one more line, moving up the module
<Down Arrow>	Display one more line, moving down the module
<carriage return>	Display one more line
<spacebar>	Display one more screenful
1 - 9	Display 1 to 9 more lines, moving down the module

When you have viewed enough source code, press “Q” to quit.

# Sample Session

**ss> step**

```
#2718
#2719
#2720     void delay (msecs)
#2721     int msecs;
#2722     {
[ Auto Into OVER(sp,1..9) Mode Quit Return ]      "M"
steps [ SOURCE Assembly ] calls [ Into OVER ]      "A"
    7840:000004f0 push ebp
[ Auto Into OVER(sp,1..9) Mode Quit Return ]      " "
    7840:000004f1 mov  ebp,esp
[ Auto Into OVER(sp,1..9) Mode Quit Return ]      "M"
steps [ SOURCE Assembly ] calls [ Into OVER ]      "S"
[ Inside ]
#2718
#2719
#2720     void delay (msecs)
#2721     int msecs;
#2722     {
[ Auto Into OVER(sp,1..9) Mode Quit Return ]      "R"
[ Returning to :CMAIN.count_task#2809 ]
#2809         c_data ();
[ Auto Into OVER(sp,1..9) Mode Quit Return ]      "Q"
```

---

**ss> task process\_token**

```
Current context: task = 5630  job = 7620
[ :CMAIN.process_task() ]
#2744
#2745
#2746
#2747     void far process_task ()
#2748     {
```

---

**ss> br -**

```
[ All breakpoints removed ]
```

Start stepping again. At the first prompt, type “M” then “A” to change to assembly level stepping. Press the `<spacebar>` to execute one instruction.

Press “M” then “S” to change the stepping mode back to source level.

Press “R” to return to the calling procedure. Press “Q” to quit.

---

Switch to the context of `PROCESS_TASK`. You can refer to a task by its token name as well as number. It is a good practice to make the tokens of tasks you want to control global variables so that you can reference them from anywhere in your application without needing to remember their numeric values.

---

Remove all breakpoints.

# Sample Session

**ss> br #2785**

```
global * :CMAIN.process_task#2785 [ Breakpoint added ]
```

---

**ss> go**

```
< Task running >
```

---

**!ss> task**

```
* 5630 [ Not currently at a breakpoint ]
6980 :CMAIN.main#2723
69b8 :CMAIN.count_task#2809
```

---

**!ss> task 6980**

```
Current context: task = 6980  job = 7620
[ :CMAIN.main#2723 ]
#2723          if (msg_count >= MAX_COUNT)
```

---

**ss> go**

```
[ Break at :CMAIN.process_task#2785 ]
#2778          (BYTE *)&msg,
#2779          0xffff,
#2780          &exception);
#2781
#2782  /*
#2783  *   Take action on the received message (fill msg.buffer)
#2784  */
#2785  for (i = 0; i < BUFF_LEN; i++) {
```

Set a breakpoint at line #2785.

---

Begin the execution of PROCESS\_TASK.

---

The TASK command reports that PROCESS\_TASK has not hit the breakpoint set above. This is because MAIN task is not running.

---

Change to MAIN task.

---

Start MAIN task again. The breakpoint set in PROCESS\_TASK is now hit.

# Sample Session

**ss> br -**

```
[ All breakpoints removed ]
```

---

**ss> go write msg.buffer[25]**

```
< Write break >
[ Break at :CMAIN.process_task#2787 ]
#2787          }
```

---

**ss> msg.buffer**

```
[0..25] . . . . . 2aH  42  '*'
[26..124] . . . . . 00H  0  \.'
```

---

**ss> br delay**

```
global * :CUTILS.delay() [ Breakpoint added ]
```

---

**ss> go**

```
[ Break at :CUTILS.delay() ]
#2718
#2719
#2720     void delay (msecs)
#2721     int msecs;
#2722     {
```

---

**ss> task**

```
69b8 :CMAIN.count_task#2809
* 5630 :CUTILS.delay()
```

Remove all breakpoints.

---

Set a breakpoint at the data reference `MSG.BUFFER[25]`. You can set up to 4 `WRITE` or `ACCESS` breakpoints. In this example, Scope-Scope III will break when `MSG.BUFFER[25]` is written to.

---

To display the contents of a variable, just type its name at the prompt. Here we confirm that the 25th element of `MSG.BUFFERS` was written to.

---

Set a breakpoint at the entry to the procedure `DELAY`. This procedure is called from both `COUNT_TASK` and `PROCESS_TASK`.

---

Go until that breakpoint is hit.

---

The `TASK` command shows that both `PROCESS_TASK` and `COUNT_TASK` have hit a breakpoint. The asterisk indicates which task is current.

# Sample Session

## ss> task count\_token

```
Current context: task = 69b8 job = 7620
[ :CMAIN.count_task#2809 ]
#2809          c_data();
```

---

## ss> go

```
[ Break at :CUTILS.delay() ]
#2718
#2719
#2720 void delay (msecs)
#2721 int msecs;
#2722 {
```

---

## ss> task

```
* 69b8 :CUTILS.delay()
 5630 :CUTILS.delay()
```

---

## ss> br - delay

```
global * :CUTILS.delay() [ Breakpoint removed ]
```

---

## ss> br :cmain#2807

```
global * :CMAIN.count_task#2807 [ Breakpoint added ]
```

---

## ss> counter

```
00000002H          +2
```

Change the context to `COUNT_TASK`.

---

Go again.

---

Once again both tasks are reported broken at the shared procedure, `DELAY()`, but `COUNT_TASK` is reported as the current task.

---

Remove the breakpoint at the procedure `DELAY()`.

---

Set a breakpoint at line #2807, where the variable `COUNTER` is incremented. Because this line is not in the current module, you must precede it with a colon and its containing module's name.

---

Examine the value of `COUNTER`

# Sample Session

**ss> go**

```
[ Break at :CMAIN.count_task#2807 ]  
#2805  
#2806     while (1) {  
#2807         counter++;
```

---

**ss> go**

```
[ Break at :CMAIN.count_task#2807 ]  
#2805  
#2806     while (1) {  
#2807         counter++;
```

---

**ss> go**

```
[ Break at :CMAIN.count_task#2807 ]  
#2805  
#2806     while (1) {  
#2807         counter++;
```

---

**ss> counter**

```
00000004H     +4
```

---

**ss> br -**

```
[ All breakpoints removed ]
```

Begin execution.

---

Go a second time.

---

Go a third time.

---

Counter has been incremented.

---

Remove all breakpoints.

# Sample Session

**ss> go**

```
[ No breakpoints are set, go anyway? (y/n) ]      "Y"  
< Task running >
```

---

**!ss> counter**

```
00000009H          +9
```

---

**!ss> counter**

```
0000000bH          +11
```

---

**!ss> suspend count\_token**

```
[ Suspend successful ]
```

---

**!ss> counter**

```
00000010H          +16
```

---

**!ss> counter**

```
00000010H          +16
```

Begin execution again. Because there are no breakpoints set you are asked if this is really what you want to do. Answer “Y”. You are returned to the “running” prompt, designated by the exclamation point, “!”. This means that Soft-Scope III’s current task is not at a breakpoint.

---

You can examine a variable while your application is running. Look at the value of COUNTER.

---

Look at it again. Its value has changed.

---

Suspend the task COUNT\_TASK which increments COUNTER.

---

Look at counter twice more.

---

Its value doesn't change because COUNT\_TASK is suspended.

# Sample Session

## !ss> vt count\_token

Object type = 2 Task

Static pri	c8	Dynamic pri	c8	Task state	susp
Suspend depth	01	Delay req	0000	Last exchange	0000
Except handler	5d30:0000610f	Except mode	00	Task flags	00
K-saved SS:SP	3340:00000f28	Containing job	7620	Interrupt task	no

---

## !ss> vk

Ready tasks: 63e8 30e0 0268

Sleeping tasks:

0e28	1538	18c0	0f00	0ea8	0ee8	1268	1290
12b8	12e0	1308	1330	1358	1380	13a8	13d0
13f8	1420	1448	1470	0ed8	14e8	1240	17a0
0e80	3120	3280	3290	4500	1a98	4980	1060
1528	30a8	2498	1f80	1ef8	2fb8	4488	4970
0fe8	0ff8	73f8	1fe0	28a0	2aa0	23a8	5718
4390	6130	1828	1110	6378	1758	0e68	6980
5630	2bf0	2c78					

---

## !ss> resume count\_token

[ Resume successful ]

Use the System Debugger (SDB) view token command to confirm that `COUNT_TASK` has been suspended. All of the SDB commands are supported as Soft-Scope III macros.

---

VK displays a list of the tokens for the currently ready and sleeping tasks. Note that these tokens vary from session to session.

---

Resume `COUNT_TASK`.

# Sample Session

**!ss> counter**

00000016H            +22

---

**!ss> counter**

00000017H            +23

---

**!ss> go :cutils.delay\_fine**

```
[ Break at :CUTILS.delay_fine() ]
#2730
#2731
#2732     static void delay_fine (count)
#2733     int count;
#2734     {
```

---

**ss> macro**

reads	writes	read	write
clearline	stackview	vu	vs
vf	vr	vo	vmo
vmi	vmf	vk	vj
vt	vh	vd	vc
vb	os	loadsegs	bpscope
suspend	resume	task	bptimeout
set_base_10	set_base_16		

Examine COUNTER again.

---

COUNTER has begun incrementing again.

---

Go to the entry of the procedure DELAY\_FINE. Because it is not in the currently open module, you must precede its name with a colon and its containing module name. Notice that the next Soft-Scope III prompt has no “!”, indicating we are again at a breakpoint.

---

The MACRO command lists currently loaded macros. These command extensions to the Soft-Scope III command set allow you to suspend and resume tasks (SUSPEND, RESUME), as well as look at iRMX objects (VT, etc.).

# Sample Session

**ss> stackview 2**

```
          0      2      4      6      8      a      c      e
3340:00000f48                                055f 0000
```

---

**ss> go delay\_fine**

```
[ Break at :CUTILS.delay_fine() ]
#2730
#2731
#2732     static void delay_fine (count)
#2733     int count;
#2734     {
```

---

**ss> stack trace**

```
[ :CUTILS.delay_fine(), Current execution point. ]
[ Return 1 - :CUTILS.delay_fine#2739 called delay_fine() ]
[ Return 2 - :CUTILS.delay_fine#2739 called delay_fine() ]
[ Return 3 - :CUTILS.delay_fine#2739 called delay_fine() ]
[ Return 4 - :CUTILS.delay_fine#2739 called delay_fine() ]
[ Return 5 - :CUTILS.delay_fine#2739 called delay_fine() ]
[ Return 6 - :CUTILS.delay_fine#2739 called delay_fine() ]
[ Return 7 - :CUTILS.delay_fine#2739 called delay_fine() ]
[ Return 8 - :CUTILS.delay_fine#2739 called delay_fine() ]
[ Return 9 - :CUTILS.delay#2728 called delay_fine() ]
[ Return 10 - :CMAIN.count_task#2808 called delay() ]
[ Return 11 - :CQ__TSTART._task_start called count_task() ]
```

STACKVIEW is a simple Scope-Scope III macro. It dumps a specified number of entries from the top of the program stack.

See *Macros, Chapter 6*, for more information.

---

Start execution and break again on DELAY\_FINE. The procedure has been recursively entered.

---

STACK TRACE displays the current source location and traces backwards, a level at a time, showing each previous caller. You may see DELAY\_FINE() called a different number of times for your session.

## Sample Session

### ss> stack trace lines

```
[ :CUTILS.delay_fine(), Current execution point. ]
[ Return 1 - :CUTILS.delay_fine#2739 called delay_fine() ]
#2739 delay_fine (--count );          /* Call self recursively.      !
[ Return 2 - :CUTILS.delay_fine#2739 called delay_fine() ]
#2739 delay_fine (--count );          /* Call self recursively.      !
[ Return 3 - :CUTILS.delay_fine#2739 called delay_fine() ]
#2739 delay_fine (--count );          /* Call self recursively.      !
[ Return 4 - :CUTILS.delay_fine#2739 called delay_fine() ]
#2739 delay_fine (--count );          /* Call self recursively.      !
[ Return 5 - :CUTILS.delay_fine#2739 called delay_fine() ]
#2739 delay_fine (--count );          /* Call self recursively.      !
[ Return 6 - :CUTILS.delay_fine#2739 called delay_fine() ]
#2739 delay_fine (--count );          /* Call self recursively.      !
[ Return 7 - :CUTILS.delay_fine#2739 called delay_fine() ]
#2739 delay_fine (--count );          /* Call self recursively.      !
[ Return 8 - :CUTILS.delay_fine#2739 called delay_fine() ]
#2739 delay_fine (--count );          /* Call self recursively.      !
[ Return 9 - :CUTILS.delay#2728 called delay_fine() ]
#2728 delay_fine (10);                /* delay_fine(10) delays for 100 ms!
[ Return 10 - :CMAIN.count_task#2808 called delay() ]
#2808 delay (1000);
[ Return 11 - :CQ_TSTART._task_start called count_task() ]
[ Module :CQ_TSTART initializing ]
```

---

### ss> go return(10)

```
[ Break at :CMAIN.count_task#2809 ]
#2809          c_data ();
```

Scope-Scope III displays the source line corresponding to the caller at each level.

Find the number of the last return before the return to your start up code. In this case it is 10.

---

Type `GO RETURN(X)`, where `X` is the number of the last return before the return to your start up code. For this sample session, that is the 10th return. It may be a different number for your sample session.

Soft-Scope III executes until it returns to 10th most recent caller, `COUNT_TASK`. `GO RETURN` would have returned one level to the `DELAY_FINE` procedure.

# Sample Session

**ss> go**

```
[ No breakpoints are set, go anyway? (y/n) ]           "Y"  
< Monitor breakpoint >  
[ Break in Unknown module (0000:00000000) ]  
< Address 0000 -> GDT[0] - Segment not present >
```

---

**ss> task**

```
* 69b8 [ Not currently at a breakpoint ]  
5630 :CUTILS.delay()
```

---

**ss> task process\_token**

```
Current context: task = 5630 job = 7620  
[ :CUTILS.delay() ]  
#2718  
#2719  
#2720 void delay (msecs)  
#2721 int msecs;  
#2722 {
```

---

**ss> bpscope**

```
Current bpscope is 'global'
```

---

**ss> bpscope task**

```
Current bpscope is 'task'
```

Attempt to begin execution again. Notice what Soft-Scope III reports. This is because `COUNT_TASK` is already running, i.e., not at break and correct register values cannot be determined for a task not at a break.

---

The `TASK` command confirms that `COUNT_TASK` is not currently at a breakpoint, and therefore cannot be entered with Soft-Scope III. At this point, you could cause `COUNT_TASK` to break by setting a breakpoint at a place you knew `COUNT_TASK` would execute.

---

Change the current context to `PROCESS_TASK`.

---

Display the current breakpoint scope. This is the default scope which is assigned to each breakpoint set. Currently, the default mode for the breakpoint scope is `GLOBAL`. Any task that executes code where a breakpoint is set will report a break.

---

Change the default scope to `TASK`. Now breakpoints set can only be triggered by the current task. Any other task executing code where a breakpoint is set will be allowed to continue.

## Sample Session

### ss> br delay

```
task * :CUTILS.delay() [ Breakpoint added ]
```

---

### ss> go

```
[ Break at :CUTILS.delay() ]  
#2718  
#2719  
#2720 void delay (msecs)  
#2721 int msecs;  
#2722 {
```

---

### ss> task

```
* 5630 :CUTILS.delay()
```

---

### ss> bpscope global

```
Current bpscope is 'global'
```

---

### ss> br delay\_fine task

```
task * :CUTILS.delay_fine() [ Breakpoint added ]
```

---

### ss> br

```
task * :CUTILS.delay_fine()  
task * :CUTILS.delay()
```

Set a breakpoint at the entrance to DELAY.

---

Begin execution again.

---

TASK reports that only PROCESS\_TASK hit the breakpoint set at the entrance to the procedure DELAY, although COUNT\_TASK also calls this procedure.

---

Change the default scope back to GLOBAL.

---

You can override the default scope by specifying a scope with the BREAKPT command.

---

List all currently set breakpoints. Notice that the scope of the breakpoint at DELAY did not change to GLOBAL when we changed the default scope to GLOBAL above.

## Sample Session

```
ss> br delay global  
  ^^^^^^^^^^^
```

```
< Breakpoint already set >
```

---

```
ss> br - delay
```

```
task * :CUTILS.delay() [ Breakpoint removed ]
```

---

```
ss> br delay
```

```
global * :CUTILS.delay() [ Breakpoint added ]
```

---

```
ss> br
```

```
global * :CUTILS.delay()  
task * :CUTILS.delay_fine()
```

---

```
ss> go
```

```
[ Break at :CUTILS.delay_fine() ]  
#2730  
#2731  
#2732     static void delay_fine (count)  
#2733     int count;  
#2734     {
```

---

```
ss> br -
```

```
[ All breakpoints removed ]
```

You cannot change a current breakpoint's scope by respecifying a different scope with the breakpoint command.

---

You must explicitly remove the breakpoint.

---

And reset it.

---

Now the breakpoint set at DELAY has the current default scope.

---

Begin execution again.

---

Remove all breakpoints.

## Sample Session

**ss> disasm #2727**

```
[ :CUTILS.delay#2727 ]
#2726
#2727      for (; sleep100 >= 0; sleep100-)
          7840:00000507 jmp 00000513H           ; $+7
          7840:0000050c mov eax,[ebp-04H]
          7840:0000050f dec eax
          7840:00000510 mov [ebp-04H],eax
          7840:00000513 cmp [ebp-04H],+00H       ; Imm = 0
          7840:00000517 js #2729                  ; $+15
```

---

**ss> br 7840:510**

```
global * Inside :CUTILS.delay#2727 (7840:00000510) [ Breakpoint added ]
```

---

**ss> go**

```
[ Break inside :CUTILS.delay#2727 (7840:00000510) ]
#2726
#2727      for (; sleep100 >= 0; sleep100--)
```

---

**ss> \$cs:\$eip**

```
7840:00000510
```

---

**ss> br -**

```
[ All breakpoints removed. ]
```

Soft-Scope III disassembles line #2727. Note that the selectors displayed for your debugging session will be different than the ones displayed here. If you have rebuilt the sample program, the offsets may also be different.

---

Set a breakpoint at the absolute address of the second MOV instruction.

---

Execute until the breakpoint we just set.

---

A look at the instruction pointer confirms that execution stopped where we set the breakpoint. Note that register symbol names must be preceded by a dollar sign "\$".

---

Remove the breakpoint at the absolute address. `BREAKPT` - without any parameters removes all currently set breakpoints.

# Sample Session

## ss> reg

```
eax=00000009 cs=7840 eip=00000510
ebx=00000e9a ss=33f8 esp=00000f38 ebp=00000f3c
ecx=00000000 ds=7600 esi=00005d30 fs=09e8
edx=000000c2 es=7600 edi=00007600 gs=0fb8
efl=00000206 [vm rf nt iopl=0 of df IF tf sf zf af PF cf]
```

---

## ss> reg all

```
eax=00000009 cs=7840 eip=00000510
ebx=00000e9a ss=33f8 esp=00000f38 ebp=00000f3c
ecx=00000000 ds=7600 esi=00005d30 fs=09e8
edx=000000c2 es=7600 edi=00007600 gs=0fb8
efl=00000206 [vm rf nt iopl=0 of df IF tf sf zf af PF cf]
cr0=7fffffff [pg et TS em MP PE] ldtr=02a0 tr=0278
cr2=00000000 [pfla=00000000] gdb=00008000 gdl=7cff
cr3=00000000 [pdbr=00000] idb=0000fd00 idl=07ff
```

---

## ss> \$esi = 20

```
[ Was ] 00005d30H 23856
```

---

## ss> \$esi

```
00000014H 20
```

---

## ss> task count\_token

```
Current context: task = 69b8 job = 7620
```

Look at the registers.

---

This displays all of the registers including those associated with the processor data structures GDT and IDT.

---

You can change the value of any register.

---

Yes, it has been changed.

---

Switch tasks to COUNT\_TASK.

# Sample Session

## ss> help

Command Syntax:

```
BPSCOPE [ TASK | JOB | GLOBAL ]
BPTIMEOUT [decnumber32]
BREAKPT [-] [coderef] [ TASK | JOB | GLOBAL ]
BREAKPT [-] WRITE|ACCESS memref [ TASK | JOB | GLOBAL ]
CONSOLE [devicename [termtype]]
[count] DISASM [ALL] [NOLINES] [coderef] [TO coderef]
[count] DUMP [ BYTE | WORD | DWORD ] [memref]
DUMP [ BYTE | WORD | DWORD ] memref [TO memref]
EVAL [memref | coderef]
EXIT
GO [WRITE | ACCESS] memref
GO coderef
GO RETURN
HELP [topic]
LINE [coderef]
[count] LIST [lineref | TO lineref]
LIST lineref TO lineref
LOAD filename
LOAD [SYMBOLS filename]
LOADSEGS segtoken jobtoken filename
LOG [devicename | filename]
```

**"Q"**

## ss> go c\_data

```
[ Break at :CUTILS.c_data() ]
#521      #include <ctype.h>
#586      #include <rmxc.h>
#2636
#2637      /* Forward declarations */
#2638      extern void delay_fine();
#2639      extern void delay();
#2640
#2641      /*****!
#2642      /*
#2643      /* C_DATA
#2644      /*
#2645      /*****!
#2646
#2647      void c_data()
#2648      {
```

If you cannot remember the syntax for a particular command, or just want more information about it, use the HELP command.

HELP alone will give you a list of commands and the syntax for each.

About 230k of information is available through this utility, All searches are quick because the help file (SS.HLP) is indexed internally.

At the end of a screenful, enter "Q" at the prompt.

---

Soft-Scope III executes until it encounters the entry to the procedure C\_DATA.

# Sample Session

**ss> name\_init**

```
[0] . . . . . 760000000094H 7600:00000094
[1] . . . . . 76000000009dH 7600:0000009d
[2] . . . . . 7600000000a6H 7600:000000a6
```

---

**ss> name\_init[2]**

```
7600000000a6H 7600:000000a6
```

---

**ss> name\_init[1..2]**

```
[1] . . . . . 76000000009dH 7600:0000009d
[2] . . . . . 7600000000a6H 7600:000000a6
```

---

**ss> string at name\_init[1]**

```
`Steve  \0' 9
```

---

**ss> string at name\_init[1] = "jessica"**

```
[ Was ] `Steve  \ 8
```

You can display an entire array.

---

You can also look at one element of an array.

---

Or a range of elements.

---

By using the string override and the AT operator, you can display the string that `NAME_INIT[1]` points to.

---

You can alter the contents of a string variable by following its name with an equal sign and a new string.

Instead of typing this command, press the *<up arrow>* key. The command you previously entered is displayed at the prompt. Soft-Scope III maintains a history of commands you enter, which is accessible with the arrow keys.

# Sample Session

**ss> dump name\_init**

```
          0 1 2 3 4 5 6 7 8 9 a b c d e f 0123456789abcdef
7600:00000070 94 00 00 00 00 76 9d 00 00 00 00 76 a6 00 00 00 ....v.....v.....
7600:00000080 00 76                                     .v
```

---

**ss> 2 dump word name\_init**

```
          0      2      4      6      8      a      c      e
7600:00000070 0094 0000
```

---

**ss> dump string at name\_init[1]**

```
          0 1 2 3 4 5 6 7 8 9 a b c d e f 0123456789abcdef
7600:00000000                                     6a 65 73                jes
7600:000000a0 73 69 63 61 00                                     sica.
```

---

**ss> go #2714**

```
[ Break at :CUTILS.c_data#2714 ]
#2714          if (oldcust != NULL)
```

---

**ss> type array1**

```
array[0..11] of char          [ local, stack-based ]
```

Soft-Scope III determines the length of `NAME_INIT` and displays that many bytes, with a hex display on the left and the corresponding ASCII field on the right. Because its display is more compact, the `DUMP` command is useful for references to variables which would create a long display.

---

You can specify a count with `DUMP`, or dump in `WORD` or `DWORD` format.

---

Soft-Scope III dumps the string that the pointer `NAME_INIT[1]` points to.

---

Soft-Scope III executes until line #2714, past the initialization of several data structures.

---

The `TYPE` command will display typing information about a variable. Here we see that `ARRAY1` is a local, stack-based array of `char` containing 12 elements.

# Sample Session

**ss> array1**

```
[0] . . . . . 20H +32  \ \
[1] . . . . . 21H +33  \ ! '
[2] . . . . . 22H +34  \ " \
[3] . . . . . 23H +35  \ # '
[4] . . . . . 24H +36  \ $ '
[5] . . . . . 25H +37  \ % '
[6] . . . . . 26H +38  \ & '
[7] . . . . . 27H +39  \ ' '
[8] . . . . . 28H +40  \ ( \
[9] . . . . . 29H +41  \ ) '
[10] . . . . . 2aH +42  \ * '
[11] . . . . . 2bH +43  \ + '
```

---

**ss> type struc1**

```
structure [ local, stack-based ]
  xchar . . . . . char
  xshort. . . . . int
  xint. . . . . long
  xuint. . . . . dword
```

---

**ss> struc1**

```
structure
  xchar . . . . . 21H +33  \ ! '
  xshort. . . . . fff6H -10
  xint. . . . . 00000001H +1
  xuint. . . . . 00000006H 6
```

---

**ss> struc1.xchar**

```
21H +33  \ ! '
```

Here you see each of the 12 elements of ARRAY1, displayed in character format, showing hex and ASCII.

---

Display typing information about the structure STRUC1.

---

Display the value of the structure STRUCT1.

---

You can qualify a structure to look at one element.

# Sample Session

**ss> type enet\_pkt**

```
structure [ local, stack-based ]
  crc . . . . . bitfield :2
  data. . . . . word
  pkt_type. . . . . bitfield :3
  source_addr . . . . . bitfield :4
  dest_addr . . . . . bitfield :4
  preamble. . . . . bitfield :3
```

---

**ss> enet\_pkt**

```
structure
  crc . . . . . 1H 1
  data. . . . . 1000H 4096
  pkt_type. . . . . 3H 3
  source_addr . . . . . 2H 2
  dest_addr . . . . . cH 12
  preamble. . . . . 7H 7
```

---

**ss> dword enet\_pkt = 0ffffffh**

[ Was ] f94c4001H 4182523905

---

**ss> enet\_pkt**

```
structure
  crc . . . . . 3H 3
  data . . . . . 3fffH 16383
  pkt_type . . . . . 7H 7
  source_addr . . . . . fH 15
  dest_addr . . . . . fH 15
  preamble . . . . . 7H 7
```

Soft-Scope III understands bit fields.

Any display of one of these fields will represent only the number of bits assigned to the field, and any assignment will be correspondingly limited. For instance, the maximum value you see for `ENET_PKT.PKT_TYPE` is 7 (111B).

---

Display the contents of `ENET_PKT`.

---

Modify the contents of the entire structure at once, using an overriding type of Dword (32-bit unsigned). This sets all 32 bits of `ENET_PKT` to 1.

**Note:** Since all hexadecimal numbers must begin with a digit, you need to type a zero at the beginning of the value.

---

Confirm that `ENET_PKT` was modified.

# Sample Session

## ss> type customerlist

```
array[0..2] of structure          [ local, stack-based ]
  name. . . . . array[0..7] of char
  phone . . . . . array[0..6] of char
  linkfor . . . . . pointer -> structure
```

---

## ss> type \*customer

```
structure
  name. . . . . array[0..7] of char
  phone . . . . . array[0..6] of char
  linkfor . . . . . pointer -> structure
```

---

## ss> go #2714

```
[ Break at :CUTILS.c_data#2714 ]
#2714          if (oldcust != NULL)
```

---

## ss> oldcust->name

```
[0] . . . . . 42H   +66 'B'
[1] . . . . . 65H   +101 'e'
[2] . . . . . 74H   +116 't'
[3] . . . . . 68H   +104 'h'
[4..7] . . . . . 20H   +32 ' '
```

---

## ss> oldcust

```
334000000f70H 3340:00000f70
```

Although CUSTOMERLIST is an array, we will use it as a linked list to demonstrate Soft-Scope III's data referencing features.

---

You can display type information about a variable by dereferencing its pointer.

---

Loop once through the initialization of CUSTOMERLIST.

---

Using C-like syntax, you can dereference the element NAME of the structure CUSTOMERLIST.

---

Let's confirm that this dereferencing is correct by looking at the value of OLDCUST.

# Sample Session

**ss> &customerlist[0]**

3340:00000f70

---

**ss> \*oldcust**

structure

name

[0]	. . . . .	42H	+66	'B'
[1]	. . . . .	65H	+101	'e'
[2]	. . . . .	74H	+116	't'
[3]	. . . . .	68H	+104	'h'
[4..7]	. . . . .	20H	+32	' '

phone

[0..2]	. . . . .	35H	+53	'5'
[3]	. . . . .	31H	+49	'1'
[4]	. . . . .	32H	+50	'2'
[5]	. . . . .	33H	+51	'3'
[6]	. . . . .	34H	+52	'4'

linkfor . . . . . 000000000000H 0000:00000000

---

**ss> type c\_data**

procedure

---

**ss> lengthof customerlist**

00000003H            3

---

**ss> sizeof customerlist**

00000048H            72

The address of `CUSTOMERLIST[0]` is the value of the pointer `OLDCUST`. The ampersand (`&`) is an address operator.

---

You can display the entire structure that `OLDCUST` points to.

---

You can also use the `TYPE` command to obtain information about procedures. If `C_DATA` returned a value, the `TYPE` command would display the type of the return value.

---

Soft-Scope III has several built-in functions that can be used in any valid expression. `LENGTHOF` returns the length of the array `CUSTOMERLIST`.

---

`CUSTOMERLIST` uses 72 bytes.

# Sample Session

## ss> eval customerlist

```
[0] structure
    name
        [0] . . . . . 42H +66 'B'
        [1] . . . . . 65H +101 'e'
        [2] . . . . . 74H +116 't'
        [3] . . . . . 68H +104 'h'
        [4..7] . . . . . 20H +32 ' '
    phone
        [0..2] . . . . . 35H +53 '5'
        [3] . . . . . 31H +49 '1'
        [4] . . . . . 32H +50 '2'
        [5] . . . . . 33H +51 '3'
        [6] . . . . . 34H +52 '4'
    linkfor . . . . . 000000000000H 0000:00000000
                Address 0000->GDT[0] - Segment not present

[1] structure
    name
        [0] . . . . . 6aH +106 'j'
        [1] . . . . . 65H +101 'e'
        [2..3] . . . . . 73H +115 's'
        [4] . . . . . 69H +105 'i'
        [5] . . . . . 63H +99 'c'
        [6] . . . . . 61H +97 'a'
[ More(sp,cr,1..9) Quit ] "Q"
```

---

## ss> eval c\_data

```
Module :CUTILS (#1 to #2717)
Code 7840:00000370 to 7840:000004ee (383 bytes)
```

---

## ss> eval selectorof oldcust

```
3340H gdt[1640] rpl=0 Offsets 00000000..00000fff
```

EVAL displays a variable in bases other than those normally displayed. If the variable is a pointer, its associated descriptor entry is displayed.

---

Procedures can also be evaluated. EVAL displays the location of the code, its length, and the corresponding high level lines.

---

You can use the EVAL command together with the SELECTOROF built-in function to display information about the descriptor entry for the pointer OLDCUST. The legal range of offsets and the requestor's privilege level are displayed.

# Sample Session

**ss> eval gdt[1640]**

Data R/W-AC 001ebda0L Lim=00ffffH DPL=0 gBaP

---

**ss> gdt**

```
[0] Empty
[1] Data R/W-AC Offsets 0000..7cff DPL=0
[2] Data R/W-AC Offsets 0000..07ff DPL=0
[3] Data R/W-AC Offsets 00000000..fffffff DPL=0
[4] Code RD Offsets 00000000..fffffff DPL=0
[5] Code RD-AC Offsets 00000000..00006263 DPL=0
[6] Data R/W-AC Offsets 00000000..0000111a DPL=0
[7] Data R/W-AC Offsets 0000..1fff DPL=0
[8] Code RD-AC Offsets 00000000..000015a2 DPL=0
[9] Data R/W-AC Offsets 00000000..0000065b DPL=0
[10] 386 Call gate 0028:00002efc Count=2 DPL=0
[11] 386 Call gate 0028:00000a4c Count=12 DPL=0
[12] 386 Call gate 0028:0000307c Count=0 DPL=0
[13] Avail 386 TSS Offsets 0000..0067 DPL=0
[14] Avail 386 TSS Offsets 0000..0067 DPL=0
[15] Empty
[16] Avail 386 TSS Offsets 0000..0067 DPL=0
[17] Avail 386 TSS Offsets 0000..0067 DPL=0
[18] Avail 386 TSS Offsets 0000..0067 DPL=0
[19] Avail 386 TSS Offsets 0000..0067 DPL=0
[20] Empty
[21] Avail 386 TSS Offsets 0000..0067 DPL=0
[22] Empty
```

[ More(sp,cr,1..9) Quit ] "Q"

Evaluating the descriptor itself displays its base and limit, access rights and which of the miscellaneous bits (such as for granularity) are set.

---

You can access 80386 built-in data structures. Press "Q" at the prompt to quit the display. You can also access a single entry, as in GDT[1640].

# Sample Session

**ss> set**

```
[ Options currently set ]  
  targ.dev. . . . . "sskernel"  
  base. . . . . "10"
```

---

**ss> set cmd.prompt = "SSIII> "**

SSIII>

---

**SSIII> quit**

SET without parameters lists the current options environment.

---

This changes the command prompt. The SET command enables you to alter many aspects of the Soft-Scope III environment.

---

Exit Soft-Scope III.



## Symbols

- !, Running prompt indicator 45
- !, Truncated line indicator 31
- &, addressof operator 71
- \*, pointer dereference operator 63
- >, structure pointer operator 63
- .., symbol operator 66
- ..., subscript range operators 61
- ...x, open-ended operators 61
- :, module operator 66
- ?, in displays 67
- ?, in task displays 53

## A

- Abort macros 96
- Addresses
  - In type overrides 70
- Applications
  - File types supported 3
- Arrays, examining 60
- ASM286 and ASM386 103
- at, (operator) 71

## B

- Base (set option)
  - Default value 80
  - Description 90
- Binary numbers 80

- Bitfields, referencing 62
- BLD386 105
- BND286 and BND386 104
- BPSCOPE command 44
- BPTIMEOUT command 45
- Break, in macros 96
- Breakpoints
  - Debug registers limitations 47
  - Displaying 44
  - Execution 46
  - Hardware 46
  - Missed 47
  - Number possible 2
  - Scope 44
- BREAKPT command 44
- Built-in functions 78

## C

- Calls
  - Displaying 50
  - Stepping into or over 36
- Change task context 52
- Character strings 84
- cmd.history (option)
  - Description 90
- cmd.initial (option)
  - Description 90
- cmd.macro (option)
  - Description 90

# Index

- cmd.prompt (option)
  - Description 90
- cmd.silent (option)
  - Description 90
- Commands
  - BPSCOPE 44
  - BPTIMEOUT 45
  - BREAKPT 44
  - CONSOLE 20
  - DISASM 48
  - DUMP 74
  - EVAL 64, 76
  - EXIT/QUIT 21
  - GO 40
  - HELP 17
  - LINE 34
  - LIST 30
  - LOAD 12
  - LOADSEGS 15
  - LOG 18
  - MACRO 94
  - MODULE 43
  - REG 76
  - RESUME 57
  - SET 88
  - SS 10
  - STACK 50
  - STEP 36
  - SUSPEND 57
  - Syntax elements 25
  - Syntax summary 26
  - SYSTEM 20
  - TASK 52
  - TYPE 65
  - VERSION 21
- CONSOLE command 20
- Context (task), changing 52
- Control registers, table of 117
- Control statements, macro 96
- CPU structures, examining 76

## D

- Data Types, table of 112
- Debug registers 47
- Decimal numbers 80
- Default command, LINE 34
- Descriptors, examining 76
- Device drivers, debugging 12
- DISASM command 48
- Display, initial 11
- Displaying
  - Assembly code 48
  - Breakpoints 44
  - Code 30
  - Code symbols 34
  - Descriptors 76
  - Help 17
  - Listing file assignments 43
  - Memory 74
  - Procedure calls 50
  - Registers 76
  - Tasks at break 52
  - Tasks from other SSIII sessions 53
  - The current execution point 34
- DUMP command 74

## E

- Echo on/off, in macros 97
- Editing functions
  - Command history 16
  - Deleting text 16
  - Moving the cursor 16
- Editorial conventions 6
- Error messages
  - General 125
- Escape sequences, string 84
  - Table of 85
- EVAL command 64, 76
- Evaluate data 64
- Execution
  - Stepping 36

Execution breakpoints 46  
 EXIT/QUIT commands 21  
 Exponential numbers 80

**F**

Faults, trapping 146

## Files

Assignments, listing files 43  
 Extensions 43  
 Macro source 95  
 Soft-Scope III 8  
 SS.HLP 17

Flags registers, table of 116

Floating-point numbers 80

FORTRAN-386 109

## Functions

In macros 97  
 SSIII built-in 78

**G**

GDT 76

General purpose registers, table of 115

GO command 40

GO RETURN 42

GO, with no breaks 40

**H**

Hardware breakpoints 46

HELP command 17

Hexadecimal numbers 80

Human interface applications 12

**I**

I/O port 78

iC-286 and iC-386 106

IDT 76

If/Else, in macros 96

Initial display 11

Installation

Optional methods 8

Procedure 9

Requirements 8

Interactive list mode 30

Interactive list mode, table of functions 32

Invoke SSIII 10

iRMX Kill command, warning 10

**K**

Kill command, iRMX 10

**L**

LDT 76

LENGTHOF 78

LINE command 34

Line length 31

LIST command 30

LOAD command 12

LOADSEGS command 15

LOG command 18

Logical addresses, as references 69

**M**

MACRO command 94

## Macros

Control statements 96

Error messages 142

Examples 99

Functions 97

Loading 94

Parameters 98

Source files 95

Memory references 68

Missed Breakpoints 47

MODULE command 43

**N**

NPX registers, table of 116

## Numbers

Default bases, table of 81

# Index

## O

- OFFSETOF 78
- Operators
  - Arithmetic 82
  - Logical 82
  - Precedence, tables of 83
  - Symbolic 82
  - Table of 114
- Options
  - base 90
  - cmd.history 90
  - cmd.initial 90
  - cmd.macro 90
  - cmd.prompt 90
  - cmd.silent 90
  - rmxload.excep 91
  - src.path 91
  - src.tab 91
  - sym.case 92
  - sym.descriptor 92
  - sym.pointer 92
  - tmp.path 92

## P

- Parameters, macro 98
- Physical addresses, as references 69
- PL/M 286 and PL/M 386 107
- Pointers
  - Dereferencing 63
  - Examining 63
- PORT 78
- Print, in macros 97
- Protected-mode registers, table of 117

## Q

- QUIT command 21

## R

- Reference scoping 66
- Reference summary 86

- Reference variables 60
- Referencing Structures
  - Arrays of structures 62
  - Individual elements 62
- REG command 76
- Registers
  - Control, table of 117
  - Display, described 77
  - Examining 76
  - Flags, table of 116
  - General purpose, table of 115
  - Modifying 77
  - NPX, table of 116
  - Protected-mode, table of 117
  - Segment, table of 116
- RESUME command 57
- Resume tasks 57
- RETURN 78
- Return, in macros 96
- Return, with the GO command 42
- rmxload.excp 91
- RQALOAD, system call 15

## S

- Sample programs
  - Directory located 8
- Scoping, references 66
- SDB commands 22
- Segment registers, table of 116
- SELECTOROF 78
- SET command 88
- SIZEOF 78
- src.path (option)
  - Description 91
- src.tab (option)
  - Description 91
- SS command 10
- SSABORT 10, 147
- SSKERNEL 146
  - Invocation 10
  - limitations 10

- STACK command 50
  - Stack-based variables, referencing 67
  - STEP command 36
  - Stepping
    - Change modes 37
    - Default mode 38
    - In assembly mode 36
    - Into and Over calls 36
    - Table of options 37
  - String escape sequences 84
    - Table of 85
  - Strings 84
  - Structures, referencing 62
  - SUSPEND command 57
  - Suspend tasks 57
  - Suspending tasks 57
  - Switching tasks 52
  - sym.case (option)
    - Description 92
  - sym.descriptor (option)
    - Description 92
  - sym.pointer (option)
    - Description 92
  - SYSTEM command 20
  - System Debug commands (SDB) 22
    - Loading with cmd.macro 22
    - Table of 23
- T**
- TASK command 52
  - Tasks
    - Changing context 52
    - Displaying 52
    - Suspending 57
  - tmp.path 92
  - Tools
    - ASM286 and ASM386 103
    - BLD386 105
    - BND286 and BND386 104
    - Fortran-386 109
    - iC-286 and iC-386 106
    - PL/M 286 and PL/M 386 107
  - Trapping faults 146
  - Troubleshooting 28
  - TYPE command 65
  - Type overrides
    - Definition and usage 70
    - Table of 112
- U**
- Unions, referencing 60
- V**
- Variables
    - Examining 60
    - Stack based 67
  - VERSION command 21
- W**
- While statement, in macros 96

