



# **dScope<sup>TM</sup> for Windows**

**Source-Level Debugger  
High-Speed Simulator  
and Target Debugger**

**User's Guide 01.97**

Information in this document is subject to change without notice and does not represent a commitment on the part of the manufacturer. The software described in this document is furnished under license agreement or nondisclosure agreement and may be used or copied only in accordance with the terms of the agreement. It is against the law to copy the software on any medium except as specifically allowed in the license or nondisclosure agreement. The purchaser may make one copy of the software for backup purposes. No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or information storage and retrieval systems, for any purpose other than for the purchaser's personal use, without written permission.

© Copyright 1990-1997 Keil Elektronik GmbH and Keil Software, Inc.  
All rights reserved.

Keil C51™ and dScope™ are trademarks of Keil Elektronik GmbH.  
Microsoft®, MS-DOS®, and Windows™ are trademarks or registered trademarks of Microsoft Corporation.  
IBM®, PC®, and PS/2® are registered trademarks of International Business Machines Corporation.  
Intel®, MCS® 51, ASM-51®, and PL/M-51® are registered trademarks of Intel Corporation.

Every effort was made to ensure accuracy in this manual and to give appropriate credit to persons, companies, and trademarks referenced herein.



# Preface

This manual describes the dScope™ symbolic, source-level debugger. dScope is both a simulator and a target debugger (when interfaced to your target hardware using a monitor program). dScope lets you debug programs written for the MCS® 51, MCS® 251, and 166 microcontroller families using the standard Microsoft Windows interface. This manual assumes that you are familiar with DOS and Microsoft® Windows™. Knowledge of a high-level programming language like C is strongly suggested.

The following chapters are included in this manual:

“Chapter 1. Introduction,” introduces dScope, describes how to compile and assemble programs for debugging, and shows how to start and exit the debugger.

“Chapter 2. User Interface,” describes the user interface components including windows and dialog boxes that you may use when debugging a target program.

“Chapter 3. Expressions,” describes how to use constants, symbols, and operators to create simple and complex expressions you may use with dScope.

“Chapter 4. Executing Code,” provides a tutorial on how to load and debug several example target programs including programs that utilize bank switching and the RTX real-time executives.

“Chapter 5. Examining and Modifying Memory,” provides a tutorial on how to view, display, and modify program variables and data in your target program.

“Chapter 6. Using Breakpoints,” explains how you can use breakpoints to easily locate problems in your target program.

“Chapter 7. Using Watchpoints,” explains how you can use watchpoints to monitor variables and expressions.

“Chapter 8. Tutorial,” gives a step-by-step explanation of how to load and debug a program.

“Chapter 9. Commands,” provides a categorical and alphabetical listing of all commands supported by dScope.

“Chapter 10. Functions,” shows how to create user functions and signal functions that augment the capabilities of dScope.

“Chapter 11. Error Messages,” lists the errors and warnings you may encounter when using dScope.

“Appendix A. CPU Driver Files,” describes the DLL drivers that are provided with dScope.

“Appendix B. Improving Debugger Performance,” describes how to change settings to improve the speed of the dScope simulator.

“Appendix C. Debugging Bank Switching Programs,” shows how to load and debug code banking applications.

---

**NOTE**

*This manual assumes that you are familiar with Microsoft Windows and the hardware and instruction sets of the 8051, 251, or 166 microcontrollers.*

---

# Document Conventions

This document uses the following conventions:

Examples	Description
<b>README.TXT</b>	Bold capital text is used for the names of executable programs, data files, source files, environment variables, and commands you enter at the MS-DOS command prompt. This text usually represents commands that you must type in literally. For example:  <div style="text-align: center;"> <b>CLS</b>                      <b>DIR</b>                      <b>BL51.EXE</b> </div> Note that you are not required to enter these commands using all capital letters.
<b>Courier</b>	Text in this typeface is used to represent information that displays on screen or prints at the printer.  This typeface is also used within the text when discussing or describing command line items.
<i>Variables</i>	Text in italics represents information that you must provide. For example, <i>projectfile</i> in a syntax string means that you must supply the actual project file name.  Occasionally, italics are also used to emphasize words in the text.
Elements that repeat...	Ellipses (...) are used to indicate an item that may be repeated.
Omitted code	Vertical ellipses are used in source code listings to indicate that a fragment of the program is omitted. For example:  <pre>void main (void) { . . . while (1);</pre>
[Optional Items]	Optional arguments in command-line and option fields are indicated by double brackets. For example:  <pre>C51 TEST.C PRINT [(filename)]</pre>
{ opt1   opt2 }	Text contained within braces, separated by a vertical bar represents a group of items from which one must be chosen. The braces enclose all of the choices and the vertical bars separate the choices. One item in the list must be selected.
<b>Keys</b>	Text in this sans serif typeface represents actual keys on the keyboard. For example, "Press <b>Enter</b> to continue."



# Contents

<b>Chapter 1. Introduction .....</b>	<b>1</b>
System Requirements .....	2
Preparing Programs for dScope .....	3
Source Code Considerations .....	3
Preparing A51 Assembler Programs .....	4
Preparing C51 Compiler Programs .....	5
Preparing PL/M-51 Compiler Programs .....	6
Preparing A251 Assembler Programs .....	7
Preparing C251 Compiler Programs .....	8
Preparing A166 Assembler Programs .....	9
Preparing C166 Compiler Programs .....	10
Starting the dScope Debugger .....	11
Loading the dScope Debugger .....	11
Loading a CPU Driver DLL .....	14
Loading a Program .....	15
Exiting the dScope Debugger .....	16
<b>Chapter 2. User Interface .....</b>	<b>17</b>
Main Window .....	19
Menu Bar .....	19
Toolbar .....	25
Debug Window .....	27
Status Bar .....	28
Toolbar .....	29
Dialog Bar .....	29
Menu .....	30
Operations .....	32
Command Window .....	38
Status Bar .....	38
Command Buffers .....	39
Command Line Editing .....	40
Comment Lines .....	41
Command Chaining .....	41
Syntax Generator .....	42
Command Files .....	43
Watch Window .....	44
Register Window .....	45
Serial Window .....	47
Performance Analyzer Window .....	49
Commands Menu .....	50
Adding and Removing Address Range .....	50
Valid Address Ranges .....	52
Memory Window .....	54

Symbol Browser Window .....	56
Symbol Display Mode .....	56
Symbol Options.....	57
Dragging Symbols.....	60
Call Stack Window .....	63
Code Coverage Window .....	65
Toolbox Window .....	67
Color Setup Dialog Box.....	69
Memory Map Dialog Box .....	72
Overview .....	72
Controls.....	74
Inline Assembler Dialog Box .....	76
Breakpoints Dialog Box.....	78
Watchpoints Dialog Box.....	81
Performance Analyzer Setup Dialog Box.....	83
<b>Chapter 3. Expressions .....</b>	<b>85</b>
Components of an Expression .....	85
Constants.....	86
System Variables.....	88
CPU Driver Symbols .....	90
Program Variables (Symbols).....	94
Line Numbers.....	100
Bit Addresses .....	101
Memory Spaces.....	102
Type Specifications.....	104
Operators.....	105
Address Expressions .....	108
Differences Between dScope and C.....	110
Expression Examples .....	111
<b>Chapter 4. Executing Code.....</b>	<b>115</b>
Starting a Program .....	115
Stopping a Program .....	116
Restarting a Program .....	117
Single-Stepping.....	118
Stepping Over a Function .....	118
Stepping Into a Function.....	119
Stepping Out of a Function .....	119
<b>Chapter 5. Examining and Modifying Memory.....</b>	<b>121</b>
Examining Memory .....	124
Using the Command Window .....	124
Using the DISPLAY Command.....	125
Using the EVALUATE Command .....	126
Using the OBJECT Command .....	127
Using the UNASSEMBLE Command .....	128
Using Memory Type Functions .....	129
Using the printf Function .....	129

---

Modifying Memory .....	130
Using the Command Window .....	130
Using the Inline Assembler Dialog Box.....	131
Using the ASM Command.....	132
Using the ENTER Command .....	133
Using Memory Type Functions.....	134
Using the memset Function .....	134
<b>Chapter 6. Using Breakpoints .....</b>	<b>135</b>
Overview .....	135
Breakpoint Types .....	137
Execution Breakpoints.....	137
Conditional Breakpoints .....	137
Access Breakpoints.....	138
Breakpoint Notes .....	139
Setting Breakpoints .....	140
Using the Debug Window .....	140
Using the Breakpoints Dialog Box .....	141
Using the BREAKSET Command.....	142
Viewing Breakpoints.....	147
Using the Breakpoints Dialog Box .....	147
Using the BREAKLIST Command .....	147
Removing Breakpoints .....	149
Using the Debug Window .....	149
Using the Breakpoints Dialog Box .....	149
Using the BREAKKILL Command.....	150
Enabling and Disabling Breakpoints.....	150
Using the Breakpoints Dialog Box .....	150
Using the BREAKENABLE Command.....	151
Using the BREAKDISABLE Command.....	152
<b>Chapter 7. Using Watchpoints .....</b>	<b>153</b>
Viewing Watchpoints .....	155
Setting Watchpoints .....	156
Using the Watchpoints Dialog Box .....	156
Using the WATCHSET Command .....	157
Removing Watchpoints .....	158
Using the Watchpoints Dialog Box .....	158
Using the WATCHKILL Command .....	158
Watching Program Variables .....	159
Watching Global Variables.....	159
Watching Local Variables .....	159
Watching Structure, Union, and Array Elements .....	160
Watching Pointers.....	161
Watching Memory Locations .....	163

<b>Chapter 8. Tutorial .....</b>	<b>165</b>
Running the HELLO Sample Program.....	165
Loading a CPU Driver .....	166
Loading a Target Program .....	167
Starting Program Execution .....	168
Viewing the Serial Port Peripheral Dialog Box .....	168
Halting Program Execution.....	168
Resetting the CPU.....	169
Single Stepping Through a Program.....	169
Exiting dScope .....	170
Running the MEASURE Sample Program.....	171
Loading a CPU Driver .....	172
Loading a Target Program .....	173
Measure Commands.....	174
Viewing the Memory Map .....	174
Viewing Debug Symbols .....	176
Viewing and Changing Memory.....	177
Program Execution.....	180
Using On-Chip Peripherals .....	187
Using the Performance Analyzer .....	189
Using dScope User and Signal Functions .....	190
Exiting dScope .....	192
<b>Chapter 9. Commands .....</b>	<b>193</b>
Memory Commands.....	194
Watchpoint Commands.....	194
Program Commands.....	194
Breakpoint Commands.....	195
General Commands.....	195
Command List.....	196
ASM .....	197
ASSIGN .....	198
BREAKDISABLE .....	200
BREAKENABLE .....	201
BREAKKILL .....	202
BREAKLIST.....	203
BREAKSET .....	204
Ctrl+C.....	208
DEFINE .....	209
DEFINE BUTTON .....	210
DIR.....	211
DISPLAY .....	216
DOS.....	218
ENTER.....	219
Esc.....	220
EVALUATE .....	221
EXIT.....	222
GO.....	223



INCLUDE.....	224
KILL .....	225
LOAD .....	226
LOG .....	233
MAP.....	234
MODE .....	237
OBJECT.....	238
OSTEP .....	239
Performance Analyzer.....	240
PSTEP.....	243
RESET .....	244
SAVE.....	245
SCOPE.....	246
SET .....	248
SETMODULE .....	251
SIGNAL.....	255
SLOG.....	256
TSTEP .....	257
UNASSEMBLE.....	258
WATCHKILL .....	259
WATCHSET .....	260
<b>Chapter 10. Functions.....</b>	<b>261</b>
Creating Functions .....	262
Using a Text Editor.....	262
Using the Command Window .....	263
Loading Functions .....	263
Invoking Functions.....	264
Function Classes.....	264
Predefined Functions .....	266
User Functions .....	276
Signal Functions .....	279
Differences Between dScope Functions and ANSI C .....	283
<b>Chapter 11. Error Messages.....</b>	<b>285</b>
<b>Appendix A. CPU Driver Files .....</b>	<b>295</b>
Purpose of CPU Drivers .....	295
Peripheral Registers (VTREG).....	296
CPU Driver List.....	297
80166.DLL .....	299
80167.DLL .....	301
80251A1.DLL.....	303
80251G1.DLL.....	304
80251S.DLL .....	305
80320.DLL .....	306
80410.DLL .....	307
8051.DLL .....	308
8051FX.DLL .....	309

---

80515.DLL .....	310
80515A.DLL .....	311
80517.DLL .....	312
80517A.DLL .....	314
8052.DLL .....	316
80552.DLL .....	317
80751.DLL .....	319
80781.DLL .....	320
MON166.DLL .....	321
MON251.DLL .....	324
MON51.DLL .....	326
RISM251.DLL .....	328
RISM51.DLL .....	330
<b>Appendix B. Improving Debugger Performance .....</b>	<b>333</b>
<b>Appendix C. Debugging Bank Switching Programs.....</b>	<b>335</b>
<b>Index .....</b>	<b>337</b>

# Chapter 1. Introduction

# 1

dScope is a software debugger which simulates the hardware of the MCS<sup>®</sup> 51, MCS<sup>®</sup> 251, and 166 microcontroller families. It executes all machine instructions and simulates integrated peripherals by means of loadable drivers.

dScope performs both symbolic debugging and source-level debugging of your target programs. All the features you expect in a high-end debugger are supported: complex breakpoints, watchpoints, performance analyzer, code coverage, memory display, in-line assembler, user and signal functions, and much more.

dScope fully simulates most aspects of numerous devices using CPU driver DLLs. When you use a particular DLL, dScope simulates the special, on-chip peripherals provided by that CPU. Refer to “Appendix A. CPU Driver Files” on page 295 for a list of the DLLs supported.

In addition to simulating your CPU, you can interface directly to your target hardware and debug in real-time using the tScope target debugger. tScope is an aspect of dScope that starts when you use a target monitor DLL. When target debugging, tScope interfaces to your target hardware via the PC's serial port. You target hardware must run a monitor program (either the Keil MON51, MON251, MON166, or the Intel RISM51 or RISM251) to communicate with tScope.

---

## NOTE

*dScope and tScope are synonymous for the dScope program. When you simulate a CPU, dScope displays in the debugger title bar. When you debug using your target hardware, tScope displays in the title bar.*

---

## 1

# System Requirements

The dScope debugger has the following hardware and software requirements.

## Hardware Requirements

- IBM PC compatible computer,
- An 80386 or higher processor,
- A minimum of 4 MB of memory (8 MB recommended),
- One floppy disk drive,
- One hard disk,
- Mouse recommended.

## Software Requirements

- DOS version 5.0 or higher,
- Microsoft Windows 3.1 running in enhanced mode, Windows 95, or Windows NT.

# Preparing Programs for dScope

There are a few steps you must follow in order to prepare your programs for debugging with dScope.

When you compile and assemble your programs using the Keil tools (A51, C51, A251, C251, A166, and C166), make sure you use the **DEBUG** command-line directive. This directive instructs the compiler and assembler to include debugging information in the object module that is generated.

When you link your application program, all debugging information (symbol names, symbol addresses, and source code line numbers) is maintained in the final object module.

---

## NOTE

*By default, the Keil tools exclude all debugging information. If you fail to explicitly include debugging information, you will only see assembly code in dScope. No symbol information will be available.*

---

## Source Code Considerations

Any valid C, PL/M-51, or Assembly program can be symbolically debugged in dScope. Here are a few hints that will make debugging your source code easier.

- Place each source statement on a separate line. C lets you place multiple statements on a single line. This does not prevent dScope from working. However, when you step through your program dScope treats the line as a single unit. For example, if you have two statements on a single line, you will only be able to set a breakpoint on the first statement.
- C lets you create macros that expand during compilation. dScope will not help you debug macros in your source code. dScope treats these as a single statement. You must expand the macros yourself before debugging them.
- C include files should not contain statements that produce executable code. This is because object modules can only be assigned a single source module. When include files produce executable code, the line numbers in the object module do not reflect the actual line numbers in the source module and dScope may not correctly synchronize line numbers.

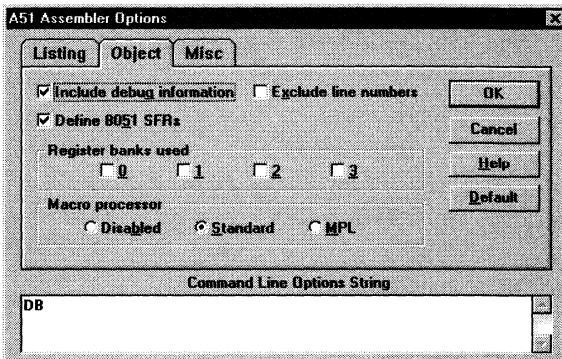
## Preparing A51 Assembler Programs

# 1

When you create programs with the A51 assembler, you must use the **DEBUG** directive to instruct the assembler to include debugging information in the object module. For example:

```
A51 MYFILE.A51 DEBUG
```

If you use the  $\mu$ Vision integrated development environment, make sure to set the Include debug information check box in the A51 Assembler Options dialog box.



### NOTE

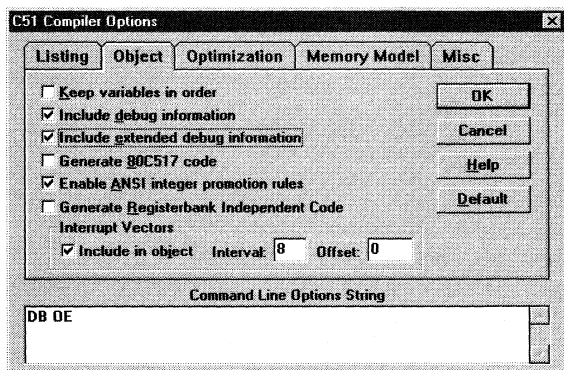
Do not check the *Exclude line numbers* check box if you wish to perform source-level debugging with dScope. This check box lets you exclude line number information for older emulators and debuggers.

## Preparing C51 Compiler Programs

When you create programs with the C51 compiler, you must use the **DEBUG** directive to instruct the assembler to include debugging information in the object module. You should also use the **OBJECTTEXTEND** directive to include extended type information in the object module. For example:

```
C51 MYFILE.C DEBUG OBJECTTEXTEND
```

If you use the  $\mu$ Vision integrated development environment, make sure to set the Include debug information and Include extended debug information check boxes in the C51 Compiler Options dialog box.



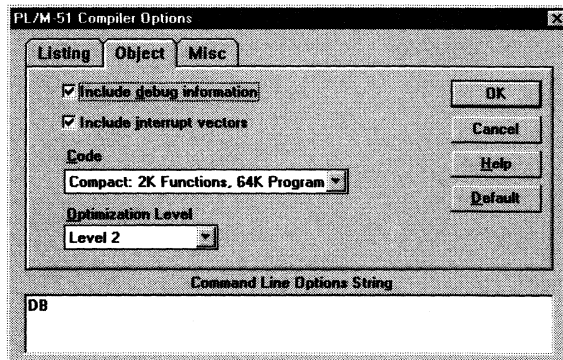
## Preparing PL/M-51 Compiler Programs

1

When you create programs with the Intel PL/M-51 compiler, you must use the **DEBUG** directive to instruct the assembler to include debugging information in the object module. For example:

```
PLM51 MYFILE.PLM DEBUG
```

If you use the  $\mu$ Vision integrated development environment, make sure to set the Include debug information check box in the PL/M-51 Compiler Options dialog box.



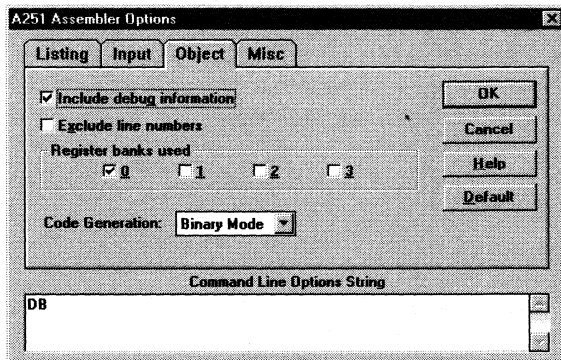


## Preparing A251 Assembler Programs

When you create programs with the A251 assembler, you must use the **DEBUG** directive to instruct the assembler to include debugging information in the object module. For example:

```
A251 MYFILE.ASM DEBUG
```

If you use the  $\mu$ Vision integrated development environment, make sure to set the Include debug information check box in the A251 Assembler Options dialog box.



### NOTE

*Do not check the Exclude line numbers check box if you wish to perform source-level debugging with dScope. This check box lets you exclude line number information for older emulators and debuggers.*

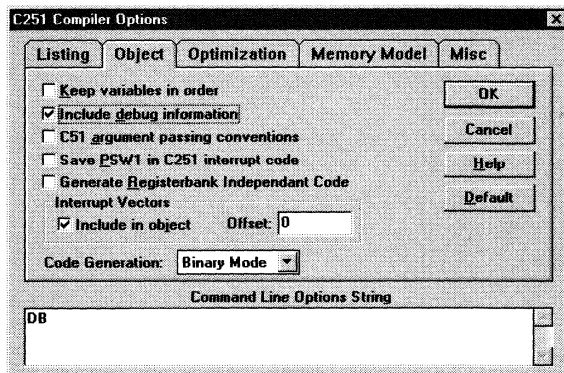
## Preparing C251 Compiler Programs

# 1

When you create programs with the C251 compiler, you must use the **DEBUG** directive to instruct the assembler to include debugging information in the object module. For example:

```
C251 MYFILE.C DEBUG
```

If you use the  $\mu$ Vision integrated development environment, make sure to set the Include debug information check box in the C251 Compiler Options dialog box.

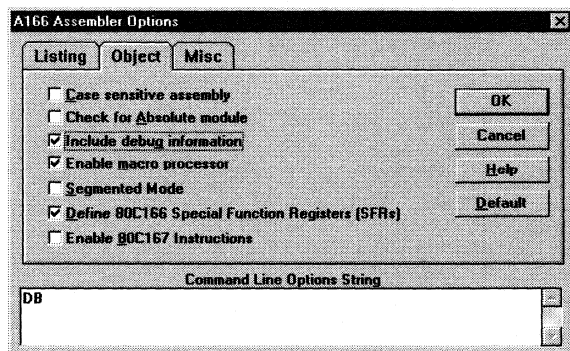


## Preparing A166 Assembler Programs

When you create programs with the A166 assembler, you must use the **DEBUG** directive to instruct the assembler to include debugging information in the object module. For example:

```
A166 MYFILE.ASM DEBUG
```

If you use the  $\mu$ Vision integrated development environment, make sure to set the Include debug information check box in the A166 Assembler Options dialog box.



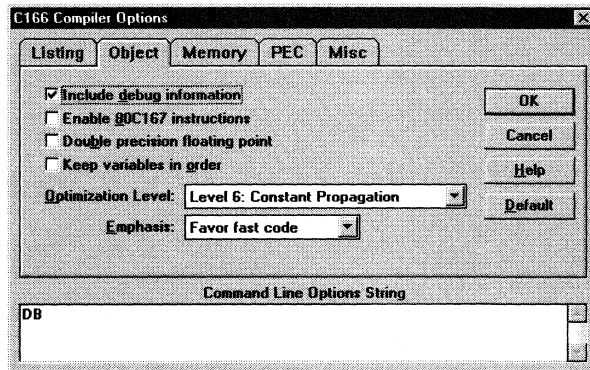
## 1

## Preparing C166 Compiler Programs

When you create programs with the C166 compiler, you must use the **DEBUG** directive to instruct the assembler to include debugging information in the object module. For example:

```
C166 MYFILE.C DEBUG
```

If you use the  $\mu$ Vision integrated development environment, make sure to set the Include debug information check box in the C166 Compiler Options dialog box.



# Starting the dScope Debugger

To start debugging with dScope, you must perform the following steps:

- Load the dScope debugger program,
- Select a CPU Driver DLL to use,
- Load your target application program.

Each of these is explained below.

## Loading the dScope Debugger

The dScope debugger must be installed on your computer before you can run any of the examples presented in this manual. Once installed, the following files are found in the \...BIN\ directory.

### For the C51 development tools...

Filename	Description
<b>DSW51.EXE</b>	dScope-51 for Windows for the 251 and 8051 families of microcontrollers.
<b>DSW51.HLP</b>	dScope-51 for Windows help file.
<b>80*.DLL</b>	CPU driver DLLs for the different microcontrollers that are supported.
<b>MON51.DLL</b>	Driver for interfacing to a target board running the Keil MON51 monitor.
<b>RISM51.DLL</b>	Driver for interfacing to a target board running the Intel RISM51 monitor.

### For the C251 development tools...

Filename	Description
<b>DSW51.EXE</b>	dScope-51 for Windows for the 251 and 8051 families of microcontrollers.
<b>DSW51.HLP</b>	dScope-51 for Windows help file.
<b>80*.DLL</b>	CPU driver DLLs for the different microcontrollers that are supported.
<b>MON251.DLL</b>	Driver for interfacing to a target board running the Keil MON251 monitor.
<b>RISM251.DLL</b>	Driver for interfacing to a target board running Intel RISM251 monitor.

## For the C166 development tools...

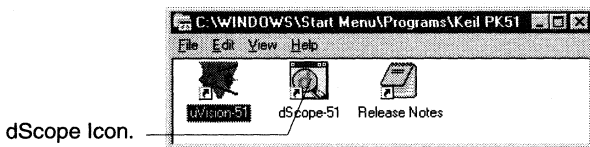
# 1

Filename	Description
<b>DSW166.EXE</b>	dScope-166 for Windows for the Siemens 80166 and C167 family of microcontrollers.
<b>DSW166.HLP</b>	dScope-166 for Windows help file.
<b>80*.DLL</b>	CPU driver DLLs for the different microcontrollers that are supported.
<b>MON166.DLL</b>	Driver for interfacing to a target board running MON166.

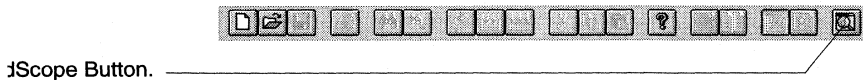
## Starting the dScope Debugger

You may start the dScope debugger:

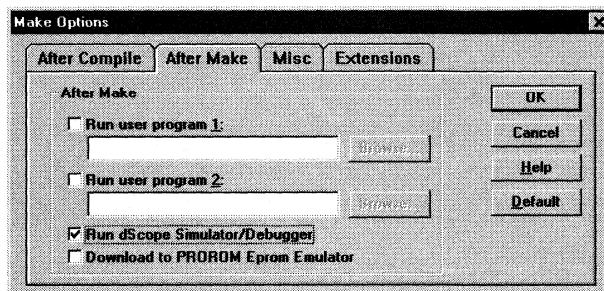
- By double-clicking on the dScope Icon in the program group created by the installation program,



- By clicking on the dScope button in the  $\mu$ Vision toolbar,



- By automatically running dScope after making a project in  $\mu$ Vision. This is accomplished by selecting the Run dScope Simulator/Debugger in the Make Options dialog box,



- By double-clicking on a shortcut you create in Windows 95 or Windows NT 4.0,

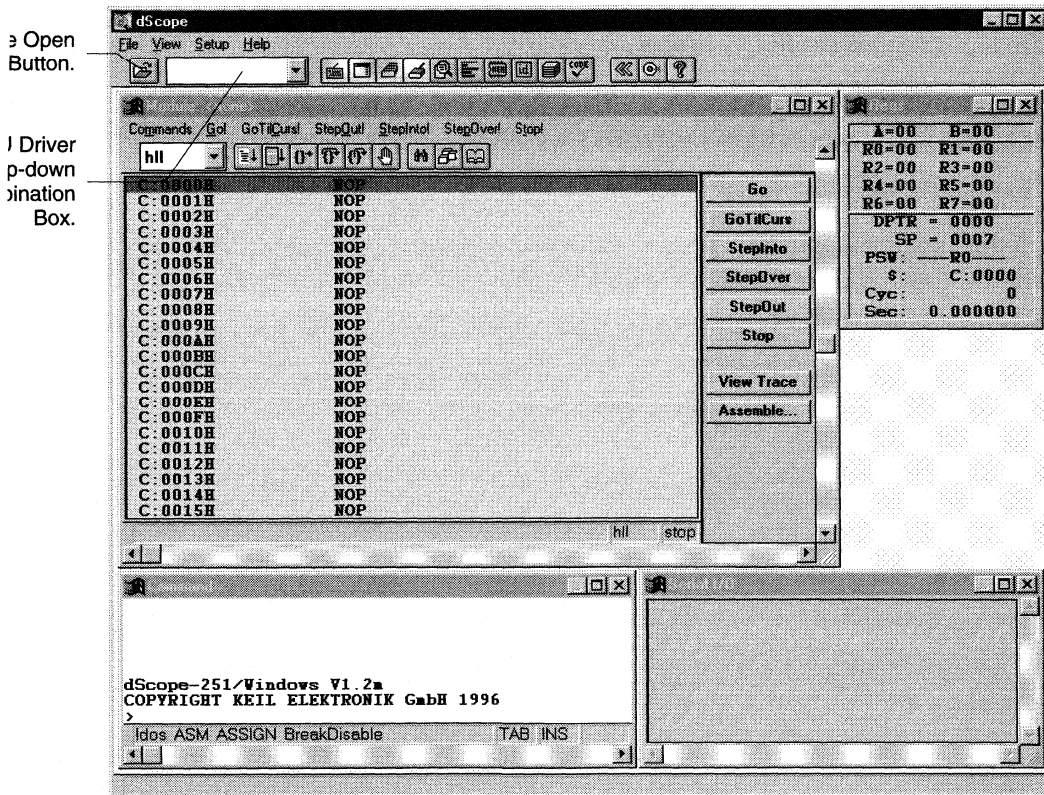


dScope-51

- By running the DSW\*.EXE program in the Windows 3.1 File Manager or Windows 95 Explorer.

## Opening Screen

When dScope starts, the following screen displays:



## 1

## Using Include Files

You may direct dScope to execute instructions from an include file using the `-i` command-line option. For example,

```
DSW51 -i COMMANDS.TXT
```

## Loading a CPU Driver DLL

To begin using dScope, you must first load a CPU driver DLL. The DLL contains instructions that tell dScope how a particular CPU works. Refer to “Appendix A. CPU Driver Files” on page 295 for a complete list of the DLLs available.

There are several ways to load a CPU driver.

- You may use the CPU driver drop-down box on the toolbar and select the driver from the list.
- You may select the Load CPU driver... command from the File menu.
- In the Command window, you may enter the Load command along with the name of the DLL to use. For example:

```
load 8052.dll
```

Refer to “LOAD” on page 226 for more information about using the **LOAD** command.



## Loading a Program

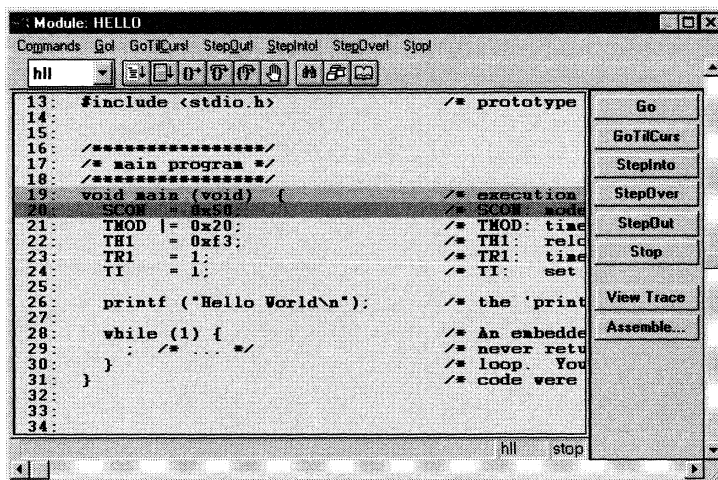
After selecting the CPU driver, you are ready to load your target program. There are several ways you may do this:

- You may use the File open button on the toolbar.
- You may select the Load object file... command from the File menu.
- In the Command window, you may enter the Load command along with the name of your target program. For example:

```
load hello
```

Refer to “LOAD” on page 226 for more information about using the **LOAD** command.

When your program loads, the Debug window displays the source code for your program.



## 1

## Exiting the dScope Debugger

You may exit the dScope debugger...

- By selecting the Exit command from the File menu,
- By clicking on the dScope Main Window Close Button (in Windows 95),
- By selecting the Close command from the dScope Main Window Control Menu (in Windows 3.1),
- By double-clicking on the dScope Main Window Control Menu box,
- By entering the Exit command in the Command window. For example:

```
exit
```

---

**NOTE**

*dScope requires that you stop simulation or target execution of your program before you exit.*

---

## Chapter 2. User Interface

dScope provides you with a powerful user interface that utilizes standard Windows components to display and gather information. These components display information (in a number of easy-to-read formats) and obtain input (from entries typed in windows and dialog boxes).

### NOTE

*Keys you type are directed to the window that is currently in focus. You may click the mouse on a window to bring that window into focus. You may alternatively use **Ctrl+Tab** to switch the focus from one window to the next.*

The following windows and dialog boxes are described in this chapter:

Window / Dialog Box	Page	Description
<b>Main Window</b>	<b>19</b>	Displays the main menu bar, toolbar, status bar, and all other dScope windows.
<b>Debug Window</b>	<b>27</b>	Displays target program source or assembly code and lets you perform execution functions. This window is the primary window used to interface to dScope.
<b>Command Window</b>	<b>38</b>	Displays commands you enter and their results.
<b>Watch Window</b>	<b>44</b>	Displays scalar, structure, union, and array variables from your target program.
<b>Register Window</b>	<b>45</b>	Displays the values of the CPU registers, instruction cycles executed, and the total computed execution time based on the current crystal frequency.
<b>Serial Window</b>	<b>47</b>	Emulates a serial terminal connected to the CPU's built-in serial port and displays all input and output activity.
<b>Performance Analyzer Window</b>	<b>49</b>	Displays performance statistics for functions and address ranges you wish to analyze.
<b>Memory Window</b>	<b>54</b>	Displays, in HEX and ASCII, the contents of various memory areas you specify.
<b>Symbol Browser Window</b>	<b>56</b>	Displays public symbols, local symbols, or line number information defined in the currently loaded target program.
<b>Call Stack Window</b>	<b>63</b>	Displays the currently nested function calls or interrupt procedures.
<b>Code Coverage Window</b>	<b>65</b>	Displays the percentage of code that has been executed in the functions in a source module.
<b>Toolbox Window</b>	<b>67</b>	Displays the currently defined toolbox buttons.

Window / Dialog Box	Page	Description
<b>Color Setup Dialog Box</b>	<b>69</b>	Lets you change colors and fonts used in the display windows.
<b>Memory Map Dialog Box</b>	<b>72</b>	Lets you change the memory map used for the target program.
<b>Inline Assembler Dialog Box</b>	<b>76</b>	Lets you assembly new code directly into code memory.
<b>Breakpoints Dialog Box</b>	<b>78</b>	Lets you create, view, and remove breakpoints.
<b>Watchpoints Dialog Box</b>	<b>81</b>	Lets you create and remove watchpoints.
<b>Performance Analyzer Setup Dialog Box</b>	<b>83</b>	Lets you create, view, and remove performance analyzer address ranges.

---

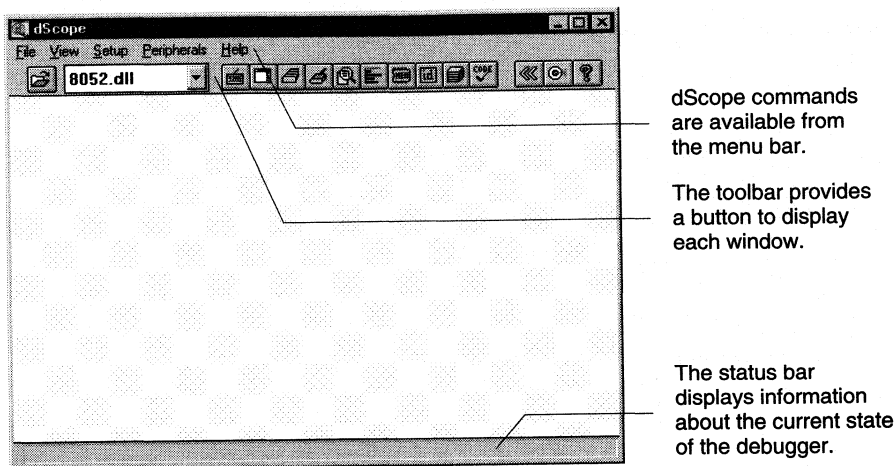
**NOTE**

*There are a number of windows dScope provides for input and output. Most debug sessions require the Main window, Debug window, and Command window. You can perform most routine debugging with only these three windows.*

---

## Main Window

The Main window is the dScope parent window which includes the main menu bar, toolbar, and status bar.

**2**

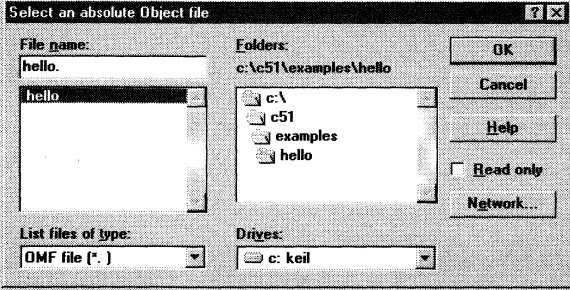
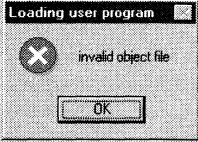
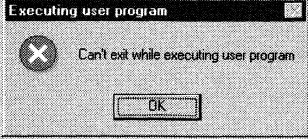
All other dScope windows are children of the Main window.

## Menu Bar

dScope provides a number of commands that are available from the menu bar. Commands are organized according to function in the File, View, Setup, Peripherals, and Help menus.

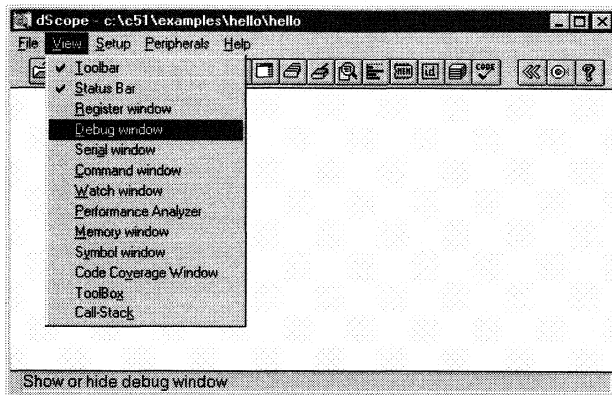
## File Menu Commands

The File menu includes commands that access files or exit dScope. The following commands are available in this menu:

Menu Command	Description
<b>Load object file...</b>	<p>This command opens a dialog box where you may select a target file you wish to load and debug. You may rapidly access this command using the Open File button on the toolbar. Refer to “Loading a Program” on page 15 for more information.</p>  <p>dScope lets you select from a number of supported file types and presents a warning message if an unrecognized or unsupported file is loaded.</p> 
<b>Load CPU driver...</b>	<p>This command drops down the toolbar combo box containing a list of available CPU drivers. From this list, you may select the appropriate driver necessary to debug your application.</p> <p>You may rapidly access this command using the CPU Driver button on the toolbar. Refer to “Loading a CPU Driver DLL” on page 14 for more information.</p>
<b>Exit</b>	<p>This command exits dScope and returns to Windows. dScope cancels this command if either the target program or a dScope user function are running.</p>  <p>To exit, close this message box, halt the user program and any active signal functions, and repeat the Exit command.</p>

## View Menu Commands

The commands in the View menu select whether or not certain windows and parts of the user interface are displayed. A check mark beside a command in this menu indicates that the corresponding window is shown. Windows are not displayed for commands without check marks.



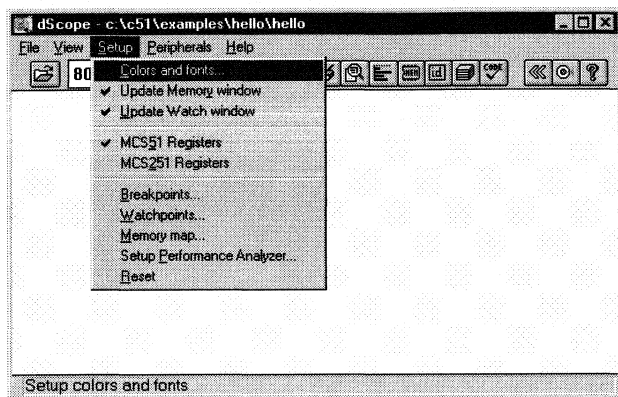
2

The following commands are available in this menu:

Command	Function
<b>Toolbar</b>	Toggles the toolbar on or off.
<b>Status Bar</b>	Toggles the status bar on or off.
<b>Register window</b>	Toggles the Register window on or off.
<b>Debug window</b>	Toggles the Debug window on or off.
<b>Serial window</b>	Toggles the Serial window on or off.
<b>Command window</b>	Toggles the Command window on or off.
<b>Watch window</b>	Toggles the Watch window on or off.
<b>Performance Analyzer</b>	Toggles the Performance Analyzer window on or off.
<b>Memory window</b>	Toggles the Memory window on or off.
<b>Symbol window</b>	Toggles the Symbol Browser window on or off.
<b>Code Coverage window</b>	Toggles the Code Coverage window on or off.
<b>Toolbox</b>	Toggles the Toolbox window on or off.
<b>Call-Stack</b>	Toggles the Call-Stack window on or off.

## Setup Menu Commands

The Setup menu contains commands that customize how dScope operates.



There are commands to:

- Select the colors and fonts used in the different windows,
- Select whether or not memory and watch windows are updated during target program execution,
- Display 8051 or 251 registers,
- Setup breakpoints and watchpoints,
- Define memory areas,
- Initialize the performance analyzer,
- Reset dScope.



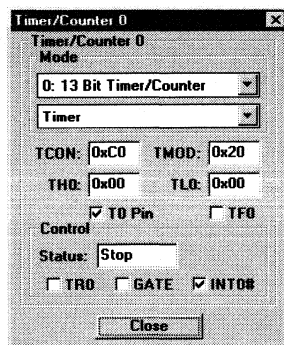
Each command is described in detail in the following table.

Command	Description
<b>Colors and fonts...</b>	This command displays the color and font configuration dialog box. You may set the colors and the fonts used in each of dScope's windows in this dialog box. Refer to "Color Setup Dialog Box" on page 69.
<b>Update Memory window</b>	This command selects whether or not the contents of the Memory window are updated during execution of the target program. If this command is checked, the Memory window is updated during target program execution. If this command is not checked, the Memory window is updated only after halting the target program.
<b>Update Watch window</b>	This command selects whether or not the contents of the Watch window are updated during execution of the target program. If this command is checked, the Watch window is updated during target program execution. If this command is not checked, the Watch window is updated only after halting the target program.
<b>MCS 51 Registers</b>	This command configures the Register window to show MCS <sup>®</sup> 51 registers (not available in dScope-166).
<b>MCS 251 Registers</b>	This command configures the Register window to show MCS <sup>®</sup> 251 registers (not available in dScope-166).
<b>Breakpoints...</b>	This command displays the Breakpoint dialog box where you may edit breakpoints in your target program. Refer to "Breakpoints Dialog Box" on page 78 for more information.
<b>Watchpoints...</b>	This command displays the Watchpoints dialog box where you may edit watchpoints for your target program. Refer to "Watchpoints Dialog Box" on page 81 for more information.
<b>Memory map...</b>	This command displays the Memory map dialog box where you may setup memory areas used in your target program. Refer to "Memory Map Dialog Box" on page 72 for more information.
<b>Setup Performance Analyzer...</b>	This command displays the Performance analyzer dialog box where you may setup portions of your target program for performance analysis. Refer to "Performance Analyzer Setup Dialog Box" on page 83 for more information.
<b>Reset</b>	<p>This command resets dScope. You may rapidly access this command using the Reset button on the toolbar. Refer to "RESET" on page 244 for more information.</p> <p>When you invoke the reset command, the current program counter (PC) and CPU driver are reset. This is equivalent to sending a hardware reset signal to the CPU.</p> <p>Note that the reset command does not affect the memory map, debug information, or any other settings.</p>

## Peripheral Menu Commands

The Peripheral menu is a dynamic menu that is specific to the CPU driver that is currently loaded. Each on-chip peripheral is supported by a dialog box you can access from this menu, for example, I/O ports, Interrupts, Timers, and Serial I/O.

The Timer/Counter 0 dialog box is part of the 8052.DLL CPU driver. When this CPU driver loads, it dynamically adds support for the timer/counter into the Peripherals menu. Refer to “Appendix A. CPU Driver Files” on page 295 for more information about the CPU drivers that are available and the on-chip peripherals that are supported.

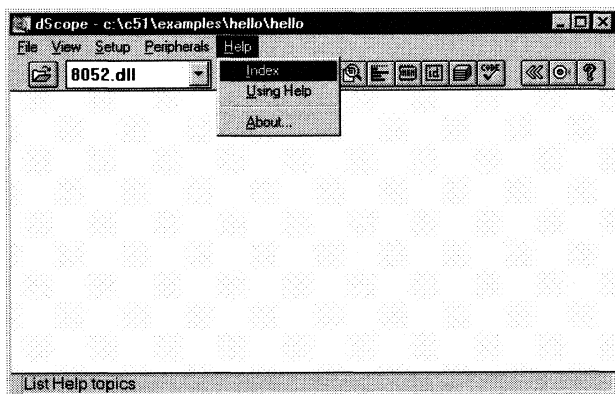


### NOTE

*The Peripheral menu is dynamically created and does not display until a CPU driver DLL is loaded.*

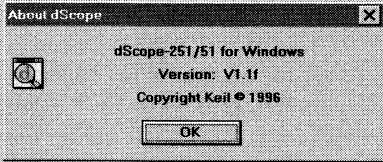
## Help Menu Commands

The Help menu contains commands that provide you with information about how to use dScope. dScope provides context sensitive help for most dialog boxes and windows using a Help button. On occasion, however, you may wish to access help information directly from the Help menu.



Each of the help commands is described in the following table.

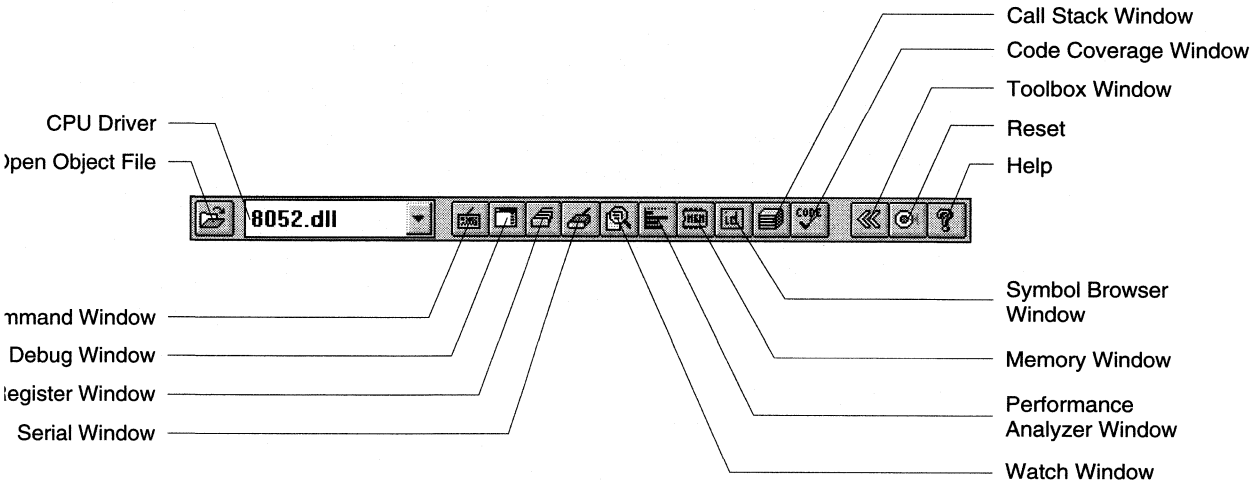
Command	Function
<b>Index</b>	This command opens the dScope help file's table of contents. From here, you may browse the dScope help file in topological order or you may search the help file for a particular topic of interest.
<b>Using Help</b>	This command opens the Windows help file that describes how to use the Windows help system. Use the information presented here to learn about the help features built into Microsoft Windows.
<b>About...</b>	This command displays a dialog box that shows the version number of the software you are using.



2

# Toolbar

dScope provides a toolbar with buttons which duplicate many of the commands found in the File menu and View menu. The following figure shows the layout of the toolbar and gives descriptions of each toolbar button.



The following table provides a description for each of the buttons on the toolbar.

Button	Description
<b>Call Stack Window</b>	Shows or hides the <b>Call Stack</b> window. Refer to “Call Stack Window” on page 63 for more information about this window.
<b>Code Coverage Window</b>	Shows or hides the <b>Code Coverage</b> window. Refer to “Code Coverage Window” on page 65 for more information about this window.
<b>Command Window</b>	Shows or hides the <b>Command</b> window. Refer to “Command Window” on page 38 for more information about this window.
<b>CPU Driver</b>	Shows the currently selected CPU driver (if one is selected). You may select a CPU driver from this drop-down combination box.
<b>Debug Window</b>	Shows or hides the <b>Debug</b> window. Refer to “Debug Window” on page 27 for more information.
<b>Help</b>	Shows the <b>About</b> dialog.
<b>Memory Window</b>	Shows or hides the <b>Memory</b> window. Refer to “Memory Window” on page 54 for more information about this window.
<b>Open Object File</b>	Starts the <b>Open Object File</b> dialog box.
<b>Performance Analyzer Window</b>	Shows or hides the <b>Performance Analyzer</b> window. Refer to “Performance Analyzer Window” on page 49 for more information about this window.
<b>Register Window</b>	Shows or hides the <b>Register</b> window. Refer to “Register Window” on page 45 for more information about this window.
<b>Reset</b>	Resets dScope. Refer to “RESET” on page 244 for more information.
<b>Serial Window</b>	Shows or hides the <b>Serial</b> window. Refer to “Serial Window” on page 47 for more information about this window.
<b>Symbol Browser Window</b>	Shows or hides the <b>Symbol Browser</b> window. Refer to “Symbol Browser Window” on page 56 for more information about this window.
<b>Toolbox Window</b>	Shows or hides the <b>Toolbox</b> window. Refer to “Toolbox Window” on page 67 for more information about this window.
<b>Watch Window</b>	Shows or hides the <b>Watch</b> window. Refer to “Watch Window” on page 44 for more information about this window.

## Getting Help for Toolbar Buttons

dScope displays help text on the status line for the toolbar buttons. Do the following to display this help text.

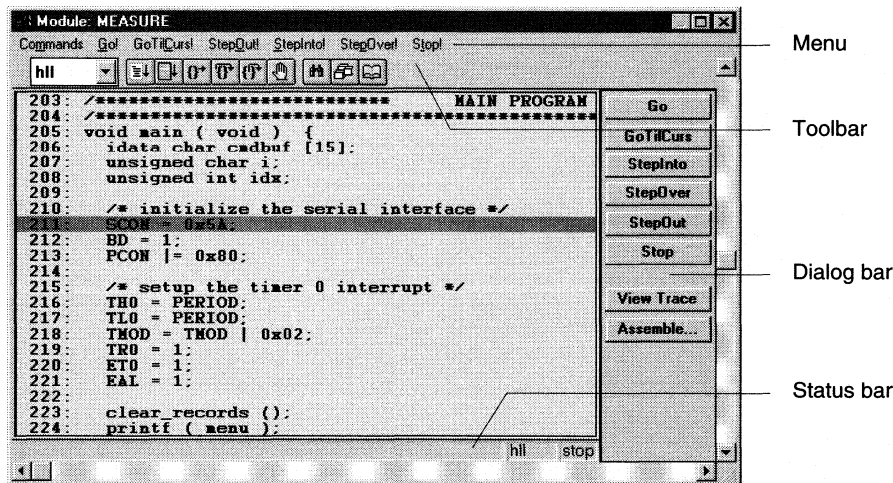
- Click a toolbar button using the left mouse button.
- *Keep the mouse button pressed* and note the help text in the status bar.
- Move the mouse pointer away from the toolbar button and release.

# Debug Window

The Debug window or Module window lets you view your target program as original source code, assembly code, or mixed source and assembly code. You may display or hide the Debug window using the Debug window command from the View menu or the toolbar Debug window button shown on the right.



This window provides all program execution functions and is the primary window used to interface to dScope.



The Debug window contains a status bar, a toolbar, a dialog bar, and a menu. The dialog bar and the toolbar duplicate the most frequently used menu commands such as Go, StepInto, and View Trace.

The Debug window...

- Displays user programs in one of three different viewing modes: HLL (high-level language), ASM (assembly), and MIXED (HLL intermixed with ASM lines). The view mode can be changed at any time. If HLL mode is selected and the code shown does not have associated line number information, the view mode automatically resorts to ASM mode. The view mode shifts back to HLL mode if the content is scrolled to a range where line number information is available.
- Facilitates the various **go** and **step** commands. Each execution command meets a specific need such as stepping over a single instruction or line of code, stepping over a function call, or stepping out of the current function.

## 2

- Provides trace recording. If trace recording is enabled, dScope records the most recent 512 instructions executed and the CPU register contents for each instruction. The Debug window can be switched to display the recorded trace frames. Depending on the associated line number information, the display shows ASM or MIXED mode lines.
- Lets you select the source module to display. dScope maintains a list of high level modules in your program. You may change the displayed module at any time via a standard dialog box.
- Assembles in-line code. The Debug window provides an in-line assembler where you can write assembly code to augment or to change your application.
- Locates text strings. The Find dialog lets you search the entire Debug window for text strings.
- Lets you select a range of text in the Debug window to write to a log file.
- Lets you quickly set and remove execution breakpoints.

## Status Bar

The Debug window's status bar displays the following:

1. Help information when a menu command is selected.
2. The actual viewing mode: HLL for high-level language, MIXED for high-level language and assembly, or ASM for assembly.
3. The current execution status: RUN or STOP.

---

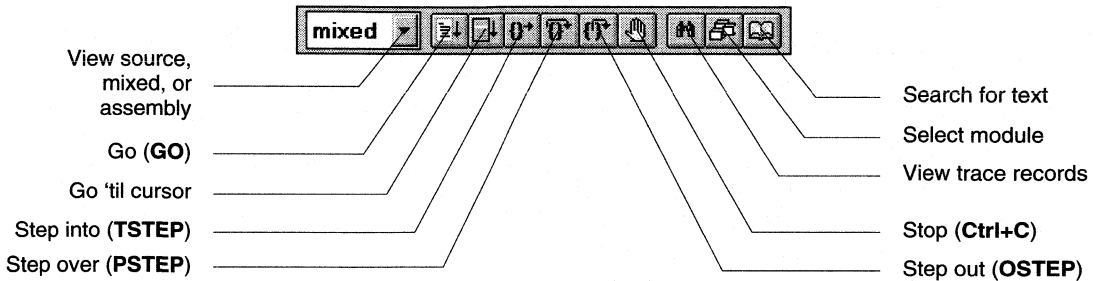
**NOTE**

*You may enable or disable the status bar from the Commands menu.*

---

## Toolbar

The toolbar in the Debug window lets you rapidly select the most frequently used debug commands. The following figure shows the layout of the toolbar and gives descriptions of each toolbar button.



Refer to “Menu” on page 30 for descriptions of these commands.

### NOTE

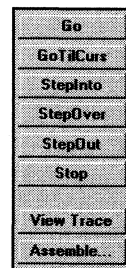
*The toolbar and the dialog bar both provide quick access to the most frequently used debugger functions. You may enable or disable either or both of these from the Commands menu.*

## Dialog Bar

The dialog bar in the Debug window lets you rapidly select the most frequently used debugger commands. Refer to “Menu” on page 30 for descriptions of these commands.

### NOTE

*The toolbar and the dialog bar both provide quick access to the most frequently used debugger functions. You may enable or disable either or both of these from the Commands menu.*





## Menu

The Debug window has a menu line containing commands necessary for debugging most programs. The following commands are available from the menu:

Menu Item	Function
<b>Commands</b>	See next table.
<b>Go!</b>	Starts execution of the user program from the current program counter ( <b>PC</b> ).  <i><b>NOTE:</b> This menu item is equivalent to the <b>GO</b> command entered in the Command window. Refer to "GO" on page 223 for more information.</i>
<b>GoTilCurs!</b>	Starts execution of the user program from the current program counter ( <b>PC</b> ).  Execution stops if the current cursor line is reached. You may move the cursor to a line by moving the mouse pointer to the desired line and clicking the left mouse button.  dScope provides an additional shortcut for the <b>GoTilCurs</b> command. Double-clicking the right mouse button at the desired line starts execution and stops if the selected line is reached.
<b>StepOut!</b>	Starts execution from the current program counter and stops if the current function returns to the statement following the function invocation. dScope internally maintains a list of currently nested function calls including the code address where a function invocation took place. If no function calls are present in the list the <b>StepOut</b> command is not available.  <i><b>NOTE:</b> This menu item is equivalent to the <b>OSTEP</b> command entered in the Command window.. Refer to "OSTEP" on page 239 for more information.</i>
<b>StepInto!</b>	Forces execution of the next statement. The definition of statement depends on the current view mode. In HLL mode a statement defines a high-level statement. In ASM or MIXED mode a statement defines a single assembler instruction. The <b>StepInto</b> command does not treat a function call as a single statement. Using <b>StepInto</b> on a function call steps into the function.  <i><b>NOTE:</b> This menu item is equivalent to the <b>TSTEP</b> command entered in the Command window.. Refer to "TSTEP" on page 257 for more information.</i>
<b>StepOver!</b>	Forces execution of the next statement. <b>StepOver</b> works much the same as <b>StepInto</b> except that <b>StepOver</b> treats a function call as a single statement. Using <b>StepOver</b> on a function call steps over that function.  <i><b>NOTE:</b> This menu item is equivalent to the <b>PSTEP</b> command entered in the Command window.. Refer to "PSTEP" on page 243 for more information.</i>
<b>Stop!</b>	Stops execution of the user program.  <i><b>NOTE:</b> This menu item is equivalent to <b>Ctrl+C</b> entered in the Command window.. Refer to "Ctrl+C" on page 208 for more information.</i>



The following table describes commands found in the Commands menu.

Commands Menu	Function
<b>View High Level</b>	Switches the viewing mode to HLL (high-level language) mode. If the currently disassembled address range has line number information and the associated source file or list file is accessible, source lines are displayed instead of the disassembly.
<b>View Mixed</b>	Switches to MIXED mode. If the currently disassembled address range has line number information and the associated source file or list file is accessible, the display shows source lines intermixed with assembly lines. This mode is intended for analyzing the compiler-generated code for a specific source line.
<b>View Assembly</b>	Switches to ASM mode. The display shows assembly lines only. Address ranges without line number information or with inaccessible source or list files resort to ASM mode.
<b>View Trace Records</b>	Switches to trace history mode (the Record Trace command must have previously been invoked). Program execution must be stopped before the trace history can be viewed.  If the trace history is not empty, the Debug window shows the trace records on the upper half of the screen. Use the cursor keys or the vertical scroll bar to move into the history. Values in the register window are saved and displayed for each line in the trace memory.
<b>Select source module</b>	Shows the currently available high-level language source modules. The Debug window synchronizes immediately when a new module is selected.
<b>Inline assemble</b>	Allows entry of assembly language instructions. The in-line assembler translates the instructions into machine code which is stored in code memory.
<b>Find</b>	Searches for text in the Debug window. You may use this command to search for text even while your program is executing.
<b>Show Dialogbar</b>	Shows or hides the dialog bar. Refer to "Dialog Bar" on page 29 for more information.
<b>Show Statusbar</b>	Shows or hides the status bar. Refer to "Status Bar" on page 28 for more information.
<b>Show Toolbar</b>	Shows or hides the toolbar. Refer to "Toolbar" on page 29 for more information.
<b>Record trace</b>	Enables and disables trace recording.
<b>Set log start line</b>	Sets the starting line for the log file.
<b>Perform File log</b>	Saves the area from the starting line to the current cursor line to the currently active log file.

## Operations

The Debug window lets you perform a number of operations on your program source code. The following operations are briefly explained on the following pages.

- Scrolling the window contents (refer to “Scrolling the Debug Window” on page 32),
- Setting and removing breakpoints (refer to “Setting and Removing Execution Breakpoints” on page 33),
- Selecting the displayed source module (refer to “Displaying Source Module” on page 33),
- Text searching (refer to “Searching for Text” on page 34),
- Assembling code in-line (refer to “Assembling In-line Code” on page 34),
- Recording the last 512 instructions executed (refer to “Trace Recording” on page 34),
- Saving information in the Debug window to a file (refer to “Debug Window Logging” on page 36).

### Scrolling the Debug Window

The Debug window supports both vertical and horizontal scrolling using either the keyboard or scroll bars. The following keys may be used.

Key	Function
<b>PgDn</b>	Scroll window contents one page downwards.
<b>PgUp</b>	Scroll window contents one page upwards.
↓	Scroll window contents one line downwards.
↑	Scroll window contents one line upwards.
→	Scroll window contents 1 character to the right.
←	Scroll window contents 1 character to the left.

Scrolling with the scroll bars works similarly to other Windows applications.

**NOTE**

*The vertical slider is always positioned in the middle of the vertical scroll bar. Dragging it in either direction selects an address within the available address space. The address is shown in the Status Bar of the Debug Window. The slider always repositions itself into the middle of the scroll bar.*

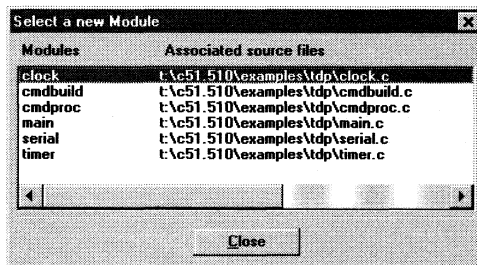
## Setting and Removing Execution Breakpoints

You may set one or more execution breakpoints by double-clicking the left mouse button on the desired line in the Debug window. The selected line is redrawn using the BP-Highlight color with the label **[BR *n*]** (where *n* represents the breakpoint number). Double-clicking the left mouse button on a line with an execution breakpoint removes that breakpoint.

Refer to “Chapter 6. Using Breakpoints” on page 135 for more information.

## Displaying Source Modules

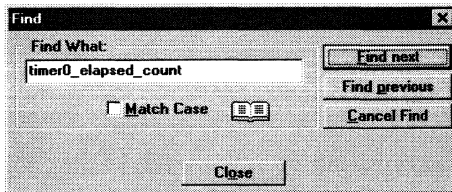
dScope uses the program counter to determine which source module to display in the Debug window. The Select source module command in the Commands menu presents the following dialog box where you may select the source module dScope displays.



All source files with line number information are listed. Selecting a source module displays the file contents in the Debug window.

## Searching for Text

You may search for specific text in the Debug window. Use the Find command in the Commands menu to display the following dialog box.



Enter the search string and select the Find Next button to find matching strings after the current line. The Find Previous button locates matching strings before the current line. The Match Case check box performs case sensitive searches.

If matching text is located, the Debug window highlights the matching text. You may search for text strings even while the user program is executing.

---

### NOTE

*The search may run for a long time, especially if the search runs over the entire 16MB range. You may select the Cancel Find button at any time to stop searching.*

---

## Assembling In-line Code

The Inline Assemble command from the Commands menu opens the Inline Assembler dialog box. Refer to “Inline Assembler Dialog Box” on page 76 for more information about this dialog box.

## Trace Recording

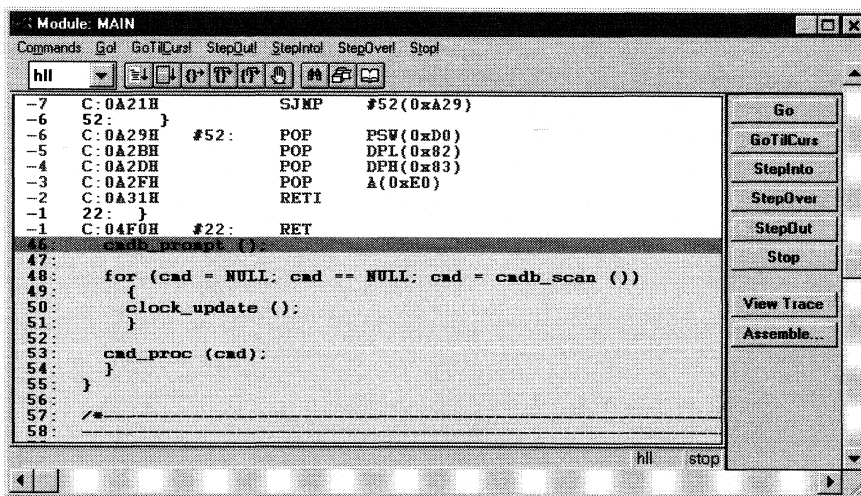
Occasionally, you may need to review the execution history (or trace history) of your embedded application. dScope maintains a trace history buffer that holds the 512 most recently executed assembler instructions. The following information is saved for each instruction:

- The Program Counter (PC),
- Program Status Word(s) (PSW),
- All CPU registers (this depends on the selected CPU driver).

To view trace history for your application:

1. Make sure the Record trace command in the Commands menu is checked. When trace recording is enabled, dScope saves the 512 most recently executed assembly instructions in the trace buffer.
2. Begin running your program using the Go, StepInto, or StepOver buttons, for example.
3. Stop your program. Execution must be stopped before you can view the trace history.
4. Use the View trace records command from the Commands menu to display the trace buffer.

2



The upper half of the Debug window shows the trace buffer. Instructions in trace memory are preceded by a negative line number that corresponds to the position in the buffer. Line number -1 is the most recently executed instruction, -2 was executed before that, and so on.

The program line corresponding to the current program counter displays below the trace memory. When you resume program execution, this is the line that executes next.

You may use the cursor keys to scroll through the trace memory. The values displayed in the Register window (refer to "Register Window" on page 45) are the values for the selected trace instruction.

When you are finished viewing trace memory and ready to continue debugging, you may select the Go, StepInto, or StepOver buttons to resume program execution and clear the trace display.

### NOTE

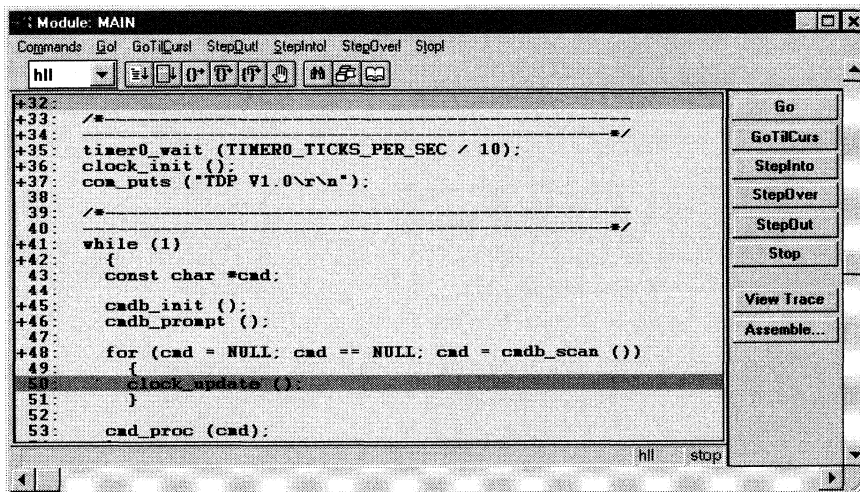
*Trace recording slows execution speed of your target program. Disable trace recording for maximum performance.*

## 2

### Debug Window Logging

dScope provides you with the ability to save a portion of the Debug window to a log file. This is useful when you want to capture part of the disassembly or execution of your program.

For example, when the following Debug window:



is logged to a file. The file appears as follows:

```
+32:
+33: /*-----
+34: -----*/
+35: timer0_wait (TIMER0_TICKS_PER_SEC / 10);
+36: clock_init ();
+37: com_puts ("TDP V1.0\r\n");
+38:
+39: /*-----
+40: -----*/
+41: while (1)
+42: {
+43:     const char *cmd;
+44:
```

```
+45:  cmdb_init ();
+46:  cmdb_prompt ();
+47:
+48:  for (cmd = NULL; cmd == NULL; cmd = cmdb_scan ())
49:  {
50:      clock_update ();
51:  }
52:
53:  cmd_proc (cmd);
```

The following steps list how to log the contents of the Debug window to a file.

1. Open a log file. In the Command window, enter the following command to open a log file.

```
Log > filename
```

This creates *filename* in the specified directory. To append the log information to an existing file, use the double greater-than sign (“>>”).

2. Select the starting line. In the Debug window, move the cursor to the line to start logging.
3. Set the starting line. In the Debug window Commands menu, select the Set log start line command.
4. Select the ending line. In the Debug window, move the cursor to the last line to log.
5. Write to the log file. In the Debug window Commands menu, select the Perform File Log command. This actually writes the text in the Debug window to the log file.
6. Close the log file. In the Command window, enter the following command:

```
Log Off
```

This command flushes and closes the log file.

---

**NOTE**

*dScope writes plain ASCII text to the log file. You may use any text editor to view and modify the contents of the log file.*

---

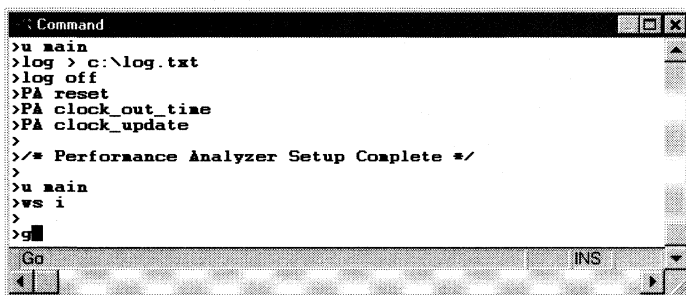
## Command Window

You use the Command window to enter and display the results of dScope commands. You may display or hide the Command window using Command window in the dScope View menu or the toolbar Command window button shown on the right.



2

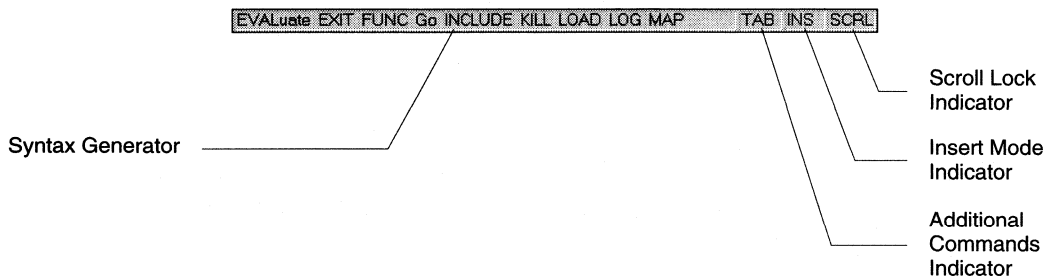
Most dScope commands may be executed using dialog boxes, however, you may also execute dScope commands directly in the Command window.



You enter commands by typing them in the Command window. All commands are terminated with the **Enter** key.

## Status Bar

The Command window has a status bar that displays at the bottom of the window.





Each section of the status bar has a different purpose.

- The Syntax Generator displays a command menu in the status bar as commands are entered. Help text is continuously displayed during command entries. The help text reflects the syntax of the command and includes information about command keywords, required parameters, and command options. This makes command entry easy since options from the help line may be transferred to the command line.
- The Additional Commands Indicator displays **TAB** if there are more commands in the Syntax Generator that may be accessed with the **Tab** key.
- The Insert Mode Indicator displays **INS** if the keyboard is in Insert Mode and **OVR** if the keyboard is in Overstrike Mode. Use the **Ins** key to toggle between Insert and Overstrike Mode.
- The Scroll Lock Indicator displays **SCRL** if Scroll Lock is enabled. Use the **Scroll Lock** key to toggle the Scroll Lock.

**2**

## Command Buffers

The Command window maintains two command buffers. The command output buffer stores the last 32K of text generated by dScope commands. The command line history buffer stores the last 64 input lines entered in the Command window.

You may review the output from prior dScope commands and you may recall previous commands for subsequent use.

### Recalling Command Lines

You may use the up and down cursor keys to scroll through the command line history buffer. The **Scroll Lock** LED on the keyboard must be off to scroll through the command line history buffer. If the **Scroll Lock** LED is on, the cursor keys scroll through the command output history.

You may edit and re-enter a command from the buffer using the cursor left, cursor right, and **Enter** keys.

## Reviewing Command Output

You may use the scroll bars in the Command window to review the output from commands previously entered in the Command window. You may also use the up and down cursor keys (in combination with **Scroll Lock**) as well as the cursor left and right keys and **PgUp** and **PgDn** to scroll through the command history.

# 2

## Command Line Editing

You may edit the command line using the following keys:

Key	Operation
<b>Enter</b>	Execute command line.
<b>Backspace</b>	Delete character left of the cursor.
<b>Ctrl+D, Ctrl+F, Del</b>	Delete character under the cursor.
<b>Esc, Ctrl+C</b>	Erase command entry without executing. Stops execution if dScope is running a target program.
<b>Home</b>	Position cursor at the beginning of the command line.
<b>End</b>	Position cursor at the end of the command line.
<b>Ins</b>	Toggle between insert and overwrite mode.
<b>Tab</b>	Display additional syntax generator information in the status bar.
← †	Move the cursor one character to the left.
→ †	Move the cursor one character to the right.
↑ †	Restore the prior command line.
↓ †	Restore a subsequent command line.

† These control keys work only when **Scroll Lock** is disabled.

## Comment Lines

dScope lets you enter comments on the command line. Comments that follow the language conventions of C and PL/M are supported with only a few restrictions:

- A comment may not extend over more than one line in command mode.
- A comment may extend over more than one line in a dScope function.

For example:

```
>BS TEST                                /* The following line is a command */
                                        /* Set breakpoint on TEST */
                                        /* Comments entered on a command line... */
                                        /* may not extend beyond that line. */

>FUNC void abc (void) {                 /* The following is a function definition */
    1:                                /* This is a comment that extends beyond
    2:                                one line in a function. */
    3: }
```

---

### NOTE

*The syntax generator stops providing context-sensitive help once a comment is entered.*

---

## Command Chaining

dScope allows you to enter multiple commands, separated by semicolons (“;”), on a single line. This is required when you wish to assign several commands to a function key. For example:

```
>SET F2 = "RESET; BS 0010h; G"
```

Assigns **F2** to reset the CPU, set a breakpoint at 0010h, and begin execution.

---

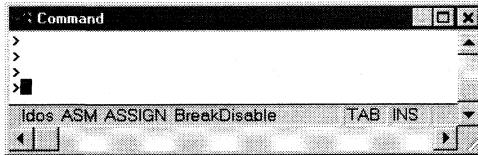
### NOTE

*The syntax generator only provides context-sensitive help for the first command entered.*

---

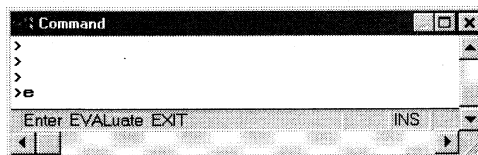
## Syntax Generator

The command menu of the syntax generator is displayed in the Command window's status bar. During command entry, the syntax generator displays possible commands, options, and parameters.

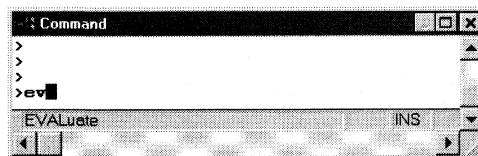


The status bar displays **TAB** to indicate additional commands are available. Use the **Tab** key to scroll through the available commands. Significant letters display in uppercase.

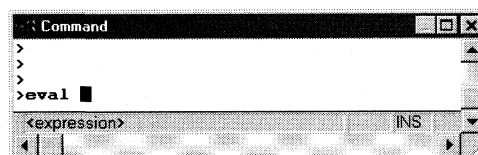
As you enter a command, the syntax generator reduces the list of likely commands to coincide with the characters you type. If you type **e** in the command window, the syntax generator responds by reducing the commands listed in the status bar to **Enter**, **EVALuate**, and **EXIT** as shown in the following figure.



Now, if you type **v** in the command window, the syntax generator changes again to show the commands that begin with **ev**.

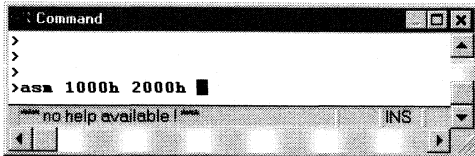


The syntax generator now displays only one command. You may type a **Space** to let the syntax generator automatically complete the command.



After entering the **EVALUATE** command, the syntax generator prompts with **<expression>** to indicate an expression should be entered.

The syntax generator leads you through dScope command entry and helps you avoid errors. If the syntax generator does not recognize the syntax of a command, it displays **\*\*\* no help available! \*\*\*** as shown below.



2

This message usually indicates an invalid command entry.

## Command Files

In addition to typing commands in the Command window, you may also direct dScope to read commands from a command file. Command files are plain ASCII text files that you may create with any text editor. Command files may contain multiple commands. They provide a convenient way to repeat complex commands over and over without the need to retype each individual command. A command file may load a CPU driver, set the memory map for the program, load an object file, set breakpoints, set watchpoints, and begin program execution.

The following example command file loads the CPU driver, the target program, begins executing the program, breaks at the **main** C function, and sets watchpoints for three variables; **var1**, **var2**, and **var3**.

```
load 8052.dll      /* Load 8052 CPU driver */
load myprog        /* Load MYPROG absolute object file */

g,main             /* GO until the main function is reached */

ws var1            /* Set a watchpoint on var1 */
ws var2            /* Set a watchpoint on var2 */
ws var3            /* Set a watchpoint on var3 */
```

Use the **include** command to read and execute the lines in a command file. For example:

```
include c:\c51\examples\cmds.txt
```

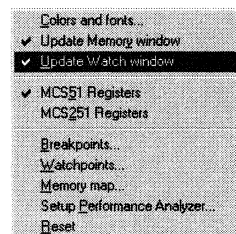
## Watch Window

The Watch window displays scalar, structure, union, and array variables from your target program. You may display or hide the Watch window using the Watch window command from the View menu or the toolbar Watch window button shown on the right. The Watch window may be active at any time, even during target program execution.

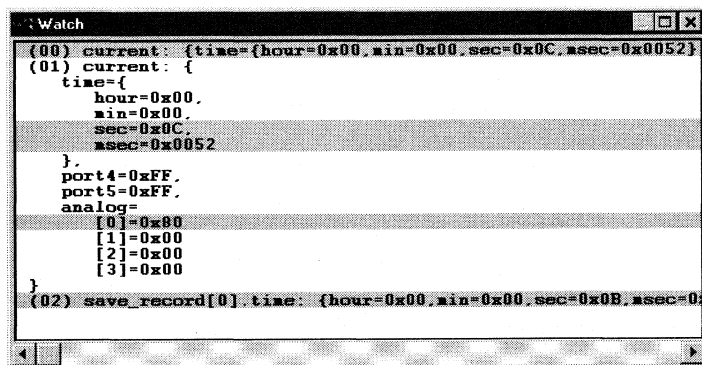


2

The contents of the Watch window are automatically updated whenever program execution stops. Values changed since the last update are highlighted. By default, the Watch window does not update while a target program is running. However, you may choose to update the Watch window during target program execution by selecting the Update Watch window command from dScope's Setup menu shown on the right. When selected, this option updates the Watch window every 1255 instructions. This lets you see variables change as your target program runs. It also slows down simulation.



Expressions that display in the Watch window are called watchpoints. They may display on a single line or on multiple lines. Single lines are truncated at 128 characters. This may not be the best choice for large data structures and arrays. In multi-line mode, each member of a structure or union and each element of an array displays on separate lines.



The figure above shows a single-line and a multi-line display for the **current** structure. Note that each watchpoint is preceded by an index in parentheses. You use this index to remove a specific watchpoint.

Refer to “Chapter 7. Using Watchpoints” on page 153 for more information about using watchpoints.

## Register Window

The Register window displays the values of the CPU registers, instruction cycles executed, and the total computed execution time based on the current crystal frequency. You may display or hide the Register window using the Register window command from the View menu or the toolbar Register window button shown on the right.



The contents of the Register window are automatically updated whenever program execution stops. Values changed since the last update are highlighted. The window is updated if it is selected by clicking the left mouse button in the Register window. If the Register window is current selected, double-clicking the left mouse button in the window updates the register values.

dScope uses different formats for the MCS<sup>®</sup> 51, MCS<sup>®</sup> 251, and 166 Register windows. The appropriate format is automatically selected when a CPU driver is loaded. When debugging using dScope for the MCS<sup>®</sup> 51 and MCS<sup>®</sup> 251, the Register window format may be changed using the MCS51 Registers and MCS251 Registers commands from the Setup menu.

### MCS<sup>®</sup> 51 Register Window

When you load a CPU driver for an 8051-compatible device, the Register window appears as shown on the right. The Register window displays the A and B registers, R0-R7, data pointer (DPTR), stack pointer (SP), and program status word (PSW). The current program counter is identified by \$. Cyc indicates the number of instruction cycles executed. Sec indicates the total execution time calculated using the current value of XTAL. Refer to "CPU Pin Registers (VTREGs)" on page 91 for more information.

Regs	
A=00	B=01
R0=4A	R1=E3
R2=03	R3=FF
R4=00	R5=0F
R6=00	R7=11
DPTR = 03E3	
SP = 0069	
PSW: ---R0---	
\$:	C: 1B28
Cyc:	12270514
Sec:	12.270514

## MCS<sup>®</sup> 251 Register Window

The Register window appears as shown on the right for 251-compatible devices. The Register window displays byte registers R0-R15, word registers WR16-WR31, data pointer (DPTR), stack pointer (SP), both program status words (PSW and PSW1), program counter (\$), program states (Sts), and execution time (Sec).

Regs	
R0:	00 00 00 0A
R4:	00 00 00 09
R8:	00 00 00 00
R12:	00 00 00 00
WR16:	0000 0000
WR20:	0000 0000
WR24:	0000 0000
WR28:	0000 0000
DPTR:	0001:03B9
SP:	0000:0033
PSW:	CA-R0----
PSW1:	CANR0----
\$:	00FF:0B40
Sts:	570258
Sec:	0.095043

## 2

### Changing Register Values

The Register window does not support directly manipulating register contents. To change register value, you must enter the new register values in the Command window. For example:

```
>A = 0xFE           /* Assign 0xFE to Accumulator */
>DPTR = 0x1234       /* Assign 0x1234 to DPTR */
>WR20 += --DPTR      /* Note C style operators */
>R0=3,R1=4,R2=5      /* Comma separated expressions */
>R7=current.time.sec-- /* Program symbols used */
>DR56=0x20000        /* Set MCS 51 xdata segment to 2 */
```



## Serial Window

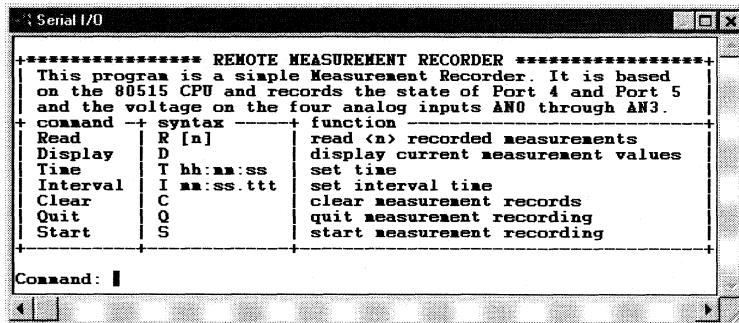
The Serial window emulates a serial terminal connected to the CPU's built-in serial port or any other special function register (like a parallel port). You may display or hide the Serial window using the Serial window command from the View menu or the toolbar Serial window button shown on the right.



All keys typed in the Serial window, except special dScope control keys, are passed to the CPU's serial port SFRs.

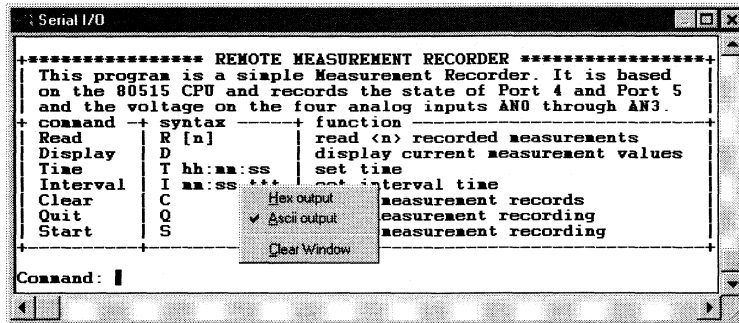
You may specify the CPU registers that get input from and provide output to the Serial window. Refer to "ASSIGN" on page 198 for more information.

2



The Serial window maintains a history buffer that holds the most recent 8K of serial data. dScope automatically wraps lines longer than 128 characters. You may use the cursor keys and horizontal and vertical scroll bars to review the contents of the history buffer.

The Serial window displays the output in ASCII or HEX format. To change the format, click the right mouse button in the Serial window. dScope presents a menu that lets you select Hex Output, ASCII Output, and Clear Window.



Data previously displayed in the Serial window does not change when you select a new output mode. Only subsequent data is affected by the output mode change. To clear the Serial window, select the Clear Window command.

#### NOTE

*The output mode does not affect the characters entered in the Serial window. The output mode only affects how characters are displayed. If a character is entered, its ASCII value is transmitted to the CPU's SFRs.*

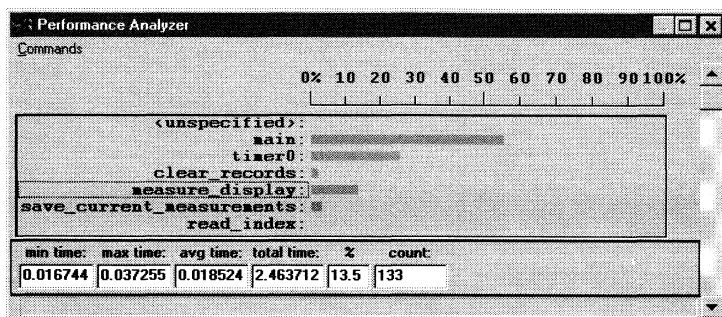
## Performance Analyzer Window

The Performance Analyzer window displays the execution time recorded for up to 256 functions and address ranges you specify. You may display or hide the Performance Analyzer window using the Performance Analyzer command from the View menu or the toolbar Performance Analyzer window button shown on the right.



Results display in the Performance Analyzer window as bar graphs. Information such as invocation count, minimum time, maximum time, and average time is displayed for the selected function or address range.

2



The Performance Analyzer window contains a Commands menu, a percentage ruler, a status bar, and a bar for time display. The Performance Analyzer window shows the starting address for each address range or function name followed by a bar that indicates the percent of time spent in that code range.

The **<unspecified>** address range is automatically generated by dScope. It shows the amount of time spent executing code that is not included in the specified functions or address ranges.

You may select any address range in the Performance Analyzer window. The timing statistics for the selected address range appear at the bottom of the window.

Each of these statistics is described in the following table.

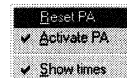
Label	Description
<b>min time</b>	The minimum time spent in the selected address range or function.
<b>max time</b>	The maximum time spent in the selected address range or function.
<b>avg time</b>	The average time spent in the selected address range or function.
<b>total time</b>	The total time spent in the selected address range or function.
<b>%</b>	The percent of the total time spent in the selected address range or function.
<b>count</b>	The total number of times the selected address range or function was executed.

#### NOTE

*The **total time** and **%** are the only fields displayed for the <unspecified> address range.*

## Commands Menu

The Performance Analyzer command menu shown on the right provides you with the following commands:



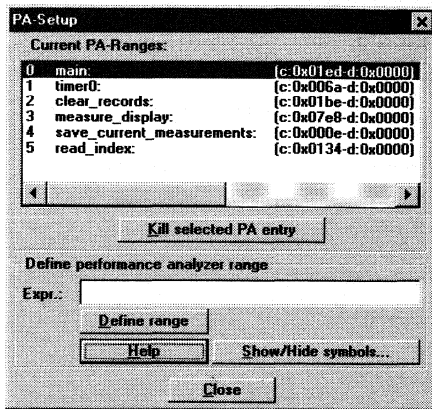
- **Reset PA:** Resets the performance analyzer by clearing the recorded time and invocation information of all currently defined functions and address ranges.
- **Activate PA:** Enables or disables time recording. If disabled, the display freezes and no performance analysis occurs.
- **Show times:** Shows or hides the additional timing statistics.

## Adding and Removing Address Range

You must define functions and address ranges to analyze before you can use the Performance Analyzer. A valid address range must have a unique entry and exit point.

You may define and remove functions or address ranges for the Performance Analyzer either in the Command window (refer to “Performance Analyzer” on page 240) or in the PA-Setup dialog box.

Select the Setup Performance Analyzer command from the dScope Setup menu to open the PA-Setup dialog box.

**2**

### To define an address range...

- Enter the address range or function name in the Expr. input line. Address ranges require that you enter the entry address and exit address separated by a comma (“,”). The address range for a function is automatically derived by dScope from the high-level debugging information included in the target program.
- Click the Define Range button to add the address range to the list of Current PA Ranges.

You may also drag symbols from the Symbol Browser window and drop them onto the Expr. input line. The Show Symbols button opens the Symbol Browser window. Refer to “Symbol Browser Window” on page 56 for more information.

### To remove an address range...

- Select the address range to remove from the Current PA Ranges list.
- Click on the Kill selected PA entry button to remove the selected address range or function from performance analysis.

## Valid Address Ranges

Address ranges specified for the Performance Analyzer must have unique entry and exit points. You may use the Scope command or the Symbol Browser window to obtain address range information about the functions available in your target program.

### 2

#### Using the Scope command...

The Scope command displays address range information for all currently defined program blocks. For example:

```
MEASURE
{CvtB} RANGE: 0xFF03B7-0xFF07E5
{CvtB} RANGE: 0xFF000B-0xFF000D
SAVE_CURRENT_MEASUREMENTS RANGE: 0xFF000E-0xFF0069      /* valid */
TIMER0 RANGE: 0xFF006A-0xFF0135                          /* valid */
_READ_INDEX RANGE: 0xFF0136-0xFF01BF                     /* valid */
_CLEAR_RECORDS RANGE: 0xFF01C0-0xFF01EE                   /* valid */
MAIN RANGE: 0xFF01EF-0xFF03B6                             /* valid */
MCOMMAND
{CvtB} RANGE: 0xFF09A6-0xFF0A23
MEASURE_DISPLAY RANGE: 0xFF07E7-0xFF084A                 /* valid */
_SET_TIME RANGE: 0xFF084B-0xFF08CA                       /* valid */
_SET_INTERVAL RANGE: 0xFF08CB-0xFF09A5                   /* valid */
GETLINE
_GETLINE RANGE: 0xFF0A24-0xFF0A87                         /* valid */
?C_FPADD
?C_FPMUL
```

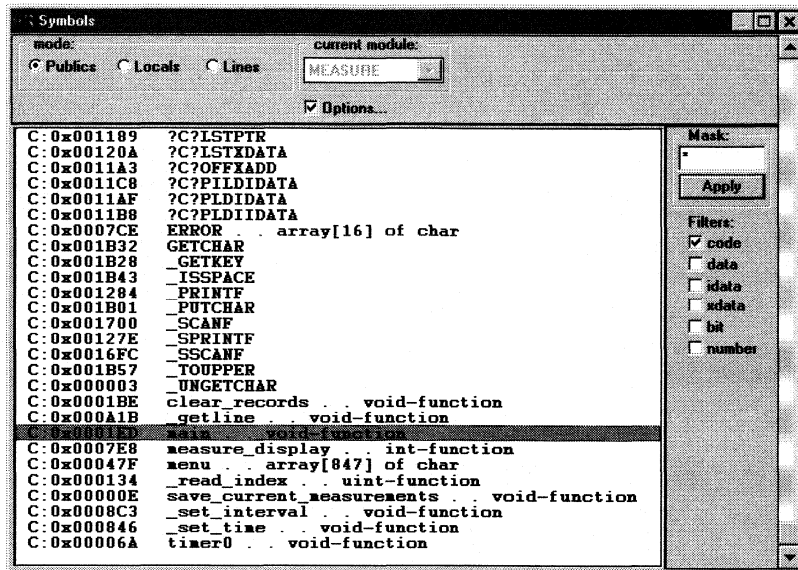
Lines beginning with {CvtB} are created by dScope for blocks that have insufficient debug information. This is normal for modules that have no debug information (like those in libraries or assembly language modules). These lines do not represent valid address ranges.

Lines with identifiers that are indented represent functions and their address ranges. You may use all such address ranges.

Refer to “SCOPE” on page 246 for more information about the Scope command.

## Using the Symbol Browser window...

The Symbol Browser window makes it easy to view and select functions for the Performance Analyzer. You can easily display the code symbols in your target program—select the Publics or Locals radio button, select the Options check box, and clear all Filters except code.



Functions listed in the Symbol Browser window may be dragged and dropped into the PA-Setup dialog box.

## Memory Window

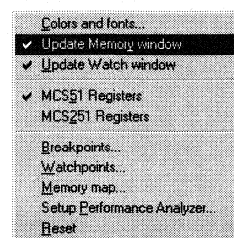
The Memory window displays the contents of the various memory areas. You may display or hide the Memory window using the Memory window command from the View menu or the toolbar Memory window button shown on the right.



2

Output in the Memory window is formatted in HEX and ASCII. The contents of the Memory window are automatically updated whenever program execution stops.

By default, the Memory window does not update while a target program runs. However, you may choose to update the Memory window during target program execution by selecting the Update Memory window command from dScope's Setup menu shown on the right. When selected, this option updates the Memory window while your target program runs. This lets you see memory locations change as your target program runs. It also slows down simulation.



Memory																	0123456789ABCDEF															
Addr	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F																
C:0280:	E4	FA	FB	12	01	34	8E	3B	8F	3C	EF	F4	70	02	EE	F4	.....4.....<..p.....															
C:0290:	70	03	02	12	10	E5	3C	65	2E	70	04	E5	3B	65	2D	70	p.....<e.p...e-p.....															
C:02A0:	03	02	02	10	30	98	0B	12	1B	28	EF	64	1B	70	03	02	...0.....<d.p.....															
C:02B0:	02	10	90	00	00	75	F0	0B	E5	3C	12	11	A3	E5	3B	75	...u.....<...>..u.....															
C:02C0:	F0	0B	A4	25	83	F5	83	E0	F4	60	33	AE	3B	AF	3C	7C	...%.....<3.....< .....															
C:02D0:	00	7D	0B	12	11	48	74	00	2F	F9	74	00	3E	FA	7B	01	...}...Ht...t...>{.....															
C:02E0:	C0	03	1B	78	3D	7C	00	AD	03	D0	03	7E	00	7F	0B	12	...x= .....<.....															
C:02F0:	10	C7	12	07	E8	7B	FF	7A	03	79	E4	12	12	84	05	3C	...{z.y.....<.....															
C:0300:	E5	3C	70	02	05	3B	64	E8	70	04	E5	3B	64	02	70	85	<p...d.p...d.p.....															
C:0310:	F5	3B	F5	3C	02	02	95	74	12	25	3A	F9	E4	FA	FB	12	<p...<t%.....<.....															
C:0320:	08	46	02	02	10	74	12	25	3A	F9	E4	FA	FB	12	08	C3	F...t%.....<.....															
C:0330:	02	02	10	7B	FF	7A	03	79	E6	12	12	84	20	98	1D	D2	...{z.y.....<.....															
C:0340:	01	20	01	FD	78	3D	7C	00	7D	00	7B	00	7A	00	79	22	...{z.y.....<.....															
C:0350:	7E	00	7F	0B	12	10	C7	12	07	E8	80	E0	12	1B	28	BF	~...x= ...{z.y...{.															
C:0360:	1B	DA	7B	FF	7A	04	79	15	12	12	84	02	02	10	7B	FF	...{z.y.....<.....															
C:0370:	7A	04	79	18	12	12	84	D2	00	02	02	10	7B	FF	7A	04	z.y.....<.....{z...															
C:0380:	79	36	12	12	84	C2	00	02	02	10	7B	FF	7A	04	79	53	y6.....<z.yS															
C:0390:	12	12	84	12	01	BE	02	02	10	75	4C	FF	75	4D	04	75	...uL...uL...u.....															
C:03A0:	4E	6F	7B	FF	7A	07	79	CE	12	12	84	7B	FF	7A	04	79	No{z.y.....<z.y															
C:03B0:	7F	12	12	84	02	02	10	22	51	55	49	54	20	4D	45	41	....."QUIT MEA															
C:03C0:	53	55	52	45	4D	45	4E	54	53	20	42	45	46	4F	52	45	SUREMENTS BEFORE															
C:03D0:	20	52	45	41	44	00	25	64	00	0A	43	6F	6D	6D	61	6E	READ.%d...Comman															
C:03E0:	64	3A	20	00	0A	00	0A	44	69	73	70	6C	61	79	20	63	d:....Display c															
C:03F0:	75	72	72	65	6E	74	20	4D	65	61	73	75	72	65	6D	65	urrent Measureme															

The address range displayed is defined with the Display Memory command entered in the Command window. Refer to "DISPLAY" on page 216 for more information.



The following example commands show you how to display different memory areas in the Memory window.

```
>D X:0x0000,X:0xFFFF /* 64K of xdata memory */  
>D I:0x00,I:0xFF /* The internal data memory */  
>D current /* 256 bytes starting from &current */  
>D save_record /* 256 bytes starting at &save_record */
```

The Display Memory command takes two arguments separated by a comma (“,”). The first parameter specifies the starting address while the second parameter specifies the ending address. If the ending address is omitted, it is assumed to be either at the end of the specified memory space or 256 bytes after the starting address, whichever is smaller.

**2**

---

### NOTES

*If the Memory window is not displayed, the results of the Display Memory command are output to the Command window.*

*dScope truncates the address range to one 64K segment. If a range from 0x000000 to 0x1FFFFF is given, dScope truncates it to 0x000000 to 0x0FFFFF.*

---

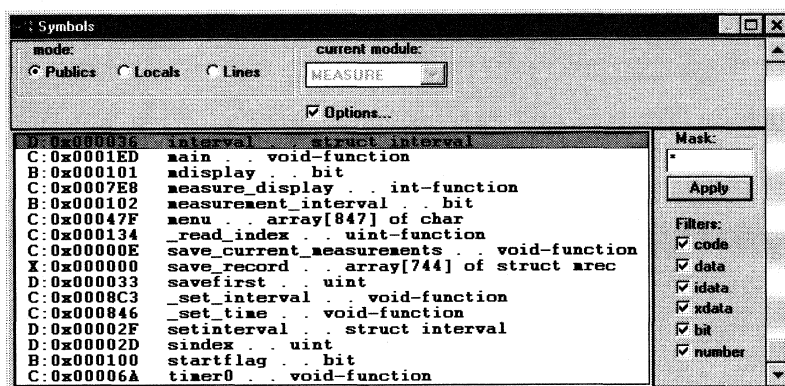
## Symbol Browser Window

The Symbol Browser window displays public symbols, local symbols, or line number information defined in the currently loaded target program. You may display or hide the Symbol Browser window using the Symbol window command from the View menu or the toolbar Symbol window button shown on the right.



2

CPU-specific symbols defined by the CPU driver are also displayed in the Symbol Browser window.

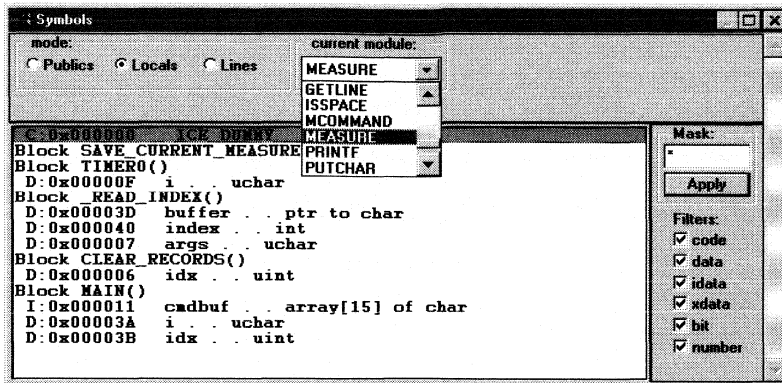


## Symbol Display Mode

You may select the type of symbols to display (Publics, Locals, or Lines) by selecting the appropriate radio button in the mode box at the top of the window.

Symbol Mode	Description
<b>Publics</b>	All public symbols in the target program are displayed. Public symbols are symbols which have application-wide scope rather than module-scope or function-scope. The CPU-specific symbols, such as SFR names and bit names, are considered public symbols.
<b>Locals</b>	Displays local symbols of the functions in the current module. The current module is selected using the current module drop-down combination box. Local symbols are indented under the name of each function.
<b>Lines</b>	Displays line numbers and associated code addresses for the current module. The current module is selected using the current module drop-down combination box.

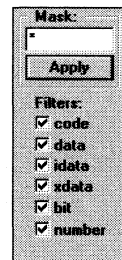
When Locals or Lines are selected, the current module drop-down combination box lets you select the module for which local symbols or line number information displays.



The current module drop-down combination box lists the names of all modules. Some modules may not contain local symbols or line information. Typically, these modules were compiled without debugging information or were included from a library.

## Symbol Options

The Options... check box in the Symbol Browser window displays or hides a window with Filter check boxes and a Mask input line. Using these controls, you may limit the displayed symbols to match the memory area and symbol name you specify. Filters and Masks are disabled when displaying line number information since line numbers are located in code space and have no names.



Filters let you specify the memory space from which symbols are displayed. For example, if you select code, only those symbols that are in the code space display.

The Mask lets you specify a regular expression that is used to match specific symbol names. The Mask may consist of alphanumeric characters plus the mask characters described in the following table.

Mask Character	Description
#	Matches a digit (0-9).
\$	Matches any character.
*	Matches zero or more characters.

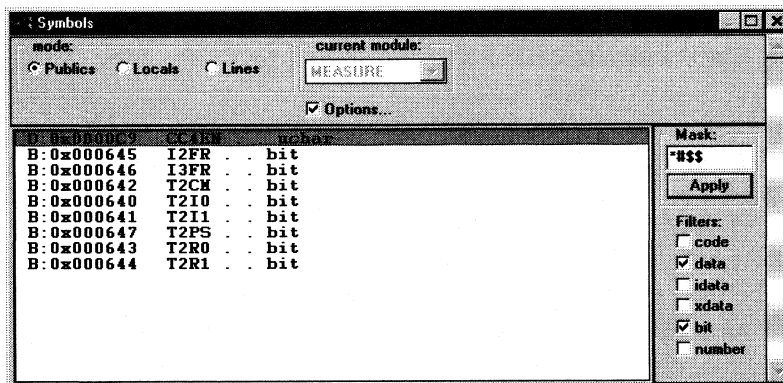
The following table provides a few examples of name masks.

Mask	Description
*	Matches any symbol. This is the default mask in the Symbol Browser.
***	Matches symbol names that contain one digit in any position.
##	Matches symbol names that end in two adjacent digits.
<u>a\$*</u>	Matches symbol names starting with an underline, followed by the letter <b>a</b> , followed by any character, followed by a digit, ending with zero or more characters. For example, <u>ab1</u> or <u>a10value</u> .
<u>*ABC*#</u>	Matches symbol names that start with an underline character, followed by zero or more characters, followed by <b>ABC</b> , followed by zero or more characters, ending with a digit.

Following are several rules which apply to the search Mask.

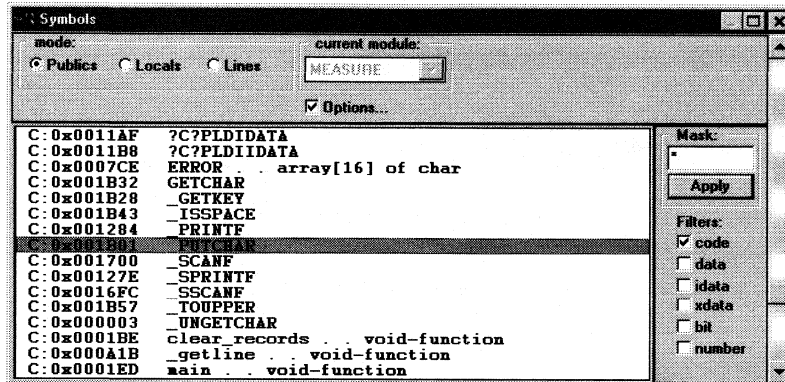
- The alphabetic characters of a search mask are case sensitive.
- A search mask may not have two adjacent asterisks ("\*\*").
- The memory space (Filter) is also taken into account for matching symbols.
- The Apply button applies the mask to the symbols and displays the updated symbol list.

The following illustration shows the symbol output for Filter and Mask selections.



## Public Symbols

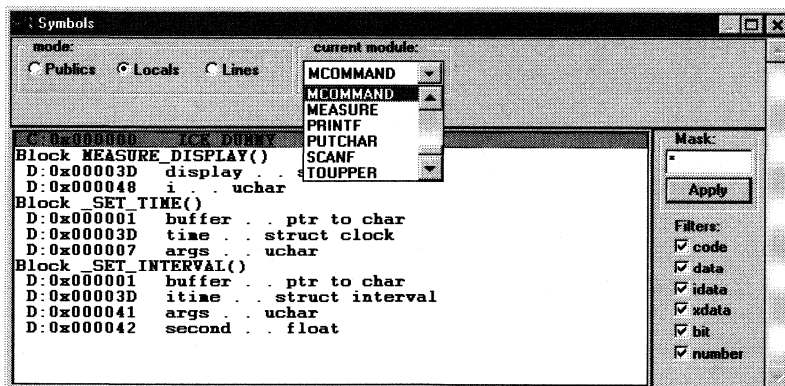
The Symbol Browser displays public symbols only when the Publics mode is selected. Since public symbols have application-wide scope, the current module drop-down combination box is disabled. You may restrict the symbol output by changing the Mask or memory space Filters.



2

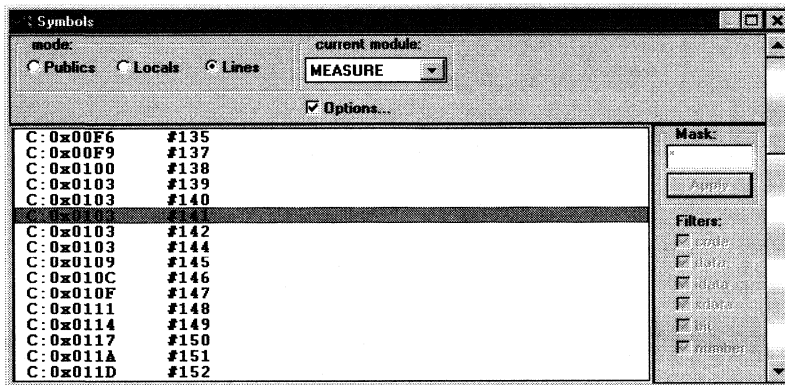
## Local Symbols

The Symbol Browser displays local symbols when the Locals mode is selected. The output corresponds to the module selected in the current module drop-down combination box. Local symbols are indented under blocks that specify the name of the function to which the symbols belong. You may restrict the symbol output by changing the Mask or memory space Filters.



## Line Numbers

The Symbol Browser displays line number information when the Lines mode is selected. Each line begins with the code address followed by the line number in the selected module. Output corresponds to the module selected in the current module drop-down combination box. Mask and memory space Filters are disabled when displaying line number information.



## Dragging Symbols

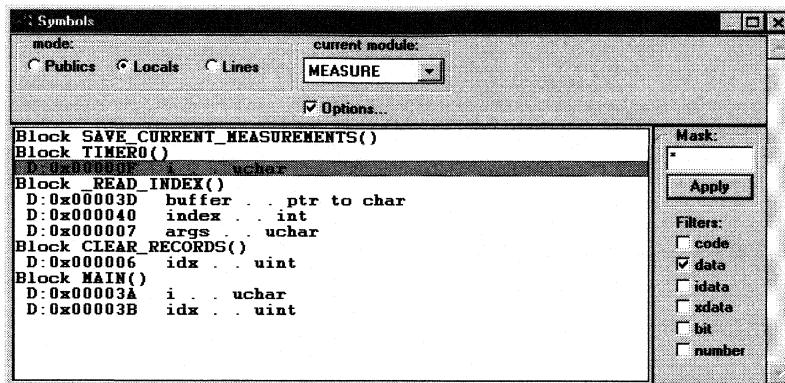
dScope lets you drag symbols from the Symbol Browser window and drop them in other windows and dialogs that require symbol name input. This includes the In-line Assembler dialog box, the Watchpoint dialog box, the Breakpoint dialog box, and the Command window.

To drag a symbol from the Symbol Browser window, click on the symbol with the left mouse button and, without releasing the mouse button, drag the symbol to the desired window or dialog. As you drag the symbol, the mouse cursor changes to indicate where you may drop the symbol name. To drop the symbol, release the left mouse button.

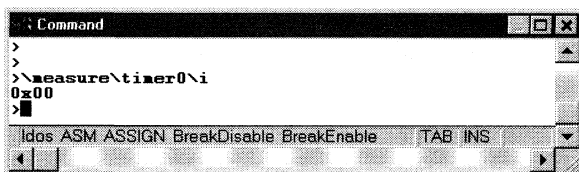
You may drag public symbols, local symbols, and line numbers from the Symbol Browser window. When you drag and drop a public symbol, the name of the public symbol is inserted. When you drag and drop a local symbol, the fully qualified name is inserted. When you drag and drop a line number, the fully qualified line number is inserted. The components in a qualified name are delimited by a backslash character.

## Fully Qualified Local Symbols

Fully qualified local symbols include the name of the module, the name of the function, and the name of the symbol. For example:

**2**

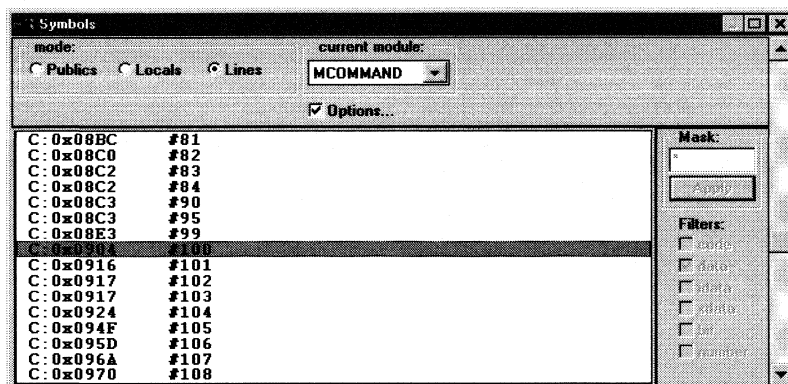
When you drag and drop the local symbol **i** from the **TIMER0** function into the Command window, the fully qualified symbol name displays as follows.



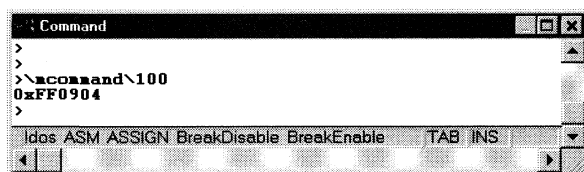
Refer to “Fully Qualified Symbols” on page 97 for more information on fully qualified local symbols.

## Fully Qualified Line Numbers

Fully qualified line numbers include the name of the module and the line number. For example:



When you drag and drop line number **100** from the **MCOMMAND** module into the Command window the fully qualified line number displays as follows.



Refer to “Fully Qualified Symbols” on page 97 for more information on fully qualified line numbers.

### NOTE

*Line numbers are associated with a module and not with the functions in a module.*

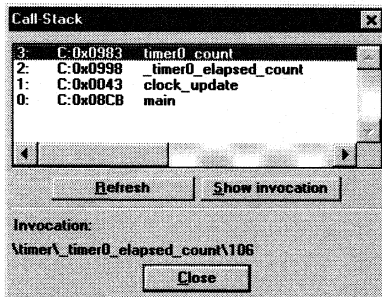


## Call Stack Window

The Call Stack window shows a list of currently nested function calls or interrupt procedures. You may display or hide the Call Stack window using the Call-Stack command from the View menu or the toolbar Call Stack window button shown on the right.

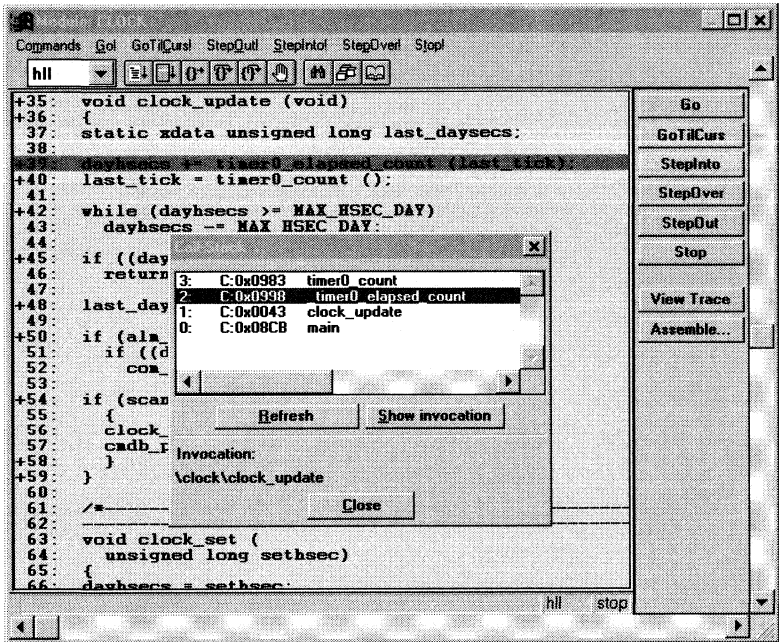


Each line in the display begins with the nesting level followed by the address of the invoked function and the symbolic name of the function if available.



The Call Stack window may be left open at all times. The Refresh button updates the window to reflect the current function nesting, even during program execution. When a function in the Call Stack list is selected, the calling function is listed at the bottom of the window.

The Show Invocation button displays, in the Debug window, the code that called the selected function in the Call Stack list.



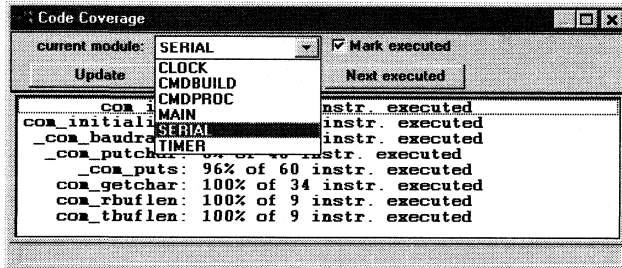
The highlighted line in the above image shows where the timer0\_elapsed\_count function is invoked.

## Code Coverage Window

The Code Coverage window shows you the percentage of code that has been executed in the functions in a source module. You may display or hide the Code Coverage window using the Code Coverage window command from the View menu or the toolbar Code Coverage window button shown on the right.

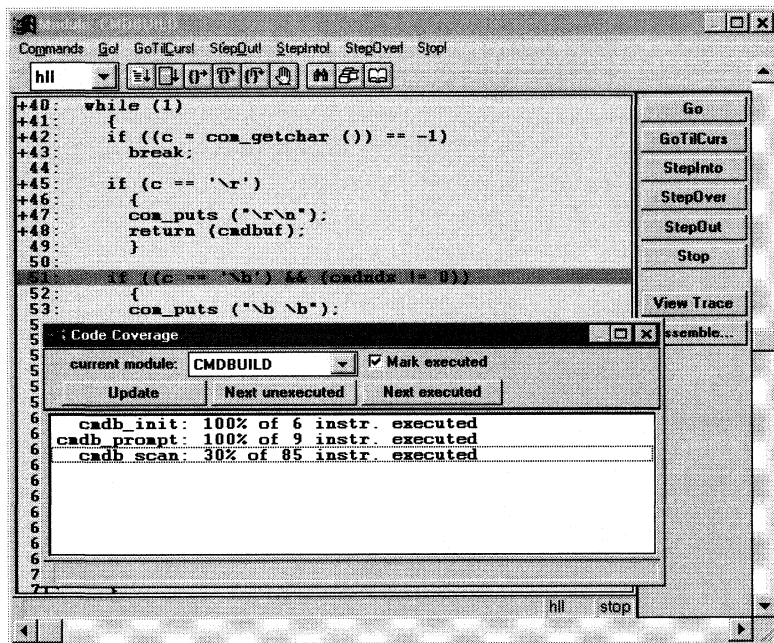


The Code Coverage window helps you identify and test infrequently executed parts of your program. You may use this information to better test your target program and exercise those sections that are rarely executed. You can use the Code Coverage window to easily locate programming errors and dead code in your target program.

**2**

The Code Coverage window lists the names of all functions which correspond to the selected module. Each line starts with the name of a function and includes the percent of instructions executed and the total number of instructions in the function.

The current module drop-down list lets you select the source module for code coverage information. Once a module is selected, the Code Coverage window shows the percent of instructions in each function that have been executed. If the Mark executed check box is checked, executed instructions in the Debug window are marked with a plus sign (“+”).



You may select a function listed in the Code Coverage window and use the Next unexecuted button to locate lines of code that have not been executed. You may use the Next executed button to locate lines of code that have been executed.

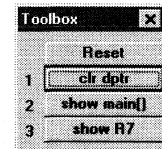
You may use the Update button to update the Code Coverage window percentages while your target program runs.

## Toolbox Window

The Toolbox window is a user-configurable dialog box that contains up to 16 command buttons you define. You may display or hide the Toolbox window using the Toolbox command from the View menu or the toolbar Toolbox window button shown on the right.



Clicking the mouse on a button in the Toolbox window executes the button's associated command. Commands in the Toolbox may be executed at any time, even while a target program is running.

**2**

### To create a toolbox button...

You may add buttons to the Toolbox window by entering the Define Button command in the Command window along with the name and command to assign to the button. The general syntax is:

**DEFINE BUTTON** "button\_label", "button\_cmd"

where:

*button\_label* is the name to display on the button in the Toolbox window,

*button\_cmd* is the dScope command to execute when the button is pressed.

The following examples show the define commands used to create the buttons in the Toolbox window shown above.

```
>DEFINE BUTTON "clr dptr", "dptr=0"  
>DEFINE BUTTON "show main()", "u main"  
>DEFINE BUTTON "show R7", "printf (\"R7=%02XH\n\",R7) "
```

---

### NOTE

*The printf command defined in the last button definition shown above introduces nested strings. The double quote characters (") of printf's format string must be escaped (\") to avoid syntax errors.*

---

When a button is defined, it is immediately added to the Toolbox window. Each button receives a button number which is displayed in the Toolbox window. This number is used to remove specific buttons from the Toolbox window.

---

**NOTE**

*Button command definitions are not saved in dScope's INI file when you exit.*

---

**2****To remove a toolbox button...**

You may remove a button from the Toolbox window by entering the Kill Button command along with the button number in the Command window. For example:

```
>Kill Button 3      /* Remove show R7 button */  
>Kill Button 2      /* Remove show main() button */
```

---

**NOTE**

*The **Reset** button in the Toolbox is created automatically by dScope and cannot be removed.*

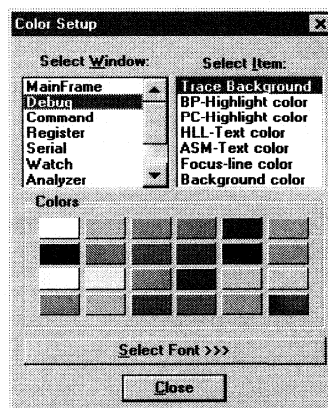
---

## Color Setup Dialog Box

dScope lets you set the colors and fonts used for each window. The colors and fonts used may be changed in the Color Setup dialog box which you open using the Colors and fonts command from the dScope Setup menu. dScope saves all color and font settings, as well as window sizes and positions, when you exit.

### NOTE

*Loading a target program before changing colors or fonts makes it easier to see the parts of a window that are affected.*

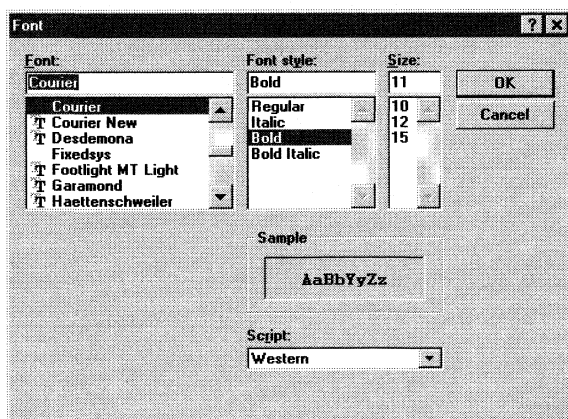


2

Setting colors and fonts for all windows is straightforward. The Debug window is more complex than other windows, but color and font selections are the same.

## To Change Fonts for a Specific Window...

1. Open the window you want to change.
2. Open the Color Setup dialog box and select the window to change from the Window list box. If the selected window allows font changes, the Select Font button displays at the bottom of the Color Setup dialog box.
3. Click on the Select Font button to open the Font dialog box.



4. Select the font, style, and size that you want and click the OK button. The window is immediately redrawn using the new font.

### NOTE

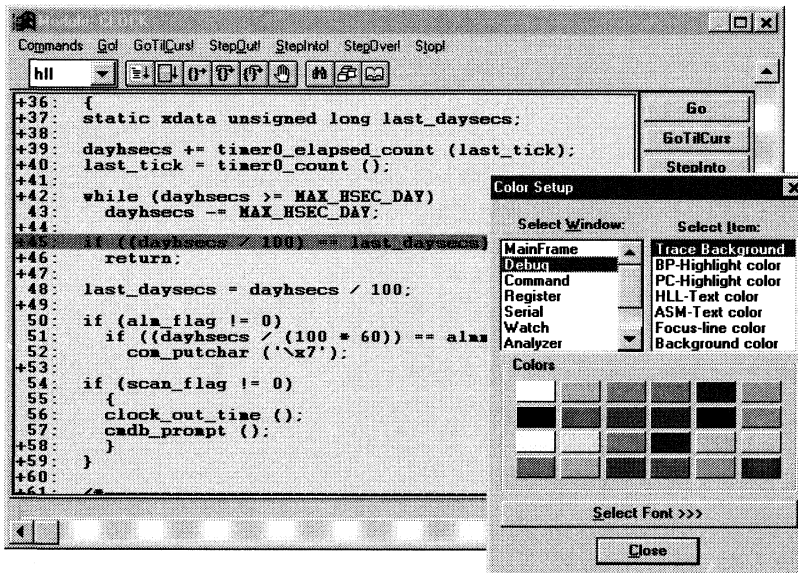
*The fonts listed in the Font listbox depend on the fonts installed and may be different on your computer.*

## 2

### To Change Colors for a Specific Window...

1. Open the window you want to change.
2. Open the Color Setup dialog box and select the window to change from the Window list box. When the window is selected, the Item list box shows the items you may change for the selected the window.

If you select the Debug window, the dScope screen should display that window and the Color Setup dialog box as shown below.



3. Select the item whose color you want to change and select a new color from the color palette. The item color in the window changes immediately.



## Debug Window Notes

The Debug window is the most complex window as far as colors are concerned. The following table describes the color items available in the Debug window.

Color Item	Description
<b>Trace Background</b>	Background color for disassembly and source code displayed from the trace buffer.
<b>BP-Highlight Color</b>	Background color for breakpoints.
<b>PC-Highlight Color</b>	Background color for the current program counter.
<b>HLL-Text Color</b>	High-level language text color.
<b>ASM-Text Color</b>	Assembly language text color.
<b>Focus-Line Color</b>	Background color for the focus line.
<b>Background Color</b>	Background color for everything else.

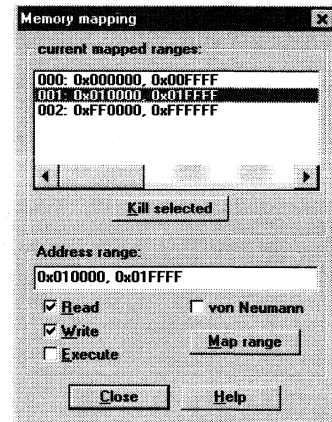
## Memory Map Dialog Box

The Memory Map dialog box lets you specify the memory areas your target program uses for data storage and program execution. You may also configure the target program's memory map using the **MAP** command. Refer to “MAP” on page 234.

### 2

Open the Memory Map dialog box using the Memory map command from the dScope Setup menu.

As your target program runs, dScope uses the memory map to verify your program does not access invalid memory areas. For each memory range, you may specify the access method: **Read**, **Write**, or **Execute**. You may also specify that a particular memory range is **von Neumann**.



## Overview

dScope uses symbolic information in your target program to automatically setup the memory map for most applications. In cases where this is not adequate, you may re-specify the memory areas your program uses in the Memory Map dialog box.

---

### NOTE

*If your program uses memory-mapped I/O devices or dynamically accesses memory through pointers, you may need to make changes to the memory map.*

---

When dScope loads, the following memory maps are defined.

CPU	Address Range	Access
<b>MCS<sup>®</sup> 51</b>	0x000000-0x00FFFF ( <b>DATA</b> )	<b>READ WRITE</b>
	0x010000-0x01FFFF ( <b>XDATA</b> )	<b>READ WRITE</b>
	0xFF0000-0xFFFFFFFF ( <b>CODE</b> )	<b>EXEC READ</b>
<b>MCS<sup>®</sup> 251</b>	0x000000-0x00FFFF ( <b>DATA</b> )	<b>READ WRITE</b>
	0x010000-0x01FFFF ( <b>XDATA</b> )	<b>READ WRITE</b>
	0xFF0000-0xFFFFFFFF ( <b>CODE</b> )	<b>EXEC READ</b>
<b>80166 &amp; 80167</b>	0x000000-0x00FFFF	<b>EXEC READ WRITE</b>

dScope supports up to 16MB of memory. This memory is divided into 256 segments of 64K each. The default MCS<sup>®</sup> 51 and MCS<sup>®</sup> 251 memory spaces are assigned by dScope to the segments with the numbers listed in the following table.

Segment Value	Memory Space
<b>0x00</b>	MCS <sup>®</sup> 51 <b>DATA</b> segment, 0x00:0x0000—0x00:0x00FF. MCS <sup>®</sup> 251 <b>EDATA</b> segment, 0x00:0x0000-0x00:0xFFFF.
<b>0x01</b>	MCS <sup>®</sup> 51 and MCS <sup>®</sup> 251 <b>XDATA</b> segment 0x01:0x0000-0x01:0xFFFF.
<b>0x80-0x9F</b>	MCS <sup>®</sup> 51 and MCS <sup>®</sup> 251 Code Bank 0 through Code Bank 31. 0x80 ~ Code Bank 0, 0x81 ~ Code Bank 1, etc.
<b>0xFE</b>	MCS <sup>®</sup> 51 and MCS <sup>®</sup> 251 <b>PDATA</b> segment 0xFE:0x0000-0xFE:0x00FF.
<b>0xFF</b>	MCS <sup>®</sup> 51 and MCS <sup>®</sup> 251 <b>CODE</b> segment 0xFF:0x0000-0xFF:0xFFFF.

For 166 derivatives, memory is viewed as a linear array of bytes. No segment identifiers like X: and D: are used.

Although dScope supports up to 16MB of target program memory, only the memory ranges required should be mapped. dScope requires two copies of each block allocated in the memory map. One copy holds the data used for reading, writing, and execution. Another copy holds the specific attributes such as access permissions and information for code coverage and performance analysis. For this reason, mapping huge amounts of memory may slow down the execution speed of dScope since disk swapping may be required.

## Controls

The following controls are available in the Memory Map dialog box.

### Current Mapped Ranges

The Current Mapped Ranges list box shows the currently defined memory map ranges. You may select one of the ranges to display the access methods in the Read, Write, Execute, and von Neumann check boxes.

The addresses shown correspond to the way dScope maps logical segments to physical segments. For example, 0x00xxxx represents an address in **DATA**, **IDATA**, or **EDATA** space; 0x01xxxx represents an address in **XDATA** space; and 0xFFxxxx represents an address in **CODE** space. Note that no such scheme is used for 166 derivatives.

### Kill Selected

The Kill Selected button removes the selected memory range from the memory map.

### Address Range

The Address Range input line shows the memory range. The memory range may include a starting and ending address, in which case every byte in the range is mapped as specified by the **Read**, **Write**, **Execute**, and **von Neumann** check boxes, or it may include only a single byte address, in which case the byte is mapped as specified.

### Read, Write, and Execute

The **Read**, **Write**, and **Execute** check boxes specify how a memory range may be accessed. If the **Read** check box is checked, the memory range may be read. If the **Write** check box is checked, the memory range may be written. If the **Execute** check box is checked, code from the memory range may be executed.

---

**NOTE**

*Memory ranges that contain executable code should be assigned both read and execute permission since MCS<sup>®</sup> 51 and MCS<sup>®</sup> 251 architectures allow read access to program code.*

---

**von Neumann**

The **von Neumann** check box identifies the specified memory range as von Neumann memory. When specified, dScope overlaps the external data memory (**XDATA**) range and code memory (0xFFxxx). Write accesses to external data memory also change code memory.

**2**

---

**NOTES**

*von Neumann memory ranges may not lie in the code area nor may they cross a 64K boundary.*

*von Neumann memory is supported in the **XDATA** space of the MCS<sup>®</sup> 51 and MCS<sup>®</sup> 251 architectures only. It is not available for the 166.*

*von Neumann memory ranges must have both **Read** and **Write** access.*

---

**Map Range**

The Map Range button adds the defined memory range to the memory map.

**Help**

The Help button opens a help window with help information about the Memory Map dialog box.

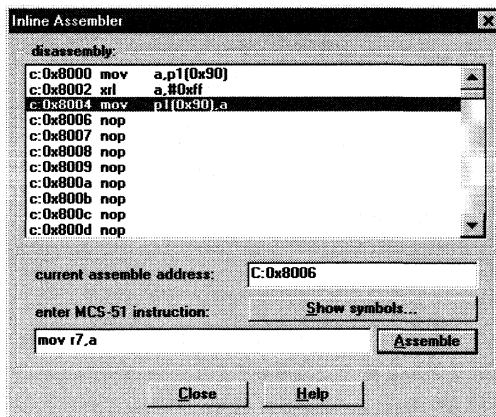
**Close**

The Close button closes the Memory Map dialog box.

## Inline Assembler Dialog Box

You may use the Inline Assembler dialog box to enter and assemble instructions directly into code memory. You use the inline assembler when you need to patch your target program without going through the process of editing the source files and recompiling and relinking the entire program.

Open the Inline Assembler dialog box using the Assemble button in the Debug window.



The following controls are available in the Inline Assembler dialog box.

### Disassembly

This list box displays assembly instructions starting from the address of the focus line in the Debug window. You may use the scroll bar to scroll through the disassembly. Double-click the mouse on an instruction to change the current assemble address to that of the selected line.

### Current Assemble Address

This input line shows the address where the next entered instruction is stored. You may change the address by typing the new address followed by the **Enter** key. You may also double-click the mouse on an instruction in the disassembly box to change the address.

---

#### NOTE

*You must press **Enter** in the Current Assemble Address input line to change the address where subsequent instructions are assembled.*

---

### Show Symbols

This button opens the Symbol Browser window. Refer to “Symbol Browser Window” on page 56 for more information.

## Instruction

Enter assembly instructions in the Instruction input line. Valid instructions depend on the CPU driver. For example, if the **80251S.DLL** driver is loaded, instructions for the MCS<sup>®</sup> 251 microcontroller are accepted. If the **8052.DLL** driver is loaded, instructions for the MCS<sup>®</sup> 51 microcontroller are accepted.

Operands for assembly instructions may be registers, numeric constants, line numbers, or other symbol names. For example, **JMP MEASURE210** or **MOV *index*,A**. You may drag symbol names from the Symbol Browser window and drop them into the Instruction input line. Refer to “Dragging Symbols” on page 60 for more information.

**2**

---

### NOTE

*Be cautious when specifying target addresses for **call** and **jmp** instructions for the MCS<sup>®</sup> 251 microcontrollers. For example, when the **80251S.DLL** driver is loaded, **SJMP 0** jumps to address 0x000000 which may be out of range and is probably not the intended target address. Instead, use **SJMP 0xFF0000** or **SJMP C:0** since these addresses correctly specify the proper code segment.*

---

When you have completed entry of an assembly instruction, click on the Assemble button. If dScope detects a problem with the instruction, an error message displays in the Command window.

## Assemble

The Assemble button assembles an instruction and inserts the instruction op-codes into program memory at the specified address. Afterwards, the current assemble address is updated.

## Help

The Help button opens a help window with help information about the Inline Assembler dialog box.

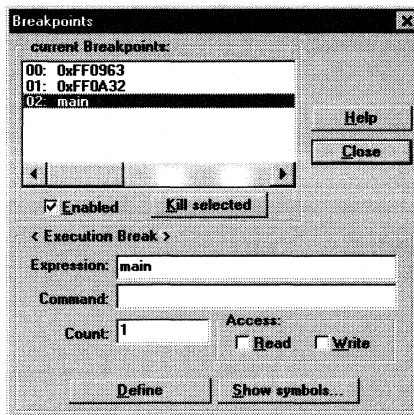
## Close

The Close button closes the Inline Assembler dialog box.

## Breakpoints Dialog Box

The Breakpoints dialog box helps you manage and maintain breakpoints for your target program. You use breakpoints to halt execution of your target program when a certain instruction is reached, test condition is met, or when a specific address is read or written from. Refer to “Chapter 6. Using Breakpoints” on page 135 for more information.

Open the Breakpoints dialog box using the Breakpoints command from the dScope Setup menu.



The following controls are available in the Breakpoints dialog box.

### Current Breakpoints

This listbox displays currently defined breakpoints. You may select one of the breakpoints to display specific information about the breakpoint definition. The breakpoint type (<Execution Break>, <Conditional Break> or <Access Break>) appears above the Expression input line when a breakpoint is selected.

Each breakpoint listed is preceded by an index that is assigned by dScope when the breakpoint is created. You use the index when you access specific breakpoints using dScope commands.

### Enabled

The Enabled check box indicates whether or not the selected breakpoint is enabled. You can change the state of the selected breakpoint using this check box. Refer to “BREAKDISABLE” on page 200 and “BREAKENABLE” on page 201 for more information about disabling and enabling breakpoints.



## Kill Selected

The Kill Selected button removes the selected breakpoint. This button is disabled when no breakpoint is selected. Refer to “BREAKKILL” on page 202.

## Expression

Enter the breakpoint expression in the Expression input line. You may set a breakpoint on a line of code, a memory access (read, write, or both), or an expression.

**2**

## Command

Enter a command to execute when the breakpoint occurs in the Command input line. If you want dScope to halt program execution, leave this input line blank. You may alternatively enter a dScope command in this input line. When the breakpoint occurs, dScope executes the command and resumes executing your target program. The command you specify here may be a user or signal function.

## Count

You may enter a count for each breakpoint in the Count input line. The count specifies the number of times the breakpoint expression is true before the breakpoint is triggered. For example, if you set a breakpoint on the first instruction of a function and if you set the count to 2, program execution halts when the function is executed the second time.

## Read

The Read check box specifies that the breakpoint occurs when the breakpoint expression is read. You may use this check box in combination with the Write check box.

## Write

The Write check box specifies that the breakpoint occurs when the breakpoint expression is written. You may use this check box in combination with the Read check box.

## Define

Use the Define button to define a breakpoint when you have entered an expression and optional command, count, and access method. Refer to “BREAKSET” on page 204.

## Show Symbols

This button opens the Symbol Browser window. You can drag and drop symbols from this window into the Expression input line. Refer to “Symbol Browser Window” on page 56 for more information.

## Help

The Help button opens a help window with help information about the Breakpoints dialog box.

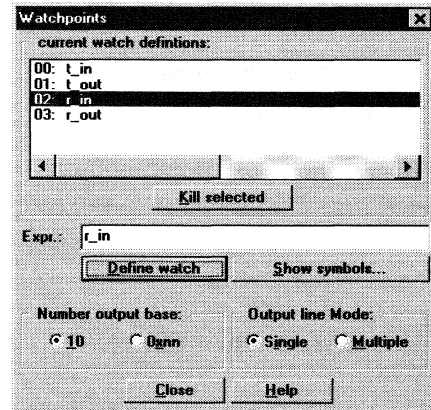
## Close

The Close button closes the Breakpoints dialog box.

## Watchpoints Dialog Box

The Watchpoints dialog box helps you manage and maintain watchpoints for your target program. Watchpoints are expressions with values that you want to watch at all times. They are displayed in the Watch window. Refer to “Watch Window” on page 44.

Open the Watchpoints dialog box using the Watchpoints command from the dScope Setup menu to access the following controls.



2

### Current Watch Definitions

The Current Watch Definitions list box shows the currently defined watchpoints. You may select one of the watchpoints to display specific information such as the expression, output base, and output line mode.

Each watchpoint listed is preceded by an index that is assigned by dScope when the watchpoint is created. You use the index when you access specific watchpoints using dScope commands.

#### NOTE

*Watchpoints for large data objects may considerably reduce the execution speed of dScope. This is due to the amount of information that must be processed to update the Watch window. Watching only the necessary parts of arrays and structures will ensure fast execution speeds.*

### Kill Selected

The Kill Selected button removes the selected watchpoint. This button is disabled when no watchpoint is selected. Refer to “WATCHKILL” on page 259.

### Expr

Enter the expression to watch in the Expr input line. You may watch variables, data addresses, complex expressions, structures, unions, and arrays.

## Define Watch

This button defines a watchpoint. Refer to “WATCHSET” on page 260.

## Show Symbols

This button opens the Symbol Browser window. You can drag and drop symbols from this window into the Expr input line. Refer to “Symbol Browser Window” on page 56 for more information.

2

## Number Output Base

The Number Output Base radio buttons determine the base of the value displayed in the Watch window. The output number base can be either decimal (10) or HEX (0xnn). The number base does not apply to floating-point values. These are always displayed using the floating-point output format.

## Output Line Mode

The Output Line Mode radio buttons let you select whether or not arrays and structures display on a single line or on multiple lines. Scalar expressions are always output in single line mode. Aggregate types such as arrays, structures, or unions may display on a single line or on multiple lines. In single line mode, aggregates display on one line (up to 128 characters). In multiple line mode, each component (array or structure element) displays on a separate line.

## Help

The Help button opens a help window with help information about the Watchpoints dialog box.

## Close

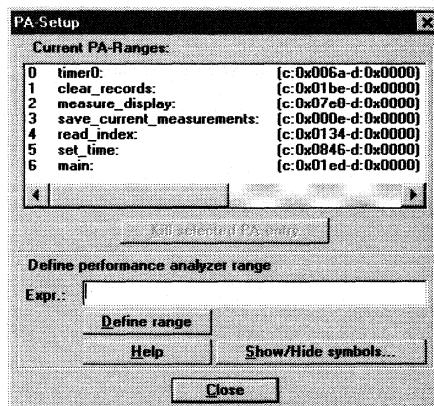
The Close button closes the Watchpoints dialog box.

## Performance Analyzer Setup Dialog Box

The PA-Setup dialog box helps you manage and maintain performance analyzer code ranges for your target program. Open this dialog box using the Setup Performance Analyzer command from the dScope Setup menu.

The performance analyzer in dScope lets you monitor the amount of time spent executing up to 255 ranges of code you specify in your target program. Results of performance analysis are constantly updated in the Performance Analyzer window. Refer to “Performance Analyzer Window” on page 49.

The following controls are available in the PA-Setup dialog box.



2

### Current PA-Ranges

The Current PA-Ranges list box displays the currently defined performance analyzer code ranges. You may select a code range and remove it using the Kill Selected PA Entry button.

Each code range listed is preceded by an index that is assigned by dScope when created. You use the index when you access specific code ranges using dScope commands. Refer to “Performance Analyzer” on page 240 for more information about the performance analyzer commands available.

### Kill Selected PA Entry

The Kill Selected button removes the selected code range. This button is disabled when no code range is selected.

## Expr

Enter the code range or function name to add to the performance analyzer in the Expr input line. A code range consists of a starting address and an ending address. A function name consists of the function name only. dScope automatically determines the ending address when the function name is specified.

## 2

## Define Range

Use the Define Range button to add a new code range to the performance analyzer.

## Show/Hide Symbols

This button opens and closes the Symbol Browser window. You can drag and drop symbols from this window into the Expr input line. Refer to “Symbol Browser Window” on page 56 for more information.

## Help

The Help button opens a help window with help information about the PA-Setup dialog box.

## Close

The Close button closes the PA-Setup dialog box.

## Chapter 3. Expressions

Most dScope commands accept numeric expressions as parameters. A numeric expression is a number or a complex expressions that contains numbers, debug objects, or operands.

Input lines entered in the Command window that do not contain commands are interpreted as expressions. For example, entering the following input line:

```
>R7                                /* Display value of R7 */
0x05                                /* R7 = 0x05 */
```

displays the value of R7.

The following input line:

```
>R7 = ACC
```

assigns the value in ACC to R7.

You may use the same operators in dScope that are available in the C programming language.

## Components of an Expression

An expression may consist of any of the following components.

Component	Description
<b>Bit Addresses</b>	Bit addresses reference bit-addressable data memory.
<b>Constants</b>	Constants are fixed numeric values or character strings.
<b>Line Numbers</b>	Line numbers reference code addresses of executable programs. When you compile or assemble a program, the compiler and assembler include line number information in the generated object module.
<b>Memory Spaces</b>	Memory spaces let you associate an expression with a physical address space of the MCS <sup>®</sup> 51 and MCS <sup>®</sup> 251 microcontroller architectures. dScope-166 does not require memory space prefixes.
<b>Operators</b>	Operators include +, -, *, and /. Operators may be used to combine subexpressions into a single expression. In dScope, you may use all operators that are available in the C programming language.
<b>Program Variables (Symbols)</b>	Program variables are those variables in your target program. They are often called symbols or symbolic names.

Component	Description
System Variables	System variables predefined dScope variables that alter or affect the way dScope operates.
Type Specifications	Type specifications let you specify the data type of an expression or subexpression.

Each of these components is described in the following sections.

## Constants

dScope accepts decimal constants, HEX constants, octal constants, binary constants, floating-point constants, character constants, and string constants.

### Binary, Decimal, HEX, and Octal Constants

By default, dScope assumes numeric constants to be decimal or base ten numbers. When you enter 10, dScope assumes this is the number ten and not the HEX value 10h. The following table shows the prefixes and suffixes that are required to enter constants in base 2 (binary), base 8 (octal), base 10 (decimal), and base 16 (HEX).

Base	Prefix	Suffix	Example
Binary:	None	Y or y	1111111Y
Decimal:	None	T or none	1234T or 1234
Hexadecimal:	0x or 0X	H or h	1234H or 0x1234
Octal:	None	Q, q, O, or o	777q or 777Q or 777o

Following are a few points to note about numeric constants.

- Numbers may be grouped with the dollar sign character (“\$”) to make them easier to read. For example, 1111\$1111Y is the same as 11111111Y.
- HEX constants must begin prefixed with a leading zero when the first digit in the constant is A-F.
- By default, numeric constants are 16-bit values. They may be followed with an L to make them long, 32-bit values. For example, 0x1234L, 1234L, and 1255HL.
- When a number is entered that is larger than the range of a 16-bit integer (0-65535), dScope automatically promotes the number to a 32-bit integer. This avoids truncating significant digits.



## Floating-Point Constants

Floating-point constants are used to enter floating-point numbers. They may be specified in one of the following formats:

*number . number*

*number e[+|-] number*

*number . number [e[+|-] number]*

where:

*number* is any valid decimal number. For example, 4.12, 0.1e3, and 12.12e-5.

### NOTE

*In contrast with the C programming language, floating-point numbers in dScope must have a digit before the decimal point. For example, .12 is not allowed. It must be entered as 0.12.*

3

## Character Constants

The rules of the C programming language for character constants apply to dScope. For example, the following are all valid character constants.

```
'a', '1', '\n', '\v', '\x0FE', '\015'
```

dScope also supports escape sequences. The following table lists the escape sequences that are supported.

Escape Sequence	Description
\\	Backslash character ("").
\"	Double quote.
'	Single quote.
\a	Alert, bell.
\b	Backspace.
\f	Form feed.
\n	Newline.
\r	Carriage return.
\t	Tab.
\0nn	Octal constant.
\Xnnn	HEX constant.

## String Constants

The rules of the C programming language for string constants also apply to dScope. For example:

```
"string\x007\n"
"value of %s = %04XH\n"
```

are both valid string constants. Embedded escape sequences are fully supported.

Nested strings may be required in some instances. For example, defining a Toolbox button. The double quotes for the nested string must be escaped. For example:

```
"printf (\"hello world!\n\")"
```

### NOTE

*In contrast with the C programming language, successive strings are not concatenated into a single string. For example, "string1+" "string2" is not combined into a single string.*

## System Variables

System variables control or indicate how a specific feature of dScope operates. You may display the value of a system variable by simply entering its name in the Command window. For example:

```
>$                                /* Display the current PC ($) system variable
*/
0xFF0000                          /* Output in the Command window */
```

Many system variables can be changed by assigning them a new value in the Command window. For example:

```
>$ = 0xFF0100                      /* Change PC ($) to 0xFF0100 */
```

System variables may be used anywhere a program variable or other expression is used. System variables may be entered in either upper or lower case. The following table lists the available system variables, the data types, and their uses.

Variable	Type	Description
<b>\$</b>	<b>unsigned long</b>	<p>\$ represents the current address stored in the program counter. The program counter is displayed in the Register window. You may use \$ to display and change the program counter. For example,</p> <p>\$ = 0x4000</p> <p>sets the program counter to address 0x4000.</p>
<b>_break_</b>	<b>unsigned int</b>	<p>The <b>_break_</b> system variable let you stop executing the target program. When you set <b>_break_</b> to a non-zero value, dScope halts target program execution. You may use this variable in user and signal functions to halt program execution. Refer to "Chapter 10. Functions" on page 261 for more information.</p>
<b>_framesize_</b>	<b>unsigned int</b>	<p>The <b>_framesize_</b> system variable reflects the number of bytes saved on the stack for interrupts. A value of 0 indicates that 2 bytes are saved (the 16-bit program counter). A value of 1 indicates that 4 bytes are saved (the 24-bit program counter and PSW1).</p> <p>If the target program was created using the Keil A251 assembler, C251 compiler, and L251 linker, dScope automatically detects the CPU operating mode and sets the <b>_framesize_</b> system variable. You may change the number of bytes saved on the stack by setting <b>_framesize_</b> to the desired value.</p> <p>When an interrupt occurs, MCS® 251 microcontrollers save PSW1 and the 24-bit return address on the stack. Since 251 microcontrollers have a 16MB address space, it is important to save all 24 bits of the program counter.</p> <p><b>NOTE:</b> <b>_framesize_</b> is relevant only on the 251.</p>
<b>_iip_</b>	<b>unsigned char</b>	<p>The <b>_iip_</b> system variable indicates the number of interrupts that are currently nested. Your user and signal functions may use this system variable to determine if an interrupt is being processed.</p> <p><b>NOTE:</b> <b>_iip_</b> is not available with dScope-166.</p>
<b>_mode_</b>	<b>unsigned int</b>	<p>The <b>_mode_</b> system variable indicates whether the MCS® 251 CPU simulator is operating in binary mode or source mode. When <b>_mode_</b> has a value of 0, the CPU is operating in binary mode. When <b>_mode_</b> has a value of 1, the CPU is operating in source mode.</p> <p>If the target program was created using the Keil A251 assembler, C251 compiler, and L251 linker, dScope automatically detects the CPU operating mode and sets the <b>_mode_</b> system variable.</p> <p><b>NOTE:</b> <b>_mode_</b> is relevant only for the 251.</p>

Variable	Type	Description
<b>cycles</b>	<b>unsigned long</b>	<p>The <b>cycles</b> system variable contains the current value of the CPU instruction cycle counter. This counter starts counting from 0 when your target program begins execution and increases for each instruction that is executed. Refer to the documentation for your chip to determine the number of CPU cycles required for each instruction.</p> <p>The CPU instruction cycle counter displays in the Register window during program execution. You can use this counter to determine the amount of time required for a function or code block.</p> <p><b>NOTE:</b> <i>cycles</i> is a read-only variable.</p>
<b>itrace</b>	<b>unsigned int</b>	<p>The <b>itrace</b> system variable indicates whether or not trace recording is performed during target program execution. When <b>itrace</b> has a value of 0, no trace recording is performed. When <b>itrace</b> has a non-zero value, trace information is recorded. Refer to “Trace Recording” on page 34 for more information.</p>
<b>radix</b>	<b>unsigned int</b>	<p>The <b>radix</b> system variable determines the base used for numeric values displayed in the Command window. Value values for <b>radix</b> are 10 and 16. The default setting is 16 for HEX output.</p>

## 3

## CPU Driver Symbols

dScope automatically defines a number of symbols when you load a CPU driver. The symbols are defined in the particular CPU driver DLL. There are two types of symbols that are defined: special function registers (SFRs) and CPU pin registers (VTREGs).

### Special Function Registers (SFRs)

Each CPU driver supports the complete set of special function registers for the simulated microcontroller. For example, the **8051.DLL** CPU driver supports special function registers for P0 to P3, Timer 0 and 1, and the Serial Port. The **8052.DLL** CPU driver supports all these plus the registers for timer 2 (T2CON, TL2, TH2).

Special function registers are defined as public symbols that reside in the internal data and bit space of the CPU. They each have an associated address and may be used in any expression.

Refer to “Appendix A. CPU Driver Files” on page 295 for a complete list of the CPU drivers that are supported.

## CPU Pin Registers (VTREGs)

CPU pin registers, or VTREGs, are symbols that are defined to let you use the CPU's simulated pins for input and output. VTREGs are not public symbols nor do they reside in a memory space of the CPU. They may be used in expressions, but their values and utilization are CPU dependent. VTREGs provide a way to specify signals coming into the CPU from a simulated piece of hardware.

After loading a CPU driver, you can list of the available VTREGs using the **DIR** command. Refer to "DIR" on page 211. Additionally, "Appendix A. CPU Driver Files" on page 295 includes a complete list of the CPU drivers and the VTREGs that each supports.

The following examples show how VTREGs may be used to aid in simulating your target program. In most cases, you use VTREGs in signal functions to simulate some part of your target hardware.

### Crystal Frequency

Most of the CPU drivers define the **XTAL** VTREG which contains the crystal frequency. The value of **XTAL** is used to calculate timing for a number of on-chip peripherals including the baudrate for the on-chip serial port. Additionally, dScope uses this value to display the number of seconds of execution in the Register window. You may change the value of **XTAL** to any crystal frequency you desire. For example:

```
XTAL = 12000000      /* Set xtal frequency to 12MHz */  
XTAL = 33000000      /* Set xtal frequency to 33MHz */  
XTAL = 11059200      /* Set xtal frequency to 11.0592MHz */
```

### I/O Ports

The **8051.DLL** CPU driver includes a VTREG for each 8-bit I/O port: **PORT0**, **PORT1**, **PORT2**, and **PORT3**. Do not confuse these VTREGs with the SFRs for each port (**P0**, **P1**, **P2**, and **P3**). The SFRs have an address inside the 8051's internal data memory. The VTREGs are the signals present on the 8051's pins.

With dScope, it is easy to simulate input from external hardware. If you have a pulse train coming into a port pin, you can use a signal function to simulate the signal. For example, the following signal function inputs a square wave on P1.1 with a frequency of 1000Hz.

```
signal void one_thou_hz (void) {
    while (1) {                                /* repeat forever */
        PORT1 |= (1 << 1);                      /* set P1.1 */
        twatch ((XTAL / 12) / 2000);           /* delay for .0005 secs */
        PORT1 &= ~(1 << 1);                    /* clear P1.1 */
        twatch ((XTAL / 12) / 2000);           /* delay for .0005 secs */
    }                                           /* repeat */
}
```

The following command starts this signal function:

```
one_thou_hz ()
```

3

Refer to “Chapter 10. Functions” on page 261 for more information about dScope’s user and signal functions.

Simulating external hardware that responds to output from a port pin is only slightly more difficult. Two steps are required. First, write a dScope user function or signal to perform the desired operations. Second, create a breakpoint that invokes the user function.

Suppose you use an output pin (P1.0) to enable or disable an LED. The following signal function uses the **PORT1** VTREG to check the output from the CPU and display a message in the Command window.

```
signal void check_p10 (void) {
    twatch (1);                                /* Delay 1 cycle for port set */
    if (PORT1 & (1 << 0)) {                    /* Test P1.0 */
        printf ("LED is ON\n"); }              /* 1? LED is ON */
    else {                                     /* 0? LED is OFF */
        printf ("LED is OFF\n"); }
    }
}
```

Now, you must add a breakpoint for writes to port 1. The following command line adds a breakpoint for all writes to P1.

```
BS WRITE P1, 1, "check_p10 ()"
```

Now, whenever your target program writes to P1, the check\_P10 signal function prints the current status of the LED. Refer to “BREAKSET” on page 204 for more information about setting breakpoints.



## Serial Ports

Several CPU drivers include VTREGs for the on-chip serial port: **STIME**, **SIN**, and **SOUT**. **SIN** and **SOUT** are the serial input and output pins on the CPU.

The **STIME** VTREG lets you specify whether the serial port timing is instantaneous (**STIME** = 0) or whether the serial port timing is relative to the specified baudrate (**STIME** = 1). When **STIME** is 1, serial data displayed in the Serial window is output at the specified baudrate. When **STIME** is 0, serial data is displayed in the Serial window much more quickly.

Simulating serial input is just as easy as simulating digital input. Suppose you have an external serial device that inputs specific data periodically (every second). You can create a signal function that feeds the data into the CPU's serial port.

```
signal void serial_input (void) {
    while (1) {                                /* repeat forever */
        twatch (XTAL / 12);                    /* Delay for 1 second */

        SIN = 'A';                             /* Send first character */
        twatch ((XTAL / 12) / 900);            /* Delay for 1 character time */
                                                /* 900 is good for 9600 baud */
        SIN = 'B';                             /* Send next character */
        twatch ((XTAL / 12) / 900);
        SIN = 'C';                             /* Send final character */
                                                /* repeat */
    }
}
```

When this signal function runs, it delays for 1 second, inputs 'A', 'B', and 'C' into the serial input line and repeats.

Serial output is simulated in a similar fashion using a user or signal function and a write access breakpoint as described above.

---

### NOTE

*Some CPU drivers support chips with 2 or more on-chip serial ports. In these cases, the serial port VTREGs are often named **S0IN** and **S0OUT** and **S1IN** and **S1OUT**. The examples above apply to these CPU drivers as well.*

---

## Program Variables (Symbols)

dScope lets you access variables, or symbols, in your target program by simply typing their name. Variable names, or symbol names, represent numeric values and addresses. They are stored in the object module as debug information generated by the assembler or compiler. Symbols make the debugging process easier by allowing you to use the same names in the debugger as you use in your program.

### Module Names

A module name is the name of an object module that makes up all or part of a target program. Source-level debugging information as well as symbolic information is stored in each module.

The module name is derived from the name of the source file (for C and assembly source files) or it is manually specified (for PL/M-51 source files).

If the target program consists of a source file named **MCOMMAND.C** and the C51, C251, or C166 compiler generates an object file called **MCOMMAND.OBJ**, the module name is **MCOMMAND**. Module names are constructed similarly for the A51, A251, and A166 assemblers.

For modules compiled with the Intel PL/M-51 compiler, the module name is constructed differently. PL/M-51 source files begin with an identifier followed by a colon. This identifier specifies the name of the module. For example:

```
MY_MODULE: DO;                /* The first statement in a PLM module */
                                /* other PLM statements                */
END;                          /* each PLM module is terminated by END; */
```

If the specified module name (**MY\_MODULE** in this case) does not match the name of the source file, dScope cannot locate the source code for the PL/M-51 module and source-level debugging is not possible. Refer to “**SETMODULE**” on page 251 for information on manually specifying the source for PL/M-51 modules.



## Symbol Information

When you load a target program into dScope, a table of module names and the symbols defined in each module is created. The symbols include local variables (declared in functions in the module), the functions declared in the modules, and the line number information for the module.

You must specify that debug information is included in the object module created when you compile and assemble your source files. Without debug information, dScope cannot perform source-level and symbolic debugging.

Refer to “Preparing Programs for dScope” on page 3 for specific details on how to compile and assemble your programs for debugging.

## Symbol Naming Conventions

The following conventions apply to symbols in dScope.

- dScope ignores the case of symbols. The symbol name **SYMBOL** is equivalent to **Symbol**.
- Symbols may be a maximum of 31 characters long.
- The first character of a symbol name must be one of the following: ‘A’-‘Z’, ‘a’-‘z’, ‘\_’, or ‘?’.
- Subsequent characters in a symbol name may be any of the following: ‘A’-‘Z’, ‘a’-‘z’, ‘0’-‘9’, ‘\_’, or ‘?’.

---

### NOTE

*When using the ternary operator (“?:”) in dScope with a symbol that begins with a question mark (“?”), you must insert a space between the ternary operator and the symbol name. For example, **r5 = acc ? ?symbol : r7**.*

---

## Symbol Classification

dScope recognizes two classes of symbols:

- **Program Symbols** are those symbols that are defined in the target program object module. When you load a program to debug, dScope automatically loads the program symbols from the object module you specify.
- **Reserved Words** are those symbols that are predefined by dScope. Commands, command options, data type names, register names, system variables, CPU symbols, and VTREGs are all reserved words.

Commands and command options are reserved words. Refer to “Chapter 9. Commands” on page 193 for a complete listing of the commands available in dScope.

Names of data types are reserved words. For example: **BIT**, **BYTE**, **CHAR**, **DWORD**, **FLOAT**, **INT**, **LONG**, **PTR**, **REAL**, **UCHAR**, **UINT**, **ULONG**, **VOID**, and **WORD** are all reserved words.

Register names and other assembly names are reserved symbols. For example: **R0-R7**<sup>†</sup>, **R0-R15**<sup>‡§</sup>, **RL0-RH7**<sup>§</sup>, **A** (accumulator)<sup>†‡</sup>, **C** (carry)<sup>†</sup>, **AB** (for **MUL** and **DIV**)<sup>†‡</sup>, **WR0-WR30**<sup>‡</sup>, **DR0-DR60**<sup>‡</sup>.

System variables are reserved words. Refer to “System Variables” on page 88 for more information about these symbols.

CPU symbols are reserved words that are defined when a CPU driver is loaded. They are the special function registers of the particular chip derivative.

VTREGs are reserved words. These symbols depend on the CPU driver that you load. Refer to “CPU Driver Symbols” on page 90 for more information.

<sup>†</sup> Indicates symbols used with the MCS<sup>®</sup> 51 family of microcontrollers.

<sup>‡</sup> Indicates symbols used with the MCS<sup>®</sup> 251 family of microcontrollers.

<sup>§</sup> Indicates symbols used with the C166 family of microcontrollers.

## Literal Symbols

Often you may find that your program symbols duplicate reserved words in dScope. When this occurs, you must literalize your program symbols using the back quote character ("`) to help dScope differentiate them from reserved words.

For example, if you define a variable named R5 in your program and you attempt to access it in dScope, you will actually access the R5 register. To access the R5 variable, you must prefix the variable name with the back quote character.

### Accessing the R5 Register

```
>R5 = 121
```

### Accessing the R5 Variable

```
>`R5 = 212
```

Normally, dScope searches for reserved words then for target program symbols. When you literalize a symbol, dScope only searches for target program symbols.

## Fully Qualified Symbols

Symbols may be entered using the only name of the variable or function they reference. Symbols may also be entered using a fully qualified name that includes the name of the module and name of the function in which the symbol is defined.

A fully qualified symbol name may include any of the following components:

- **Module Name** which identifies the module where a symbol is defined.
- **Line Number** which identifies the address of the code generated for a particular line in the module.
- **Function Name** which identifies the function in a module where a local symbol is defined.
- **Symbol Name** which identifies the name of the symbol.

You may combine these four names as shown in the following table to create a fully qualified symbol name.

Symbol Components	Description
<b>\ModuleName\LineNumber</b>	This fully qualified symbol references the address of the code generated for line number <b>LineNumber</b> in <b>ModuleName</b> .
<b>\ModuleName\FunctionName</b>	This fully qualified symbol references the address of the <b>FunctionName</b> function in <b>ModuleName</b> .
<b>\ModuleName\SymbolName</b>	This fully qualified symbol references the address of the global symbol <b>SymbolName</b> in <b>ModuleName</b> .
<b>\ModuleName\FunctionName\SymbolName</b>	This fully qualified symbol references the address of the local symbol <b>SymbolName</b> in the <b>FunctionName</b> function in <b>ModuleName</b> .

Several examples of fully qualified symbol names are shown in the following table.

Full Qualified Symbol Name	Description
<b>\MEASURE\clear_records\idx</b>	Identifies the local symbol <b>idx</b> in the <b>clear_records</b> function in the <b>MEASURE</b> module.
<b>\MEASURE\MAIN\cmdbuf</b>	Identifies the <b>cmdbuf</b> local symbol in the <b>MAIN</b> function in the <b>MEASURE</b> module.
<b>\MEASURE\sindx</b>	Identifies the <b>sindx</b> symbol in the <b>MEASURE</b> module.
<b>\MEASURE225</b>	Identifies line number 225 in the <b>MEASURE</b> module.
<b>\MCOMMAND\82</b>	Identifies line number 82 in the <b>MCOMMAND</b> module.
<b>\MEASURE\TIMER0</b>	Identifies the <b>TIMER0</b> symbol in the <b>MEASURE</b> module. This symbol may be a function or a global variable.

Non-Qualified Symbols

When you enter a fully qualified symbol name, dScope quickly determines if the symbol exists and reports an error if it does not. For symbols that are not fully qualified, dScope must search a number of tables to locate the symbol.

dScope searches for non-qualified symbols in the following order until a matching symbol name is found. If no matching symbol is found, dScope reports an error that the symbol or line was not found.

1. Local Variables in the Current Function

dScope searches the local symbols of the current function in the target

program. The current function is determined by the value of the program counter.

**2. Global or Static Variables in the Current Module**

dScope searches the table of symbols in the current module. As with the current function, the current module is determined by the value of the program counter. Symbols in the current module represent variables that were declared in the module but outside a function. This includes file-scope or static variables.

**3. Symbols Created using the DEFINE Command**

dScope searches the table symbols created by the **DEFINE** command. These symbols are created in dScope and are not a part of the target program. Refer to “DEFINE” on page 209 for more information.

**4. System Variables**

dScope searches the table of system variables. These symbols are by dScope to provide a way to monitor and change debugger characteristics. They are not a part of the target program. Refer to “System Variables” on page 88 for more information. If a global variable in your target program shares the same name as a system variable, you may access the global variable using a literal symbol name. Refer to “Literal Symbols” on page 97 for more information.

**5. Global or Public Symbols**

dScope searches the table of global or public symbols for your target program. SFRs defined by the CPU driver are considered to be public symbols and are also searched.

**6. CPU Driver Symbols (VTREGs)**

dScope searches the VTREG symbols defined by the CPU driver. Refer to “Appendix A. CPU Driver Files” on page 295 for a complete list of the VTREG symbols defined by each CPU driver.

---

**NOTE**

*The search order for symbols changes when creating user or signal functions in dScope. dScope first searches the table of symbols defined in the user or signal function. Then, the above list is searched. Refer to “Chapter 10. Functions” on page 261 for more information about user and signal functions.*

---

## Line Numbers

Line number information is produced by the compiler to enable source-level debugging. The line number information includes the line number in the source or listing module and the physical address of the associated program code. dScope uses this information with the value of the program counter to display the source code for the current assembler instruction.

Since a line number represents a code address, dScope lets you use in an expression. The syntax for a line number is shown in the following table.

Line Number Symbol	Description
<code>\LineNumber</code>	This symbol references the address of the code generated for line number <b>LineNumber</b> in the current module. If this line number does not exist, dScope generates an error.
<code>\ModuleName\LineNumber</code>	This fully qualified symbol references the address of the code generated for line number <b>LineNumber</b> in <b>ModuleName</b> . If this line number does not exist, dScope generates an error.

### Example

```
\measure\108          /* Line 108 in module "MEASURE" */
\143                  /* Line 143 in the current module */
```

## Bit Addresses

Bit addresses represent bits in the bit-addressable memory. This included bits in special function registers. You may specify a bit address as follows:

*expression . bit\_position*

where:

*expression* is a numeric value that corresponds to a valid bit-addressable byte. For MCS<sup>®</sup> 51 microcontrollers, this is an address from 0x20-0x2F or 0x80, 0x88, 0x90, 0x98, 0xA0, ... 0xF8. For MCS<sup>®</sup> 251 microcontrollers, this is an address from 0x20-0xFF.

*bit\_position* is a numeric value from 0-7.

### Example

```
20H.0          /* Bit 0 of the 8051 */
0x2F.7         /* Bit 127 of the 8051 */
ACC.0          /* Bit 0 of the 51/251 accumulator (0xE0) */
0xFD00.15      /* 80166 bit address */
```

## Memory Spaces

The MCS<sup>®</sup> 51 and MCS<sup>®</sup> 251 microcontrollers provide a number of different memory areas where variables and program code may be located.

dScope provides a number of prefixes you must use with expressions to access a specific memory area. Prefixes must be separated from the address with a colon (“:”). The prefixes available are listed in the following table.

Prefix	Memory Space	Description
<b>B</b>	<b>BIT</b>	Bit-addressable RAM. (MCS <sup>®</sup> 51 and MCS <sup>®</sup> 251)
<b>C</b>	<b>CODE</b>	Code Memory. (MCS <sup>®</sup> 51 and MCS <sup>®</sup> 251)
<b>CO</b>	<b>CONST</b>	Memory range for constants. (MCS <sup>®</sup> 251)
<b>D</b>	<b>DATA</b>	Internal, directly-addressable RAM. (MCS <sup>®</sup> 51 and MCS <sup>®</sup> 251)
<b>EB</b>	<b>EBIT</b>	Extended bit-addressable RAM. (MCS <sup>®</sup> 251)
<b>ED</b>	<b>EDATA</b>	Extended data RAM. (MCS <sup>®</sup> 251)
<b>HC</b>	<b>HCONST</b>	Huge memory range for constants. (MCS <sup>®</sup> 251)
<b>I</b>	<b>IDATA</b>	Internal, indirectly-addressable RAM. (MCS <sup>®</sup> 51 and MCS <sup>®</sup> 251)
<b>P</b>	<b>VTREG</b>	Peripheral memory of the CPU driver. (MCS <sup>®</sup> 51 and MCS <sup>®</sup> 251)
<i>Note that when you use this prefix for VTREG symbols you must specify the symbol name rather than an address. For example, <b>P:PORT1</b>. Refer to “Appendix A. CPU Driver Files” on page 295 for a complete list of VTREG symbols available for each CPU driver.</i>		
<b>X</b>	<b>XDATA</b>	External RAM. (MCS <sup>®</sup> 51 and MCS <sup>®</sup> 251)

### NOTE

*These prefixes are not necessary with symbols since symbolic names typically have an associated memory space.*

Memory space prefixes may also be numeric constants from 0x00-0xFF. These numeric constants reference a segment.



The relationship between the segment number and the memory area is shown in the following table.

Segment Value	Memory Space
<b>0x00</b>	MCS <sup>®</sup> 51 <b>DATA</b> segment, 0x00:0x0000—0x00:0x00FF. MCS <sup>®</sup> 251 <b>EDATA</b> segment, 0x00:0x0000-0x00:0xFFFF.
<b>0x01</b>	MCS <sup>®</sup> 51 and MCS <sup>®</sup> 251 <b>XDATA</b> segment 0x01:0x0000-0x01:0xFFFF.
<b>0x80-0x9F</b>	MCS <sup>®</sup> 51 and MCS <sup>®</sup> 251 Code Bank 0 through Code Bank 31. 0x80 ≈ Code Bank 0, 0x81 ≈ Code Bank 1, etc.
<b>0xFE</b>	MCS <sup>®</sup> 51 and MCS <sup>®</sup> 251 <b>PDATA</b> segment 0xFE:0x0000-0xFE:0x00FF.
<b>0xFF</b>	MCS <sup>®</sup> 51 and MCS <sup>®</sup> 251 <b>CODE</b> segment 0xFF:0x0000-0xFF:0xFFFF.

### Example

```

C:0x100          /* Address 0x100 in code memory */
I:100            /* Address 0x64 in internal RAM of the 8051 */
X:0FFFFH         /* Address 0xFFFF in the external data memory */
B:0x7F           /* Bit address 127 or 2FH.7 */
0xFF:0x0100      /* Equivalent to C:0x100 */
0x01:0x1234      /* Equivalent to X:0x1234 */
0x00:0x20        /* Equivalent to D:0x20 or I:0x20 */

```

## Type Specifications

dScope automatically performs implicit type casting in an expression. You may explicitly cast expressions to specific data types. Type casting in dScope following the conventions used in the C programming language. The major difference is that data types in dScope are simplified to make debugging easier. For example, in dScope **uint** corresponds to **unsigned int** in C.

The following types may be used in dScope:

Type	Description
<b>BIT</b>	A single bit ( <b>bit</b> ).
<b>BYTE</b>	An unsigned character ( <b>unsigned char</b> ).
<b>CHAR</b>	A signed character ( <b>signed char</b> ).
<b>DWORD</b>	An unsigned long integer ( <b>unsigned long</b> ).
<b>FLOAT</b>	A floating-point number ( <b>float</b> ).
<b>INT</b>	A signed integer ( <b>signed int</b> ).
<b>LONG</b>	A signed long integer ( <b>signed long</b> ).
<b>PTR</b>	A generic pointer (*).
<b>REAL</b>	A floating-point number ( <b>float</b> ).
<b>UCHAR</b>	An unsigned character ( <b>unsigned char</b> ).
<b>UINT</b>	An unsigned integer ( <b>unsigned int</b> ).
<b>ULONG</b>	A signed long integer ( <b>unsigned long</b> ).
<b>WORD</b>	An unsigned integer ( <b>unsigned int</b> ).

### Example

```
(float) 0x100          /* Produces 256 from type float */
float(c:0x100)         /* Displays value from code address 0x100 */
float(c:0x100)=3.1     /* Assigns 3.1 to code address 0x100 */
float(c:0x100)         /* Displays 3.1 */
ulong(c:0x100)         /* Displays 0x3F8CCD */
```

# Operators

dScope supports all operators of the C programming language. The operators have the same meaning as their C equivalents unless otherwise noted.

## Primary Operators

Operator	Description
<b><i>const</i></b>	Numeric constant. For example, 1, 34, 1.03. Refer to "Constants" on page 86 for more information.
<b><i>string_const</i></b>	String constant. For example, "ds51\n". Refer to "String Constants" on page 88 for more information.
<b><i>system_variable</i></b>	System variable symbol. For example, CYCLES, _IIP_. Refer to "System Variables" on page 88 for more information.
<b><i>register_name</i></b>	Register name. For example, R0, R7, A, C.
<b><i>type(mspace:const)</i></b>	Memory reference. For example, int (x:0x12). Refer to "Memory Spaces" on page 102 for more information.
<b><i>mspace primary</i></b>	Memory-specific address. For example, C:0x1000. Refer to "Memory Spaces" on page 102 for more information.
<b><i>pspace identifier</i></b>	Peripheral-specific address. For example, P:port5, P:ain0. Refer to "Memory Spaces" on page 102 for more information.
<b><i>id (expr)</i></b>	dScope function invocation. For example, printf("end\n"). Refer to "Chapter 10. Functions" on page 261 for more information.
<b><i>id</i></b>	Non-qualified symbol. For example, myvar. Refer to "Non-Qualified Symbols" on page 98 for more information.
<b><i>\modfunc\id</i></b>	Fully qualified symbol. For example, \MEASURE\myfunc\myvar. Refer to "Fully Qualified Symbols" on page 97 for more information.
<b><i>\modid</i></b>	Fully qualified global symbol. For example, \MEASURE\counter. Refer to "Fully Qualified Symbols" on page 97 for more information.
<b><i>\const</i></b>	Line number. For example, \123. Refer to "Line Numbers" on page 100 for more information.
<b><i>\modconst</i></b>	Fully qualified line number. For example, \MEASURE\123. Refer to "Line Numbers" on page 100 for more information.
<b><i>( expr )</i></b>	Parentheses. For example, (ACC + 5).

## Postfix Operators

Operator	Description
<i>expr++</i>	Post-increment operator. For example, ACC++.
<i>expr--</i>	Post-decrement operator. For example, ACC--.
<i>expr.id</i>	Structure member operator. For example, interval.msec.
<i>expr-&gt;id</i>	Structure pointer operator. For example, pint->msec.
<i>expr.const</i>	Bit address operator. For example, ACC.7, 0x20.5.
<i>expr [expr]</i>	Array index operator. For example, save_record [1].

## Unary Operators

3

Operator	Description
<i>++expr</i>	Pre-increment operator. For example, ++ACC.
<i>--expr</i>	Pre-decrement operator. For example, --ACC.
<i>&amp;expr</i>	Address operator. For example, &ACC, &R0.
<i>*expr</i>	Pointer operator. . For example, *read_index.buffer.
<i>+expr</i>	Unary plus operator. For example, +ACC.
<i>-expr</i>	Unary minus operator. For example, -0x10, -R7.
<i>~expr</i>	One's complement operator.. For example, ~ACC, ~R4.
<i>!expr</i>	Logical negation operator. For example, !R0.
<i>sizeof (expr)</i>	Size of operator. For example, sizeof(R5).
<i>sizeof (type)</i>	Size of operator. For example, sizeof (float).

## Arithmetic Operators

Operator	Description
<i>expr + expr</i>	Addition operator. For example, CYCLES + R5.
<i>expr - expr</i>	Subtraction operator. For example, CYCLES - R7.
<i>expr * expr</i>	Multiplication operator. For example, ACC * R5.
<i>expr / expr</i>	Division operator. For example, CYCLES / 2.
<i>expr % expr</i>	Modulo operator. For example, ACC % 7.

## Relational and Logical Operators

Operator	Description
<i>expr &lt; expr</i>	Less than operator. For example, R5 < R7.
<i>expr &gt; expr</i>	Greater than operator. For example, R5 > R0.
<i>expr &gt;= expr</i>	Greater or equal operator. For example, DPTR >= 200.
<i>expr &lt;= expr</i>	Less or equal operator. For example, DPL < DPH.
<i>expr == expr</i>	Equality operator. For example, DPL == (DPH + ACC).
<i>expr != expr</i>	Inequality operator. For example, DPL != ACC.
<i>expr &amp;&amp; expr</i>	Logical AND operator. For example, ACC==3 && !R7.
<i>expr    expr</i>	Logical OR operator. For example, ACC==3    R5==R2.

## Bitwise Operators

Operator	Description
<i>expr &gt;&gt; expr</i>	Right shift operator. For example, ACC >> 3.
<i>expr &lt;&lt; expr</i>	Left shift operator. For example, ACC << 3.
<i>expr &amp; expr</i>	Bitwise AND operator. For example, ACC & 7.
<i>expr ^ expr</i>	Bitwise XOR operator. For example, DPH ^ ACC.
<i>expr   expr</i>	Bitwise OR operator. For example, DPTR   0x10.

## Assignment Operators

Operator	Description
=	Simple assignment. For example, ACC = R7.
+=	Addition assignment. For example, R5 += 5.
-=	Subtraction assignment. For example, ACC -= R3.
*=	Multiplication assignment. For example, DPL *= 4.
/=	Division assignment. For example, ACC /= 2.
%=	Modulo assignment. For example, DPH %= 7.
<<=	Left shift assignment. For example, ACC <<= 1.
>>=	Right shift assignment. For example, R5 >>= ACC.
&=	Bitwise AND assignment. For example, R5 &= 1.
=	Bitwise OR assignment. For example, ACC  = 2.
^=	Bitwise XOR assignment. For example, DPL ^= ACC.

Other Operators

Operator	Description
<i>(type) expr</i>	Type conversion. For example, (long) charval.
<i>expr, expr</i>	Comma operator. For example, R5, R4, -1.
<i>expr ? expr : expr</i>	Conditional operator. For example, ACC==1 ? R5 : R7.

Address Expressions

Many expressions you enter in dScope are simply resolved as numeric values. Address expressions are those expressions which are resolved into addresses in the memory space of the target CPU.

Ambiguous Addresses with dScope Commands

Many dScope commands require that you enter an address or address range. For example, the **DISPLAY** command lets you specify a starting address for memory contents to display. Refer to “DISPLAY” on page 216.

The following command:

```
D 0x80 /* Display memory command */
```

specifies an ambiguous address: 0x80. This could mean **CODE** address 0x80, **XDATA** address 0x80, or **IDATA** address 0x80. The **DISPLAY** command actually displays data from address 0x000080.

The following command:

```
D C:0x100 /* Display memory command */
```

operates as expected because the starting address (0x100) as well as the memory space (**CODE**) are known. Refer to “Memory Spaces” on page 102.

Ambiguous Addresses with dScope Functions

Many dScope functions also require an address expression. The **memset** function (see page 271) initializes a specified memory area with a value. As with the **DISPLAY** command, it is possible to enter ambiguous addresses for the **memset** function invocation.



The following command:

```
memset (0, 0x1000, 5)
```

fills the memory range from 0x000000 to 0x001000 with the value 5. If the intention was to fill the **XDATA** memory range the address specified was incorrect.

The following command:

```
MEMSET (X:0, X:0x1000, 5)
```

correctly fills the **XDATA** memory space from 0x0000 to 0x1000 with the value 5.

## Symbols With Implicit Memory Spaces

Most expressions you use will include a symbol with an implicit memory space. For example:

```
D ACC
```

displays the contents of the accumulator (which is a predefined symbol of the 8051 that is in internal RAM at address 0xE0).

When an address expression is required and a symbol is found, the address of the symbol, not its contents, is used. For example:

```
D &ACC /* Address of accumulator */
```

also displays the contents of the accumulator.

---

### NOTE

*When an address expression is expected, dScope implicitly adds the & address operator. This prevents the contents of the accumulator from being used as the address in the prior example.*

---

## Differences Between dScope and C

There are a number of differences between expressions in dScope and expressions in the C programming language:

- dScope does not differentiate between uppercase and lowercase characters for symbolic names and command names.
- dScope does not support converting an expression to a typed pointer like **char \*** or **int \***. Pointer types are obtained from the symbol information in the target program. They cannot be created.
- Function calls entered in dScope's Command window refer to dScope functions. You cannot invoke functions in your target from the Command window. Refer to "Chapter 10. Functions" on page 261 for more information.
- dScope does not support structure assignments.
- dScope lets you reference the contents of a union or structure using a pointer and the **OBJECT** command. Refer to "OBJECT" on page 238. When you specify the \* pointer operator with a structure or union pointer, the **OBJECT** command displays each element in the structure or union. Without the \* pointer operator, the **OBJECT** command displays the address of the pointer. For example, if a structure is declared as follows:

```
struct time { char hour; char min; char sec; } time, *ptime;
```

The following dScope commands displays have the specified effect.

```
*ptime          /* Displays the pointer value of ptime */
OBJ *ptime      /* Displays the structure pointed to by ptime */
WS *ptime       /* Defines a watchpoint with structure output */
```



## Expression Examples

The following expressions were entered in dScope's Command window. All applicable output is included with each example. The 80517.DLL CPU driver and the 8051 MEASURE example program were used for all examples.

### Constant Examples

```
>0x1234                                /* Simple constant */
0x1234                                /* Output */
>EVAL 0x1234
4660T 11064Q 1234H '....4'          /* Output in several number bases */
```

### Register Examples

```
>ACC                                /* Interrogate value of the accum */
D:0xE0 = 0x00                        /* Address from ACC = 0xE0, mem type = D: */
>ACC = --R7                          /* Set accum and R7 equal to value R7-1 */
>ACC,R7                              /* Interrogation, comma exp. are valid */
D:0xE0 = 0xFF                        /* Output for ACC */
D:0x7 = 0xFF                         /* Output for R7 */
```

### dScope Function Invocation Examples

```
                                /* String constant within printf() */
>printf ("dScope is coming now!\n")
dScope is coming now!              /* Output */
```

### Function Symbol Examples

```
>main                                /* Get addr of main() from MEASURE.C */
C:0x01EF                             /* Reply, means C:0x01EF */

>&main                                /* Same as before */
0xFF01EF

>d main                              /* Display: address = main, mem type = C: */
C:0231      75 98 5A D2 DF 43 87-80 75 8C 06 75 8A 06 43 u.Z.C.u.u.C
C:0240      89 02 D2 8C D2 A9 D2 AF-12 01 F3 75 4C 05 75 4D .....uL.uM
C:0250      04 75 4E FD 12 14 17 75-4C 05 75 4D 04 75 4E 57 uN.uL.uM.uNW
C:0260      12 14 17 75 40 69 75 41-0F 12 0A D1 E4 F5 3D 74 .u@iuA...=t
```

### Address Utilization Examples

```
>main + 0x10                          /* Address calculation */
C:0x241

>uchar (C:0x1EF)                     /* Read byte from code addr 0x01EF */
0x75                                 /* Reply */
```

## Symbol Output Examples

```
>dir \measure\main                /* Output sym from main() as mod MEASURE */
FUNCTION:  MAIN                    /* Output */
I:0x000067H . . . . cmdbuf . . array[15] of char
D:0x00003CH . . . . i . . uchar
D:0x00003DH . . . . idx . . uint
```

## Program Counter Examples

```
>$ = main                        /* Set program counter to main() */
>dir                            /* points to local mem sym. from main() */
FUNCTION:  MAIN                    /* Output */
I:0x000067H . . . . cmdbuf . . array[15] of char
D:0x00003CH . . . . i . . uchar
D:0x00003DH . . . . idx . . uint
```

## 3

## Program Variable Examples

```
>cmdbuf                        /* Interrogate address from cmdbuf */
I:0x0067                      /* Output of addr due to aggr type (Array)*/
>cmdbuf[0]                    /* Output cont. of first array element */
0x00
>OBJ cmdbuf                   /* Output the contents of entire array */
"\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0"
>i                             /* Output contents from i */
0x00
>idx                          /* Output contents from idx */
0x0000
>idx = $                      /* Set contents from index equal to the PC */
>idx                          /* Output contents from idx */
0x01EF
```

## Line Number Examples

```
>\211                          /* Address of the line number #104 */
0xFF01EF                      /* Reply */
>main                          /* \211 is like main */
C:0x01EF
>
> BD                           /* BD bit literalized, else conflict with */
                                /* BreakpointDisable ! */
0                               /* Reply */
>PCON                          /* Output contents of the PCON register */
0x00
>\MCOMMAND#91                  /* A line number of module "MCOMMAND" */
0xFF08CB
>set_interval                  /* Output a value of the PUBLIC variable */
C:0x08CB
```

## Operator Examples

```
>--ACC                      /* Auto-decrement also for 8051 registers */
0xFE
>mdisplay                    /* Output a PUBLIC bit variable */
0
>mdisplay = 1                /* Change */
>mdisplay                    /* Check result */
1

>ACC == 0xFE ? R7 : $        /* Conditional operation */
0xFF                        /* Condition "ACC==0xFE" is true */
>--ACC
0xFD
>ACC == 0xFE ? R7 : $        /* Condition no longer true --ACC */
0x231
>ACC == 0xFD && R7 == 0xFF && $ == main /* Logical expression */
0x01                        /* 0x01 = TRUE, 0x00 = FALSE */
```

## Structure Examples

```
>save_record[0]              /* Address of a record */
X:0x4000
>obj save_record[0]          /* Output complete record contents */
{time={hour=0x00,min=0x00,sec=0x00,msec=0x00},
port4=0x00,port5=0x00,analog={0x00,0x00,0x00,0x00}}
>

>save_record[0].time.hour = R7 /* Change struct element of records
change */
>save_record[0].time.hour      /* Interrogation */
0xFF
>save_record[1].time.hour = save_record[0].time.hour
>save_record[0].time.hour, save_record[1].time.hour
0xFF
0xFF
```

## Array Examples

```
>menu                        /* Address of an array */
0x04FE
>
>menu [0], menu[1], menu[2], menu[3] /* Output array elements */
0x0A
0x2B
0x2A
0x2A
>
>ACC /= 2                    /* C short form for "ACCU = ACCU / 2" */
0x7E                        /* Output */
>ACC /= 2
0x3F
>ACC /= 2
0x1F>(!R0)                  /* Parenthesis , else DOS cmd! */
0x01
```

## Operator Examples

```
>DPTR--                                /* C short form for "DPTR = DPTR - 1" */
0x0
>++DPTR                                /* C short form for "DPTR = DPTR + 1" */
0xFFFF
>ACC << 3                             /* ACC SHL 3 */
0xF8
```

## dScope Function Invocation Examples

```
>printf ("RegBank = %d\n", (PSW & 0x18) >> 3)
RegBank = 0
>PSW |= 0x18                           /* Adjust register bank */
0x19
>printf ("RegBank = %d\n", (PSW & 0x18) >> 3)
RegBank = 3
```

# 3

## Fully Qualified Symbol Examples

```
>--\measure\main\idx                   /* Auto INC/DEC valid for qual symbol */
0x01EF
>++\measure\main\idx
0x01F0
>++\measure\main\idx
0x01F1
>A <= B + 2                            /* C short form for "A = A SHL (B+2)" */
0x7C
>intcycle
0x00
>intcycle |= ~1                        /* C short for */
0xFE                                  /* "intcycle = intcycle OR NOT 1" */
>intcycle
0xFE
```

## Structure Examples

```
>                                     /* example useful in DS51 function */
>interval.min = getint ("enter integer: ")
enter integer: 21
>interval.min                          /* Interrogation */
0x15
>obj interval                          /* Output structure contents */
{min=0x15,sec=0x00,msec=0x0}
>obj interval,10t                      /* Output struct contents, decimal base */
{min=21,sec=0,msec=0}

>                                     /* Change structure elements */
>interval.min = interval.sec = interval.msec = A / 2
>obj interval                          /* Control */
{min=0x3E,sec=0x3E,msec=0x3E}
```

## Chapter 4. Executing Code

Simulating the execution of your target program is one of fundamental features of a debugger like dScope. To provide you with the greatest flexibility in debugging, dScope provides a number of ways to run the code in your target program. You can easily start and stop program execution, step over instructions, step into and out of functions, and reset the target CPU.

The following commands are available for executing your target code.

Command	Action
<b>Ctrl+C</b>	Stops program execution.
<b>Esc</b>	Stops program execution.
<b>GO</b>	Starts program execution. dScope executes the target program until a breakpoint is reached.
<b>OSTEP</b>	Steps out of a function and returns to the calling function or procedure.
<b>PSTEP</b>	Single-steps over source or assembly instructions but does not step into procedures or functions.
<b>RESET</b>	Resets the CPU.
<b>TSTEP</b>	Single-steps over source or assembly instructions and into procedures and functions.

# 4

You may also use buttons and menu commands in the Debug window to perform these operations.

---

### NOTE

*In addition to its simulation capabilities, dScope also lets you debug code running on your target hardware. When you use MON51, MON251, MON166, or the Intel RISM51 or RISM251 in your target hardware, dScope lets you symbolically debug your program “in-circuit.” Almost all execution operations that are available while simulating are available while target debugging.*

---

## Starting a Program

dScope provides a number of ways for you to start executing your target program. Remember that unless you specifically setup breakpoints, dScope does not halt execution of your target programs—you must manually stop program execution.

Perform the following operations to start dScope running your target program.



### In the Command window...

- Enter the **GO** command (refer to “GO” on page 223).

### In the Debug window...

- Click the Go button on the dialog bar,
- Click the Go button on the toolbar,
- Select the Go! command on the menu.

When your target program is running, dScope display a status message in the Debug window status line that indicates your target program is running. Refer to “Status Bar” on page 28.

## Stopping a Program

# 4

There are a number of ways you can stop a target program that is running.

### In the Command window...

- Type **Ctrl+C** or **Esc** to halt program execution (refer to “Ctrl+C” on page 208).

### In the Debug window...

- Click the Stop button on the dialog bar,
- Click the Stop button on the toolbar,
- Select the Stop! command on the menu.

---

#### **NOTE**

*When you halt program execution, dScope displays the code that was executing in the Debug window. If you halt program execution inside a library function, you may use the **OSTEP** command to step out of each nested function and eventually return to your target program.*

*The **OSTEP** command is not available when debugging on a target using **MON51**, **MON251**, **MON166**, **RISM51**, or **RISM251**.*

---

## Restarting a Program

You may reset the target CPU, both under simulation and under target debugging with a monitor program. Resetting the CPU returns the state of the microcontroller and all SFRs, registers, and internal memory to that just after a power-on-reset. You may need to reset the CPU if you debug past a critical point and need to restart your debug session. Performing a reset has no effect on breakpoints, watchpoints, or user functions.

dScope provides a two ways for you to reset the CPU.

### In the Command window...

- Enter the **RESET** command (refer to “RESET” on page 244).

### In the Main window...

- Click the Reset button on the toolbar (refer to “Toolbar” on page 25).

---

#### **NOTE**

*If you reset the CPU while your target program is running, dScope immediately begins running your program starting at the reset vector. Halt program execution before performing a reset if you do not want dScope to immediately begin executing your target program after a reset.*

*The **RESET** command does not reset all CPU SFRs when debugging using MON51, MON251, MON166, RISM51, and RISM251. This is because dScope cannot determine the exact CPU derivative that is used. To guarantee that all SFRs are properly initialized, use the hardware reset on your target board.*

---

## Single-Stepping

Often it is not so important to merely start and stop program execution. Often you need to execute individual instructions and immediately ascertain their effects. Single-step operations let you step over one instruction at a time. You may choose to treat functions as a single entity (stepping over a function) or you may single-step into a function call (stepping into a function). If you single-step into a function that holds no interest for your debugging session, you may choose to step out of the function.

dScope shows the current instruction or line of code at the program counter. When you single-step, dScope executes the instruction or line, updates the registers, and stops. You may then examine registers, variable contents, or memory before continuing.

## Stepping Over a Function

### 4

When you single-step, dScope executes one instruction or line of code and waits for you to proceed with the next instruction. When the current instruction is a call instruction or when the current line of code invokes a function, you can select whether or not dScope single-steps into the function.

When dScope steps over a function, the function executes as a single entity and is not interrupted (unless a breakpoint occurs). Single-stepping over functions is useful for skipping library routines like **printf** and **strcpy** when you debug.

dScope provides several ways to single-step over a function.

### In the Command window...

- Enter the **PSTEP** command (refer to “PSTEP” on page 243).

### In the Main window...

- Click the StepOver button on the dialog bar,
- Click the StepOver button on the toolbar,
- Select the StepOver! command on the menu.



## Stepping Into a Function

When you single-step, dScope executes one instruction or line of code and waits for you to proceed with the next instruction. When the current instruction is a call instruction or when the current line of code invokes a function, you can select whether or not dScope single-steps into the function.

When dScope steps into a function, the name of the function or the function address are added to the call stack which you can view using the Call Stack window. You may step out of the function if you so choose (refer to “Stepping Out of a Function” on page 119).

dScope provides several ways to single-step into a function.

### In the Command window...

- Enter the **TSTEP** command (refer to “TSTEP” on page 257).

### In the Main window...

- Click the StepInto button on the dialog bar,
- Click the StepInto button on the toolbar,
- Select the StepInto! command on the menu.

## Stepping Out of a Function

When you find yourself stepping through a function in which you have no interest, you may direct dScope to step out of the function. In doing so, dScope runs your program until the next return instruction or return from interrupt instruction executes. dScope then displays the current instruction or line of code (which follows the function invocation).

dScope provides several ways to step out of a function.

### In the Command window...

- Enter the **OSTEP** command (refer to “OSTEP” on page 239).

### In the Main window...

- Click the StepOut button on the dialog bar,
- Click the StepOut button on the toolbar,
- Select the StepOut! command on the menu.

---

#### **NOTE**

*If you manually halt program execution, the program counter may be in the middle of a library routine or another uninteresting part of your program. You may step out of the function to return to the function invocation.*

*The **OSTEP** command is not available when debugging on a target using MON51, MON251, MON166, RISM51, or RISM251.*

---

## Chapter 5. Examining and Modifying Memory

The dScope debugger provides numerous methods of displaying and changing data like expressions, symbols, memory, CPU registers, SFRs, and VTREGs. The different ways of displaying data in dScope's windows and modifying memory, variables, and CPU registers are discussed in this chapter.

---

### NOTE

*Typically, you use expressions with the commands described in this chapter. The expression evaluator in dScope recognizes most of the expressions supported by the C programming language. Refer to "Chapter 3. Expressions" on page 85 for more information.*

---

You use dScope commands, dialog boxes, and predefined functions to display and change memory.

### Commands that Display and Change Memory

The following table lists the commands you can use to examine and display memory.

Command	Description
<b>ASM</b>	Stores assembler instructions in memory starting at the specified address.
<b>DISPLAY</b>	Displays memory from the specified memory area and address in the Memory window (if it is open) or in the Command window.
<b>ENTER</b>	Changes memory starting at the specified address.
<b>EVALUATE</b>	Displays the specified expression in decimal, octal, HEX, and ASCII format.
<b>OBJECT</b>	Displays structures, unions, and arrays in the Command window. You may also use this command to display scalar variables.
<b>UNASSEMBLE</b>	Displays code starting at the specified address in the Debug window.

## Predefined Functions that Display and Change Memory

The following table lists the predefined functions that you can use to display memory.

Function	Description
<b>bit ()</b>	Displays or sets the <b>bit</b> variable at the specified address.
<b>byte ()</b>	Displays or sets the <b>byte</b> variable ( <b>unsigned char</b> ) at the specified address.
<b>char ()</b>	Displays or sets the <b>char</b> variable at the specified address.
<b>double ()</b>	Displays or sets the <b>double</b> variable at the specified address.
<b>dword ()</b>	Displays or sets the <b>dword</b> variable ( <b>unsigned long</b> ) at the specified address.
<b>float ()</b>	Displays or sets the <b>float</b> variable at the specified address.
<b>int ()</b>	Displays or sets the <b>int</b> variable at the specified address.
<b>long ()</b>	Displays or sets the <b>long</b> variable at the specified address.
<b>printf ()</b>	Displays or sets the specified format string replacing format parameters with the specified arguments.
<b>ptr ()</b>	Displays or sets the <b>ptr</b> variable ( <b>void *</b> ) at the specified address.
<b>real ()</b>	Displays or sets the <b>real</b> variable ( <b>float</b> ) at the specified address.
<b>uchar ()</b>	Displays or sets the <b>uchar</b> variable ( <b>unsigned char</b> ) at the specified address.
<b>uint ()</b>	Displays or sets the <b>uint</b> variable ( <b>unsigned int</b> ) at the specified address.
<b>ulong ()</b>	Displays or sets the <b>ulong</b> variable ( <b>unsigned long</b> ) at the specified address.
<b>word ()</b>	Displays or sets the <b>word</b> variable ( <b>unsigned int</b> ) at the specified address.

# 5

The memory type functions; **bit**, **byte**, **char**, **double**, **dword**, **float**, **int**, **long**, **ptr**, **real**, **uchar**, **uint**, **ulong**, and **word**; let you display or set a data type at a specific memory address.

The **printf** predefined function lets you display symbols, variables, and memory contents.

The **memset** predefined function lets you fill an area of memory with a specified value.

## Windows

There are several windows in which dScope displays data. They are each listed, along with a description, in the following table.

Window	Description
<b>Watch Window</b>	dScope displays the values for the defined watchpoints in this window. Refer to "Chapter 7. Using Watchpoints" on page 153 for more information about defining and viewing watchpoints.
<b>Register Window</b>	dScope always displays the current CPU register contents in this window. The Register window is for display purposes only. Values in the window may not be directly manipulated. However, all registers may be changed from the Command window. Refer to "Register Window" on page 45 for more information.
<b>Memory Window</b>	dScope displays the results of the <b>DISPLAY</b> command in this window. You may use the scroll bars to rapidly scroll through memory. You may choose to update the Memory window during target program execution by selecting the Update Watch window command from dScope's Setup menu shown on the right. When selected, this option updates the Watch window every 1255 instructions. This lets you see memory change as your target program runs. Refer to "Watch Window" on page 44 for more information.
<b>Debug Window</b>	The Debug window shows program code in a source-level, assembly-level, or mixed source and assembly display. You may enter or modify the assembly code in your program from this window. Refer to "Debug Window" on page 27 for more information.
<b>Command Window</b>	You enter commands and view their results in the Command window. You use the Command window to change the memory area displayed in the Memory window, to adjust the values of CPU registers, and to change the value of variables in your target program.

The windows listed above display various data of your target program. While the Watch window, Register window, and Memory window only display information, the Debug window and the Command window are much more interactive.

## Examining Memory

The following commands and predefined functions display memory contents.

### Using the Command Window

You may simply type the name of a variable in the Command window to display its contents. The following example shows a simple C function.

```
void func (void)
{
    unsigned int x;                /* loop variable */
    for (x = 0; x < 100; x++)      /* count from 0 to 99 */
    {
        printf ("x = %d\n", (int) x); /* display x */
    }
}
```

To display the contents of the variable *x*, enter the following in the Command window.

```
>x
```

You may use the following command to display the address where *x* is stored.

```
>&x
```

5

---

#### NOTE

Variables that display in the Command window are formatted in HEX by default. You may use the **radix** system variable (see “System Variables” on page 88) to change the number base used.

---

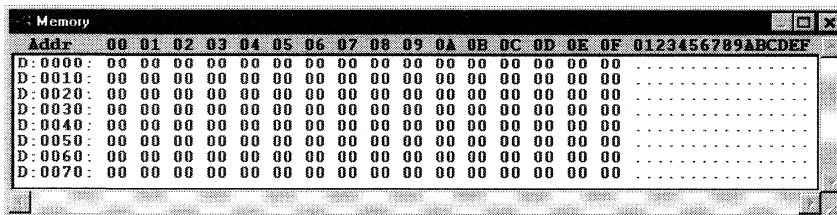
## Using the DISPLAY Command

The **DISPLAY** command lets you display a range of memory. Memory ranges display in HEX and ASCII. Display lines consists of the address of the first byte and up to 16 HEX bytes and ASCII characters. Dots (“.”) display for non-printing ASCII values.

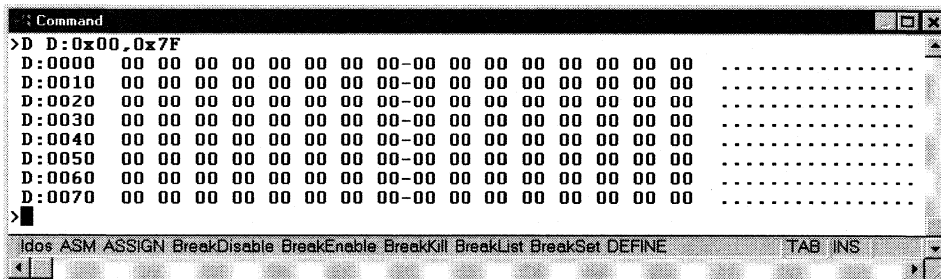
For example, the following command displays the contents of the internal data memory on the MCS<sup>®</sup> 51 microcontrollers.

```
>D D:0x00,0x7F
```

If the Memory window is open, the memory range displays there.



If the Memory window is not open, the range displays in the Command window.



Refer to “DISPLAY” on page 216 for more information.

## Using the EVALUATE Command

The **EVALUATE** command lets you display the result of an expression in decimal, octal, HEX, and ASCII.

For example, the following command evaluates the expression ‘-1’.

```
>eval -1  
16777215T 7777777Q 0xFFFFFFFF '....'
```

The following command evaluates the expression ‘intcycle’ which is a program variable.

```
>eval intcycle  
1T 1Q 0x01 '....'
```

Refer to “EVALUATE” on page 221 for more information.



## Using the OBJECT Command

The **OBJECT** command lets you display structures, unions, or arrays. You may also use the **OBJECT** command to display scalar variables.

You may optionally specify whether the output is formatted as decimal or HEX and whether or not the object displays on a single line or multiple lines.

For example, the following command displays the *current* structure in decimal on a single line.

```
>OBJ current,10 LINE
{time={hour=0,min=0,sec=0,msec=0},port4=0,port5=0, ...
```

The following command displays the *current* structure in HEX on multiple lines.

```
>OBJ current                                /* multi-line, radix 16 */
{
    time={                                  /* a nested structure */
        hour=0x00,                          /* the structure members ... */
        min=0x00,
        sec=0x00,
        msec=0x0000
    },
    port4=0x00                               /* a scalar */
    port5=0x00                               /* another scalar */
    analog=                                 /* an array ... */
        [0]: 0x00
        [1]: 0x00
        [2]: 0x00
        [3]: 0x00
}
```

Refer to “OBJECT” on page 238 for more information.

## Using the UNASSEMBLE Command

The **UNASSEMBLE** command disassembles code memory and displays it in the Debug window. Refer to “Debug Window” on page 27 for more information about the display modes and controls available.

For example, the following command disassembles code starting from C:0000h. This is the address of the reset vector in the MCS<sup>®</sup> 51 microcontroller family. The Debug window updates to reflect the disassembled code.

```
U C:0x0 /* Disassemble from address C:0x0000 */
```

The following command disassembles code starting from the *main* C function.

```
U main /* Disassemble starting at address "main" */
```

The following command disassembles code starting at the current program counter. Use this command to restore the Debug window to the current execution address after you have disassembled other parts of your program.

```
U $ /* Disassemble starting at the program counter */
```

Refer to “UNASSEMBLE” on page 258 for more information.

## Using Memory Type Functions

There are a number of standard operators you may use like functions: **bit**, **byte**, **char**, **double**, **dword**, **float**, **int**, **long**, **ptr**, **real**, **uchar**, **uint**, **ulong**, and **word**. You may use these operators to access a data type at a specific address.

The following command displays the unsigned character at address C:0000h.

```
>byte (c:0x0000)          /* display the character at C:0000h */
```

The following command displays the signed integer at address D:50h.

```
>int (d:0x50)              /* display the unsigned int at D:50h */
```

The following command displays the floating-point value at address X:1090h.

```
>float (x:0x1090)          /* display the float at X:1090h */
```

## Using the printf Function

The **printf** function works like the ANSI C library function but displays its results in the Command window. For example, the following command line output the current value of the variable *x*.

```
>printf ("x = %d\n", x)  
x = 25
```

## Modifying Memory

The following dialog box, commands, and predefined functions modify the contents of memory.

### Using the Command Window

You may enter an assignment in the Command window to assign a new value to a variable. The following example shows a simple C function.

```
void func (void)
{
    unsigned int x;                /* loop variable */
    for (x = 0; x < 100; x++)      /* count from 0 to 99 */
    {
        printf ("x = %d\n", (int) x); /* display x */
    }
}
```

You may change the value of *x* by assigning it a new value. For example:

```
>x = 10
```

# 5

---

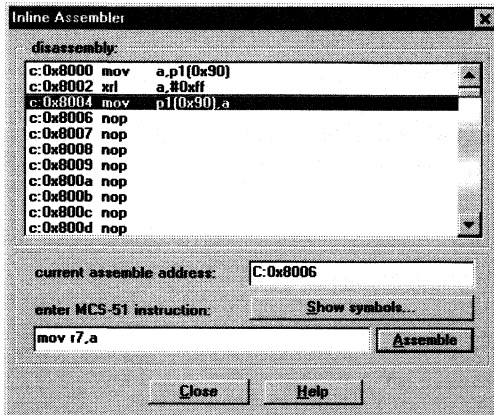
#### NOTE

*dScope is not case sensitive. Even though most of the examples in this manual refer to variables using uppercase letters, you may use uppercase or lowercase letters to reference symbols from your target program.*

---

## Using the Inline Assembler Dialog Box

The Inline Assembler dialog box lets you enter and assemble instructions directly into code memory.



As with the **ASM** command, you may use the Inline Assembler dialog box to correct mistakes or to make temporary changes to the target program you are debugging.

The following steps show you how to enter assembly instructions in the Inline Assembler dialog box.

1. Open the Inline Assembler dialog box using the Assemble button in the Debug window.
2. Enter the assembly address in the current assemble address input line and press **Enter**. You may also double-click the mouse on an instruction in the disassembly box to change the address.
3. Enter an assembly instruction in the Instruction input line.
4. Click the mouse on the Assemble button and dScope assembles the instruction and stores it in memory. If there is a problem with the instruction, dScope displays an error message in the Command window.

Refer to “Inline Assembler Dialog Box” on page 76 for more information.

## Using the ASM Command

The **ASM** command sets the address where assembler instructions are stored and lets you enter assembly instructions to store. When you enter an assembly instruction, dScope stores the resulting opcode in code memory. You may use the in-line assembler to correct mistakes or to make temporary changes to the target program you are debugging.

The following command sets the starting address for in-line assembly.

```
>ASM C:0x8000 /* set address to C:8000h (8051 and 251) */
```

The following commands store three **nop** instructions at addresses C:8000h, C:8001h, and C:8002h.

```
>ASM nop  
>ASM nop  
>ASM nop
```

Refer to “ASM” on page 197 for more information.

## Using the ENTER Command

The **ENTER** command lets you change the contents of memory starting at the specified address. When you use the **ENTER** command, you specify a data type (**bit**, **byte**, **char**, **dword**, **float**, **int**, **long**, **ptr**, **real**, **uchar**, **uint**, **ulong**, or **word**), an address, and an assignment to make. You may specify multiple expressions, separated by commas (“,”), which are converted into the specified data types and finally stored in consecutive addresses in memory.

For example, the following command stores 4 long values; 1, 2, 3, and 4; at X:1000h, X:1004h, X:1008h, and X:100Ch respectively

```
>E long X:0x1000 = 1,2,3,4
```

The following command stores the string “Buffer Overflow” in sequential bytes starting at X:8000h.

```
>E char X:0x8000 = "Buffer Overflow"
```

Refer to “ENTER” on page 219 for more information.

## Using Memory Type Functions

There are a number of standard operators you may use like functions: **bit**, **byte**, **char**, **double**, **dword**, **float**, **int**, **long**, **ptr**, **real**, **uchar**, **uint**, **ulong**, and **word**. You may use these operators to access a data type at a specific address and, contrary to C conventions, to assign a value of a specific type to a specific address.

The following command assigns the floating-point value 3.1415 to address X:1000h.

```
>float (x:0x1000) = 3.1415 /* assign 3.1415 float to X:1000h */
```

The following command assigns the unsigned long value 8051 to address D:24h.

```
>ulong (d:0x24) = 8051 /* assign the ulong value 8051 to D:24h */
```

## Using the memset Function

The **memset** function works like the ANSI C library function and fills a memory range with a specified value. Use the following command to fill the entire **XDATA** memory area on the 8051 with the value 5.

```
>MEMSET (X:0, X:0xFFFF, 5) /* Write XDATA RAM 0000 to FFFF with 5 */
```



## Chapter 6. Using Breakpoints

Breakpoints let you halt program execution or execute a dScope command when a particular code address, test condition is true, or when a specific address is read or written. Breakpoints are often the quickest way to locate a problem in your program.

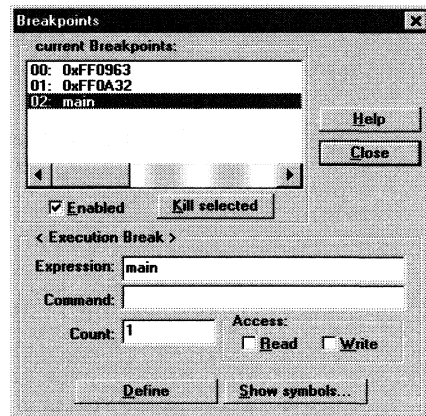
You may manipulate breakpoints using the Breakpoints Dialog Box described on page 78 as well as the following breakpoint commands:

Command	Action
<b>BREAKSET</b>	Described on page 204, this command defines and enables a breakpoint.
<b>BREAKDISABLE</b>	Described on page 200, this command disables a previously defined breakpoint.
<b>BREAKENABLE</b>	Described on page 201, this command enables a previously defined breakpoint.
<b>BREAKKILL</b>	Described on page 202, this command deletes a breakpoint.
<b>BREAKLIST</b>	Described on page 203, this command displays the list of breakpoints.

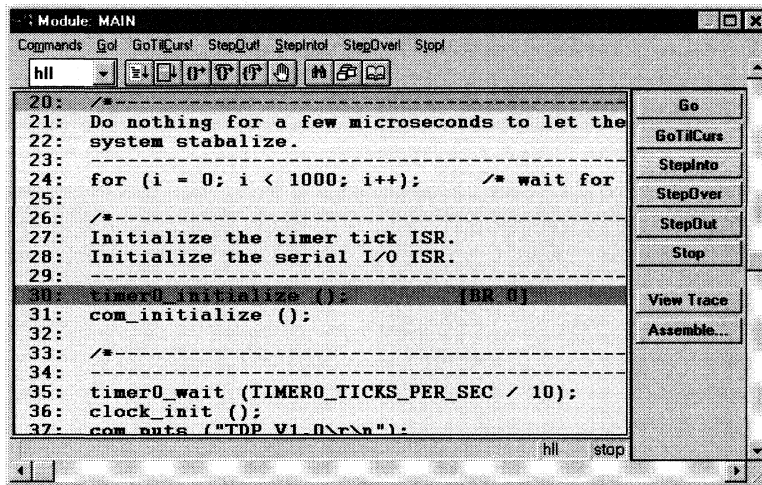
As you can see from the list of commands, there are a number of operations you can perform with breakpoints. The remainder of this chapter discusses each of these along with examples.

### Overview

dScope maintains an indexed table of defined breakpoints. As you define new breakpoints, dScope adds them to this table and increments the index number. dScope displays the index along with the breakpoint information in the Breakpoints dialog box and in the Command window (when you use the **BREAKLIST** command).



Additionally, the index appears for execution breakpoints in the Debug window as shown below.



When you are finished using a breakpoint, you can remove it using the **BREAKKILL** command. Alternatively, if you want to disable the breakpoint for a brief time and enable it again later, the **BREAKDISABLE** and **BREAKENABLE** commands are available.

### NOTE

*If you use a large number of breakpoints, you can create user functions to disable and enable breakpoints for a particular debugging session. You can even assign these functions to buttons in the Toolbox window for even more convenience. Refer to “Command Files” on page 43 and “Toolbox Window” on page 67 for more information.*

## 6

When a breakpoint is triggered, dScope can either halt program execution or can execute a command. Each breakpoint has an associated command string which, if specified, dScope executes. When a command is specified for a breakpoint, program execution is not automatically suspended.

If you wish to execute dScope commands and halt program execution, specify, for the command string, a user function which performs the desired commands and which sets the **\_break\_** system variable to 1. Refer to “System Variables” on page 88 for more information.

## Breakpoint Types

There are three types of breakpoints you can use with dScope: Execution Breakpoints, Conditional Breakpoints, and Access Breakpoints.

### Execution Breakpoints

An execution breakpoint is simply an address in your target program where execution stops or where a dScope command executes. Execution breakpoints are the easiest breakpoint to understand and use.

You may use execution breakpoints to check the control flow of your target program, to halt execution if a particular function is reached, or to rapidly skip over parts of your program that are already correct.

When you create an execution breakpoint, you must guarantee that the breakpoint address specifies an opcode address (the first byte of an instruction).

### Conditional Breakpoints

Conditional breakpoints are used in situations that cannot be described by execution breakpoints or access breakpoints. Conditional breakpoints include a test case (the condition) that is checked after each assembly instruction. If the test is true, program execution stops or a dScope command executes.

For example, if you have a variable *index* that gets corrupted during program execution, you can set a conditional breakpoint to test for cases when *index* is greater than 10 ( $\text{index} > 10$ ). After each assembly instruction executes, dScope tests the conditional expression ( $\text{index} > 10$ ). If it is true, the instruction just executed altered the variable *index* and the breakpoint occurs.

If the source of a problem is unknown, conditional breakpoints may be the only way to locate it.

---

#### NOTE

*Checking a conditional expression for each assembly instruction introduces a lot of overhead and slows down program execution speed considerably. However, this may be the only way to catch spurious program errors.*

---

## Access Breakpoints

Access breakpoints let you capture memory accesses to a specific address. Access breakpoints can be triggered by **READ** and/or **WRITE** accesses. Access breakpoints require that a memory address and a data type are specified.

You may use access breakpoints to monitor a number of program operations like:

- **Stack Space.** A **WRITE** access breakpoint may be set deep in the stack to warn of an impending stack overflow condition.
- **Buffer Overflow.** A **WRITE** access breakpoint may be set near the end of a large buffer to warn you when the buffer is nearly full.
- **Null Pointer Access.** A **READ** access breakpoint may be set at address 0x0000 to warn you when your program reads the value at a NULL pointer.
- **Null Pointer Assignment.** A **WRITE** access breakpoint may be set at address 0x0000 to warn you when your program writes to a NULL pointer.
- **Memory-mapped I/O.** A **WRITE** access breakpoint may be set for accesses to a memory-mapped I/O address. When the I/O address is written, the access breakpoint can be used to execute a user function that simulates the I/O device. Refer to “CPU Pin Registers (VTREGs)” on page 91 for more information.

## Breakpoint Notes

There are a number of points you should consider when using breakpoints.

- Execution breakpoints do not slow the execution of your target program.
- You may specify only one execution breakpoint for a particular address. Multiple breakpoints at the same address are not allowed.
- Conditional breakpoints slow the execution of your target program since they must test an expression after each assembly instruction executions. Reduce the number of conditional breakpoints that are enabled, or disable them entirely to increase execution speed.
- You may specify any valid dScope command to execute when a breakpoint is triggered. You may even change the program counter by assigning a new value to the dollar sign character (“\$”).
- Access breakpoints do not slow execution of your target program.
- Access breakpoints monitor the first byte of the object you specify in the breakpoint expression. For this reason, when you specify a **long int** object (32 bits) in the breakpoint expression, dScope triggers the breakpoint only when the first byte (in memory) of the object is accesses. Since 8051 variables are stored MSB first, this may not be the intended operation.
- When defining access breakpoints in dScope-166 for SFRs, you must specify the ampersand character (“&”) before the SFR symbol. For example,

```
BS WRITE &BUSCON
```

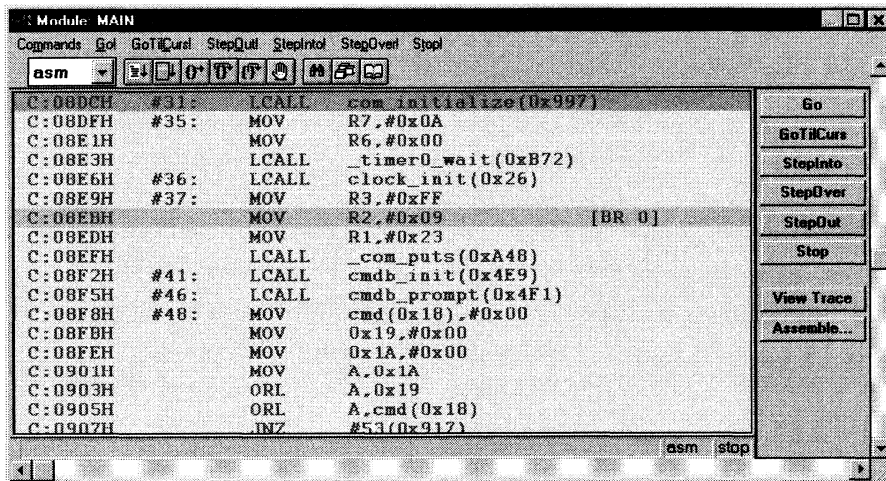
This restriction applies only to dScope-166.

## Setting Breakpoints

You may define breakpoints from the Command window using the **BREAKSET** command, from the Breakpoints dialog box using the provided controls, or from the Debug window using the mouse.

## Using the Debug Window

The Debug window displays the source and assembly code for your target program. You may set an execution breakpoint in the Debug window by double-clicking the left mouse button on the desired line in your program. If an execution breakpoint is already defined for the line it is removed.

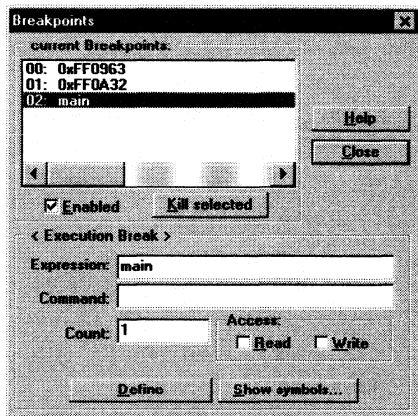


# 6

When an execution breakpoint is defined for a line displayed in the Debug window, the line is highlighted using the BP-Highlight color and a breakpoint label (**[BR n]** where **n** represents the breakpoint index) displays on the line.

## Using the Breakpoints Dialog Box

The Breakpoints dialog box lets you quickly define breakpoints.



The following steps show you how to define a breakpoint using the Breakpoints dialog box.

1. Open the Breakpoints dialog box using the Breakpoints command in the dScope Setup menu.
2. Enter the breakpoint expression in the Expression input line. The expression may be a code address (for execution breakpoints), a conditional expression (for conditional breakpoints), or a memory address (for access breakpoints).
3. Enter the dScope command to perform in the Command input line. If no command is entered, dScope halts program execution when the breakpoint is triggered. If a command is entered, dScope executes that command but does not halt program execution.
4. In the Count input line, enter the number of times the breakpoint condition is met before the breakpoint triggers. For example, if count is 2, the breakpoint is not triggered until the 2<sup>nd</sup> time the breakpoint expression is true. By default, the value of count is 1.
5. Check the Read or Write check boxes to specify the type of memory access for access breakpoints. If you define an access breakpoint, you must select one of these checkboxes.
6. Click the Define button.

## Rules for Access Breakpoints

Access breakpoints have two basic rules that apply to the breakpoint expression.

1. Expressions used in an access breakpoint must have a single, unique memory type. Expressions that include multiple objects are not allowed.
2. Only a few operators (&, &&, <, <=, >, >=, ==, and !=) are allowed in an access breakpoint expression. The expression left of the operator must adhere to rule number 1. The expression right of the operator may be of any type or complexity and is not required to adhere to rule number 1.

If an access breakpoint is defined that conforms to rule number 2, dScope uses the expression left of the operator for the memory address and data type. When an access to that address is detected, dScope then evaluates the expression right of the operator and triggers the interrupt if the result is true (assuming the breakpoint count is 1).

## Using the BREAKSET Command

The **BREAKSET** command lets you define execution breakpoints, conditional breakpoints, and access breakpoints. Each type of breakpoint is defined using the **BREAKSET** command in a slightly different manner, however, the basics for all breakpoints are the same. Refer to “Breakpoint Types” on page 137 for complete descriptions of each breakpoint type.

### Execution Breakpoints

Execution breakpoints are defined using the following command.

```
BS expression [, count] [, "command"]
```

where:

*expression* is the address of an instruction in your program. You may enter a qualified line number for the expression. Refer to “Fully Qualified Symbols” on page 97 for more information.

*count* is an expression that determines the number of times the instruction executes before dScope halts program execution or executes the specified *command*. The default value is 1.



*command* is a command that dScope executes when the breakpoint triggers. When no command is specified, dScope halts program execution when the breakpoint triggers. If the command requires quotation marks, you must escape them using the backslash character (“\”).

## Examples

The following command sets an execution breakpoint on the *main* function.

```
>BS main
```

The following command sets an execution breakpoint on line 133 of the *measure* program module. The breakpoint is triggered the 10<sup>th</sup> time line 133 executes.

```
>BS \measure\133, 10
```

The following example sets an execution breakpoint on the *sio\_isr* function. The breakpoint triggers when *sio\_isr* is invoked the second time (because the count is 2). When the breakpoint is triggered, dScope invokes the **printf** command and displays the message “sio\_isr received” in the Command window.

```
>BS sio_isr, 2, "printf (\"sio_isr received\\n\\n\")"
```

## Conditional Breakpoints

Conditional breakpoints are defined as follows:

```
BS expression [ , count ] [ , "command" ]
```

where:

*expression* is a conditional expression that dScope evaluates between each instruction execution.

*count* is an expression that determines the number of times the conditional *expression* is true before dScope halts program execution or executes the specified *command*. The default value is 1.

*command* is a command that dScope executes when the breakpoint triggers. When no command is specified, dScope halts program execution when the breakpoint triggers.

## Examples

The following command shows how to set a conditional breakpoint when a member of the *siobuf* array is set to 0xFF.

```
>BS siobuf [6] == 0xFF
```

The following breakpoint is triggered when the variable *index* is between 0x0A and 0x25 inclusive or when *index* has the value 0x81. When triggered, the breakpoint evaluates the value of *index* and displays the result in the Command window.

```
>BS (index >= 0x0A && index <= 0x25) || index == 0x81, 1, "eval index"
```

The following breakpoint halts program execution the 1000<sup>th</sup> time your target program enters the *timer0* function when *index* is greater than 5.

```
>BS $ == timer0 && index > 5, 1000
```

---

### NOTE

*Be careful to avoid changing the program counter in a breakpoint expression (for example, BS \$ = timer0) since dScope redraws the Debug window whenever the program counter changes. If the program counter is changed inside a breakpoint expression, the response time of the debugger is slowed greatly. If this occurs, enter the **BK \*** command in the Command window to remove all breakpoints and **Ctrl+C** to halt program execution.*

---

## Access Breakpoints

Memory access breakpoints are defined as follows:

```
BS {READ | WRITE | READWRITE} expression [, count] [, "command"]
```

where:

- |                  |  |
|------------------|--|
| <b>READ</b>      | indicates that the breakpoint triggers on READ accesses to the specified <i>expression</i> .           |
| <b>WRITE</b>     | indicates that the breakpoint triggers on WRITE accesses to the specified <i>expression</i> .          |
| <b>READWRITE</b> | indicates that the breakpoint triggers on READ and WRITE accesses to the specified <i>expression</i> . |

<i>expression</i>	is an expression that dScope evaluates during program execution.
<i>count</i>	is an expression that determines the number of times the <i>expression</i> is true before dScope halts program execution or executes the specified <i>command</i> . The default value is 1.
<i>command</i>	is a command that dScope executes when the breakpoint triggers. When no command is specified, dScope halts program execution when the breakpoint triggers.

Access breakpoints have two basic rules that apply to the breakpoint expression.

1. Expressions used in an access breakpoint must have a single, unique memory type. Expressions that include multiple objects are not allowed.
2. Only a few operators (&, &&, <, <=, >, >=, ==, and !=) are allowed in an access breakpoint expression. The expression left of the operator must adhere to rule number 1. The expression right of the operator may be of any type or complexity and is not required to adhere to rule number 1.

If an access breakpoint is defined that conforms to rule number 2, dScope uses the expression left of the operator for the memory address and data type. When an access to that address is detected, dScope then evaluates the expression right of the operator and triggers the interrupt if the result is true (assuming the breakpoint count is 1).

## Examples

The following command sets an access breakpoint for an byte in the stack. If this breakpoint is triggered, the stack may be too shallow for the target program to successfully operate.

```
>BS WRITE BYTE(D:0xF8)
```

The following command sets a breakpoint for WRITES to the character variable *sindex* when the value of *sindex* equals 0x133.

```
>BS WRITE sindex && sindex == 0x133
```

In the above example, if *sindex* was an **int** variable, the breakpoint may never be triggered. If *sindex* is an **int** variable it occupies two bytes in memory. In the MCS<sup>®</sup> 51 architecture, *sindex* is stored MSB first (so the high-order byte is at the

lowest address). The above breakpoint expression detects WRITES to the high-order byte but not to the low-order byte. To specify a breakpoint expression that detects WRITES to the low-order byte, you should use `&index + 1` as shown in the following example.

```
>BS WRITE &index+1 && index == 0x133
```

---

### NOTE

*Unlike C expressions where `&index + n` means address of `index` plus `n` times the size of `index`, dScope's address expressions do not scale the offset `n`. `&index+1` always targets the address of `index` plus one byte, regardless of the variable type. Using the C rules of scaling, it is not possible to address part of a memory object.*

---

The expression `&index + 1` now specifies the low-order byte of `index`. The reason for using the LSB in the breakpoint expression is that the low-order byte always gets written. The high-order byte may get written only if an overflow from the low-order byte occurs.

To specify the LSB of a 4-byte variable in the above example, add 3 to the base address of the variable as shown in the following command line.

```
>BS WRITE &index+3 && index == 0x133
```

---

### NOTE

*On occasion, you may want to trigger a breakpoint when a large data object (like a structure or array) is accessed. You may define an access breakpoint for each byte in the data object. However, this only makes sense for the smallest objects. A better method for detecting accesses in large objects is the **MAP** command. For example, use the following command to detect WRITES to the structure `current`.*

```
>MAP &current, &current + sizeof (current) - 1 READ
```

*This command maps the structure for read access. Any writes cause dScope to halt program execution and report an access violation.*

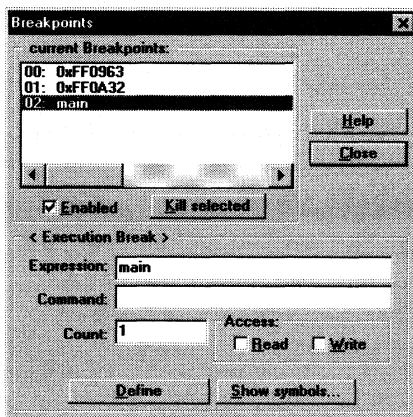
---

## Viewing Breakpoints

You may view the current breakpoints either from the Command window using the **BREAKIST** command, or from the Breakpoints dialog box.

## Using the Breakpoints Dialog Box

The Breakpoints dialog box displays a list of the currently defined breakpoints. You may use the scroll bars to scroll through the list.



To open the Breakpoints dialog box, select the Breakpoints command in the dScope Setup menu.

You may select a breakpoint to see the breakpoint expression, the command to execute when the breakpoint is triggered, the count, the access type (READ or WRITE), and whether or not the breakpoint is enabled.

## Using the BREAKLIST Command

The **BREAKLIST** command displays the list of currently defined breakpoints in the Command window. All enabled and disabled breakpoints are listed.

The following command line:

```
>BL /* List current breakpoints
*/
```

lists the currently defined breakpoints shown below.

```
0: (E C: 0xFF01EF) 'main', CNT=1, enabled
1: (E C: 0xFF006A) 'timer0', CNT=10, enabled
   exec ("MyRegs()")
2: (C) 'sindex == 8', CNT=1, enabled
3: (C) 'save_record[5].time.sec > 5', CNT=3, enabled
4: (A RD 0x000037) 'READ interval.min == 3', CNT=1, enabled
5: (A WR 0x000034) 'WRITE savefirst==5 && acc==0x12', CNT=1, enabled
```

Breakpoints are listed one per line using the following format:

```
number: (type) 'expression', CNT=count, enable_flag
        exec ("command")
```

where:

- number* is the index for the breakpoint. dScope assigns this number when a breakpoint is created. Use it when enabling, disabling, or removing a breakpoint with the **BREAKENABLE**, **BREAKDISABLE**, or **BREAKKILL** commands,
- type* is the breakpoint type which may be an execution breakpoint (**E**: followed by the address), a conditional breakpoint (**C**), or an access breakpoint (**A**: followed by **RD** for read, **WR** for write, or **RW** for read/write followed by the address).
- expression* is the original text of the breakpoint definition.
- count* is the pass counter for the breakpoint. If *count* has a value of 2, dScope halts program execution (or executes the specified command) when the breakpoint is reached the second time.
- enable\_flag* displays **enabled** for an enabled breakpoint or **disabled** for a disabled breakpoint.
- command* is the command to execute when the breakpoint is reached.

## Removing Breakpoints

You may remove breakpoints using the **BREAKKILL** command, from the Breakpoints dialog box using the provided controls, or from the Debug window using the mouse.

## Using the Debug Window

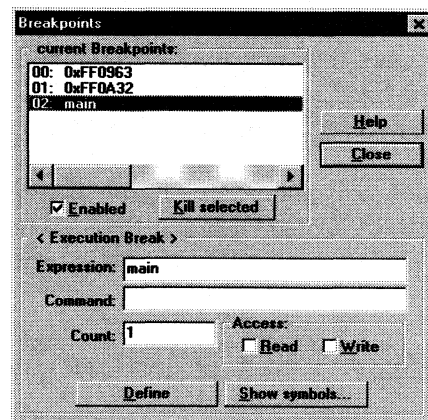
The Debug window displays the source and assembly code for your target program. You may set an execution breakpoint in the Debug window by double-clicking the left mouse button on the desired line in your program. If an execution breakpoint is already defined for the line it is removed.

When an execution breakpoint is defined for a line displayed in the Debug window, the line is highlighted using the BP-Highlight color and a breakpoint label (**[BR n]** where **n** represents the breakpoint index) displays on the line.

## Using the Breakpoints Dialog Box

The Breakpoints dialog box lets you quickly select and remove breakpoints. The following steps show you how to remove a breakpoint using the Breakpoints dialog box.

1. Open the Breakpoints dialog box using the Breakpoints command in the dScope Setup menu.
2. Select the breakpoint to remove from the current breakpoints list.
3. Click the Kill Selected button.



## Using the BREAKKILL Command

The **BREAKKILL** command lets you delete one or more breakpoints. You may delete all breakpoints using **BK \***. You may delete specific breakpoints by specifying **BK** along with the breakpoint index displayed using the **BREAKLIST** command or the Breakpoints dialog box.

The following command removes breakpoints 0 and 3.

```
>BK 0,3
```

The following command removes all breakpoints.

```
>BK *
```

## Enabling and Disabling Breakpoints

Once a breakpoint is defined, you may enable or disable it. When a breakpoint is disabled, the breakpoint condition is no longer tested and the breakpoint cannot be triggered. When the breakpoint is enabled, its condition is tested and the breakpoint can be triggered.

Rather than remove a breakpoint definition you may need later, you can disable the breakpoint and re-enable it when necessary. This is often more convenient than re-defining a complex breakpoint.

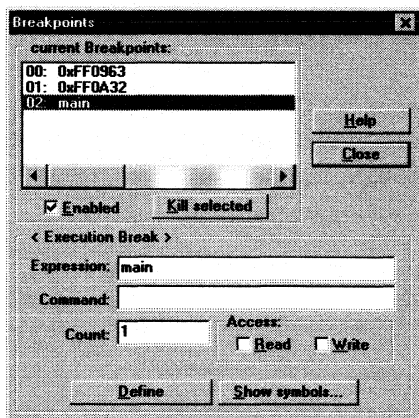
You may enable and disable breakpoints from the Command window using the **BREAKENABLE** and **BREAKDISABLE** commands, or from the Breakpoints dialog box using the provided controls.

# 6

## Using the Breakpoints Dialog Box

The Breakpoints dialog box lets you quickly enable and disable breakpoints.





The following steps show you how to enable or disable a breakpoint using the Breakpoints dialog box.

1. Open the Breakpoints dialog box using the Breakpoints command in the dScope Setup menu.
2. Select the breakpoint to remove from the current breakpoints list.
3. Click the Enabled check box to toggle whether or not the breakpoint is enabled. When this check box is checked, the breakpoint is enabled. When it is cleared, the breakpoint is disabled.

## Using the BREAKENABLE Command

The **BREAKENABLE** command lets you enable one or more breakpoints. You may enable all breakpoints using **BE \***. You may enable specific breakpoints by specifying **BE** along with the breakpoint index displayed using the **BREAKLIST** command or the Breakpoints dialog box.

The following command enables breakpoints 0 and 1.

```
>BE 0,1
```

The following command enables all breakpoints.

```
>BE *
```

## Using the BREAKDISABLE Command

The **BREAKDISABLE** command lets you disable one or more breakpoints. You may disable all breakpoints using **BD \***. You may disable specific breakpoints by specifying **BD** along with the breakpoint index displayed using the **BREAKLIST** command or the Breakpoints dialog box.

The following command disables breakpoints 1 and 2.

```
>BD 1,2
```

The following command disables all breakpoints.

```
>BD *
```

## Chapter 7. Using Watchpoints

Watchpoints are a powerful feature of dScope. They let you watch variables and memory areas affected by your target program.

There are three basic operations you may perform with watchpoints. You may:

- View watchpoints using the Watch Window described on page 44,
- Define watchpoints using the Watchpoints Dialog Box described on page 81 and using the **WATCHSET** command described on page 260,
- Remove watchpoints using the Watchpoints Dialog Box and using the **WATCHKILL** command described on page 259.

This chapter describes how to view, define, and remove watchpoints using each of the methods outlined above.

### Watchpoint Lifetime

Watchpoints remain in memory until you do one of the following.

- Remove them using the Watchpoints dialog box or **WATCHKILL** command,
- Load a new target program,
- Load a new CPU driver,
- Exit the dScope debugger.

Since watchpoints remain in memory, you can easily enter a complex set of watchpoints and execute your target program several times.

You may create a command file that stores the watchpoints you want to use each time you debug your target program. The **INCLUDE** command reads the command file and sets the specified watchpoints. This saves time if your watchpoints are lengthy or complex. Refer to “Command Files” on page 43 for more information.

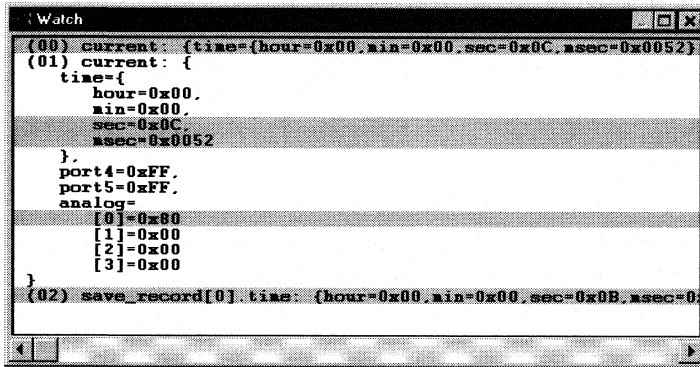
## Watchpoint Notes

There are a number of points you should take into consideration when you use watchpoints.

- Setting a watchpoint using a symbol name or variable name is a convenient feature of the dScope debugger. dScope resolves the symbol name to a data type and memory address.
- Once a watchpoint is defined, dScope always displays the value at the watchpoint expression address; even when the variables are not in scope. Automatic variables and function arguments may not always display correctly when they are not in scope. This is because their storage space may be on the stack or overlaid in memory.
- To enter a watchpoint using a local variable or function argument, the program counter must be in the function where the variable is defined. If the program counter is located outside the function, you must use the fully qualified symbol name to enter a watchpoint for that variable. Refer to “Fully Qualified Symbols” on page 97.
- Watchpoints require some overhead and slow the simulation speed of dScope. Complex watchpoints like large structures and arrays take longer to evaluate and display. For faster simulation speeds, reduce the number and complexity of watchpoints.
- dScope can automatically update the Watch window as your target program runs. To enable this feature, select the Update Watch Window command from the Setup menu. Note that updating the Watch window also slows simulation speed.

## Viewing Watchpoints

The Watch window displays the watchpoints you define. Watchpoints may display on a single line or on multiple lines depending on how they are defined. Single lines are truncated at 128 characters which may not be the best choice for large data structures and arrays. In multi-line mode, each member of a structure or union and each element of an array display on separate lines.



As you step through your target program, the Watch window updates to reflect the current value for each watchpoint.

---

### NOTE

*Each watchpoint displayed in the Watch window is preceded by an index in parentheses. In the above figure, the index for the `save_record[0].time` watchpoint displays as (02). Use the index to remove a specific watchpoint.*

---

## Setting Watchpoints

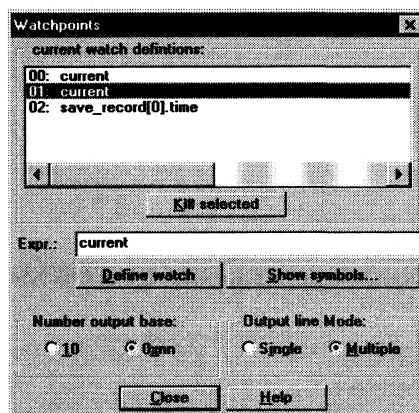
You may define watchpoints either from the Command window using the **WATCHSET** command, or from the Watchpoints dialog box using the provided controls.

## Using the Watchpoints Dialog Box

The Watchpoints dialog box provides a number of controls you use to create and define watchpoints to add to the Watch window.

You must perform the following steps to define a watchpoint using the Watchpoints dialog box.

1. Open the Watchpoints dialog box using the Watchpoints command in the dScope Setup menu.
2. Specify the variable to watch on the Expression input line. You may enter the name of a symbol or an expression. You may also drag symbols from the Symbol Browser window and drop them onto the Expression input line. The Show Symbols button opens the Symbol Browser window. Refer to “Symbol Browser Window” on page 56 for more information.
3. Select any appropriate options (number output base and output line mode). Scalars such as integer, byte, bit, or floating-point are always displayed on a single line.
4. Click the Define Watch button to define the watchpoint. If the specified watchpoint is valid it is added to the list of current watch definitions. An error is displayed if there is a problem with the watchpoint.



## Using the WATCHSET Command

The **WATCHSET** command lets you define watchpoint expressions to display in the Watch window. You may optionally specify the number base (10 or 16) in which to display the value of the expression and you may optionally use the **LINE** parameter to specify that structures, unions, and arrays display on a single line (as opposed to displaying on multiple lines).

The following command creates a watchpoint for **interval**. The watchpoint displays in base 10 on a single line.

```
>WS interval,10 LINE
```

The following command creates a watchpoint for the **analog** element of the **struct** or **union** at **save\_record[0]**.

```
>WS save_record[0].analog
```

## Removing Watchpoints

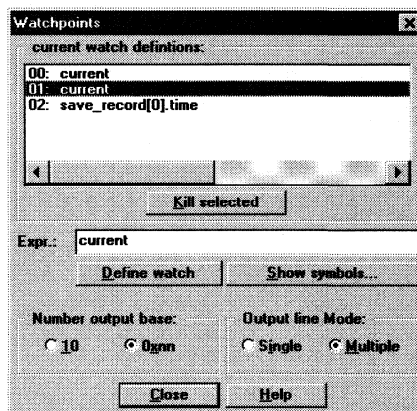
You may remove watchpoints either from the Command window using the **WATCHKILL** command, or from the Watchpoints dialog box using the provided controls.

## Using the Watchpoints Dialog Box

The Watchpoints dialog box lets you quickly select and remove a watchpoint.

The following steps show you how to remove a watchpoint using the Watchpoints dialog box.

1. Open the Watchpoints dialog box using the Watchpoints command in the dScope Setup menu.
2. Select the watchpoint to remove from the current watch definitions list.
3. Click the Kill Selected button.



## Using the WATCHKILL Command

The **WATCHKILL** command lets you delete one or more watchpoints from the Watch window. You may delete all watchpoints using **WK \***. You may delete specific watchpoints by specifying **WK** along with the watchpoint index displayed the Watch window.

The following command removes watchpoints 0 and 2.

```
>WK 0,2 /* Delete watchpoints 0 and 2 */
```

The following command removes all watchpoints.

```
>WK * /* Delete all watchpoints */
```



# Watching Program Variables

dScope lets you watch both local variables and global variables in your target programs. You can watch complex variables like structures, unions and arrays. dScope even supports watchpoints that use pointers to indirectly access variables.

## Watching Global Variables

To add a watchpoint for a program variable you enter the variable name as the watch expression. For example, in the following function,

```
int i;  
  
void func (void)  
{  
  for (i = 0; i < 1000; i++)  
  {  
    printf ("%d\n", (int) i);  
  }  
}
```

to watch the global variable **i**, enter the following command in the Command window:

```
>WS i
```

The defined watchpoint displays the value of **i** in the Watch window.

## Watching Local Variables

Watching a local variable is just as easy as watching a global variable. For example, in the following function,

```
void func (void)  
{  
  int t;  
  
  for (t = 0; t < 1000; t++)  
  {  
    printf ("%d\n", (int) t);  
  }  
}
```

to watch the local variable **t**, enter the following command in the Command window:

```
>WS t
```

If **t** is in scope, the watchpoint is added to the Watch window. If **t** is not in scope, you must use the fully qualified symbol name to add the watchpoint. Refer to “Fully Qualified Symbols” on page 97 for more information.

## Watching Structure, Union, and Array Elements

Watching C structures, unions, and arrays is easy to do. The following program example defines a structure, a union, and an array.

```
struct msg_st
{
    unsigned char id;
    unsigned char buffer [100];
    unsigned char length;
} message;

union ulong_st
{
    unsigned long number;
    unsigned char array [4];
} ulongconv;

long tenpow [] = { 1L, 10L, 100L, 1000L, 10000L, 100000L };
```

The following command creates a watchpoint for the entire **message** structure:

```
>WS message
```

The following command creates a watchpoint for the **id** element of the **message** structure:

```
>WS message.id
```

The following command creates a watchpoint for the **number** element of the **ulongconv** union:

```
>WS ulongconv.number
```

The following command creates a watchpoint for the **ulongconv** union:

```
>WS ulongconv
```

The following command creates a watchpoint for the **tenpow** array:

```
>WS tenpow
```

The following command creates a watchpoint for the first element in the **tenpow** array:

```
>WS tenpow[0]
```

---

## NOTES

*Your target application must be compiled with full debug information to in order to support full type information for structures and unions. If the debugging information available is insufficient, dScope displays an error message. Refer to “Preparing Programs for dScope” on page 3 for more information.*

*When you add a watchpoint for a union, dScope displays all elements of the union in the Watch window. Since the members of a union occupy the same memory area, you can use this feature of dScope to see the effects of assigning different values to the different members.*

---

## Watching Pointers

dScope lets you use pointers in watchpoint expressions. You can watch the value of the pointer and you can watch the object that the pointer references. The difference is in how you specify the pointer in the watchpoint expression. The following program example defines a structure and pointers that reference it.

```
struct node
{
    struct node    *next;        /* a ptr to a struct */
    unsigned char  op;           /* operation */
    unsigned char  type;         /* operation type */
};

struct node nodes[10];          /* an array of [10] nodes */
struct node nptr = &nodes[0]; /* node pointer */
```

The following watchpoint displays the address stored in the **nptr** structure pointer. It does not display the contents of **nodes [0]**.

```
>WS nptr
```

The following watchpoint displays the structure pointed to by the **nptr** pointer. For the above example, this is the contents of **nodes [0]**.

```
>WS *nptr
```

The following watchpoint displays the address stored in the **next** member of the structure pointed to by **nptr**. It does not display the contents of the structure pointed to by **next**.

```
>WS nptr->next
```

The following watchpoint displays the structure pointed to by the **next** member in the structure pointed to by **nptr**.

```
>WS *nptr->next
```

---

**NOTE**

*Your target application must be compiled with full debug information to in order to support full type information for pointers. If the debugging information available is insufficient, dScope displays an error message. Refer to “Preparing Programs for dScope” on page 3 for more information.*

---

## Watching Memory Locations

dScope lets you create watchpoints for memory locations. You can watch any of the following data types:

<b>bit</b>	<b>float</b>	<b>uchar</b>
<b>byte</b>	<b>int</b>	<b>uint</b>
<b>char</b>	<b>long</b>	<b>ulong</b>
<b>double</b>	<b>ptr</b>	<b>word</b>
<b>dword</b>	<b>real</b>	

To create a watchpoint for a particular memory address, you must specify one of the above data types followed by an address in parentheses. For example, the following command defines a watchpoint for an **int** at X:0x1000.

```
>WS int(X:0x1000)
```

The following watchpoint displays the floating-point number at D:0x10.

```
>WS float(D:0x10)
```



## Chapter 8. Tutorial

This chapter provides a tutorial on how to use the dScope debugger to load and test your target programs. It assumes that you have already installed the dScope debugger and know how to start it. Refer to “Starting the dScope Debugger” on page 11 for more information.

### Running the HELLO Sample Program

Keil development tools for the 8051, 251, and 166 come with a number of sample programs you can use to help you get started. The HELLO sample program prints the text “Hello World” to the on-chip serial port of the 8051, 251, or 166. While this is not a very sophisticated program, it does help demonstrate how to use the entire toolset including the dScope debugger.

---

#### NOTES

*The first time you invoke dScope, you may need to change the fonts and colors used for the different windows. Refer to “Color Setup Dialog Box” on page 69 for more information.*

*This example uses the HELLO program with the 8051 tools. The concepts presented here also apply to the 251 and 166.*

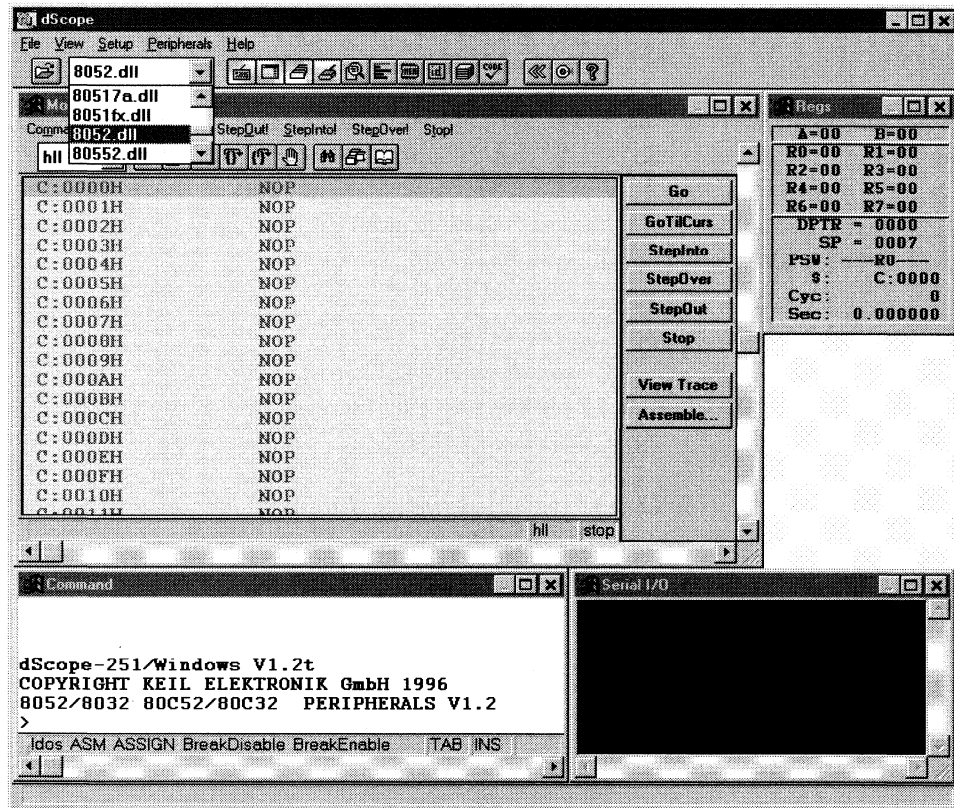
---

## 8

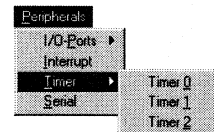
## Loading a CPU Driver

After you start dScope, you must first load a CPU driver so that dScope can properly simulate the on-chip peripherals of the particular CPU you are using. The HELLO sample program uses the 8052 CPU.

Load the 8052.DLL CPU driver as shown in the following figure.



When the CPU driver loads, note that Peripherals appears in the Menu. The Peripherals menu lists dialog boxes you may use to view and manipulate on-chip peripherals associated with the simulated CPU. For example, the 8052 DLL supports 4 I/O ports (P0, P1, P2, and P3), 6 interrupts, 3 timers, and a serial port.



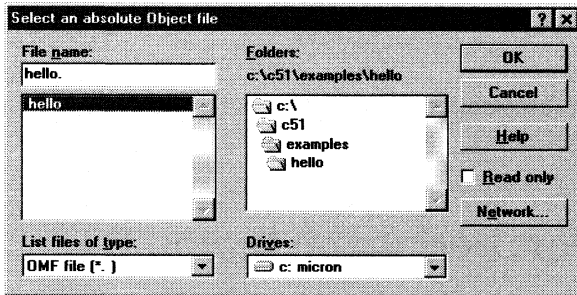
Other DLLs may support additional on-chip peripherals. Refer to “Appendix A. CPU Driver Files” on page 295 for a complete list of DLLs provided with dScope.



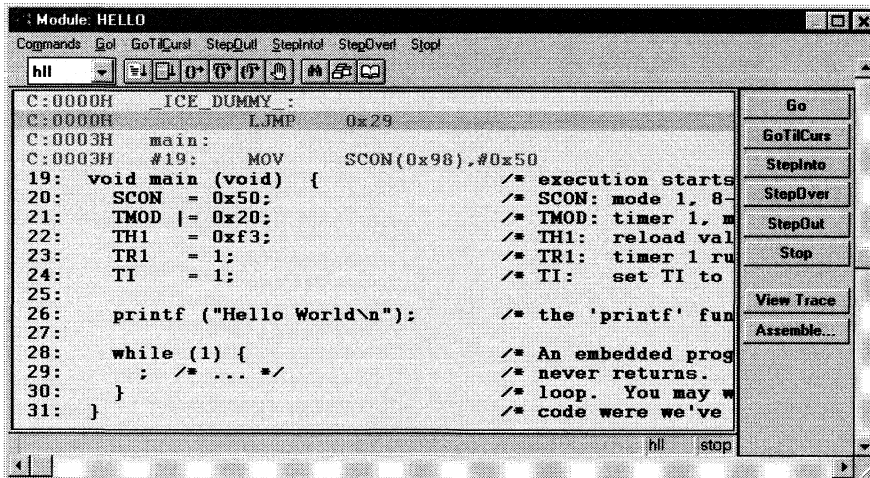
## Loading a Target Program

After the CPU driver is loaded, you may load a target program. The target program is an application program you create using the assembler, compiler, and linker. The OMF file created by the linker is what you load into dScope. It is also sometimes called an absolute object module or an absolute OMF file.

To load the HELLO sample program, use the Open object file button to open the File Open dialog. Then, switch to the C51\EXAMPLES\HELLO directory and select the HELLO sample program.



When HELLO loads, the Debug window displays the source code for the program.

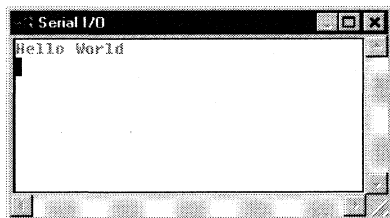


The HELLO program is very short. It initializes the serial port, prints “Hello World”, and enters an infinite loop.

## 8

## Starting Program Execution

To run the HELLO program, click on the Go button in the Debug window or enter **g** in the Command window. The HELLO program displays “Hello World” in the Serial window.

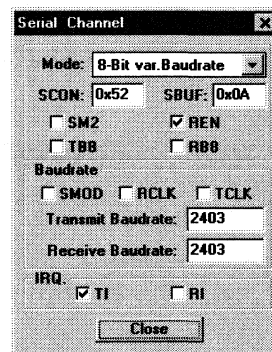


This happens pretty quickly. On a fast Pentium-based PC, simulation occurs at almost real-time rates.

## Viewing the Serial Port Peripheral Dialog Box

One of the unique features of dScope is its ability to support the on-chip peripherals of many derivatives of the 8051, 251, and 166.

Select the Serial command from the Peripherals menu to display the SFRs associated with the serial port on the 8052. The dialog box displays the current status of all SFRs and even displays the current baudrate. You may use the controls in the dialog box to change the state of any of the SFRs listed.



## Halting Program Execution

To stop the HELLO program, click on the Stop button in the Debug window or type **Ctrl+C** or **Esc** in the Command window. Note that when program execution stops, the status bar in the Debug window updates to reflect the current status: RUN or STOP.

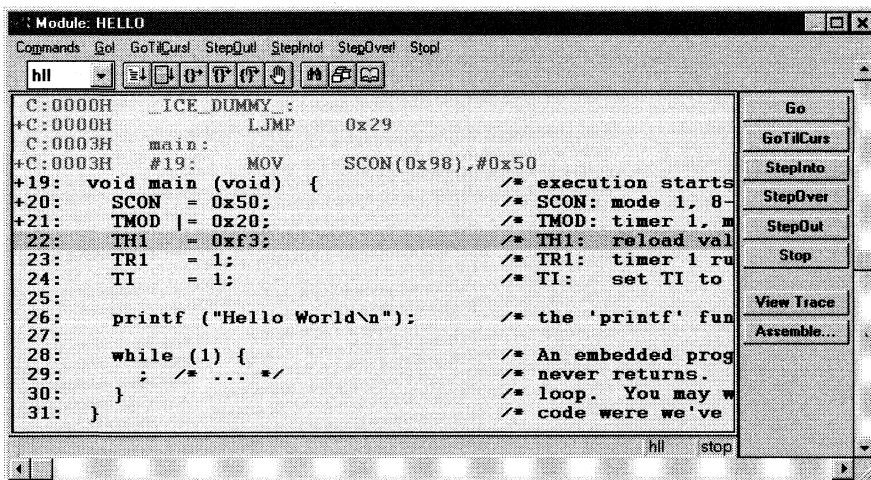
## Resetting the CPU

To reset the HELLO program so you can run it again, click on the Reset button on the tool bar or enter the **RESET** command in the Command window. Refer to “RESET” on page 244 for more information.

When you reset the CPU, the program counter is reset to 0 and all SFRs are initialized to their power-on state.

## Single Stepping Through a Program

Stepping through your program one statement or instruction at a time is easy with dScope. After halting program execution and resetting the CPU, click several times on the StepOver button in the Debug window.



Each time you step over a statement, dScope updates the program counter and displays a highlight bar to show the next instruction or statement to execute.

As you step through the HELLO program, you will see the SFRs in the Serial Port dialog box update. Finally, when you step over the printf function call, the Serial window displays the “Hello World” message.

### NOTE

Refer to “Chapter 4. Executing Code” on page 115 for more information about the commands you may use to step through and run your target program.

## 8

## Exiting dScope

To exit dScope, you may...

- Select the Exit command from the File menu,
- Click on the dScope Main Window Close Button (in Windows 95),
- Select the Close command from the dScope Main Window Control Menu (in Windows 3.1),
- Double-clicking on the dScope Main Window Control Menu box,
- Enter the Exit command in the Command window. For example:

```
exit
```

---

### NOTE

*dScope requires that you stop simulation or target execution of your program before you exit. Click on the Stop button or type **Ctrl+C** in the Command window.*

---

## Running the MEASURE Sample Program

The MEASURE sample program runs a remote measurement system that collects analog and digital data like data acquisition systems found in weather stations and process control applications. MEASURE records data from two 8-bit digital ports and four 8-bit analog-to-digital inputs. A timer controls the sample rate which may be configured from 1 millisecond to 60 minutes. Each measurement saves the current time and all of the input channels to an 8 Kbyte RAM buffer.

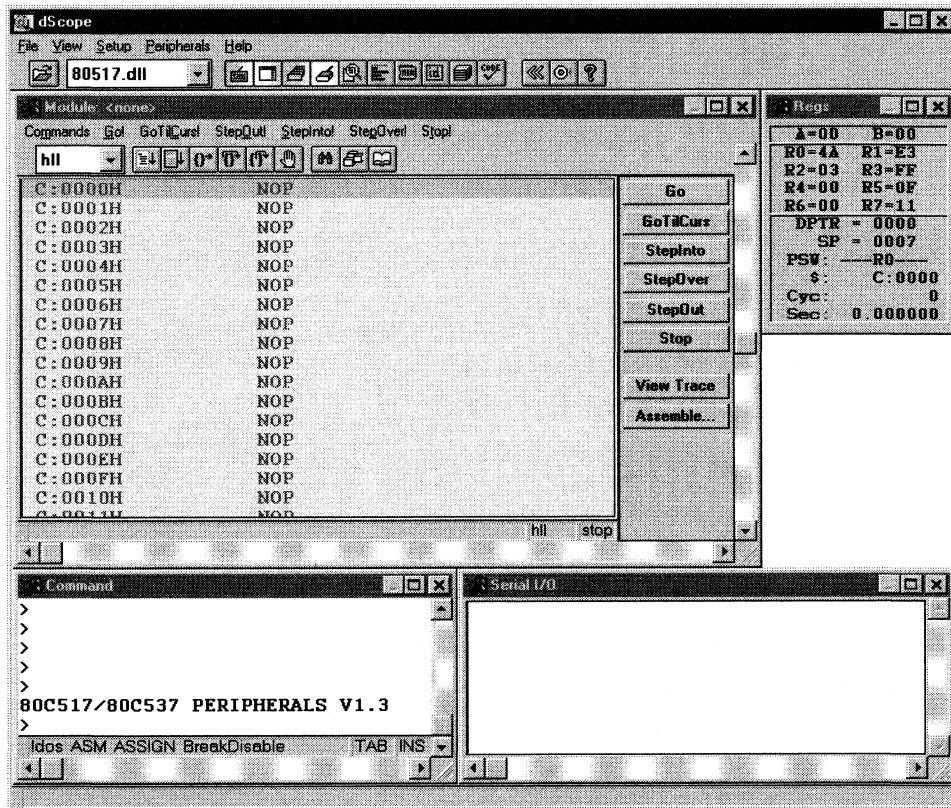
The MEASURE sample program is designed to accept commands received by the on-chip serial port. If you have actual target hardware, you can use a host computer or dumb terminal to communicate with the CPU. If you do not have target hardware, you can use dScope to simulate the hardware and provide commands to MEASURE using the Serial window.

## 8

## Loading a CPU Driver

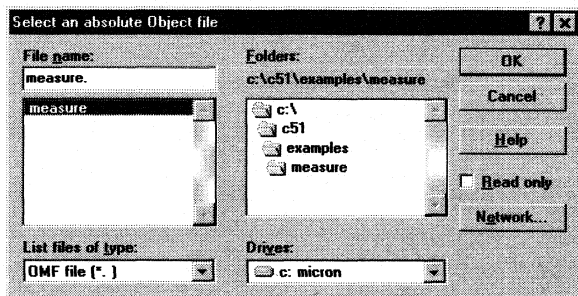
The hardware for MEASURE is based on the 80517 CPU. This microcontroller provides analog and digital input capability. Port 4 and port 5 are used for the digital inputs and AN0 through AN3 are used for the analog inputs. You do not actually need target hardware because dScope lets you simulate all the hardware required for this program.

Load the 80517.DLL CPU driver as shown in the following figure.

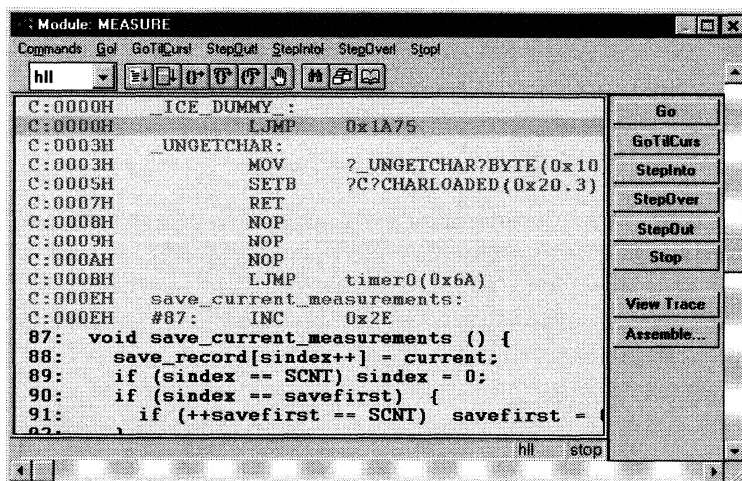


## Loading a Target Program

To load the MEASURE sample program, use the Open object file button to open the File Open dialog. Then, switch to the C51\EXAMPLES\MEASURE directory and select the MEASURE sample program.



When MEASURE loads, the Debug window displays the source code for the program.



## 8

## Measure Commands

The serial commands that MEASURE supports are listed in the following table. These commands are composed of ASCII text characters.

Command	Serial Text	Description
<b>Clear</b>	C	Clears the measurement record buffer.
<b>Display</b>	D	Displays the current time and input values.
<b>Time</b>	T <i>hh:mm:ss</i>	Sets the current time in 24-hour format.
<b>Interval</b>	I <i>mm:ss.ttt</i>	Sets the interval time for the measurement samples. The interval time must be between 0:00.001 (for 1ms) and 60:00.000 (for 60 minutes).
<b>Start</b>	S	Starts the measurement recording. After receiving the start command, MEASURE samples all data inputs at the specified interval.
<b>Read</b>	R [ <i>count</i> ]	Displays the recorded measurements. You may specify the number of most recent samples to display with the read command. If no count is specified, the read command transmits all recorded measurements. You can read measurements on the fly if the interval time is more than 1 second. Otherwise, the recording must be stopped.
<b>Quit</b>	Q	Quits the measurement recording.

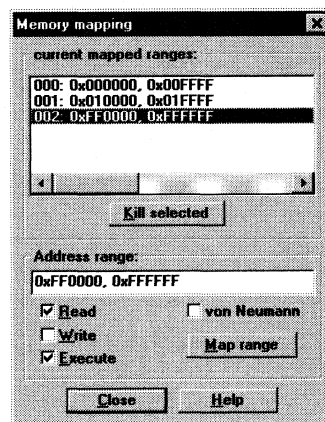
## Viewing the Memory Map

dScope maintains a memory map for your target program where you specify a memory range and the type of access methods (read, write, or execute) that may be performed there. If there are invalid memory accesses in a range or if there are any access outside the defined range, dScope prompts you with a warning. Refer to “Memory Map Dialog Box” on page 72.

Open the Memory Map dialog box using the Memory map command in the dScope Setup menu.

When you click on a memory range in the list of current mapped ranges, the Address range, Read, Write, and Execute boxes are filled-in.

dScope automatically maps memory according to definitions in the MEASURE program. If there are additional memory regions the MEASURE program requires, like memory mapped I/O, you could enter them here.





The memory ranges listed for MEASURE are as follows:

- 0x000000, 0x00FFFF designates the data space of the CPU (**DATA**).
- 0x010000, 0x01FFFF designates the external data space of the CPU (**XDATA**).
- 0xFF0000, 0xFFFFFFFF designates the code space of the CPU (**CODE**).

Since dScope supports up to 16 megabytes of mapped memory, individual memory spaces of the CPU are mapped to one of the 256 possible segments. **CODE** is mapped to segment 0xFF (0xFF0000-0xFFFFFFFF), **XDATA** is mapped to segment 0x01 (0x010000-0x01FFFF), and **DATA** is mapped to 0x00 (0x000000-0x00FFFF).

---

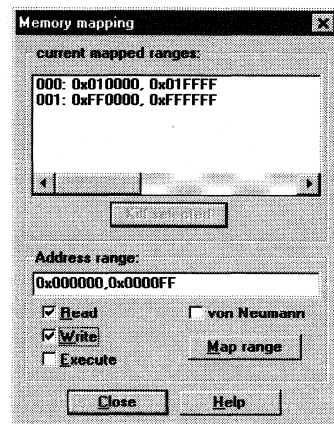
**NOTE**

*The **DATA** space mapped is much larger than required. This is done by default to provide upwards compatibility with the new MCS<sup>®</sup> 251 architecture.*

---

dScope makes it easy to change the memory map for the **DATA** area to include only the memory from 0x000000 to 0x0000FF. The following steps show how to remove the existing map definition and replace it with a new one.

1. Select the current memory map for the **DATA** area (0x000000, 0x00FFFF).
2. Use the Kill selected button to remove the memory map definition.
3. In the Address range input line, enter:  
0x000000, 0x0000FF.
4. Click the Read and Write check boxes to indicate that the range may be written to and read from. The Execute check box should be empty since code may not be executed from the internal data memory of the 8051.
5. Use the Map range button to add the new memory range to the memory map.
6. Now, the **DATA** memory range is remapped to be 256 bytes from 0x000000 to 0x0000FF.

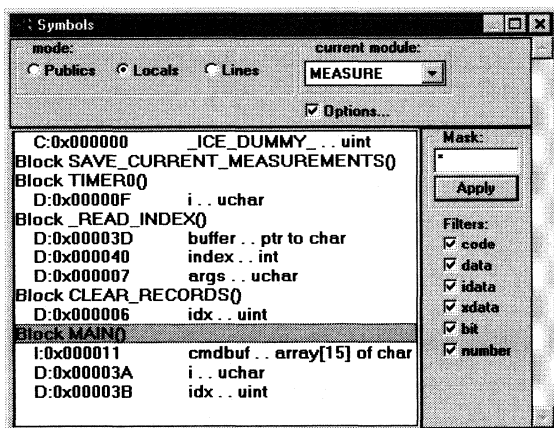


When you are finished viewing and changing the memory close the Memory Map dialog box using the Close button.

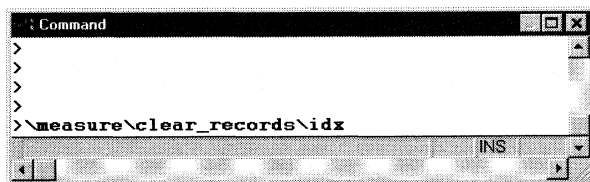
## 8

## Viewing Debug Symbols

The MEASURE sample program is configured for full debug information and includes public and local symbols, line numbers, and high-level data type information. To view the symbol information, click on the Symbol Browser button on the tool bar to open the Symbol Browser window. Then, select the Locals radio button and the Options check box as shown below.



dScope supports drag and drop and lets you access the symbols this way. For example, you may use the mouse to drag the **idx** symbol from the Symbol Browser window and drop it into the Command window. The fully qualified symbol name with module name and function name are inserted as shown below.



Select the Command window and press **Enter** to display the value of **idx**.

You may filter the symbols displayed in the Symbol Browser window by selecting the memory space filter. If you clear the data check box, all symbols in the data memory area are removed from the display. You may also specify a search mask to limit the symbols displayed. To limit the symbol list to those beginning with the letter I, enter **I\*** and click the Apply button.

For more information about using the Symbol Browser, refer to “Symbol Browser Window” on page 56.

## Viewing and Changing Memory

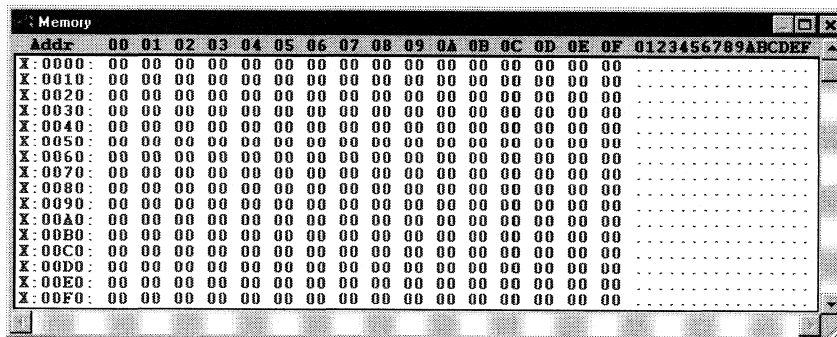
dScope provides a number of ways you can view and change variables, registers, and memory contents. In addition to the methods discussed in this chapter, refer to “Chapter 5. Examining and Modifying Memory” on page 121 for more information about viewing and changing memory.

### Dumping Memory Contents

dScope displays memory in HEX and ASCII in the Memory window. Open the Memory window by clicking on the Memory button on the tool bar. Then, enter the address range you want to view in the Command window. For example:

```
D X:0x0000, X:0xFFFF
```

When you enter the above command, memory from the **XDATA** area displays in the Memory window. Since the Memory window cannot show the entire memory range at once, you may use the scroll bars to scroll through the memory area. The bounds for scrolling are defined by the address range specified, which is 0x0000 to 0xFFFF for this example.



To display the on-chip data memory, enter the following in the Command window.

```
D I:0x0000, I:0xFF
```

dScope can dynamically update the Memory window while your application is running. To toggle dynamic updating, select the Update Memory window command from the Setup menu. When Update Memory window is checked, dynamic updating is enabled.

Refer to “Memory Window” on page 54 for more information.

## 8

## Filling Memory Areas

dScope does not provide a Memory-Fill command. Nonetheless, filling memory can be done using the predefined dScope function **memset**. The **memset** function requires three arguments:

- The first argument is the starting address.
- The second argument is the ending address.
- The third argument is an 8-bit value to write to the memory.

For the first and second arguments, you should supply addresses with a memory space as shown in the following examples.

```
MEMSET (X:0x0000, X:0xFFFF, 0xA5)      /* clear 64K xdata memory */
MEMSET (D:0x20, D:0x7F, 0)              /* clear on-chip data memory */
MEMSET (C:0x0, C:0xFFFF, 0)            /* clear code memory to nops */
MEMSET (0xFF0000, 0xFFFFFFFF, 0)        /* same as C:0, C:0xffff ! */
MEMSET (0x010000, 0x01FFFF, 0)          /* same as X:0, X:0xffff ! */
```

Note that the addresses and values may be address expressions and expression values and not just constants.

---

### NOTE

*If you applied the **memset** function to code memory, you should reload the MEASURE program to continue with the remaining examples.*

---

## Viewing Registers and Variables

In dScope, to view the contents of a register, simply type the name of the register in the Command window and type **Enter**. dScope displays the contents of the register in the Command window.

Viewing values of variables or addresses of functions is similar, simply enter the name.

```
>A                /* get value of the Accumulator */
>DPTR            /* DPTR's value */
>current.time.sec /* get members value */
>C              /* get Carry flag */
>sindex         /* get value of 'sindex' */
>current.time.sec = 12 /* set nested struct member */
>\MEASURE\main\cmdbuf[0] /* fully qualified */
>OBJ current    /* get values of complete structure */
```

If the name of a variable is the same as the name of a dScope command, you must literalize the variable name in order to avoid a syntax error:

```
>BD = 1                /* set Bit 'BD' */
                        /* Note: gives a syntax error since BD is the dScope */
                        /* command BreakpointDisable !!! */
                        /* to avoid an error, literalize BD: */
>`BD = 1              /* now it works. */
```

## Changing Registers and Variables

To change the contents of a register, simply type the name of the register, an equal sign (“=”), the new value for the register, and finally type **Enter**. dScope changes the register's contents.

The following examples show how this is done.

```
>A = 0xFE          /* assign 0xFE to Accumulator */
>DPTR = 0x1234     /* assign 0x1234 to DPTR */
>--DPTR           /* meaning: DPTR = DPTR - 1 */
>R0=3,R1=4,R2=5   /* comma separated expressions */
>R7=current.time.sec-- /* program symbols used */
>C=1              /* set Carry flag from PSW */
```

Changing the value of a variable is similar. You simply enter the name of the variable followed by an assignment:

```
>sindex = 0x64     /* assign 100 to 'sindex' */
>current.time.sec = 12 /* set nested struct member */
>\MEASURE\_READ_INDEX\index = 0 /* fully qualified symbol */
>current.time.msec = save_record[10].time.msec
```

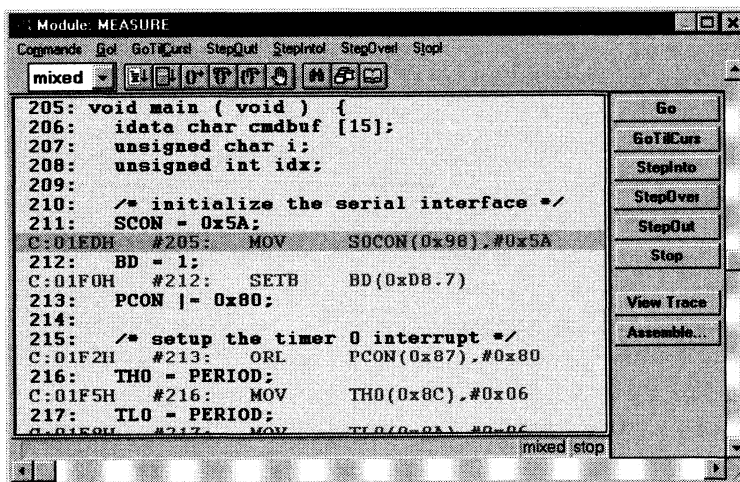
## 8

## Program Execution

dScope gives you a number of options when executing code in your target program. In addition to the methods discussed in this chapter, refer to “Chapter 4. Executing Code” on page 115 for more information.

### Changing the View Mode

dScope lets you change the view mode in the Debug window. Display the Debug window using the debug button on the tool bar. Then, to change the view mode, open the Commands menu in the Debug window and select View High level, View Mixed, or View Assembly. For example, View Mixed changes to the mixed source and assembly display.

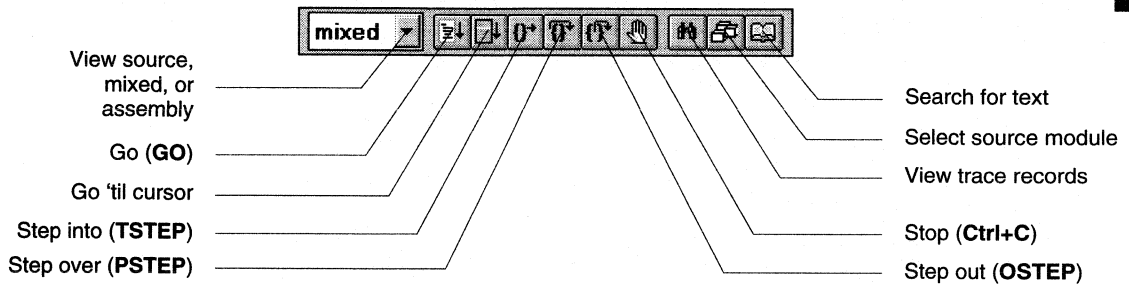


The Debug window shows intermixed source and assembly lines.

## Using the Toolbar and Dialog Bar

The toolbar in the Debug window lets you rapidly select the most frequently used debug commands. The following figure shows the layout of the toolbar and gives descriptions of each toolbar button.

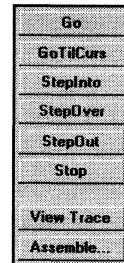
8



In addition to the toolbar, the dialog bar in the Debug window also lets you rapidly select the most frequently used debugger commands.

### NOTE

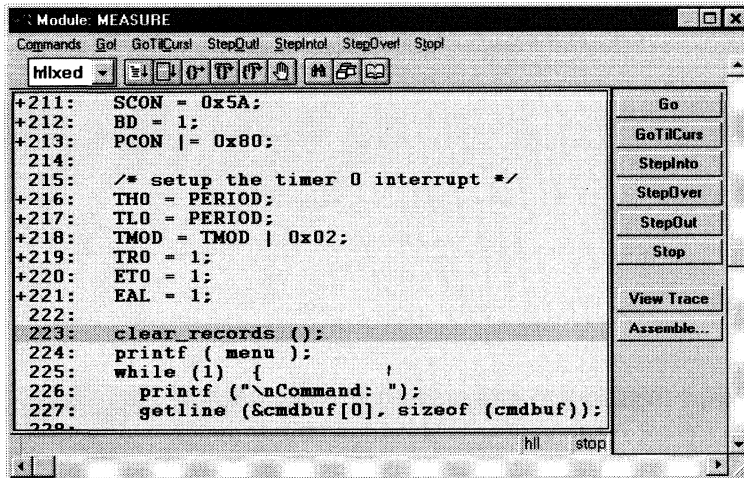
*When you begin simulating the MEASURE program, use the Debug, Register, and Serial buttons on the tool bar to display the Debug, Register, and Serial windows. You may disable other windows if your screen is not large enough.*



## 8

## Single Stepping

From the toolbar, select the reset button to reset dScope. In the Debug window, select the View HLL command from the Commands menu. Then, click on the StepInto button several times.

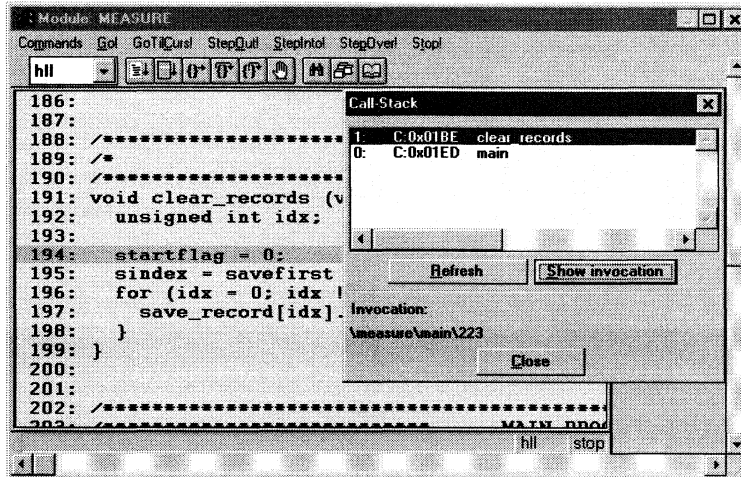


The StepInto button lets you single-step through your application and into function calls. Click on the StepInto button a few more times to get to the `clear_records` function call. Then, StepInto the `clear_records` function.



## Viewing the Call Stack

As you execute through your target program, dScope internally tracks function nesting using **call** and **ret** assembler instructions. The function nesting is known as the Call Stack. You may view the call stack at any time by opening the Call Stack window. Once you have single stepped into the **clear\_records** function, use the Call Stack button on the tool bar to display the Call Stack window.

**8**

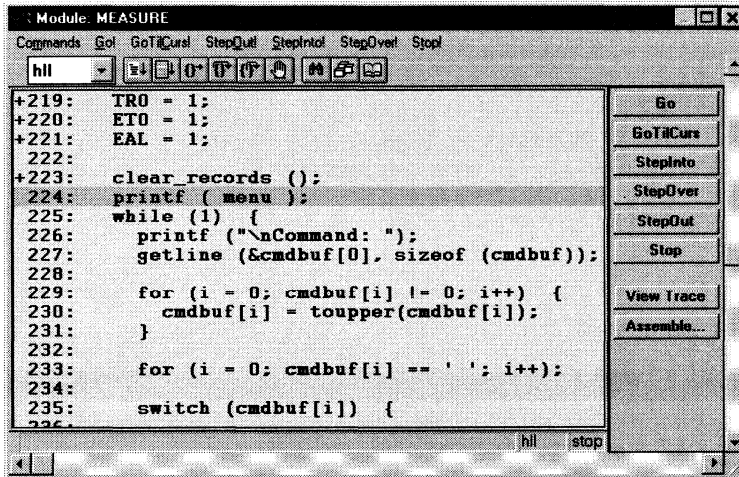
This window lists all currently nested functions. Each line contains a nesting level number, the numeric address of the invoked function, and the symbolic name of the function if debug information is available.

You can display the caller of a function by selecting the function from the list. Then, you can use the Show invocation button to display the function call in the Debug window.

## 8

## Stepping Out of a Function

From inside the **clear\_records** function, you can return to the calling point in the **main** function by clicking on the StepOut button. This command completes execution of the current function and returns to the statement immediately following the function call.

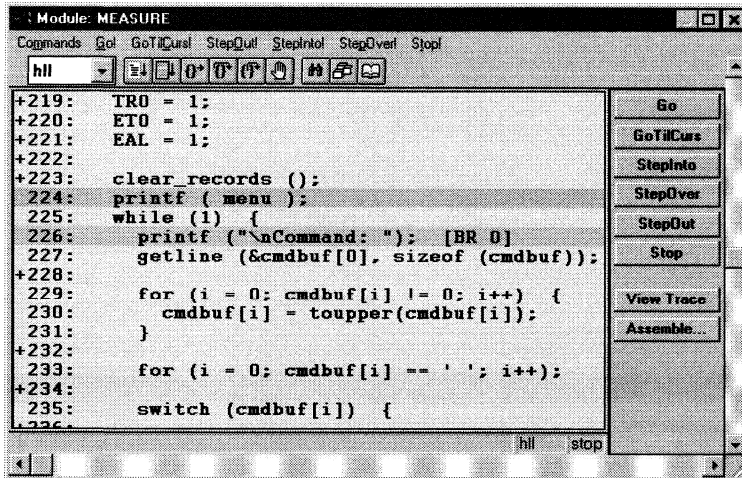


### NOTE

You cannot **StepOut** from the **main** function because it is invoked by a long `jmp` (`LJMP`) rather than a `call` instruction.

## Setting and Removing Breakpoints

You may set execution breakpoints from within the Debug window by double-clicking on the desired source line. The selected line is highlighted and a [BR n] label is displayed at the end of the line. A breakpoint set on the `printf` statement is shown in the following Debug window.



Click on the Go button and dScope starts execution from the current program counter and stops when the breakpoint is reached. To remove a breakpoint, double-click on the line containing the breakpoint.

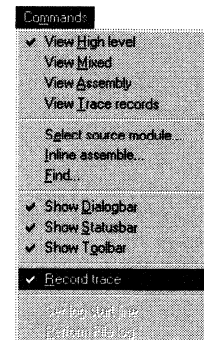
Refer to “Chapter 6. Using Breakpoints” on page 135 for more information about other types of breakpoints.

## Trace Recording

It is common during debugging to reach a breakpoint where you require information like register values and other circumstances which led to the breakpoint. Trace recording lets you record each assembly instruction and all register values in a 512-instruction circular buffer.

To enable or disable trace recording, select the Record trace command from the Commands menu to toggle instruction trace recording.

You may use trace recording with the MEASURE example.

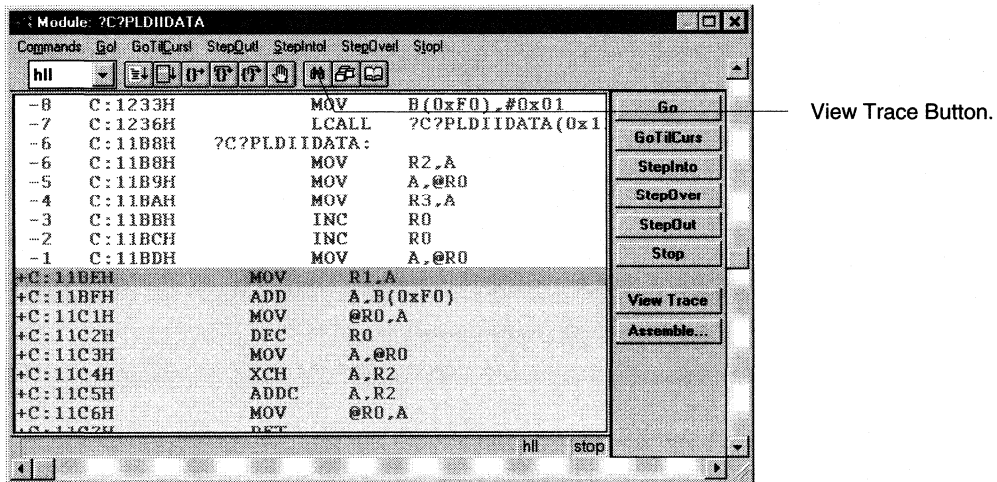


## 8

Enable trace recording, start running the MEASURE program (click on the Go button in the Debug window), and select the Serial window. MEASURE displays a menu and waits for input after displaying **Command**.

In the Serial window, enter **a** and MEASURE begins to display measurement values, the record time, two port values, and finally the analog input values.

Click on the Stop button in the Debug window. This halts program execution immediately. Click on the View Trace button to view the trace buffer.



The upper portion of the Debug window shows the trace history. The lower portion of the Debug window shows instructions starting from the current program counter. The program counter line is the delimiter between the trace history and instructions not yet executed.

The trace history lines begin with negative numbers. The newest trace buffer entry is -1. The oldest entry is -511. When the buffer overflows, the oldest entries are removed to make space for new entries.

You may scroll into the trace buffer using the keyboard or the mouse. The Register window shows the register contents for the selected instruction in the trace buffer.

### NOTE

*Program execution must be stopped before you can view the trace buffer.*

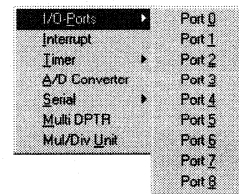
## Using On-Chip Peripherals

dScope provides several ways you can provide input to the on-chip peripherals used in your target program. You may use the dialog boxes available from the Peripheral menu to visually inspect and change the peripherals, you may use the Command window to assign I/O values to VTREG symbols (peripheral pin registers), and you may create user functions and signal functions in dScope to automatically write new values to the on-chip peripherals as your target program runs.

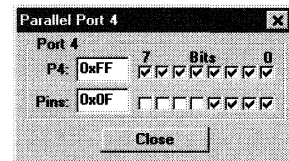
8

### Using Peripheral Dialog Boxes

When you load a CPU driver, dScope automatically attaches dialog boxes defined in the driver to the Peripheral menu in the dScope Main window. For the 80517.DLL driver, the Peripheral menu includes dialog boxes for: I/O Ports, Interrupts, Timers, A/D Converter, Serial Ports, Multiple Data Pointers, and High-Speed Arithmetic Unit.



You may change the values coming into Port 4 and Port 5 using the dialog boxes for those two ports. Each of these dialog boxes displays the values of the bits in the P4 or P5 SFRs as well as the values at the simulated pins on the 80517 device. To change the inputs to the 80517, change the values of the Pins. The Parallel Port 4 dialog box shows that the signals coming into the upper four bits of Port 4 (P4.7-P4.4) are low while the signals coming into the lower four bits (P4.3-P4.0) are high.



### Using VTREG Symbols

You may use the VTREG symbols defined by the CPU driver to change the signals coming into the simulated 80517. In the Command window, you may make assignments to the VTREG symbols just like variables and registers. For example, the commands change port values for Port 4 and channel 1 of the A/D converter.

```
PORT4=0x23
AIN1=3.3
```

```
set digital input PORT3 to 0x23.
set analog input AIN1 to 3.3 volts.
```

In the above example, PORT4 and AIN1 specify peripheral names (VTREG symbols) defined by the CPU driver. You may display a complete list of VTREG symbols using the **DIR** command in the Command window.

```
>DIR VTREG                                /* Output all PIN registers and values */
PORT0:  uchar, value = 0xFF
PORT1:  uchar, value = 0xFF
PORT2:  uchar, value = 0xFF
PORT3:  uchar, value = 0xFF
PORT4:  uchar, value = 0xFF
PORT5:  uchar, value = 0xFF
PORT6:  uchar, value = 0xFF
PORT7:  uchar, value = 0x00
PORT8:  uchar, value = 0x00
AIN0:   float, value = 4.5                /* Changed value */
AIN1:   float, value = 0
.
.
.
S0IN:   uint, value = 0xFFFF
S0OUT:  uint, value = 0xA
S1IN:   uint, value = 0x0
S1OUT:  uint, value = 0x0
VAGND:  float, value = 0
VAREF:  float, value = 5
```

Refer to “Appendix A. CPU Driver Files” on page 295 for more information about the VTREG symbols that are defined for each CPU driver.

## Using dScope User and Signal Functions

You may combine the use of VTREG symbols defined by the CPU driver and dScope user and signal function to create a sophisticated method of providing external input to your target programs. An example of a signal function using the analog inputs is provided in “Using dScope User and Signal Functions” on page 190. Also refer to “Chapter 10. Functions” on page 261 for more information about dScope functions.

## Using the Performance Analyzer

dScope lets you perform timing analysis of your applications using the integrated performance analyzer. You may specify an address range or a function for dScope to analyze. To prepare for timing analysis, halt execution of MEASURE and enter the following commands in the Command window.

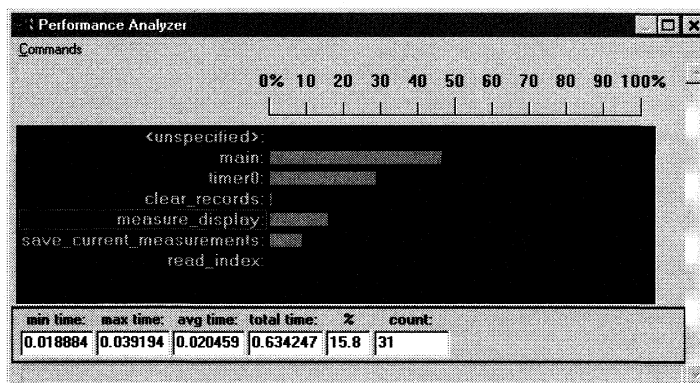
```
PA main
PA timer0
PA clear_records
PA measure_display
PA save_current_measurements
PA read_index
RESET PA                                     /* Initialize PA */
```

These commands create the performance analyzer address ranges for timing statistics. You may also create and view performance analyzer address ranges with the Performance Analyzer dialog box available from the Setup menu.

Perform the following steps to see the performance analyzer in action:

1. Open the Performance Analyzer window using the button on the tool bar. The display shows the ranges defined above. The **<unspecified>** line accumulates all execution time outside the defined ranges.
2. Reset dScope and start program execution by clicking on the Go button in the Debug window.
3. Select the Serial window and type **S Enter D Enter**.

The Performance Analyzer window shows a bar graph for each range. The bar graph dynamically updates and shows the percent of the time spent executing code in each range. Click on the range to see timing statistics for each individual range.



Refer to “Performance Analyzer Setup Dialog Box” on page 83 for more information.

## 8

## Using dScope User and Signal Functions

dScope user and signal functions let you expand the capabilities of dScope and make it easy for you to test parts of your program that interface to external hardware. Although a brief introduction to dScope functions is provided here, you should refer to “Chapter 10. Functions” on page 261 for more detailed information.

### NOTE

*The user and signal functions described below may be found in the MEASURE sample program directory.*

### User Functions

User functions are functions you create to immediately perform a specific task. These functions may be invoked at any time, even during program execution.

The following user function, **MyStatus**, displays the analog input values for analog channels 0 to 4.

```
/*-----*/
/* MyStatus shows analog input values          */
/*-----*/
FUNC void MyStatus (void) {
2:   printf ("=====\n");
3:   printf (" Analog-Input-0:  %f\n", ain0);
4:   printf (" Analog-Input-1:  %f\n", ain1);
5:   printf (" Analog-Input-2:  %f\n", ain2);
6:   printf (" Analog-Input-3:  %f\n", ain3);
7:   printf ("=====\n");
8: }
```

Use the Command window to define and enter a user function. Once the function is entered, it may be invoked using the following command line.

```
>MyStatus ()
```

The output displays in the Command window as follows:

```
=====
Analog-Input-0:  0.000000
Analog-Input-1:  0.000000
Analog-Input-2:  0.000000
Analog-Input-3:  0.000000
=====
```



## Signal Functions

Signal functions are a little more complex than user functions because they are functions that execute in the background while your target program runs. Signal functions are typically used to provide input to the on-chip peripherals that would normally come from your target hardware or another external device.

For the MEASURE program, one good use for a signal function would be to provide input to the A/D converter input of the 80517. The following signal function varies the input to analog input 0 from 0.0 volts to the maximum voltage you specify as an argument.

```

/*-----*/
/* Analog0() simulates analog input values given */
/* channel-0 (AIN0) of the 80C517 */
/*-----*/
SIGNAL void Analog0 (float limit) {
2:   float volts;
3:
4:   printf ("ANALOG0 (%f) ENTERED\n", limit);
5:   while (1) {           /* forever */
6:       volts = 0;
7:       while (volts <= limit) {
8:           ain0 = volts;    /* analog input-0 */
9:           twatch (30000); /* 30000 Cycles Time-Break */
10:          volts += 0.5;    /* increase voltage */
11:      }
12:      volts = limit-0.5;
13:      while (volts >= 0.5) {
14:          ain0 = volts;
15:          twatch (30000); /* 30000 Cycles Time-Break */
16:          volts -= 0.5;    /* decrease voltage */
17:      }
18:  }
19: }

```

You may invoke this function as follows:

```
>Analog0 (5.0) /* invoke with limit at 5V */
```

The function responds with a message indicating that it is running.

```
ANALOG0 (5.000000) ENTERED
```

Now, when the MEASURE program runs, the **Analog0** signal function continuously changes the input on analog input 0.

Click on the Go button to begin executing the MEASURE program. Then, in the Serial window enter **d**. The output displayed in the Serial window shows the changing voltages produced by the **Analog0** signal function.

## 8

## Exiting dScope

To exit dScope, you may...

- Select the Exit command from the File menu,
- Click on the dScope Main Window Close Button (in Windows 95),
- Select the Close command from the dScope Main Window Control Menu (in Windows 3.1),
- Double-clicking on the dScope Main Window Control Menu box,
- Enter the Exit command in the Command window. For example:

```
exit
```

---

### NOTE

*dScope requires that you stop simulation or target execution of your program before you exit. Click on the Stop button or type **Ctrl+C** in the Command window.*

---

## Chapter 9. Commands

dScope supports a number of commands you may enter in the Command window. Commands may be separated into categories that describe the common function of the commands.

Category	Description
<b>Breakpoint Commands</b>	These commands let you create and delete breakpoints. You use breakpoints to halt program execution or to execute dScope commands or user functions when a particular instruction is reached.
<b>General Commands</b>	These commands let you perform a number of miscellaneous debugging operations.
<b>Memory Commands</b>	These commands let you display and change the contents of memory.
<b>Program Commands</b>	These commands let you execute your target program and analyze its performance.
<b>Watchpoint Commands</b>	These commands let you define memory areas and symbols to watch in the Watch window.

This chapter lists the commands by category and provides an alphabetical reference section that provides details about each command.

---

**NOTE**

*Use the underlined characters in the command names to enter commands. For example, the **WATCHSET** command must be entered as **WS**.*

---

## Memory Commands

The following memory commands let you display and alter memory contents.

Command	Description
<b><u>ASM</u></b>	Assembles in-line code.
<b><u>DEFINE</u></b>	Defines typed symbols that you may use with dScope functions.
<b><u>DISPLAY</u></b>	Display the contents of memory.
<b><u>ENTER</u></b>	Enters values into a specified memory area.
<b><u>EVALUATE</u></b>	Evaluates an expression and outputs the results.
<b><u>MAP</u></b>	Specifies access parameters for memory areas.
<b><u>OBJECT</u></b>	Displays values for structures and arrays.
<b><u>UNASSEMBLE</u></b>	Disassembles program memory.

## Watchpoint Commands

The following memory commands let you watch variables in your target program.

Command	Description
<b><u>WATCHKILL</u></b>	Removes a watch variable from the Watch window.
<b><u>WATCHSET</u></b>	Adds a watch variable to the Watch window.

## Program Commands

Program commands let you run code and step through your program one instruction at a time. Additionally, dScope provides a sophisticated performance analyzer that lets you easily determine hot-spots in your target application.

Command	Description
<b>Ctrl+C</b>	Stops program execution.
<b>Esc</b>	Stops program execution.
<b><u>G</u>O</b>	Starts program execution.
<b><u>Performance</u>Analyze</b>	Initializes the built-in performance analyzer.
<b><u>P</u>STEP</b>	Steps over instructions but does not step into procedures or functions.
<b><u>Q</u>STEP</b>	Steps out of the current function.
<b><u>I</u>STEP</b>	Steps over instructions and into functions.

## Breakpoint Commands

dScope provides breakpoints you may use to conditionally halt the execution of your target program. Breakpoints can be set on read operations, write operations and execution operations.

Command	Description
<b><u>BREAKDISABLE</u></b>	Disables one or more breakpoints.
<b><u>BREAKENABLE</u></b>	Enables one or more breakpoints.
<b><u>BREAKKILL</u></b>	Removes one or more breakpoints from the breakpoint list.
<b><u>BREAKLIST</u></b>	Lists the current breakpoints.
<b><u>BREAKSET</u></b>	Adds a breakpoint expression to the list of breakpoints.

9

## General Commands

The following general commands do not belong in any other particular command group. They are included in dScope to make debugging easier and more convenient.

Command	Description
<b><u>ASSIGN</u></b>	Assigns input and output sources for the Serial window.
<b><u>DEFINE BUTTON</u></b>	Creates a Toolbox button.
<b><u>DIR</u></b>	Generates a directory of symbol names.
<b><u>DOS</u></b>	Opens a DOS box.
<b><u>EXIT</u></b>	Exits dScope.
<b><u>INCLUDE</u></b>	Reads and executes the commands in a command file.
<b><u>KILL</u></b>	Deletes dScope functions and Toolbox buttons.
<b><u>LOAD</u></b>	Loads CPU drivers, object modules, and HEX files.
<b><u>LOG</u></b>	Creates log files, queries log status, and closes log files for the Debug window.
<b><u>MODE</u></b>	Sets the baud rate, parity, and number of stop bits for PC COM ports.
<b><u>RESET</u></b>	Resets dScope, resets memory map assignments, and resets predefined variables.
<b><u>SAVE</u></b>	Saves a memory range in an Intel HEX386 file.
<b><u>SCOPE</u></b>	Displays address assignments of modules and functions of a target program.
<b><u>SET</u></b>	Sets the string value for predefined variable.
<b><u>SETMODULE</u></b>	Assigns a source file to a module for PL/M-51 programs.
<b><u>SIGNAL</u></b>	Displays signal function status and removes active signal functions.
<b><u>SLOG</u></b>	Creates log files, queries log status, and closes log files for the Serial window.

## Command List

All commands available in dScope are listed on the following pages. A command summary and description are listed at the top of each page. For example:

Syntax	Description
<b><u>B</u>REAKLIST</b>	Lists all breakpoints.

9

Command abbreviations are indicated by underlined characters in the command name. In the above command summary, the abbreviation for the **BREAKLIST** command is **BL**.

Following the command summary is a complete command description where details of command usage and references to other commands are provided.

An example section follows the command description. Numerous examples are provided for each command. The following example section is taken from the **BREAKLIST** command.

### Example

```
>BL /* List current breakpoints */
0: (E C: 0xFF01EF) 'main', CNT=1, enabled
1: (E C: 0xFF006A) 'timer0', CNT=10, enabled
   exec ("MyRegs()")
2: (C) 'sindex == 8', CNT=1, enabled
3: (C) 'save_record[5].time.sec > 5', CNT=3, enabled
4: (A RD 0x000037) 'READ interval.min == 3', CNT=1, enabled
5: (A WR 0x000034) 'WRITE savefirst==5 && acc==0x12', CNT=1, enabled
```

## ASM

Syntax	Description
<b>ASM</b>	Displays the address where in-line assembly instructions are stored.
<b>ASM address</b>	Sets the in-line assembly address to <b>address</b> .
<b>ASM instruction</b>	Assembles the specified <b>instruction</b> and stores the resulting opcode in memory at the current in-line assembly address. The address is increased by the number of bytes in the instruction.

The **ASM** command displays or sets the current assembly address and lets you enter assembly instructions. When instructions are entered, the resulting opcode is stored in code memory. You may use the in-line assembler to correct mistakes or to make temporary changes to the target program you are debugging.

The in-line assembler accepts mnemonics for assembly instructions based on the CPU driver loaded.

- If an MCS<sup>®</sup> 51 driver is loaded, 8051 instructions are supported.
- If an MCS<sup>®</sup> 251 driver is loaded, 80251 instructions are supported.
- If the 80166 driver is loaded, 166 instructions are supported.
- If the 80167 driver is loaded, 167 instructions are supported.

Refer to “Inline Assembler Dialog Box” on page 76 for information about assembling in-line code using the Inline Assembler dialog box.

### Example

```
>ASM 0xFF0000          /* set assemble address to C:0x0000 (8051 & 251) */
>ASM mov a,#12
>ASM mov r0,#0x20
>ASM movx @r0,A
>ASM inc r0
>ASM movx @r0,A
>ASM jmp C:0x8000

>ASM C:0020H          /* set assemble address to C:0x0020 (8051 & 251) */
>ASM CLR A
```



## ASSIGN

Syntax	Description
<b>ASSIGN</b>	Display serial channel I/O assignments.
<b>ASSIGN</b> <i>channel</i> < <i>inreg</i> > <i>outreg</i>	Change serial channel I/O assignments.

The **ASSIGN** command lets you display and change the input and output sources for the specified serial channel.

The following table shows the supported *channels* and their default input register (*inreg*) and output register (*outreg*) settings.

Channel	Default Inreg	Default Outreg	Description
<b>COM1</b>	None	None	COM port 1 on the PC.
<b>COM2</b>	None	None	COM port 2 on the PC.
<b>COM3</b>	None	None	COM port 3 on the PC.
<b>COM4</b>	None	None	COM port 4 on the PC.
<b>WIN</b>	<b>SIN</b> or <b>S1IN</b>	<b>SOUT</b> or <b>S1OUT</b>	The Serial window.

Currently, dScope for windows supports the following channels: **WIN**, **COM1**, **COM2**, **COM3**, and **COM4**. The **WIN** channel is the Serial window. The **COMx** channels represent the serial communication ports of the PC which you may use to communicate with the simulated CPU and your target program. You may set the parameters for the PC serial ports using the **MODE** command. Refer to “MODE” on page 237 for more information.

The names of the input register and output register are defined by the CPU driver. Refer to “Appendix A. CPU Driver Files” on page 295 for more information about VTREGs supported by each CPU driver. Refer to “CPU Driver Symbols” on page 90 for more information about using VTREGs.

---

### NOTE

*You may not use the **ASSIGN** command if a CPU driver is not loaded.*

---



### Example

```
>LOAD 80517.DLL          /* LOAD a CPU driver */
>ASSIGN                  /* Display serial assignment */
  WIN: <S0IN >S0OUT       /* S0IN provides serial input & */
                          /* S0OUT provides serial output */
>
>ASSIGN WIN <S1IN >S1OUT  /* Assign S1IN & S1OUT to the Serial window */
>ASSIGN                  /* And display the assignment */
  WIN: <S1IN >S1OUT
>ASSIGN COM1 < SIN > SOUT /* Assign SIN & SOUT to COM1 */
```

### NOTE

*The ASSIGN command may not be used with the tScope target debugger.*

## BREAKDISABLE

Syntax	Description
<b>BREAKDISABLE</b> <i>number</i> [ , <i>number...</i> ]	Disables, but does not delete, the specified breakpoints. Numbers are assigned to breakpoints as they are defined.
<b>BREAKDISABLE</b> *	Disables all breakpoints.

# 9

The **BREAKDISABLE** command disables a previously defined breakpoint. dScope normally halts execution or executes a specified command when a breakpoint is reached. Disabling a breakpoint leaves the definition in the breakpoint list, but causes dScope to ignore the breakpoint while executing the target program.

Refer to “Breakpoints Dialog Box” on page 78 for information about setting breakpoints using the Breakpoint dialog box.

### Example

```
>BL                                /* List breakpoints */

0: (E C: 0xFF01EF) 'main', CNT=1, enabled
1: (E C: 0xFF006A) 'timer0', CNT=10, enabled
   exec ("MyRegs()")

>BD 0                              /* Disable breakpoint 0 */
>BD *                              /* Disable all breakpoints */
>BL                                /* List breakpoints */

0: (E C: 0xFF01EF) 'main', CNT=1, disabled
1: (E C: 0xFF006A) 'timer0', CNT=10, disabled
   exec ("MyRegs()")
```

## BREAKENABLE

Syntax	Description
<b>BREAKENABLE</b> <i>number</i> [, <i>number...</i> ]	Enables the specified breakpoints. Numbers are assigned to breakpoints as they are defined.
<b>BREAKENABLE</b> *	Enables all breakpoints.

The **BREAKENABLE** command enables a previously defined breakpoint that was disabled with the **BREAKDISABLE** command. dScope normally halts execution or executes a specified command when a breakpoint is reached.

Refer to “Breakpoints Dialog Box” on page 78 for information about setting breakpoints using the Breakpoint dialog box.

### Example

```
>BE 0,1          /* Enable breakpoints 0 and 1 */
>BE *            /* Enable all breakpoints */
>BL              /* List current breakpoints */

0: (E C: 0xFF01EF) 'main', CNT=1, enabled
1: (E C: 0xFF006A) 'timer0', CNT=10, enabled
   exec ("MyRegs()")
```

## BREAKKILL

Syntax	Description
<b>BREAKKILL</b> <i>number</i> [ , <i>number...</i> ]	Removes the specified breakpoints. Numbers are assigned to breakpoints as they are defined.
<b>BREAKKILL</b> *	Removes all breakpoints.

# 9

The **BREAKKILL** command removes a breakpoint that was previously defined with the **BREAKSET** command.

Refer to “Breakpoints Dialog Box” on page 78 for information about setting breakpoints using the Breakpoint dialog box.

### Example

```
>BK 0,1          /* Remove breakpoints 0 and 1 */
>BK *            /* Remove all breakpoints */
```

## BREAKLIST

Syntax	Description
<b>BREAKLIST</b>	Lists all breakpoints.

The **BREAKLIST** command lists all breakpoints. Breakpoints are listed one per line using the following format:

```
number: (type) 'expression', CNT=count, enable_flag
        exec ("command")
```

where:

- number* is the index for the breakpoint. dScope assigns this number when a breakpoint is created. Use it when enabling, disabling, or removing a breakpoint with the **BREAKENABLE**, **BREAKDISABLE**, or **BREAKKILL** commands,
- type* is the breakpoint type which may be an execution breakpoint (**E**: followed by the address), a conditional breakpoint (**C**), or an access breakpoint (**A**: followed by **RD** for read, **WR** for write, or **RW** for read/write followed by the address).
- expression* is the original text of the breakpoint definition.
- count* is the pass counter for the breakpoint. If *count* has a value of 2, dScope halts program execution (or executes the specified command) when the breakpoint is reached the second time.
- enable\_flag* displays **enabled** for an enabled breakpoint or **disabled** for a disabled breakpoint.
- command* is the command to execute when the breakpoint is reached.

Refer to “Breakpoints Dialog Box” on page 78 for information about setting breakpoints using the Breakpoint dialog box.

### Example

```
>BL /* List current breakpoints
*/
0: (E C: 0xFF01EF) 'main', CNT=1, enabled
1: (E C: 0xFF006A) 'timer0', CNT=10, enabled
   exec ("MyRegs()")
2: (C) 'sindex == 8', CNT=1, enabled
3: (C) 'save_record[5].time.sec > 5', CNT=3, enabled
4: (A RD 0x000037) 'READ interval.min == 3', CNT=1, enabled
5: (A WR 0x000034) 'WRITE savefirst==5 && acc==0x12', CNT=1, enabled
```

## BREAKSET

Syntax	Description
<b>BREAKSET</b> <i>exp</i> [, <i>cnt</i> [, " <i>cmd</i> "]]	Sets an execution or conditional breakpoint.
<b>BREAKSET READ</b> <i>exp</i> [, <i>cnt</i> [, " <i>cmd</i> "]]	Sets a read access breakpoint.
<b>BREAKSET WRITE</b> <i>exp</i> [, <i>cnt</i> [, " <i>cmd</i> "]]	Sets a write access breakpoint.
<b>BREAKSET READWRITE</b> <i>exp</i> [, <i>cnt</i> [, " <i>cmd</i> "]]	Sets a read/write access breakpoint.

The **BREAKSET** command sets a breakpoint for the specified expression (*exp*). Breakpoints are program addresses or expressions that, when true, halt execution of your target program or execute a specified command.

You may specify the following parameters for a breakpoint definition:

<i>exp</i>	is an address specification or an expression that dScope evaluates during execution.
<i>cnt</i>	is an expression that determines the number of times a breakpoint condition is met before the target program halts or the specified command is executed. The default count value is 1.
<i>cmd</i>	is a command string that specifies a dScope command to execute when the breakpoint occurs. When no command is specified, dScope halts program execution when a breakpoint occurs. If a command is specified, dScope only executes the command and the target program does not stop. The command may specify a dScope user or signal function. Refer to "System Variables" on page 88 for information about the <b>_break_</b> variable and halting program execution.

dScope recognizes three types of breakpoints: execution breakpoints, conditional breakpoints, and access breakpoints. dScope categorizes each breakpoint using the following rules:

1. When a memory access mode (READ, WRITE, or READWRITE) is specified, the breakpoint is an access breakpoint.
2. When the specified expression is a simple address, the breakpoint is an execution breakpoint.
3. When the specified expression cannot be reduced to an address, the breakpoint is a conditional breakpoint.

The breakpoint types are described below.

## Execution Breakpoints

Execution breakpoints halt program execution or execute a command when the specified code address is reached. Execution speed is not affected by execution breakpoints.

The code address specified must be the first byte of an instruction. If you set an execution breakpoint on the second or third byte of an instruction, the breakpoint never occurs.

An execution breakpoint for a code address may be specified only once. Multiple definitions are not allowed.

## Conditional Breakpoints

Conditional breakpoints halt program execution or execute a command when a specified conditional expression is true. The conditional expression is recalculated after each assembly instruction. Program execution speed may slow down considerably depending on the complexity of the conditional expression. Execution speed is also affected by the number of conditional breakpoints.

Conditional breakpoints are the most flexible type of breakpoint since optional conditions may be calculated by the expression.

## Access Breakpoints

Memory access breakpoints halt program execution or execute a command when a specific address is read, written, or accessed or when a specific address is read or written with a certain value. Program execution speed is not dramatically affected by memory access breakpoints since access expressions are evaluated only with the specified access event occurs.

# 9

Expressions specified for memory access breakpoints must reduce to a memory address and memory type. The following rules apply to memory access breakpoints:

1. Expressions used in a memory access breakpoint must have unique memory types. Expressions that include multiple objects are not allowed.
2. Only a few operators (&, &&, <, <=, >, >=, ==, and !=) are allowed in a memory access breakpoint expression.

For example, the following breakpoint:

```
BS WRITE timer.sec /* Valid expression */
```

is valid while the following breakpoint:

```
BS WRITE timer.sec + i0 /* Invalid expression */
```

is invalid because adding two values (timer.sec and i0) does not result in a memory type.

---

### NOTE

When listed using the **BREAKLIST** command, memory access breakpoints are preceded by **READ**, **WRITE**, or **READWRITE**.

---

Refer to “Breakpoints Dialog Box” on page 78 for information about setting breakpoints using the Breakpoint dialog box.



## Example

The following example sets an execution breakpoint on the address of the *main* function.

```
>BS main
```

The following example sets an execution breakpoint on the address of the *timer0* function. The breakpoint occurs and the command “MyRegs()” is executed only after the 10<sup>th</sup> invocation of *timer0*. Program execution continues after the command executes.

```
>BS timer0,10,"MyRegs()"
```

The following example sets a conditional breakpoint on the *sindex* symbol. When *sindex* is equal to 8, program execution halts.

```
>BS sindex == 8
```

The following example sets a conditional breakpoint on the *save\_record* array. Program execution halts on the third time *save\_record[5].time.sec* is greater than 3.

```
>BS save_record[5].time.sec > 5, 3
```

The following example sets a memory access breakpoint on READ accesses of the *interval.min* symbol. Program execution halts when the *min* element of the *interval* structure or union is 3.

```
>BS READ interval.min == 3
```

The following example sets a memory access breakpoint on WRITE accesses to the *savefirst* symbol. Program execution halts when *savefirst* is 5 and the accumulator (*acc*) is 0x12 after *savefirst* is written.

```
>BS WRITE savefirst == 5 && acc == 0x12
```

## Ctrl+C

When you type **Ctrl+C** in the Command window, dScope stops running your target program. After the target program execution stops, the Register window, Watch window, Debug window, and other windows are updated to reflect the new CPU status.

---

### **NOTE**

*You may not use **Ctrl+C** with the `exec` predefined function.*

---

# DEFINE

Syntax	Description
<b>DEFINE</b> <i>type identifier</i>	Defines a symbol named <i>identifier</i> with the specified <i>type</i> .

The **DEFINE** command lets you create a symbol with a type that can be used to hold a value. Symbols created this way may be used to hold return values for dScope functions (refer to “Chapter 10. Functions” on page 261) or to specify input to dScope functions.

Symbols created with the **DEFINE** command are not placed in the memory space of the simulated or target CPU. They are just symbolic names for values of a specified type. A symbol created by **DEFINE** may be used just like any other public symbol.

The possible symbol *types* are listed in the following table.

Type	Description
<b>BIT</b>	A single bit ( <b>bit</b> ).
<b>BYTE</b>	An unsigned character ( <b>unsigned char</b> ).
<b>CHAR</b>	A signed character ( <b>signed char</b> ).
<b>DWORD</b>	An unsigned long integer ( <b>unsigned long</b> ).
<b>FLOAT</b>	A floating-point number ( <b>float</b> ).
<b>INT</b>	A signed integer ( <b>signed int</b> ).
<b>LONG</b>	A signed long integer ( <b>signed long</b> ).
<b>REAL</b>	A floating-point number ( <b>float</b> ).
<b>UCHAR</b>	An unsigned character ( <b>unsigned char</b> ).
<b>UINT</b>	An unsigned integer ( <b>unsigned int</b> ).
<b>ULONG</b>	A signed long integer ( <b>unsigned long</b> ).
<b>WORD</b>	An unsigned integer ( <b>unsigned int</b> ).

The *identifier* is the name of the symbol. It must conform to the rules for variables or symbols.

## Example

```
>DEFINE BYTE TmpByte          /* define TmpByte to be a byte value */
>DEFINE FLOAT TmpFloat        /* define TmpFloat to be a float value */

>TmpFloat = 3.14159           /* give TmpFloat a value */
>TmpFloat                     /* display the value of TmpFloat */
3.14159
```

## DEFINE BUTTON

Syntax	Description
<b>DEFINE BUTTON</b> " <i>label</i> ", " <i>cmd</i> "	Define a Toolbox command button. Refer to "Toolbox Window" on page 67 for more information.

Use this command to add a button to the Toolbox window.

*label* is the name to assign to the button.

*cmd* is the dScope command or commands to assign to the button. This command is executed when the button is clicked.

### Example

```
>DEFINE BUTTON "clr dptr", "dptr=0"
>DEFINE BUTTON "show main()", "u main"
>DEFINE BUTTON "show r7", "printf (\"R7=%02XH\n\",R7)"
```

### NOTE

*The printf command defined in the last button definition shown above introduces nested strings. The double quote characters (") of printf's format string must be escaped (\") to avoid syntax errors.*

When a button is defined, it is immediately added to the Toolbox window. Each button receives a button number which is displayed in the Toolbox window. This number is used to remove specific buttons from the Toolbox window. Refer to "KILL" on page 225 for more information about removing Toolbox buttons.

## DIR

Syntax	Description
<b><u>DIR</u></b>	Displays symbol names for the current module.
<b><u>DIR</u> \module</b>	Displays symbols of <i>module</i> .
<b><u>DIR</u> \module LINE</b>	Displays line number information for the current or specified module or function.
<b><u>DIR</u> \module\func LINE</b>	Displays symbols of function <i>func</i> contained in <i>module</i> .
<b><u>DIR</u> BFUNC</b>	Displays the names of all predefined dScope functions.
<b><u>DIR</u> DEFSYM</b>	Displays the names of symbols created with the <b>DEFINE</b> command.
<b><u>DIR</u> FUNC</b>	Displays the names of all dScope functions.
<b><u>DIR</u> LINE</b>	Displays line number information for the current module.
<b><u>DIR</u> MODULE</b>	Displays the names of modules in your target program.
<b><u>DIR</u> PUBLIC</b>	Displays the names of all global symbols.
<b><u>DIR</u> SIGNAL</b>	Displays the names of all user-defined signal functions.
<b><u>DIR</u> UFUNC</b>	Displays the names of all user-defined functions.
<b><u>DIR</u> VTREG</b>	Displays the names of all CPU-pin registers that are supported by the CPU driver.

Use the **DIR** command to display various types of symbols. When **DIR** is invoked with no additional arguments, the symbol names of the current module display. The current module is the module whose address space is indicated by the program counter (\$). dScope determines the address areas assigned to the module when the target program loads.

dScope maintains various internal symbol tables whose contents can be displayed using the **DIR** command with its various options.

DIR Command Option	Description
<b>\module</b> <b>\modulefunc</b>	Displays the symbols of the specified module or function are output.
<b>BFUNC</b>	Displays the names of predefined dScope functions. These functions are always available and cannot be deleted or redefined. Refer to "Predefined Functions" on page 266 for more information.
<b>DEFSYM</b>	Displays the symbols created using the <b>DEFINE</b> command. Refer to "DEFINE" on page 209 for more information.  Using this option, the output of the symbols created by the ' <b>DEFINE &lt;type&gt;&lt;name&gt;</b> ' command is permitted.
<b>FUNC</b>	Displays the names and prototypes for all currently defined dScope functions. This includes: predefined functions, user-defined functions, and signal functions.  Note that dScope functions are not the same as functions in your target program. Refer to "Chapter 10. Functions" on page 261 for more information.
<b>LINE</b> <b>\module LINE</b> <b>\modulefunc LINE</b>	Displays line numbers for the current module, the specified module, or the specified function.
<b>MODULE</b>	Displays the names of all modules in your target program. The target program must be loaded.
<b>PUBLIC</b>	Displays all global symbol names. These objects have the attribute <b>PUBLIC</b> in assembly language or PL/M51 modules. Non-static C variables are also included in this group.
<b>SIGNAL</b>	Displays the names of signal functions. Signal functions are user functions that process in the background. These are used to produce signal forms for the port inputs. Refer to "Signal Functions" on page 279 for more information.
<b>UFUNC</b>	Displays the names of dScope user functions. User functions are those function defined by the user. Refer to "User Functions" on page 276 for more information.
<b>VTREG</b>	Displays the names of CPU-pin registers supported by the currently loaded CPU driver. These are only available if a CPU driver is loaded. Refer to "Appendix A. CPU Driver Files" on page 295 for more information about the registers available for each CPU driver. Refer to "CPU Driver Symbols" on page 90 for more information about using VTREGs.

### Example

The following example were created using the 80517.DLL CPU driver along with the MEASURE example program.

## DIR MODULE

```
>DIR MODULE      /* all module names */
MEASURE
MCOMMAND
GETLINE
?C_FPADD
?C_FPMUL
...
```

## DIR \module

```
>DIR \MEASURE    /* module 'MEASURE' */
MODULE: MEASURE
C:0x000000 . . . . _ICE_DUMMY_ . . uint
FUNCTION: {CvtB} RANGE: 0xFF03B7-0xFF07E5
C:0x000000 . . . . _ICE_DUMMY_ . . uint
FUNCTION: {CvtB} RANGE: 0xFF000B-0xFF000D
C:0x000000 . . . . _ICE_DUMMY_ . . uint
FUNCTION: SAVE_CURRENT_MEASUREMENTS RANGE: 0xFF000E-0xFF0069
FUNCTION: TIMER0 RANGE: 0xFF006A-0xFF0135
D:0x00000F . . . . i . . uchar
FUNCTION: READ_INDEX RANGE: 0xFF0136-0xFF01BF
D:0x00003F . . . . buffer . . ptr to char
D:0x000042 . . . . index . . int
D:0x000007 . . . . args . . uchar
FUNCTION: CLEAR_RECORDS RANGE: 0xFF01C0-0xFF01EE
D:0x000006 . . . . idx . . uint
FUNCTION: MAIN RANGE: 0xFF01EF-0xFF03B6
I:0x000067 . . . . cmdbuf . . array[15] of char
D:0x00003C . . . . i . . uchar
D:0x00003D . . . . idx . . uint
```

## DIR /module LINE

```
>DIR \MEASURE LINE /* Lines of module 'MEASURE' */
MODULE: MEASURE
C:0x000E . . . . #87
C:0x000E . . . . #88
C:0x003A . . . . #89
C:0x0049 . . . . #90
...
C:0x03B6 . . . . #291
C:0x03B6 . . . . #292
```



## DIR PUBLIC

```
>DIR PUBLIC          /* all PUBLIC symbols */
B:0x000640 . . . . T2I0 . . bit
B:0x000641 . . . . T2I1 . . bit
...
D:0x000023 . . . . current . . struct mrec
C:0x0007CD . . . . ERROR . . array[16] of char
X:0x004000 . . . . save_record . . array[744] of struct mrec
C:0x00000E . . . . save_current_measurements . . void-function
C:0x0001EF . . . . main . . void-function
C:0x00047E . . . . menu . . array[847] of char
D:0x000030 . . . . setinterval . . struct interval
...
B:0x000601 . . . . IEX2 . . bit
B:0x000600 . . . . IADC . . bit
```

## DIR VTREG

```
>DIR VTREG          /* Show Pin-Registers and Values */
PORT0:  uchar, value = 0xFF
PORT1:  uchar, value = 0xFF
PORT2:  uchar, value = 0xFF
PORT3:  uchar, value = 0xFF
PORT4:  uchar, value = 0xFF
PORT5:  uchar, value = 0xFF
PORT6:  uchar, value = 0xFF
PORT7:  uchar, value = 0x00
PORT8:  uchar, value = 0x00
AIN0:   float, value = 0
AIN1:   float, value = 0
AIN2:   float, value = 0
AIN3:   float, value = 0
AIN4:   float, value = 0
AIN5:   float, value = 0
AIN6:   float, value = 0
AIN7:   float, value = 0
AIN8:   float, value = 0
AIN9:   float, value = 0
AIN10:  float, value = 0
AIN11:  float, value = 0
SOIN:   uint, value = 0x0000
SOOUT:  uint, value = 0x0000
SIIN:   uint, value = 0x0000
SIOUT:  uint, value = 0x0000
VAGND:  float, value = 0
VAREF:  float, value = 5
XTAL:   ulong, value = 0xB71B00
PE_SW:  uchar, value = 0x00
STIME:  uchar, value = 0x00
```

## DIR

```
>$ = MAIN          /* set current execution point to main() */
>DIR              /* now, the main() symbols are preselected */
FUNCTION: MAIN RANGE: 0xFF01EF-0xFF03B6
I:0x000067 . . . . cmdbuf . . array[15] of char
D:0x00003C . . . . i . . uchar
D:0x00003D . . . . idx . . uint
```



## DIR DEFSYM

```
>DIR DEFSYM /* those created by 'DEFINE <type> <name>' */
            word00: uint, value = 0x0000
            byte00: uchar, value = 0x00
            dword00: ulong, value = 0x0
            float00: float, value = 0
```

## DIR FUNC

```
>DIR FUNC /* predefined dScope functions */
predef'd: void MEMSET (ulong, ulong, uchar)
predef'd: void TWATCH (ulong)
predef'd: int RAND (uint)
predef'd: float GETFLOAT (char *)
predef'd: long GETLONG (char *)
predef'd: int GETINT (char *)
predef'd: void EXEC (char *)
predef'd: void PRINTF (char *, ...)
```

## DISPLAY

Syntax	Description
<b>DISPLAY</b> [ <i>startaddr</i> [, <i>endaddr</i> ]]	Display memory from <i>startaddr</i> to <i>endaddr</i> in the Memory window (if open) or in the Command window.

The **DISPLAY** command displays a range of memory in the Memory window (if it is open) or in the Command window. Memory areas are displayed in HEX and in ASCII. Refer to “Memory Window” on page 54 for more information about displaying memory ranges in the Memory window.

Displayed memory lines consists of the address of the first byte, a maximum of 16 HEX bytes, and ASCII characters for each displayed HEX byte. Dots (“.”) display for non-printing ASCII values.

If address specifications (*startaddr* and *endaddr*) are omitted, memory displays starting from the end of a prior **DISPLAY** command. If no previous **DISPLAY** command was entered, memory display starts at code address 0x0000.

If address specifications are included, memory displays from the starting address (*startaddr*) to the ending address (*endaddr*) or until the next 256-byte page boundary.

Starting addresses may be prefixed with the memory space specifiers described in the following table. For example, X:0x0000 refers to **XDATA** address 0x0000.

Specifier	Description
<b>B</b>	Bit-addressable RAM memory ( <b>BIT</b> ).
<b>C</b>	Code memory ( <b>CODE</b> ).
<b>CO</b>	Memory range for constants (MCS <sup>®</sup> 251 <b>CONST</b> ).
<b>D</b>	Internal directly-addressable RAM memory of the 8051 ( <b>DATA</b> ).
<b>EB</b>	Extended bit-addressable RAM memory (MCS <sup>®</sup> 251 <b>EBIT</b> ).
<b>ED</b>	Extended data RAM memory (MCS <sup>®</sup> 251 <b>EDATA</b> ).
<b>HC</b>	Huge memory range for constants (MCS <sup>®</sup> 251 <b>HCONST</b> ).
<b>I</b>	Internal indirectly-addressable RAM memory of the 8051 ( <b>IDATA</b> ).
<b>P</b>	Peripheral memory (simulated CPU pins <b>VTREGS</b> ).
<b>X</b>	External RAM memory ( <b>XDATA</b> ).



## DOS

Syntax	Description
!	Open a DOS box. Enter <b>EXIT</b> at the COD command prompt to close the DOS box.

The exclamation point (“!”) opens a DOS box. You may proceed to execute any valid DOS command in the DOS box. Enter **EXIT** at the command prompt to close the DOS box.

### Example

```
!
```

---

### NOTE

*The exclamation point (“!”) is a valid C operator. For this reason, expressions that begin with an exclamation point must be enclosed in parentheses. For example, **!dptr** opens a DOS box while **(!dptr)** displays the results of the expression.*

---

# ENTER

Syntax	Description
<b>ENTER</b> <i>type address = expr</i> [, <i>expr ...</i> ]	Changes the contents of memory starting at <b>address</b> using expressions ( <b>expr</b> ) of the specified data <b>type</b> .

The **ENTER** command lets you change the contents of memory starting at the specified *address*. You may enter the following data types.

Type	Description
BIT	A single bit ( <b>bit</b> ).
BYTE	An unsigned character ( <b>unsigned char</b> ).
CHAR	A signed character ( <b>signed char</b> ).
DWORD	An unsigned long integer ( <b>unsigned long</b> ).
FLOAT	A floating-point number ( <b>float</b> ).
INT	A signed integer ( <b>signed int</b> ).
LONG	A signed long integer ( <b>signed long</b> ).
PTR	C51 generic 3-byte pointer.
REAL	A floating-point number ( <b>float</b> ).
UCHAR	An unsigned character ( <b>unsigned char</b> ).
UINT	An unsigned integer ( <b>unsigned int</b> ).
ULONG	A signed long integer ( <b>unsigned long</b> ).
WORD	An unsigned integer ( <b>unsigned int</b> ).

You may specify multiple expressions, separated by commas (“,”), which are converted into the specified data types and finally stored in consecutive addresses in memory.

### Example

```
>E CHAR x:0 = 1,2,"-dScope-" /* Enter Characters */
>D x:0
X:0000 01 02 2D 64 53 63 6F 70 65 2D 00 00 00 00 00 00 ..-dScope-
.....

>E FLOAT x:0x2000 = 3,4,15.2,0.33 /* Enter Float */
>EP x:0x1000 = main,timer0, &current /* Enter Pointer */
```

## Esc

When you type **Esc** in the Command window, dScope stops running your target program. After the target program execution stops, the Register window, Watch window, Debug window, and other windows are updated to reflect the new CPU status.

---

### **NOTE**

*You may not use **Esc** with the `exec` predefined function.*

---

## EVALUATE

Syntax	Description
<b>EVALUATE</b> <i>expression</i>	Displays the result of the expression in decimal, octal, HEX, and ASCII.

The **EVALUATE** command calculates the specified expression and outputs the result in decimal, octal, HEX and, in ASCII format. Expressions entered without the **EVALUATE** command display only in the current number base selected by the **RADIX** system variable. Refer to “System Variables” on page 88 for more information. The expression may contain several subexpressions separated by commas.

### Example

```
>eval -1
16777215T 77777777Q 0xFFFF '....'

>eval intcycle
0T 0Q 0x0 '....'

>intcycle = 0x12
>eval intcycle
18T 22Q 0x12 '....'

>eval 'a'+'b'+'c'
294T 446Q 0x126 '...&'

>eval main
16712175T 77600757Q 0xFF01EF '....'

>eval save_record[1].time
81931T 240013Q 0x1400B '...@.'

>eval save_record[1].time.sec
0T 0Q 0x0 '....'

>save_record[1].time.sec = 1
>eval save_record[1].time.sec
1T 1Q 0x1 '....'

>eval save_record[1].time.sec = 0
0T 0Q 0x0 '....'
```

## EXIT

Syntax	Description
<b>EXIT</b>	Exits the dScope Debugger.

The **EXIT** command closes all open files and exits the dScope debugger.

This command may not be invoked from a command file loaded by the **INCLUDE** command. It is not allowed as an argument to the **exec** function.

The **EXIT** command is canceled if dScope is still executing the target program or if a dScope function is still active. If this happens, stop the target program or kill the active function and reenter the **EXIT** command.



## GO

Syntax	Description
<b>GO</b> [ <i>startaddr</i> ][, <i>stopaddr</i> ]	Starts program execution from <i>startaddr</i> , if specified, and stops at <i>stopaddr</i> , if specified.

The **GO** command instructs dScope to begin running your target program.

Execution begins from the address specified by *startaddr*. If *startaddr* is not specified, execution begins from the current program counter. Generally, it is not necessary to specify the start address after the current program counter is used as the start address.

Target program execution stops at the address specified by *stopaddr*. If *stopaddr* is specified, dScope sets a temporary breakpoint which is deleted when execution stops. If *stopaddr* is not specified, target program execution continues until a breakpoint is reached or until execution is stopped by clicking the Stop button in the Debug window.

After the target program execution stops, the Register, Watch, Debug, and other windows are updated to reflect the new CPU status.

---

### NOTE

*When conditional breakpoints are used, dScope must check break conditions after each instruction. For this reason, dScope runs the target program in single-step mode even though the **GO** command was used to begin execution.*

---

### Example

```
>G,main          /* Run starting at $ up to address "main" */
>G               /* Start at $. Break with Ctrl+C or breakpoint */
```

## INCLUDE

Syntax	Description
<b>INCLUDE</b> [ <i>path</i> ] <i>filename</i>	Opens <i>filename</i> and proceeds to read and execute dScope commands from the file.

### 9

The **INCLUDE** command lets you specify a command file from which commands are read and passed, line by line, to dScope for execution. You may use **INCLUDE** files to perform repetitive operations in dScope. For example, you might want to create an **INCLUDE** file that loads the CPU driver DLL, loads your target program, runs the program to the main C function, initializes a Toolbox button, and creates several user functions.

**INCLUDE** files may be nested up to 4 levels deep. You must stop target program execution to use the **INCLUDE** command.

### Example

```
INCLUDE measure.ini
```

## KILL

Syntax	Description
<b>KILL BUTTON</b> <i>number</i>	Removes a Toolbox buttons.
<b>KILL FUNC</b> *	Deletes all dScope functions.
<b>KILL FUNC</b> <i>function_name</i>	Deletes the <i>function_name</i> dScope function.

The **KILL** command lets you delete previously defined Toolbox buttons and dScope functions.

The **KILL BUTTON** command lets you remove a previously defined Toolbox button. When you use this command, you must specify the *number* of the Toolbox button to remove. This number displays in the Toolbox window in front of each button. Refer to “Toolbox Window” on page 67 for more information.

The **KILL FUNC** \* command removes all previously defined user functions and signal functions. Predefined dScope functions are not deleted. Refer to “Chapter 10. Functions” on page 261 for more information about dScope user functions.

The **KILL FUNC** *function\_name* command removes the specified user function or signal function.

### Example

```
>KILL FUNC ANALOG          /* Delete user function "analog" */
>KILL FUNC myregs          /* Delete user function "myregs" */
>KILL FUNC *               /* Delete all user functions */

>KILL BUTTON 3             /* Kill Toolbox Button number 3 */
>KILL BUTTON 1             /* Kill Toolbox Button number 1 */
```

# LOAD

Syntax	Description
<b>LOAD</b> [ <i>path</i> ] <i>filename</i> [ <b>NOCODE</b> ]	Loads a CPU driver DLL, absolute object file, or Intel HEX file.  <b>NOCODE</b> directs dScope to load only the symbolic information and ignore code records. <b>NOCODE</b> is relevant only if a CPU driver for a monitor (MON51, MON251, or MON166) was previously loaded.

The **LOAD** command lets you specify a file for the dScope debugger to load. You may load a CPU driver DLL, object module, or Intel HEX file.

- **CPU Driver DLL**  
A CPU Driver DLL is a dynamic-link library that dScope uses to simulate a target CPU. The DLL contains support for the CPU's on-chip peripherals as well as all SFRs. CPU drivers are available for a number of different chips.
- **Target Driver DLL**  
A Target Driver DLL is a dynamic-link library that dScope uses to interface to monitor programs like MON51, MON251, or MON166. Target drivers let you perform target debugging via the serial port using your own target hardware.
- **Intel OMF-51 Object File**  
OMF-51 object files are produced by the A51 assembler, the C51 compiler, and L51 and BL51 linker/locator. They contain complete symbolic debug information, type information, and line numbers. Source-level and symbolic debugging are fully supported.
- **Banked OMF-51 Object File**  
The BL51 code banking linker lets you create code banking programs that use up to 32 code banks. Banked object modules are created for code banking programs. The banked object module contains complete symbolic debug information, type information, and source-level debugging information (line numbers).

Code banks are stored in code segments 0x80-0x9F which correspond to bank 0 through bank 31 respectively.

Note that you must use the OC51 Banked Object File Converter to convert a banked object module into Intel OMF-51 object modules. Refer to the *8051 Utilities User's Guide* for more information.

- **Intel OMF-251 Object File**

OMF-251 object files are produced by the A251 assembler, the C251 compiler, and L251 linker/locator. They contain complete symbolic debug information, type information, and line numbers. Source-level and symbolic debugging are fully supported.

- **OMF-166 Object File**

OMF-166 object files are produced by the A166 assembler, the C166 compiler, and L166 linker/locator. They contain complete symbolic debug information, type information, and line numbers. Source-level and symbolic debugging are fully supported.

- **Intel HEX/Intel HEX386 Format**

Intel HEX files are produced by the Object to Hex Converter programs (OH51, OH251, and OH166). They contain no symbolic debugging information, no type information, and no line number information. Program testing is supported only at the assembler level. Source-level and symbolic debugging are not supported.

---

**NOTE**

*dScope analyzes the contents of the specified file to determine the file type. If the file type (DLL, OMF-51, OMF-251, OMF-166, or HEX) cannot be determined, the file is not loaded and an error message displays.*

---

## Loading a CPU Driver DLL

CPU drivers are Windows dynamic link libraries that dScope loads for complete simulation of specific chip derivatives. Refer to “Appendix A. CPU Driver Files” on page 295 for a complete description of the supported DLLs.

CPU driver DLLs are stored in the `\C51\BIN`, `\C251\BIN`, or `\C166\BIN` directory. Do not enter a path for the driver, simply enter the driver name. dScope automatically searches for the driver in proper directory.

If no specific CPU driver is loaded, dScope simulates a standard 8051 or 80166. However, none of the on-chip peripherals are activated and using the Special Function Registers (SFRs) has no affect. SFR names are not recognized and communication via the Serial window is not possible.

---

### NOTE

*If a CPU driver is loaded after a user program or after another CPU driver, dScope clears the current program and all debugging information from memory.*

---

### Example

The following command loads the 80517 CPU Driver. The name and version number of the driver display after the command.

```
>LOAD 80517.DLL  
80C517/80C537 PERIPHERALS V1.0
```

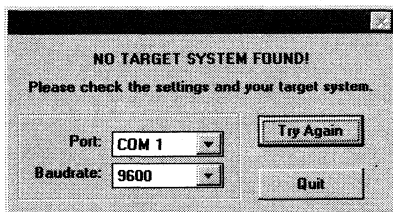
Once loaded, the CPU driver automatically defines I/O names and performs an initialization. This specific example makes dScope capable of completely simulating all of the peripherals of the 80517 CPU. The I/O names and the current values and types can be listed using the **DIR VTREG** command. Refer to “CPU Driver Symbols” on page 90 for more information.

## Loading a Target Driver DLL

Target drivers interface dScope with target systems like MON51, MON251, MON166, or third-party emulators. Target drivers change the name of dScope to tScope signifying that the debugger is now a target debugger. Refer to “Appendix A. CPU Driver Files” on page 295 for a complete description of the supported DLLs.

Target driver DLLs are stored in the \C51\BIN, \C251\BIN, or \C166\BIN directory. Do not enter a path for the driver, simply enter the driver name. dScope automatically searches for the driver in proper directory.

When a target driver loads, it attempts to initiate communications with the target board via the serial port. If there is a communications error, dScope presents you with a dialog box where you may select the proper COM port and baud rate settings.



dScope saves the settings you enter for the target driver for subsequent debugging sessions.

---

### NOTE

*If a target driver is loaded after a user program or after another CPU or target driver, dScope clears the current program and all debugging information from memory.*

---

### Example

The following command loads the MON51 Target Driver. The name and version number of the driver display after the command.

```
>LOAD MON51.DLL  
MONITOR-51 Driver V1.6
```

## Loading an Object File

OMF-51, OMF-251, and OMF-166 object files include information for symbolic and source-level debugging. They are produced by the various Keil C51, C251, and C166 compiler packages. The following table lists the compilers and assemblers that generate OMF object files along with the required command-line debugging options. When the debugging options are not included, debug information is excluded from the object file.

Assemblers & Compilers	Options	Type Information	Line Number Information	HLL Test
<b>A166</b>	<b>DEBUG</b>	yes, scalar types	yes	yes
<b>A251</b>	<b>DEBUG</b>	yes, scalar types	yes	yes
<b>A51</b>	<b>DEBUG</b>	no	yes	no
<b>C166</b>	<b>DEBUG</b>	yes	yes	yes
<b>C251</b>	<b>DEBUG</b>	yes	yes	yes
<b>C51</b>	<b>DEBUG</b>	no	yes	yes
<b>C51</b>	<b>DEBUG</b> <b>OBJECTTEXTEND</b>	yes †	yes	yes
<b>PL/M-51</b>	<b>DEBUG</b>	no	yes	yes

† Symbolic debugging is supported for programs compiled with C51 only when the **DEBUG** and **OBJECTTEXTEND** command-line options are used.

dScope lets you perform source-level debugging as long as the object module contains line number information and when the associated source or listing file can be located. Refer to “SETMOD” on page 251 for more information about associating the object module with a source file or listing file.

Symbolic debugging is supported when complete type information is available in the object module. For programs compiled with the C51 compiler, you must specify both the **DEBUG** and **OBJECTTEXTEND** command-line options to get complete type information in the generated object module.

Object files generated by the Keil assemblers and compilers include the name of the source file. The path specification is not included. When dScope loads an object module, source files are assumed to be in the same directory as the object module. You may use the **SET SRC** command to specify alternate paths to source files. Refer to “SET” on page 248 for more information.

Object files generated by the Intel PL/M-51 compiler reference statements in the listing file. They do not reference lines in the source file. Additionally, PL/M-51 object modules do not include the name of the source file or listing file.



You must manually specify the name of the listing file if you want to perform source-level debugging for PL/M-51 modules. Refer to “SETMOD” on page 251 for information about associating the object module with a source file or listing file.

---

**NOTES**

*C header files should not contain statements that produce executable program code. When this happens, dScope cannot synchronize the Debug window with the source code. This is because the object module can be associated with only one source file.*

*The C51 **OBJECTEXTEND** option is effective only when specified with the **DEBUG** option. You should always compile C51 programs with both options specified.*

*PL/M-51 refers to the Intel PL/M-51 Compiler.*

*When you load an object module or a CPU driver, debug information that was previously loaded is erased.*

---

**Example**

```
LOAD C:\C51\EXAMPLES\MEASURE\MEASURE
```

This command line loads MEASURE from the C:\C51\EXAMPLES\MEASURE directory. dScope also searches for source files in this directory.

## Loading Intel HEX and HEX386 Files

HEX files do not contain any symbolic or line number information. For this reason you cannot perform source-level or symbolic debugging with HEX files. You may debug programs at the assembly language level. You may also view the complete register set and SFR set.

### Example

```
LOAD MYPROG.HEX
```

## LOG

Syntax	Description
<b><u>LOG</u></b> > [ <i>path</i> ] <i>filename</i>	Creates the file <i>filename</i> as a log file. Output from the Command window is written to this file.
<b><u>LOG</u></b> >> [ <i>path</i> ] <i>filename</i>	Opens an existing log file named <i>filename</i> for appending log information. If <i>filename</i> does not exist, it is created. Output from the Command window is appended to this file.
<b><u>LOG</u></b>	Displays the status of the log file.
<b><u>LOG OFF</u></b>	Closes the log file.

Use the **LOG** command to create, append to, check status, or close a log file. Output displayed in the Command window is copied to the log file. The filename you specify may contain a driver letter and path specification. Filenames may be entered as character strings, for example, C:\USR\TMP\LOGFILE.

### Example

```
LOG >C:\TMP\dslog          /* Create a new log file */
LOG                         /* Interrogate log file status */
  command log file: C:\TMP\dslog
LOG OFF                     /* Close the log file */
```

## MAP

Syntax	Description
<b>MAP</b>	Displays the current memory map.
<b>MAP start, end [READ][WRITE][EXEC][VNM]</b>	Maps the specified memory range ( <i>start-end</i> ) accesses as specified.
<b>MAP start, end CLEAR</b>	Clears a mapped memory range.

# 9

Target programs you debug with dScope access and use memory. dScope uses the symbol information in your target program to automatically setup the memory map for most applications. The **MAP** command lets you specify the memory areas your program uses that are not automatically detected by dScope.

When you run your target program, dScope checks each memory access to determine if it is outside the memory map. If an invalid access is made, dScope reports an access violation error. This helps you locate and correct memory problems in your program.

### NOTE

*If your program uses memory-mapped I/O devices or dynamically accesses memory through pointers, you may need to make changes to the memory map.*

You specify an address range with the **MAP** command along with the accesses allowed for that range. Read (**READ**), write (**WRITE**), and execution (**EXEC**) accesses (or any combination) may be specified. The memory map supports 1-byte granularity.

The **VNM** option identifies the specified memory range as von Neumann memory. When **VMN** is specified with an address range, dScope overlaps external data memory (**XDATA**) and code memory. Write accesses to external data memory also change code memory. Memory ranges specified with **VNM** may not be a range from the code area and may not cross a 64K boundary. The address range specified must be from the external data area.

The **MAP** command, when entered with no other parameters, displays the current memory map for your target program. This lets you check your memory map settings.

The **CLEAR** option lets you remove an address range previously specified with the **MAP** command.

When dScope loads, the following memory maps are defined.

CPU	Address Range	Access
MCS <sup>®</sup> 51	0x000000-0x00FFFF ( <b>DATA</b> )	<b>READ WRITE</b>
	0x010000-0x01FFFF ( <b>XDATA</b> )	<b>READ WRITE</b>
	0xFF0000-0xFFFFFFFF ( <b>CODE</b> )	<b>EXEC READ</b>
MCS <sup>®</sup> 251	0x000000-0x00FFFF ( <b>DATA</b> )	<b>READ WRITE</b>
	0x010000-0x01FFFF ( <b>XDATA</b> )	<b>READ WRITE</b>
	0xFF0000-0xFFFFFFFF ( <b>CODE</b> )	<b>EXEC READ</b>
166 Family	0x000000-0x00FFFF	<b>EXEC READ WRITE</b>

dScope supports up to 16MB of memory. This memory is divided into 256 segments of 64K each. The default MCS<sup>®</sup> 51 and MCS<sup>®</sup> 251 memory spaces are assigned by dScope to the segments with the numbers listed in the following table.

Segment Value	Memory Space
0x00	MCS <sup>®</sup> 51 <b>DATA</b> segment, 0x00:0x0000—0x00:0x00FF. MCS <sup>®</sup> 251 <b>EDATA</b> segment, 0x00:0x0000-0x00:0xFFFF.
0x01	MCS <sup>®</sup> 51 and MCS <sup>®</sup> 251 <b>XDATA</b> segment 0x01:0x0000-0x01:0xFFFF.
0x80-0x9F	MCS <sup>®</sup> 51 and MCS <sup>®</sup> 251 Code Bank 0 through Code Bank 31. 0x80 = Code Bank 0, 0x81 = Code Bank 1, etc.
0xFE	MCS <sup>®</sup> 51 and MCS <sup>®</sup> 251 <b>PDATA</b> segment 0xFE:0x0000-0xFE:0x00FF.
0xFF	MCS <sup>®</sup> 51 and MCS <sup>®</sup> 251 <b>CODE</b> segment 0xFF:0x0000-0xFF:0xFFFF.

Although dScope supports up to 16MB of target program memory, only the memory ranges required should be mapped. dScope requires two copies of each block allocated in the memory map. One copy holds the data used for reading, writing, and execution. Another copy holds the specific attributes such as access permissions and information for code coverage and performance analysis. For this reason, mapping huge amounts of memory may slow down the execution speed of dScope since disk swapping may be required.

The **RESET MAP** command clears all mapped segments and restores the default mapping shown above. Refer to “RESET” on page 244 for more information.

Refer to “Memory Map Dialog Box” on page 72 for information about mapping address ranges using the Memory Map dialog box.

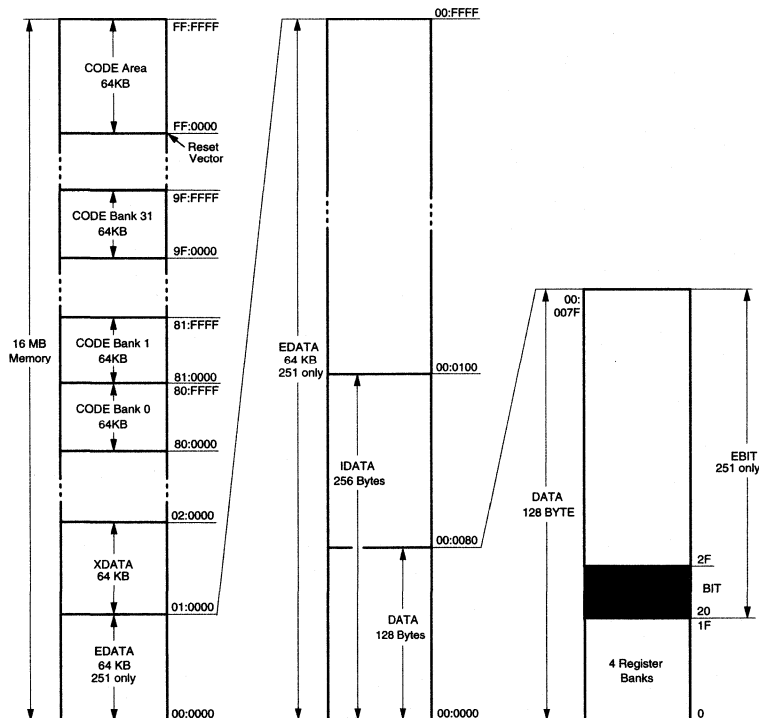
**NOTE**

Mapping less than 256 bytes of *XDATA* for 8051-compatible devices may cause problems. The 8051 supports a fast method of accessing 256 bytes of *XDATA* using the *MOVX @Ri* instructions. When more than 256 *XDATA* bytes are mapped, dScope uses the contents of *P2* and *P0* to resolve the full 16-bit address. When fewer than 256 *XDATA* bytes are mapped, dScope uses only the contents of *P0* and *P2* is ignored. Therefore, *XDATA* memory must start at the beginning of a segment (by default is 0x01:0x0000 to 0x01:0x00FF).

## 9

## Memory Map for MCS<sup>®</sup> 51 and MCS<sup>®</sup> 251 Microcontrollers

The following figure shows the memory map for MCS<sup>®</sup> 51 and MCS<sup>®</sup> 251 Microcontrollers.



### Example

```
MAP 0x10000,0x1FFFF read write      /* Enable 64K XDATA RAM */
RESET MAP                             /* Reset MAP */
MAP 0xFF0000,0xFFFFFFF exec read     /* Enable default code segment */
MAP 0x018000,0x01FFFF read write VNM /* Enable von Neumann memory */
```

## MODE

Syntax	Description
<b>MODE COMx, baudrate, parity, stopbits</b>	Change COM port settings.

dScope lets you change the settings for your PC's COM ports. The *baudrate* parameter must specify a valid baud rate like 1200, 2400, 9600, or 19200. The *parity* parameter may be **N** for none, **O** for odd, or **E** for even. The *stopbits* parameter may be **1** for 1 stop bit, **1.5** for 1½ stop bits, or **2** for 2 stop bits.

The **MODE** command may be used with the **ASSIGN** command to set the parameters for the simulated CPU serial input and output channels. Refer to "ASSIGN" on page 198 for more information.

### Example

```
>MODE COM2, 19200, N, 1      /* Setup COM2 for 19,200 bps, */
                             /* no parity, */
                             /* 1 stop bit */
```

## OBJECT

Syntax	Description
<b>OBJECT</b> <i>expression</i> [ , <i>base</i> ] [ <i>LINE</i> ]	Displays the contents of a structure or an array ( <i>expression</i> ), on a single <i>LINE</i> , if specified, using the specified <i>base</i> .

# 9

The **OBJECT** command displays complete structures or arrays. You may also use the **OBJECT** command to display scalar variables, however, this is no different from simply entering the scalar variable alone.

An optional number *base* of 10 or 16 may be specified.

The **LINE** option parameter displays the object on a single line. The output is truncated to 128 characters. If **LINE** is omitted structures and arrays are displayed on multiple lines.

### NOTE

*You may display complex data types (structures and arrays) if the target program was compiled with debug information enabled. This requires that the **DEBUG** compiler and assembler option is used. Refer to “Preparing Programs for dScope” on page 3 for more information.*

### Example

```
>OBJ current LINE                /* single line mode */
{time={hour=0x00,min=0x00,sec=0x00,msec=0x0000},port4=0x00, ...
>
>OBJ current,10                  /* multiline, radix 10 */
{
    time={                        /* a nested structure */
        hour=0,                  /* the structure members ... */
        min=0,
        sec=0,
        msec=0
    },
    port4=0,                      /* a scalar */
    port5=0,                     /* another scalar */
    analog=                      /* an array ... */
        [0]: 0
        [1]: 0
        [2]: 0
        [3]: 0
}
```



## OSTEP

Syntax	Description
<b>OSTEP</b>	Steps out of the current function invocation. If no function has been called, dScope issues an error.

The **OSTEP** command starts execution from the current program counter and stops if the current function returns to the statement following the function invocation. dScope internally maintains a list of currently nested function calls including the code address where a function invocation took place. If no function calls are present in the list, dScope issues an error when the **OSTEP** command is used.

Refer to “PSTEP” on page 243 and “TSTEP” on page 257 for information on other trace commands.

### Example

```
>0 /* Steps out of the current function */
```

## Performance Analyzer

Syntax	Description
<b>PerformanceAnalyze</b>	Displays all currently defined performance analyzer ranges and the timing results for these ranges.
<b>PerformanceAnalyze start</b> [ , end ]	Defines a new range to add to the performance analyzer.
<b>PerformanceAnalyze KILL *</b>	Removes all currently defined performance analyzer ranges.
<b>PerformanceAnalyze KILL item</b> [ , item... ]	Removes one or more currently defined performance analyzer ranges.
<b>PerformanceAnalyze RESET</b>	Resets the performance analyzer.

9

You may use the performance analyzer in dScope to tune your target programs for maximum performance. You specify the parts of your program you want to analyze and the performance analyzer gathers execution statistics for them during program execution. The fastest, slowest, and average execution times are maintained for each part of your program you chose to analyze.

You may analyze up to 256 ranges of code. The performance analyzer records the number of times a range of code is executed as well as the amount of time spent in each range.

A range is simply an address range. It normally starts at the first instruction in a function and ends at the last instruction in that function. However, you can specify a range that includes only a few instructions from any part of your target program.

As your program runs, the results of the performance analyzer display in the Performance Analyzer window. See “Performance Analyzer Window” on page 49.

The following commands are available for working with the performance analyzer.

#### **PA**

The **PA** command, when entered with no additional arguments, displays each defined performance analyzer address range, its index number (used to remove a specific range), the execution count, and the minimum, maximum, and total execution time (in clock cycles).

#### **PA start** [*start*, *end*]

The **PA** command, when followed by the name of a function in your target program or by an address range, creates a new address range to analyze. If *start* is the name of a function, dScope automatically obtains the starting and ending address for the function. If *start* is an address, *end* must specify the ending address of the range to analyze. Your target application should be compiled with full debug information enabled to best support performance analysis.

---

#### **NOTE**

*Performance analyzer address ranges must have unique entry and exit points and may not include an intermediate return (RET) instruction. A new address range may not overlap an existing range.*

---

#### **PA KILL \***

The **PA KILL \*** command removes all address ranges from the performance analyzer.

#### **PA KILL item**

The **PA KILL** command, when followed by an index number, removes the corresponding address range from the performance analyzer. dScope assigns an index to each address range and function you add to the performance analyzer. You may display these numbers using the **PA** command.

#### **PA RESET**

The **PA RESET** command clears all recorded information for all defined address ranges. You may use this command to begin recording after your target program has run a while.

Refer to “Performance Analyzer Setup Dialog Box” on page 83 for more information about defining program ranges for the performance analyzer using the Performance Analyzer Setup dialog box.

## Example

```

>PA main                                /* Define a range for main() */
>PA timer0                              /* Define a range for timer0() */
>PA clear_records                       /* Define more ranges */
>PA measure_display
>PA save_current_measurements
>PA read_index
>PA set_time
>PA set_interval
>
>PA                                     /* display all PA ranges */
0: main: (FF01EF-FF03B6)                /* FF01EF = C:0x01EF */
1: timer0: (FF006A-FF0135)
2: clear_records: (FF01C0-FF01EE)
3: measure_display: (FF07E7-FF084A)
4: save_current_measurements: (FF000E-FF0069)
5: read_index: (FF0136-FF01BF)
6: set_time: (FF084B-FF08CA)
7: set_interval: (FF08CB-FF09A5)

/* After execution of the user program ... */

>PA                                     /* Display ranges and stats */
0: main: (FF01EF-FF03B6)
   count=1, min=-1, max=0, total=167589
1: timer0: (FF006A-FF0135)
   count=2828, min=33, max=254, total=226651
2: clear_records: (FF01C0-FF01EE)
   count=1, min=27086, max=27086, total=27086
3: measure_display: (FF07E7-FF084A)
   count=10, min=19495, max=19503, total=185027
4: save_current_measurements: (FF000E-FF0069)
   count=491, min=205, max=209, total=100665
5: read_index: (FF0136-FF01BF)
6: set_time: (FF084B-FF08CA)
7: set_interval: (FF08CB-FF09A5)
>
>PA KILL 7                             /* Remove set_interval */
>PA KILL 6                             /* Remove set_time */
>PA KILL 5                             /* Remove read_index */
>PA
0: main: (FF01EF-FF03B6)
   count=1, min=-1, max=0, total=167589
1: timer0: (FF006A-FF0135)
   count=2828, min=33, max=254, total=226651
2: clear_records: (FF01C0-FF01EE)
   count=1, min=27086, max=27086, total=27086
3: measure_display: (FF07E7-FF084A)
   count=10, min=19495, max=19503, total=185027
4: save_current_measurements: (FF000E-FF0069)
   count=491, min=205, max=209, total=100665
>
>PA RESET                             /* Clear all recorded information */
>PA
0: main: (FF01EF-FF03B6)
1: timer0: (FF006A-FF0135)
2: clear_records: (FF01C0-FF01EE)
3: measure_display: (FF07E7-FF084A)

```

# PSTEP

Syntax	Description
<b>PSTEP</b> [ <i>expression</i> ]	Executes or steps over <i>expression</i> program lines or assembly instructions. The <b>PSTEP</b> command steps over functions and subroutines.

The **PSTEP** command executes one or more source-level instructions or assembly instructions depending on the display mode selected in the Debug window. The **PSTEP** command does not step into function calls. For information about how to step into function calls, refer to “TSTEP” on page 257.

The **PSTEP** command steps over source-level instructions or assembly instructions depending on the Debug window display modes outlined in the following table.

Display Mode	Description
<b>Assembly</b>	<b>PSTEP</b> steps over assembly instructions but does not step into subroutines or functions.
<b>Mixed</b>	<b>PSTEP</b> steps over assembly instructions but does not step into subroutines or functions.
<b>High-Level Language</b>	<b>PSTEP</b> steps over program statements in your C or PL/M-51 program but does not step into subroutines or functions.

## Example

```
>P 100 /* Execute 100 steps, ignoring calls */
>P /* Execute a step, ignoring calls */
```

## RESET

Syntax	Description
<b>RESET</b>	Resets the simulated or target CPU.
<b>RESET MAP</b>	Resets <b>MAP</b> assignments.
<b>RESET</b> <i>variable</i>	Resets <b>SET</b> variables.

# 9

The **RESET** command has three distinct uses depending on how it is entered.

- When **RESET** is entered with no additional arguments, the dScope debugger resets the simulated or target CPU. This is equivalent to a processor reset where the program counter is set to 0x0000 and all special function registers are reset to their default values. Your target program remains loaded along with all debug information. Any active signal functions are deactivated.
- The **RESET MAP** command resets map assignments and clears memory of any loaded target program and debug information. Refer to “MAP” on page 234 for more information.
- The **RESET** command may be entered along with the name of a SET variable to clear the string assigned to that variable. Valid variable names are: **SRC**, **F1**, **F2**, **F3**, **F4**, **F5**, **F6**, **F7**, **F8**, **F9**, **F11**, and **F12**. Refer to “SET” on page 248 for more information.

### Example

```
>RESET                /* RESET the CPU */
>RESET MAP            /* RESET the memory MAP assignments */
>RESET F3             /* RESET the F3 key assignment */
```

## SAVE

Syntax	Description
<b>SAVE</b> [ <i>path</i> ] <i>filename addr1, addr2</i>	Saves the address range <i>addr1</i> to <i>addr2</i> to <i>filename</i> in HEX386 format.

The **SAVE** command saves a memory image to *filename*. The file is saved in HEX386 format. The contents of memory from *addr1* to *addr2* are saved. The contents of this file may later be loaded into dScope using the **LOAD** command.

---

### NOTE

*You cannot save the internal RAM of the 8051 and 251 from I:0x80 to I:0xFF.*

---

## SCOPE

Syntax	Description
<b>SCOPE</b> [ <i>\module</i> [ <i>\function</i> ]]	Displays the address range of the specified program module or function.

The **SCOPE** command displays:

- The address ranges of all functions in the target program (if no module and function names are entered),
- The address ranges of all functions in a program module (if only a module name is specified),
- The address range of a single function (if a module and function are specified).

When an OMF-51, OMF-251, or OMF-166 program with debug information loads, dScope creates an internal table of address assignments and the associated symbol information. Non-qualified symbols can automatically be selected from this table.



## Examples

In the following examples, indented lines reflect the layout of the source program for each module.

```
>scope \measure\main                                /* show scope range of main() */
MAIN_RANGE: 0xFF01EF-0xFF03B6
>
>scope \measure                                      /* show scope for module 'measure' */
MEASURE
  {CvtB} RANGE: 0xFF03B7-0xFF07E5                    /* dScope dummy scope block */
  {CvtB} RANGE: 0xFF000B-0xFF000D
  SAVE_CURRENT_MEASUREMENTS_RANGE: 0xFF000E-0xFF0069
  TIMER0_RANGE: 0xFF006A-0xFF0135
  _READ_INDEX_RANGE: 0xFF0136-0xFF01BF
  CLEAR_RECORDS_RANGE: 0xFF01C0-0xFF01EE
  MAIN_RANGE: 0xFF01EF-0xFF03B6
>
>scope                                                /* show all scope ranges */
MEASURE
  {CvtB} RANGE: 0xFF03B7-0xFF07E5
  {CvtB} RANGE: 0xFF000B-0xFF000D
  SAVE_CURRENT_MEASUREMENTS_RANGE: 0xFF000E-0xFF0069
  TIMER0_RANGE: 0xFF006A-0xFF0135
  _READ_INDEX_RANGE: 0xFF0136-0xFF01BF
  CLEAR_RECORDS_RANGE: 0xFF01C0-0xFF01EE
  MAIN_RANGE: 0xFF01EF-0xFF03B6
MCOMMAND
  {CvtB} RANGE: 0xFF09A6-0xFF0A23
  MEASURE_DISPLAY_RANGE: 0xFF07E7-0xFF084A
  _SET_TIME_RANGE: 0xFF084B-0xFF08CA
  _SET_INTERVAL_RANGE: 0xFF08CB-0xFF09A5
GETLINE
  _GETLINE_RANGE: 0xFF0A24-0xFF0A87
?C_FPADD
?C_FPMUL
?C_FPDIV
?C_FPCMP
...
```

### NOTE

Blocks named **{CvtB}** are created by dScope and are the result of insufficient debug information. This is normal for library modules and assembly language modules without debug information and for blocks with a valid name but no scope range.

## SET

Syntax	Description
<b>SET</b> <i>variable</i>	Displays a <b>SET</b> <i>variable</i> assignment.
<b>SET</b> <i>variable</i> = " <i>string</i> "	Assigns <i>string</i> to the specified <b>SET</b> <i>variable</i> .

The **SET** command set the string associated with a predefined variable. You may also use the **SET** command to display the string currently associated with a predefined variable. The **RESET** command lets you clear the string associated with a predefined variable. Refer to “RESET” on page 244 for more information.

The following predefined variables may be used with the **SET** and **RESET** commands.

Variable Name	Description
<b>SRC</b>	The path or paths to search for source or listing files required for source-level debugging. Up to 10 path specifications may be specified for the <b>SRC</b> variable one at a time.
<b>F1</b>	The <b>F1</b> function key.
<b>F2</b>	The <b>F2</b> function key.
<b>F3</b>	The <b>F3</b> function key.
<b>F4</b>	The <b>F4</b> function key.
<b>F5</b>	The <b>F5</b> function key.
<b>F6</b>	The <b>F6</b> function key.
<b>F7</b>	The <b>F7</b> function key.
<b>F8</b>	The <b>F8</b> function key.
<b>F9</b>	The <b>F9</b> function key.
<b>F11</b>	The <b>F11</b> function key.
<b>F12</b>	The <b>F12</b> function key.

Note that the **F10** is not a valid predefined variable.  
You may not assign a text string to the **F10** key.

## Example

The following example shows how to assign a command to a function key.

```
>SET F5="LOAD 8052.DLL"          /* F5 assignment */
>SET F5                          /* Interrogation */
F5 = LOAD 8052.DLL              /* SET Output */
```

Pressing **F5** executes the **LOAD** command and displays as follows in the Command window.

```
8051/8031 80C51/80C31 PERIPHERALS V1.1
```

The following example shows how to reset a function key assignment.

```
>RESET F5                      /* Reset F5 key */
>SET F5                        /* F5 interrogation */
F5 =                           /* Unassigned */
```

When you load an object module, the path specified is added to the **SRC** paths that are searched.

```
>LOAD \OBJS\MEASURE            /* Load mod. "MEASURE" */
>SET SRC                      /* Display paths assigned to SRC */
\objs
```

You may add additional path specifications so search for source and listing files.

```
>SET SRC \SRC                 /* Additional path specification */
>SET SRC                      /* Display paths assigned to SRC */
\objs
\src
```

## Escape Characters

Backslash characters (“\”) must be escaped inside strings. This is illustrated in the following examples.

### Incorrect

```
>SET F5 = "load \objs\measure"
```

### Correct

```
>SET F5 = "load \\objs\\measure"
```

## Confusion with Public Symbols

**F1** is a public symbol defined in many 8051 and 251 programs (it is a bit in the PSW). This may cause a syntax error when using the **SET** command to assign a string to the **F1** key. For example:

```
>SET F1 = "dir"  
-----^  
Error 114: syntax error
```

9

If this happens, put the **F1** in double quotes as shown below.

```
>SET "F1" = "dir"
```

## SETMODULE

Syntax	Description
<b>SETMODULE</b> <i>\module = source_file</i>	Associates a source file or a listing file with the specified module.
<b>SETMODULE</b> [ <i>\module</i> ]	Displays the name of the source file or listing file currently associated with the specified module.

dScope must know the name of the source file associated with each module in your program in order to perform source-level debugging. The **SETMODULE** command lets you associate a source file or a listing file with a module of a loaded program. **SETMODULE** also displays the name of each module and its associated source file.

With the A51, A251, and A166 Assemblers and the C51, C251, and C166 Compilers you do not need to manually specify the module and source file associations. The Keil assemblers and compilers generate the module name automatically from the name of the source file.

However, for modules created with the Intel PL/M-51 compiler, you must use the **SETMODULE** command to specify the relationship between the module and the source file. For PL/M-51 modules, the module does not correspond to the filename. The module name is derived from the first source statement in the PL/M-51 module which usually appears as follows:

```
PLMODULE: DO;  
.  
.  
.
```

Therefore, dScope can not locate the source file for the module unless you manually specify the relationship using the **SETMODULE** command.

## Example

This example demonstrates a program which consists of two modules. The C module named CA.C.

```

/*****
/* C MODULE "CA.C" */
/*****
/* Module name = CA */
extern alien char PLMFUNC (char x, char y, int z);
char a,x,c;

main() {
    a = 2;
    x = 4;
    c = 5;
    c = PLMFUNC (a, x, c);
    if (c == a) --x;
    else ++a;
    while (1);
}

alien char cafunc (char q1, char q2) { /* Called from PLMMODULE */
    return (q1 + q2 + 2);
}

```

And the PL/M-51 module named PLM.P51.

```

/*****
/* PL/M-51 MODULE: "PLM.P51" */
/*****
PLMMODULE: DO; /* Module name = "PLMMODULE" */
cafunc: procedure (q0, q1) byte external; /* In CA.C */
    declare (q0, q1) byte;
end cafunc;
plmfunc: procedure (x, y, z) byte public;
    declare (x, y) byte;
    declare z word;
    z = cafunc (x, y);
    z = z + (x * y);
    return (z);
end plmfunc;
END PLMMODULE;

```

The following DOS commands compile and link the program.

```

PLM51 PLM.P51 DEBUG CODE
C51 CA.C CODE DEBUG OBJECTTEXTEND
BL51 CA.OBJ, PLM.OBJ

```

The linked program is stored in the file CA. When this program loads into dScope, source-level debugging works for the C module but not for the PL/M-51 module.

The following dScope **SETMODULE** command shows the initial module assignments.

```
>LOAD CA                      /* Load program */
>SETMOD                      /* Display module assignments */
Module CA, Src/Lst: CA.C
Module PLMMODULE, Src/Lst: <none>
```

Note that **PLMMODULE** is not associated with a source file. It should be associated with **PLM.P51**. The following **SETMOD** command shows how to make that association.

```
>SETMOD \PLMMODULE=PLM.LST    /* Perform assignment */
>SETMOD                      /* Output assignment */
Module CA, Src/Lst: CA.C
Module PLMMODULE, Src/Lst: PLM.LST
```

Now, you may view the source code in dScope for both modules.

```
>U main                      /* Disassemble main from CA.C */
>U plmfunc                   /* Disassemble plmfunc of PLM.LST */
```

---

## NOTES

*For modules created with the C51 compiler the C source file is automatically used for the source-level debugging only if the **DEBUG** and **OBJECTEXTEND** controls are specified when you compile.*

*For PL/M-51 modules created with the Intel PL/M-51 compiler, you may specify a list file (using the **SETMODULE** command) to use for source-level debugging. PL/M-51 source files may not be used because the PL/M-51 compiler puts statement numbers in the object module instead of line numbers (like C).*

*dScope always attempts to locate a source file or a listing file for each module. To do this dScope uses paths from: a preceding **LOAD** command (page 226), the **SET SRC** command (page 248), and the current directory. dScope also tries the **.LST** and **.C** extensions if the initial attempt fails. (dScope uses the first few bytes in a file to determine if it is a source file or a listing file.*

*Source-level debugging is supported only for modules that have line number information.*

---

**NOTE**

*C include files should not contain statements that produce executable code. If header files which produce code are included in your C programs, the source-level display may not be synchronized with the assembly code generated. This is because OMF-51 and OMF-166 support only one source file or listing file per module. OMF-251 does not have this limitation.*

---



## SIGNAL

Syntax	Description
<b>SIGNAL KILL</b> <i>function_name</i>	Deactivates the specified active signal function.
<b>SIGNAL STATE</b>	Displays the active signal functions.

The **SIGNAL** command lets you display or kill active signal functions.

### Example

```
>SIGNAL KILL ANALOG0          /* Deactivate the analog0 signal function */
```

The above example shows how to deactivate an active signal function.

```
>SIGNAL STATE
0 idle      Signal = ANALOG0 (line 10)
```

The above example displays the active signal functions. The function number, function status (idle or running), function name (ANALOG0 in this case), and line number of the last executed statement are displayed.

## SLOG

Syntax	Description
<b>SLOG</b> > [ <i>path</i> ] <i>filename</i>	Creates the file <i>filename</i> as a log file. Input and output from the Serial window is written to this file.
<b>SLOG</b> >> [ <i>path</i> ] <i>filename</i>	Opens an existing log file named <i>filename</i> for appending log information. If <i>filename</i> does not exist, it is created. Input and output from the Serial window is appended to this file.
<b>SLOG</b>	Displays the status of the log file.
<b>SLOG OFF</b>	Closes the log file.

Use the **SLOG** command to create, append to, check status, or close a log file. Input and output displayed in the Serial window is copied to the log file. The filename you specify may contain a driver letter and path specification. Filenames may be entered as character strings, for example, C:\USR\TMP\LOGFILE.

### Example

```

SLOG >C:\TMP\dslog          /* Create a new log file */
SLOG                        /* Interrogate log file status */
  serial log file: C:\TMP\dslog
SLOG OFF                    /* Close the log file */

```

## TSTEP

Syntax	Description
<b>TSTEP</b> [ <i>expression</i> ]	Executes or steps <i>expression</i> program lines or assembly instructions. The <b>TSTEP</b> command steps into functions and subroutines.

The **TSTEP** command executes one or more source-level instructions or assembly instructions depending on the display mode selected in the Debug window. The **TSTEP** command steps into function calls. Refer to “PSTEP” on page 243 for information about the **PSTEP** command.

The **TSTEP** command steps over source-level instructions or assembly instructions depending on the Debug window display modes outlined in the following table.

Display Mode	Description
<b>Assembly</b>	<b>TSTEP</b> steps over assembly instructions and steps into subroutines and functions.
<b>Mixed</b>	<b>TSTEP</b> steps over assembly instructions and steps into subroutines and functions.
<b>High-Level Language</b>	<b>TSTEP</b> steps over program statements in your C or PL/M-51 program and steps into subroutines and functions.

### Example

```
>T 100 /* Execute 100 steps */  
>T /* Execute a step */
```

# UNASSEMBLE

Syntax:	Description
<b><u>UNASSEMBLE</u></b> [ <i>address</i> ]	Displays disassembled code memory in the Debug window. Disassembly starts at <b><i>address</i></b> , if specified, or continues from the previous <b>UNASSEMBLE</b> command.

The **UNASSEMBLE** command disassembles code memory and displays it in the Debug window. Code displays in one of three modes: high-level language, high-level language and assembly, or assembly. Refer to “Debug Window” on page 27 for more information.

**NOTE**

*dScope requires an object module that contains line number information in order to perform source-level debugging.*

**Example**

```
U main          /* Disassemble starting at address "main" */
U              /* Continue where the previous U stopped */
U C:0x0         /* Disassemble from address C:0x0000 */
```

## WATCHKILL

Syntax	Description
<b>WATCHKILL</b> <i>number</i> [, <i>number...</i> ]	Deletes the specified watchpoint expression or expressions.
<b>WATCHKILL</b> *	Deletes all watchpoints.

The **WATCHKILL** command deletes one or more watchpoint expressions from the Watch window. Refer to “Watch Window” on page 44.

You may delete all watchpoints using **WK \***. You may delete specific watchpoints by specifying **WK** along with the watchpoint number from the Watch window.

Refer to “Watchpoints Dialog Box” on page 81 for information about setting watchpoints using the Watchpoints dialog box. Refer to “Watch Window” on page 44 for information about viewing watchpoints.

### Example

```
WK 0,2          /* Delete watchpoints 0 and 2 */
WK *            /* Delete all watchpoints */
```

## WATCHSET

Syntax	Description
<b><u>WATCHSET</u></b> <i>expression</i> [ , <i>base</i> ] [ <i>LINE</i> ]	Defines a watchpoint expression to add to the Watch window. The <i>base</i> parameter specifies the number base to use (10 or 16). The <i>LINE</i> parameter specifies that structure, union, and array expressions display on a single line.

# 9

The **WATCHSET** command lets you define watchpoint expressions to display in the Watch window. The Watch window is updated after each program execution command such as **GO**, **OSTEP**, **PSTEP**, and **TSTEP**.

You may optionally specify the number base (10 or 16) in which to display the value of the expression. You may use the **LINE** parameter to specify that structures, unions, and arrays display on a single line (as opposed to displaying on multiple lines).

Refer to “Watchpoints Dialog Box” on page 81 for information about setting watchpoints using the Watchpoints dialog box. Refer to “Watch Window” on page 44 for information about viewing watchpoints.

### NOTE

*Use \* with structure or union pointers to watch the contents of the structure or union. Without the \* pointer operator, the Watch window displays the address of the pointer.*

### Example

```
>WS interval,0x0a LINE
>WS save_record[0].analog
>WS save_record[0]
>WS sindex
```

## Chapter 10. Functions

This chapter discusses a powerful aspect of dScope: functions. You may use functions to extend the capabilities of dScope. You may create functions that generate external interrupts, log memory contents to a file, update analog input values periodically, and input serial data to the on-chip serial port.

---

### NOTE

*Do not confuse dScope functions with the functions of a target program. dScope functions are entered or included by typing commands in dScope's command window.*

---

dScope functions utilize a subset of the C programming language. The basic capabilities and restrictions are as follows:

- The flow control statements **if**, **else**, **while**, **do**, **switch**, **case**, **break**, **continue**, and **goto** may be used in dScope functions. All of these statements operate in dScope as they do in ANSI C.
- Local scalar variables are declared in dScope functions in the same way they are declared in ANSI C. Arrays are not allowed in dScope functions.

Refer to “Differences Between dScope Functions and ANSI C” on page 283 for a complete description of the differences between dScope functions and C functions.

This chapter discusses the following aspects of dScope functions:

- “Creating Functions” on page 262.
- “Loading Functions” on page 263.
- “Function Classes” on page 264.

Refer to these sections to learn how to create and load dScope functions.

---

### NOTE

*This chapter assumes that the user is familiar with and understands the C programming language.*

---

## Creating Functions

You create dScope functions in one of two ways.

1. You may create a function, using a text editor, and save it in a file which you may access in dScope. A file may contain numerous functions.
2. You may create a function in dScope's command window. You must be very careful when you manually enter a function this way because you cannot change the text on a line once it is entered.

---

### NOTE

*Typically, it is not very useful to manually create functions in dScope's command window because the function text is not available after you leave dScope.*

---

10

Both methods of creating dScope functions are described below.

## Using a Text Editor

When you use a text editor, like  $\mu$ Vision, to create a dScope function, you enter the body of the function and save it in a text file. You may also include any valid dScope commands in the text file.

Then, in dScope you use the **INCLUDE** command to read and process the contents of the text file. For example, if you type the following command in the command window, dScope reads and interprets the contents of **MYREGS.FNC** one line at a time.

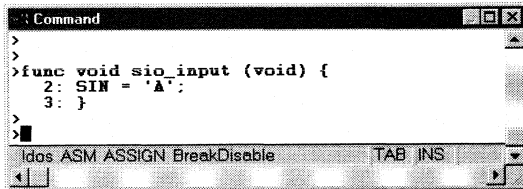
```
>INCLUDE MYREGS.FNC
```

**MYREGS.FNC** may contain dScope commands and may also include function definitions.



## Using the Command Window

When you create dScope functions in the command window, you enter the function definition one line at a time. The following figure shows a simple function that puts a character into the serial input channel.



If you make an error entering a function in the command window, dScope presents an error message. However, the command window only provides a line editor, so there is no way to correct an error. For this reason, it is better to create an include file for your dScope function definitions.

**10**

## Loading Functions

When you enter a function, manually or using the **INCLUDE** command, dScope analyzes it using an integrated function compiler and generates internal intermediate code which allows dScope functions to execute very quickly.

The function compiler in dScope performs strict type checking. For example, the number of parameters, the return type, and generally the compatibility of the types are all tested.

If there are no errors during compilation, the name of the function is added to the internal list of functions. The function may then be invoked directly from the command line, by typing the function name along with any required parameters, or from within other functions.

If an error occurs during compilation, the function is ignored. Errors may be corrected with the text editor and the function may be re-submitted to dScope for compilation.

Functions that are no longer needed may be deleted using the **KILL** command. Refer to “KILL” on page 225 for more information.

## Invoking Functions

To invoke or run a dScope function you must type the name of the function and any required parameters in the command window. For example, to run the **printf** built-in function to print “Hello World,” enter the following text in the command window:

```
>printf ("Hello World\n")
```

dScope responds by printing the text “Hello World” in the command window.

# 10

## Function Classes

dScope supports the following three classes of functions: Predefined Functions, User Functions, and Signal Functions.

- **Predefined Functions**

These functions are predefined by dScope and cannot be deleted or redefined. You may use predefined functions in your user and signal functions to perform useful tasks like waiting for a period of time or printing a message.

- **User Functions**

User functions are functions you create to extend the capabilities of dScope. User functions as well as signal functions can process the same expressions allowed at the dScope command level. You may use the predefined function **exec**, to execute dScope commands from user and signal functions.

- **Signal Functions**

Signal functions simulate the behavior of a complex signal generator. This class of function lets you create various input signals to your target application. For example, signals can be applied on the input lines of the CPU under simulation. Signal functions run in the background during your target program’s execution. Signal functions are coupled via dScope’s cycle counter which has a resolution of instruction cycle or 1 microsecond at 12 MHz. A maximum of 19 signal functions may be active simultaneously.

As functions are defined, they are entered into the internal table of user or signal functions. You may use the **DIR** command to list the predefined, user, and signal functions available.

The command **DIR BFUNC** displays the names of all built-in functions.

```
>DIR BFUNC
predef'd: void  MEMSET ( ulong, ulong, uchar )
predef'd: void  TWATCH ( long )
predef'd: int   RAND ( uint )
predef'd: uchar TIMEWAIT ( uint )
predef'd: void  KEYWAIT ( char * )
predef'd: float GETFLOAT ( char * )
predef'd: long  GETLONG ( char * )
predef'd: int   GETINT ( char * )
predef'd: void  EXEC ( char * )
predef'd: void  PRINTF ( char *, ... )
```

The command **DIR UFUNC** displays the names of all user functions.

```
>DIR UFUNC
user: void  SHOWREGS ( void )
```

The command **DIR SIGNAL** displays the names of all signal functions.

The command **DIR FUNC** displays the names of all user, signal, and built-in functions.

## Predefined Functions

dScope includes a number of predefined functions that are always available for use. They cannot be redefined or deleted. Predefined functions are provided to assist the user and signal functions you create.

There are a number of standard operators you may use like functions: **bit**, **byte**, **char**, **double**, **dword**, **float**, **int**, **long**, **ptr**, **real**, **uchar**, **uint**, **ulong**, and **word**. You may use these operators to access a data type at a specific address. You may also use them, contrary to C conventions, to assign a value of a specific type to a specific address.

The following table lists all predefined dScope functions.

10

Return Type	Name	Parameter
bit	bit	( <i>address</i> )
uchar	byte	( <i>address</i> )
char	char	( <i>address</i> )
double	double	( <i>address</i> )
ulong	dword	( <i>address</i> )
void	exec	( <i>"command_string"</i> )
float	float	( <i>address</i> )
float	getfloat	( <i>"prompt_string"</i> )
int	getint	( <i>"prompt_string"</i> )
long	getlong	( <i>"prompt_string"</i> )
int	int	( <i>address</i> )
long	long	( <i>address</i> )
void	memset	( <i>start_address, end_address, uchar val</i> )
void	printf	( <i>"string", ...</i> )
ptr	ptr	( <i>address</i> )
int	rand	( <i>int seed</i> )
float	real	( <i>address</i> )
void	twatch	( <i>ulong cycles</i> )
uchar	uchar	( <i>address</i> )
uint	uint	( <i>address</i> )
ulong	ulong	( <i>address</i> )
uint	word	( <i>address</i> )

Each of the predefined functions is described below.

## bit (*address*)

The **bit** function returns the value of the bit at *address*. You may also use this function to assign a bit value to a specific address. The range of values for a bit is 0 to 1.

### Example

```
>bit (d:0x20.5)      /* display the bit at address D:20H.5 */  
>bit (d:0x20.1) = 1  /* set bit at address D:20H.1      */
```

## byte byte (ulong *address*)

The **byte** function returns the value of the byte at *address* as an unsigned character. You may also use this function to assign an unsigned character value to a specific address. The range of values for an unsigned character is 0 to 255.

**10**

### Example

```
>byte (c:0x0000)      /* display the character at C:0000h */  
>byte (x:0x1000)      /* display the character at X:1000h */  
>byte (x:0x1000) = 'A' /* assign unsigned char 'A' to X:10000h */
```

## char char (ulong *address*)

The **char** function returns the value of the byte at *address* as a signed character. You may also use this function to assign a signed character value to a specific address. The range of values for a signed character is -128 to 127.

### Example

```
>char (c:0x0000)      /* display the character at C:0000h */  
>char (x:0x1000)      /* display the character at X:1000h */  
>char (x:0x1000) = 'Z' /* assign the character 'Z' to X:10000h */
```

## double double (ulong *address*)

The **double** function returns the double floating-point value of the eight bytes starting at *address*. You may also use this function to assign a double floating-point value to a specific address.

### Example

```
>double (c:0x1090)      /* display the double float at C:1090h */
>double (x:0x8000)      /* display the double float at X:8000h */
>double (x:0x2000) = 3.14 /* assign the double value 3.14 to X:2000h */
```

---

### NOTE

*The double function is available only in dScope-166.*

---

**10**

## ulong dword (ulong *address*)

The **dword** function returns the unsigned long integer value of the four bytes starting at *address*. You may also use this function to assign an unsigned long integer value to a specific address. The range of values for an unsigned long integer is 0 to 4294967295.

### Example

```
>dword (i:0x90)         /* display the unsigned long at i:90h */
>dword (x:0x8000)       /* display the unsigned long at X:8000h */
>dword (x:0x2000) = 123  /* assign the unsigned long 123 to X:2000h */
```

## **void exec ("command\_string")**

The **exec** function lets you invoke dScope commands from within your user and signal functions. The *command\_string* may contain several commands separated by semicolons.

The *command\_string* is passed to the dScope command interpreter and must be a valid dScope command. Additionally, the following commands may not be invoked using the **exec** function.

- **ASM**
- **ENTER**
- **EXIT**
- **FUNC**
- **LOAD** (except to load HEX files)
- **RESET**
- **SIGNAL**

**10**

### **Example**

```
>exec ("DIR PUBLIC; eval dptr + r7")
>exec ("BS timer0")
>exec ("BK *")
```

## **float float (ulong address)**

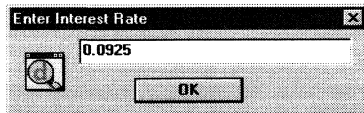
The **float** function returns the floating-point value of the four bytes starting at *address*. You may also use this function to assign a floating-point value to a specific address.

### **Example**

```
>float (c:0x1090)          /* display the float at C:1090h */
>float (x:0x8000)          /* display the float at X:8000h */
>float (x:0x2000) = 3.14   /* assign the float value 3.14 to X:2000h */
```

## float getfloat (“*prompt\_string*”)

The **getfloat** function prompts you to enter a floating-point number and, upon entry, returns the floating-point value of the number entered. If no entry is made, the value 0 is returned. The following illustration shows the **getfloat** dialog box.



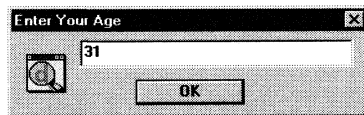
### Example

```
>getfloat ("Enter Interest Rate")
```

# 10

## int getint (“*prompt\_string*”)

The **getint** function prompts you to enter an integer number and, upon entry, returns the 16-bit integer value of the number entered. If no entry is made, the value 0 is returned. The following illustration shows the **getint** dialog box.

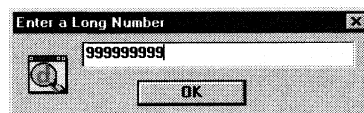


### Example

```
>getint ("Enter Your Age")
```

## int getlong (“*prompt\_string*”)

The **getlong** function prompts you to enter a long integer number and, upon entry, returns the 32-bit value of the number entered. If no entry is made, the value 0 is returned. The following illustration shows the **getlong** dialog box.



### Example

```
>getlong ("Enter a Long Number")
```



## int int (ulong *address*)

The **int** function returns the signed integer value of the two bytes starting at *address*. You may also use this function to assign a signed integer value to a specific address. The range of values for a signed integer is -32768 to 32767.

### Example

```
>int (d:0x50)           /* display the unsigned int at D:50h */
>int (x:0x8000)          /* display the unsigned int at X:8000h */
>int (x:0x1000) = 123    /* assign the int value 123 to X:1000h */
```

## long long (ulong *address*)

The **long** function returns the signed long integer value of the four bytes starting at *address*. You may also use this function to assign a signed long integer value to a specific address. The range of values for a signed long integer is -2147483648 to 2147483647.

### Example

```
>long (i:0x90)           /* display the unsigned long at I:90h */
>long (x:0x8000)          /* display the unsigned long at X:8000h */
>long (x:0x2000) = 999    /* assign the 32-bit value 999 to X:2000h */
```

## void memset (*start address*, *end address*, uchar *value*)

The **memset** function fills a memory range with the specified value. The *startaddr* argument must specify a unique memory space and the starting address. The *endaddr* argument specifies the ending address. The *value* argument specifies the value to store.

### Example

```
>MEMSET (X:0, X:0xFFFF, 'a') /* Write XDATA RAM 0000 to FFFF with "a" */
>MEMSET (C:0x0, C:0x1FFF, 0) /* Clear CODE memory range */
>MEMSET (D:0x30, D:0x7F, r7) /* DATA area 0x30-0x7F, value from R7 */
```

## void printf ("*format\_string*", ...)

The **printf** function works like the ANSI C library function. The first argument is a format string. Following arguments may be expressions or strings. The conventional ANSI C formatting specifications apply to **printf**.

### Example

```
>printf ("random number = %04XH\n", rand(0))
random number = 1014H

>printf ("random number = %04XH\n", rand(0))
random number = 64D6H

>printf ("%s-%d %s\n", "dScope", 51, "Windows")
dScope-51 Windows

>printf ("%lu\n", (ulong) -1)
4294967295

>printf ("%u\n", (ulong) -1)
65535

>printf ("%ld\n", (ulong) -1)
-1
```

## ptr ptr (ulong *address*)

The **ptr** function returns the pointer stored at *address*. You may also use this function to assign a pointer value to a specific address.

### Example

```
>ptr (C:0x1000)          /* display the pointer at C:1000h */
>ptr (x:0x8000)          /* display the pointer at X:8000h */
>ptr (x:0x1000) = 0xFF0000 /* assign a pointer to X:1000h */
```

## int rand (int *seed*)

The **rand** function returns a random number in the range -32768 to +32767. The random number generator is reinitialized each time a non-zero value is passed in the *seed* argument.

You may use the **rand** function to delay for a random number of clock cycles or to generate random data to feed into a particular algorithm or input routine.

### Example

```
>rand (0x1234)          /* Initialize random generator with 0x1234 */
0x3B98

>rand (0)                /* No initialization */
0x64BD

>rand (0)
0x12B5
```

**10**

## real real (ulong *address*)

The **real** function returns the floating-point value of the four bytes starting at *address*. You may also use this function to assign a floating-point value to a specific address.

### Example

```
>real (c:0x1090)         /* display the float at C:1090h */
>real (x:0x8000)         /* display the float at X:8000h */
>real (x:0x2000) = 3.14   /* assign the float value 3.14 to X:2000h */
```

## void twatch (long cycles)

The **twatch** function may be used in a signal function to delay continued execution for the specified number of CPU cycles. dScope updates the cycle counter while executing your target program.

### Example

The following signal function toggles the INT0 input (P3.2) every second.

```
signal void int0_signal (void) {
    while (1) {
        PORT3 |= 0x04;          /* pull INT0(P3.2) high */
        PORT3 &= ~0x04;         /* pull INT0(P3.2) low and generate interrupt */
        PORT3 |= 0x04;          /* pull INT0(P3.2) high again */
        twatch (XTAL / 12);      /* wait for 1 second */
    }
}
```

### NOTE

*The twatch function may be called only from within a signal function. Calls outside a signal function are not allowed and result in an error message.*

### Example

```
>twatch (4096)                /* Allowed only from within signal functions */
>twatch (4096)                /* Invalid at the command level */
-^
ERROR 145: TimeWatch(): not within signal()
```

## uchar uchar (ulong address)

The **uchar** function returns the value of the byte at *address* as an unsigned character. You may also use this function to assign an unsigned character value to a specific address. The range of values for an unsigned character is 0 to 255.

### Example

```
>uchar (c:0x0000)             /* display the character at C:0000h */
>uchar (x:0x1000)             /* display the character at X:1000h */
>uchar (x:0x1000) = 'A'       /* assign unsigned char 'A' to X:10000h */
```

## uint uint (ulong *address*)

The **uint** function returns the unsigned integer value of the two bytes starting at *address*. You may also use this function to assign an unsigned integer value to a specific address. The range of values for an unsigned integer is 0 to 65535.

### Example

```
>uint (d:0x50)           /* display the unsigned int at D:50h */
>uint (x:0x8000)          /* display the unsigned int at X:8000h */
>uint (x:0x8000) = 123    /* assign unsigned int 123 to X:8000h */
```

## ulong ulong (ulong *address*)

The **ulong** function returns the unsigned long integer value of the four bytes starting at *address*. You may also use this function to assign an unsigned long integer value to a specific address. The range of values for an unsigned long integer is 0 to 4294967295.

**10**

### Example

```
>ulong (i:0x90)          /* display the unsigned long at I:90h */
>ulong (x:0x8000)         /* display the unsigned long at X:8000h */
>ulong (x:0x2000) = 999    /* assign the 32-bit value 999 to X:2000h */
```

## word word (ulong *address*)

The **word** function returns the unsigned integer value of the two bytes starting at *address*. You may also use this function to assign an unsigned integer value to a specific address. The range of values for an unsigned integer is 0 to 65535.

### Example

```
>word (d:0x50)           /* display the unsigned int at D:50h */
>word (x:0x8000)          /* display the unsigned int at X:8000h */
>word (x:0x8000) = 123    /* assign unsigned int 123 to X:8000h */
```

## User Functions

User functions are functions you create to use with dScope. You may enter user functions directly in the command window or you may use the **INCLUDE** command to load a file that contains one or more user functions.

---

### NOTE

*dScope provides a number of system variables you may use in your user functions. Refer to “System Variables” on page 88 for more information.*

---

### Overview

User functions begin with **FUNC** keyword and are defined as follows:

```
FUNC return_type fname (parameter_list) {
    statements
}
```

where:

**return\_type** is the type of the value returned by the function and may be one of the following: **bit**, **char**, **float**, **int**, **long**, **uchar**, **uint**, **ulong**, **void**. You may use **void** if the function does not return a value. If no return type is specified the type **int** is assumed.

**fname** is the name of the function.

**parameter\_list** is the list of arguments that are passed to the function. Each argument must have a type and a name. If no arguments are passed to the function, use **void** for the **parameter\_list**. Multiple arguments are separated by commas.

**statements** are instructions the function carries out.

**{** is the open curly brace. The function definition is complete when the number of open braces is balanced with the number of the closing braces (“}”).

---

### NOTE

*The function return type, function name, argument list, and opening brace (“{”) must reside on the first line of the user function definition. This is illustrated in the following code examples.*

---

## Example

The following example shows a user function that displays the contents of the registers R0 through R7.

```
FUNC void ShowRegs (void) {
    printf ("----- REGISTERS -----\\n");
    printf (" R0 R1 R2 R3 R4 R5 R6 R7\\n");
    printf (" %02X %02X %02X %02X %02X %02X %02X %02X\\n",
            R0, R1, R2, R3, R4, R5, R6, R7);
    printf ("-----\\n");
}
```

You may either enter this function in the command window or load it from a text file. If you create a text file for the function, named **SHOWREGS.INC**, for example, you may load the user function as follows:

```
>INCLUDE SHOWREGS.INC
```

To invoke this function, type the following in the command window.

```
ShowRegs()
```

When invoked, the **ShowRegs** function displays the contents of the registers and appears similar to the following:

```
----- REGISTERS -----
R0 R1 R2 R3 R4 R5 R6 R7
10 A3 1F 1F BC 65 2D DD
-----
```

## Restrictions

- dScope checks user functions to ensure they return values that correspond to the function return type. Functions with a **void** return type must not return a value. Functions with a non-**void** return type must return a value. Note that dScope does not check each return path for a valid return value.
- The function return type, name, arguments, and opening curly brace must all reside on the first line of the function definition. For example, the following shows the correct definition of a user function.

```
func void MyFunc (void) {                               /* { on first line */  
    /* ... statements */  
}
```

The following shows the incorrect definition of a user function.

```
func void MyFunc (void)                                /* { not on first line ! */  
{  
    /* ... statements */  
}
```

- User functions may not invoke signal functions or the **twatch** function.
- The value of a local object is undefined until a value is assigned to it.
- Remove user functions using the **KILL FUNC** command.



## Signal Functions

Signal functions let you repeat operations, like signal inputs and pulses, in the background while dScope executes your target program. Signal functions help you simulate and test serial I/O, analog I/O, port communications, and other repetitive *external* events.

Signal functions execute in the background while dScope simulates your target program. Therefore, a signal function must call the **twatch** function at some point to delay and let dScope run your target program. dScope reports an error for signal functions that never call **twatch**.

---

### NOTE

*dScope provides a number of system variables you may use in your signal functions. Refer to “System Variables” on page 88 for more information.*

---

**10**

## Overview

Signal functions begin with the **SIGNAL** keyword and are defined as follows:

```
SIGNAL void fname (parameter_list) {  
    statements  
}
```

where:

*fname* is the name of the function.

*parameter\_list* is the list of arguments that are passed to the function. Each argument must have a type and a name. If no arguments are passed to the function, use **void** for the *parameter\_list*. Multiple arguments are separated by commas.

*statements* are instructions the function carries out.

{ is the open curly brace. The function definition is complete when the number of open braces is balanced with the number of the closing braces (“}”).

---

### NOTE

*The function name, argument list, and opening brace (“{”) must reside on the first line of the user function definition. This is illustrated in the following code examples.*

---

## Example

The following example shows a signal function that puts the character 'A' into the serial input buffer once every 1,000,000 clock ticks.

```
SIGNAL void StuffSin (void) {  
    while (1) {  
        sin = 'A';  
        twatch (1000000);  
    }  
}
```

You may enter this function in the command window or load it from a text file. If you create a text file for the function, named **SINPUT.INC**, for example, you may include the user function by including it as follows:

```
>INCLUDE SINPUT.INC
```

To invoke this function, type the following in the command window.

```
StuffSin()
```

When invoked, the **StuffSin** signal function puts an ASCII character 'A' in the serial input buffer, delays for 1,000,000 clock cycles, and repeats.

## Restrictions

The following restrictions apply to signal functions:

- The return type of a signal function must be **void**.
- A signal function may have a maximum of eight function parameters.
- A signal function may invoke other predefined functions and user functions.
- A signal function may not invoke another signal function.
- A signal function may be invoked by a user function.
- A signal function must call the **twatch** function at least once. Signal functions that never call **twatch** do not allow the target program time to execute. Since you cannot use **Ctrl+C** to abort a signal function, dScope may enter an infinite loop.

## Managing Signal Functions

dScope maintains a signal function queue in which active signal functions are maintained. A signal function may either be idle or running. A signal function that is idle is delayed while it waits for the number of CPU cycles specified in a call to **twatch** to expire. A signal function that is running is executing instruction inside the function. When the signal function invokes the **twatch** function, it becomes idle.

When you invoke a signal function, dScope adds that function to the signal function queue and marks it as running. If the function is already in the queue, dScope emits a warning (signal functions may only be activated once). Use the **SIGNAL STATE** command to view the status of active signal functions. Use the **SIGNAL KILL** command to remove an active signal from the queue.

When a signal function that is running invokes the **twatch** function, it is marked as idle and remains idle for the number of processor cycles passed to **twatch**. After remaining idle for the specified number of processor cycles, the signal function is reactivated and is marked as running. Execution continues at the statement after the **twatch** function was invoked and continues until **twatch** is invoked again.

If a signal function exits, because of a return statement, it is automatically removed from the queue of active signal functions. If a signal function is removed from the queue (by exiting or because of the **SIGNAL KILL** function) it may be invoked and added to the queue again.

## Analog Example

The following example shows a signal function that varies the input to analog input 0 on an 80517. The function increases and decreases the input voltage by 0.5 volts from 0V and an upper limit which is specified as the signal function's only argument. This signal function repeats indefinitely, delaying 25,000 cycles for each voltage step.

The function text was created by an editor and stored in a file named **ANALOG**. The function requires the 80517.DLL and uses **AIN0**, the first analog input of the 80C517 controller.

```
>  
>INCLUDE ANALOG  
>SIGNAL void analog0 (float limit) {  
    2:   
    3: float volts;
```

```

4:
5: printf ("ANALOG0 (%f) ENTERED\n", limit);
6: while (1) {                                     /* Indefinite */
7:     volts = 0;
8:     while (volts <= limit) {
9:         ain0 = volts;                             /* Analog input 0 */
10:        twatch (25000);                             /* 25000 cycles */
11:        volts += 0.5;                             /* Increase voltage */
12:    }
13:    volts = limit-0.5;
14:    while (volts >= 0.5) {
15:        ain0 = volts;
16:        twatch (25000);                             /* 25000 cycles */
17:        volts -= 0.5;                             /* Decrease voltage */
18:    }
19: }
20: }
>

```

## 10

The signal function **analog0** can then be invoked as follows:

```

>ANALOG0 (5.0)                                     /* Start of 'ANALOG()' */
ANALOG0 (5.000000) ENTERED

```

You may use the **SIGNAL STATE** command to display the current state of the **analog0** signal function.

```

>SIGNAL STATE
0 idle      Signal = ANALOG0 (line 10)

```

The leading number is an internal code and is followed by the status of the signal function: **idle** or **running**. The name of the signal function displays after the status and is followed by the line number that is executing.

Since the status of the signal function is **idle**, you can infer that **analog0** executed the **twatch** function (on line 10 of **analog0**) and is waiting for the specified number of cycles to elapse. When 25,000 cycles pass, **analog0** continues execution until the next call to **twatch** in line 10 or line 16.

The following command removes the **analog0** signal function from the queue of active signal functions.

```

>SIGNAL KILL ANALOG0

```

## Differences Between dScope Functions and ANSI C

There are a number of differences between ANSI C and the subset of features support in dScope user and signal functions.

- dScope does not differentiate between uppercase and lowercase. The names of objects and control statements may be written in either uppercase or lowercase.
- dScope has no preprocessor. Preprocessor directives like **#define**, **#include**, and **#ifdef** are not supported in dScope.
- dScope does not support global declarations. Scalar variables must be declared within a function definition. You may define symbols with the **DEFINE** command and use them like you would use a global variable.
- In dScope, variables may not be initialized when they are declared. Explicit assignment statements must be used to initialize variables.
- dScope functions only support scalar variable types. Structures, arrays, and pointers are not allowed. This applies to the function return type as well as the function parameters.
- dScope functions may only return scalar variable types. Pointers and structures may not be returned.
- dScope functions cannot be called recursively. During function execution, dScope recognizes recursive calls and aborts function execution if one is detected.
- dScope functions may only be invoked directly using the function name. Indirect function calls via pointers are not supported.
- dScope supports the new ANSI style of function declaration for functions with a parameter list. The old K&R format is not supported. For example, the following ANSI style function is acceptable.

```
func test (int pa1, int pa2) {                               /* ANSI type, correct */
    /* ... */
}
```

The following K&R style function is not acceptable.

```
func test (pa1, pa2)                                       /* Old K&R style is */
int pa1, pa2;                                           /* not supported */
{
    /* ... */
}
```

# 10

## Chapter 11. Error Messages

When dScope for Windows encounters an error, a numbered error message displays. The point where the error was recognized may be marked as shown below.

```
acc + r0 + r500
-----^
ERROR 125:  symbol or line not found

load c:\objs\measure
-----^
ERROR 109:  file does not exist
```

The following table lists the error messages that may be generated by dScope.

Error Number	Message and Description
100	<b>Illegal Digit In Number</b> An illegal digit in a number was detected. Make sure the digits of the number conform to the number base. For example, a digit greater than 7 in an octal number generates this error.
101	<b>Illegal Scope Qualifier</b> A scope qualifier consists of a module name specifier, a function name specifier, or a line number. If a function name is specified, an optional local symbol specifier such as <b>MEASUREMAIN</b> cmdbuf, <b>MEASURE</b> 225, or <b>MEASURE</b> index is also required.
104	<b>Too Many Qualifiers</b> A scope qualifier consists of at most three components: a module, a function or line number, and a local symbol (when a function is specified). See Error 101.
105	<b>Invalid Object File</b> An attempt was made to load an object file which is either corrupt or which does not conform to the OMF format specification.
106	<b>More Than 16000 Lines In One Module</b> A single program module may not contain more than 16000 line numbers. If this error occurs, you should break the large module into two smaller modules.
107	<b>Invalid Hex File</b> An attempt was made to load an Intel HEX file which is either corrupt or which does not conform to the Intel HEX, HEX86, or HEX386 format specifications.
108	<b>Checksum Error In Hex File</b> A checksum error was detected in the Intel HEX file being loaded.
109	<b>File Does Not Exist</b> The file specified in a <b>LOAD</b> or <b>INCLUDE</b> command does not exist.
110	<b>Literal Expected</b> A literal was expected but was not found. A literal is normally used to enter a file name with an optional path specification. A character string may also be used as a literal. For example: <pre>LOAD "c:\\user1\\project.90\\newcmd" /* String literal */ LOAD c:\\user1\\project.90\\newcmd /* Equivalent */</pre>



Error Number	Message and Description
111	<b>Illegal Character Constant</b> The specified character constant is invalid. Either the constant is not terminated with a single-quote ("'), or it contains an escape sequence for a number whose value is greater than 0xFF (255).
112	<b>Out Of Range Escape Value</b> An escape sequence in a string or character constant must be a value in the range 0x00 to 0xFF. Values outside this range are invalid.
113	<b>Unclosed String</b> The specified string is not closed. The string must be enclosed with double quote characters ("), for example "string". If strings are nested, the double quotes of the inner string must be escaped, for example "printf (^hello\n")".
114	<b>Syntax Error</b> An unexpected term was found at the marked position.
115	<b>Illegal Register Combination</b> The tokens found do not form a valid register name for the in-line assembler, for example, @DPTR+A instead of @A+DPTR or @R4 where only @R0 and @R1 are valid.
116	<b>Unclosed Comment</b> An unclosed comment was found during the analysis of a function. Comments must be opened and closed on the same line. Multiline comments are not allowed.
117	<b>Expression Too Complex</b> This error occurs when very complex expressions (possibly separated by commas) are analyzed. In command mode, expressions can be simplified or divided into several command lines. In function definition mode, comma expressions should be avoided. Separate statements should be used instead.
118	<b>') Expected</b> dScope expected a closing parenthesis, but found a different token. dScope marks the position where the parenthesis is expected.
119	<b>(' Expected</b> dScope expected an opening parenthesis, but found a different token. dScope marks the position where the parenthesis is expected.
120	<b> ';' Expected</b> dScope expected a semicolon, but found a different token. dScope marks the position where the semicolon is expected.
121	<b> '}' Expected</b> dScope expected a closing brace, but found a different token. dScope marks the position where the brace is expected.
122	<b> ']' Expected</b> dScope expected a closing bracket, but found a different token. dScope marks the position where the bracket is expected.
123	<b>Identifier Expected</b> An identifier was expected but some other token was found.
124	<b> ':' Expected</b> dScope expected a colon, but found a different token. dScope marks the position where the colon is expected.
125	<b>Symbol Or Line Not Found</b> The specified symbol or line number does not exist.



Error Number	Message and Description
126	<p><b>'Void' Expected</b></p> <p>This error is caused by an invalid or incomplete function definition. One of the following is required when starting a function definition:</p> <ol style="list-style-type: none"> <li>1. A parameter list <code>func void test (int i0, long l1)</code></li> <li>2. An empty parameter list <code>func void test ()</code></li> <li>3. A void parameter list <code>func void test (void)</code></li> </ol>
127	<p><b>Illegal Expression Token</b></p> <p>An expression contains a token that is not valid in the current context.</p>
128	<p><b>Illegal Type 'Void'</b></p> <p>A local variable was declared with type <b>void</b>. For example:</p> <pre>void variable;</pre> <p>Local variables with a function cannot be declared with type <b>void</b>.</p>
129	<p><b>Illegal Type</b></p> <p>An invalid type was used. For example, the following definition:</p> <pre>DEFINE void var</pre> <p>generates this error message. Refer to "DEFINE" on page 209 for more information about defining dScope symbols.</p>
130	<p><b>Illegal Or Unknown Memory Space</b></p> <p>The specifies expression does not include a valid memory space. For example, expressions used with the display or breakpoint commands require a memory space.</p>
131	<p><b>Illegal Type Conversion</b></p> <p>An invalid combination of operands was detected. This usually happens when the operand types are not compatible.</p>
132	<p><b>Unknown Struct/Union Member</b></p> <p>The specified structure member is not defined. Make sure you used the correct structure member name.</p>
133	<p><b>Undefined Peripheral I/O Location</b></p> <p>The name specified in an I/O register specifier is undefined. Use the <b>DIR VTREG</b> command to list the I/O register names available.</p>
134	<p><b>Operator Requires Lvalue</b></p> <p>The address of operator (&amp;) must have an expression on the right side which represents the address of an object. For example:</p> <pre>&amp;ACC /* Valid */ &amp;(ACC + R0) /* Invalid because of + */</pre>
135	<p><b>Illegal Bit Position</b></p> <p>An invalid bit position was detected. The bit position in a bit address must evaluate to a constant value from 0 to 7.</p>
136	<p><b>Invalid Base For Bit Type Expression</b></p> <p>Byte bases in bit addresses must adhere to the following rules:</p> <ol style="list-style-type: none"> <li>1. The address must be in the range 0x20 to 0x2F.</li> <li>2. The address must be in the range 0x80 to 0xFF and be evenly divisible by 8 (0x80, 0x88, 0x90, ...).</li> <li>3. There are no restrictions on 251 bit addresses. The address may be anywhere in the range 0x20 to 0xFF.</li> </ol>

Error Number	Message and Description
137	<b>Left Side Of '.Id' Requires Struct/Union</b> An invalid structure or union was detected. The expression on the left of the dot operator (.) must be the name of a structure or union. The dot operator is used to access members of structures and unions.
138	<b>Left Side Of '-&gt;Id' Requires Struct/Union Pointer</b> An invalid pointer to a structure or union was detected. The expression on the left of the pointer operator (->) must be the name of a pointer to a structure or union. The pointer operator is used to access members of structures and unions.
139	<b>[Dim] Applied To Non Array</b> An invalid array index was detected. An array index ([ ]) was applied to an expression that is not a pointer or an array.
140	<b>'&amp;' Requires Lvalue</b> An invalid expression was detected. The address of (&) operator was applied to a constant expression or another expression that does not have an address.
141	<b>'*': Invalid Indirection</b> The indirection with the pointer operator (*) requires an expression that points to a particular type. Typeless pointers cannot be dereferenced.
142	<b>Bad Op In Float Type Expression</b> dScope detected a bad operator in a floating-point expression. For example, the not operator (~) cannot be used with a floating-point expression.
143	<b>Invalid Left Hand Side Of Assignment Expression</b> An assignment was attempt to a constant or function. The left side of an assignment operator must contain an expression whose address represents a lvalue. Assignments to constants or functions are not possible.
144	<b>Call To Undefined Function</b> A call to an undefined dScope function was detected.
145	<b>Timewatch(): Not Within Signal()</b> A call to the <b>twatch</b> function from outside a signal function was detected. Calls to the predefined <b>twatch</b> function may only occur from within a signal function.
146	<b>Can't Activate Signal() From User Function</b> A attempt was made to invoke a signal function from another user function. Signal functions may not be invoked from any other function, signal or non-signal. Signal functions may be activated only at the command level.
147	<b>Incompatible Parameter Type</b> Some predefined functions like <b>printf</b> and <b>getint</b> expect a character string as a parameter. Anything else generates this error message.
148	<b>Missing Function Parameter</b> Too few parameters were specified in a function call. The number of parameters must agree with the function definition.
149	<b>Too Many Actual Parameters In Function Call</b> Too many parameters were specified in a function call. The number of parameters must agree with the function definition.
150	<b>Improper Operand</b> The specified operand does not correspond to the requirements of the assembler instruction.
151	<b>Div/Mod By Zero Error</b> A division or modulo by zero has been detected in an expression.

Error Number	Message and Description
152	<p><b>Function Difference Encountered</b> This error occurs under the following conditions:</p> <ol style="list-style-type: none"> <li>1. A function (func1) was defined.</li> <li>2. Another function (func2) was defined that calls func1.</li> <li>3. The return type, the number of parameters, or the parameter types of func1 are subsequently changed.</li> <li>4. The function func2 is invoked.</li> </ol> <p>dScope determines while executing func2 that func1 does not return the expected type, use the same number of parameters, or use the same parameter types.</p>
153	<p><b>Too Many Parameters</b> More parameters were passed to a dScope function than were declared in the function.</p>
154	<p><b>Recursive User Function Activation</b> Calls to dScope functions must not be recursive. If a recursive call is detected at run-time, the function is aborted.</p>
155	<p><b>Getnumber-Functions: Invalid Number</b> The predefined dScope functions <b>getint</b>, <b>getlong</b>, and <b>getfloat</b> expect numeric parameters without any operators. An invalid character was found in the input stream.</p>
156	<p><b>Unknown Identifier</b> An identifier from a scope qualifier is undefined.</p>
157	<p><b>Undefined Line Number</b> An unqualified line number, for which there is no associated code address, was used in an expression. For example, the following command:</p> <pre style="margin-left: 40px;">\123</pre> <p>when entered in the Command window generates this error if there is no code address associated with line number 123 for the current module.</p>
158	<p><b>{, Scope Stack Overflow (16)</b> A maximum of 16 levels of block nesting is allowed in function definitions.</p>
159	<p><b>Invalid 'Break/Continue'</b> The statements <b>break</b> and <b>continue</b> are valid only in <b>for</b>, <b>while</b>, <b>do</b>, and <b>switch</b> blocks.</p>
160	<p><b>'Case/Default': Missing Enclosing Switch</b> A <b>case</b> statement was found outside the scope of a <b>switch</b> block.</p>
161	<p><b>More Than One 'Default'</b> More than one <b>default</b> statement was found in a <b>switch</b> block. Only one <b>default</b> statement may exist in a <b>switch</b> block.</p>
162	<p><b>Duplicate Label</b> A duplicate label was detected in a function. Unique label names must be used within a single function. To correct this problem, change the name of the specified label.</p>
163	<p><b>Missing Return Expression</b> A function with a non-void return type did not return a value. This function must contain at least one <b>return</b> statement with a return value. Functions declared without an explicit return type default to an <b>int</b> return type.</p>
164	<p><b>Return Value On Void Function</b> A <b>void</b> function attempted to return a value to the caller. Functions declared with a <b>void</b> return type may not return a value.</p>



Error Number	Message and Description
165	<b>Undefined Label</b> A label referenced by a <b>goto</b> statement is not defined.
166	<b>Not An Integer Constant Expression</b> Expressions used in <b>case</b> statements must be one of the following types: <b>char</b> , <b>uchar</b> , <b>int</b> , or <b>uint</b> . Other types like <b>float</b> are not valid.
167	<b>Invalid Type On Controlling Expression</b> dScope has detected a non-integral expression used in a control loop ( <b>for</b> , <b>while</b> , and <b>do</b> ). Only integral expressions are allowed in control loops. Other types like structures, unions, and arrays are not allowed.
168	<b>Duplicate Identifier (Parm Or Local)</b> A duplicate variable name was detected. Names of parameters and local variables used in user functions must be unique.
169	<b>More Than 8 Parameters</b> Too many parameters were passed to a user function. dScope for Windows limits the number of parameters to a maximum of 8.
170	<b>Undefined Function</b> The specified function call does not reference a dScope function that is currently defined. Make sure the function name is correct.
171	<b>Can't Redefine Built-In Function</b> An attempt was made to redefine a built-in function. Refer to "Predefined Functions" on page 266 for a complete list of built-in functions.
172	<b>Can't Remove Built-In Function</b> A attempt was made to remove a predefined, built-in function. Functions such as <b>printf</b> and <b>memset</b> may not be removed. Refer to "Predefined Functions" on page 266 for a complete list of built-in functions.
173	<b>Signal Function: Can't Receive Or Return Value(s)</b> An attempt was made to define a signal function with a return value or with an argument list. Signal functions cannot receive or return any parameters and, therefore, must be declared as follows: <pre> signal void sigfunc (void) {     /* statements */ }</pre>
174	<b>Duplicate Case Label</b> A duplicated <b>case</b> statement was detected in a <b>switch</b> block. A <b>case</b> statement's constant expression must evaluate to a unique number for each <b>case</b> statement in a <b>switch</b> block.
175	<b>Insufficient Memory</b> This error indicates that dScope for Windows has run out of memory. Close any other open programs and check your system setup to determine the amount of memory remaining.
176	<b>Function Too Big</b> dScope for Windows encountered a user function that is too large. User functions are limited to a maximum of 512 labels and an internal code size of 32K.
177	<b>Signal Function Must Contain A Twatch() Call</b> dScope for Windows has detected a signal function that does not include a call to the <b>twatch</b> built-in function. Signal functions execute continuously in the background and cannot be aborted using <b>Ctrl+C</b> . For this reason, signal functions that never delay consume all available CPU time. A signal function must call the <b>twatch</b> function to delay and let dScope continue with the simulation.

Error Number	Message and Description
178	<p><b>Internal: Function Execution Error</b> dScope for Windows detected an internal error while executing a user function.</p> <p><b>NOTE:</b> Consult technical support or your distributor with the details of the function that caused the error.</p>
179	<p><b>This Signal() Already Activated</b> An attempt was made to activate a signal function that is already active. To reactivate a signal function, first remove the function from the active queue using the <b>SIGNAL KILL</b> command. Refer to "SIGNAL" on page 255 for more information.</p>
180	<p><b>Too Many Signal Functions</b> An attempt was made to active a excessive number of signal functions. At most, 10 signal functions may be active concurrently.</p>
181	<p><b>No Such Signal Function</b> The function specified in a <b>SIGNAL KILL</b> command is not in the queue of active signal functions. Refer to "SIGNAL" on page 255 for more information.</p>
182	<p><b>Limit Exceeded: Function Nesting (20)</b> In dScope for Windows, user functions may be nested a maximum of 20 levels. This error occurs if this limit is exceeded. When this happens, user function execution is aborted.</p>
183	<p><b>Command Not Allowed Now</b> The specified command cannot be executed. For example, this happens when the <b>RESET MAP</b> command is entered while dScope is executing the target program.</p>
184	<p><b>Include Nesting Limit Exceeded</b> Include files in user functions may be nested a maximum of 3 or 4 levels depending on the command context. This error occurs if this limit is exceeded. When this happens, user function execution is aborted.</p>
185	<p><b>Invalid Restricted Access Break Expression</b> While specifying a breakpoint with an access specification (<b>READ</b>, <b>WRITE</b>, or <b>READWRITE</b>), one of the following rules was neglected:</p> <ol style="list-style-type: none"> <li>1. The address portion of an expression must have a unique memory type. To specify a structure element, a constant offset is included here.</li> <li>2. The only permissible operators are &amp;, &amp;&amp;, &lt;, &lt;=, &gt;, &gt;=, ==, and !=. An expression corresponding to rule 1 must be on right side of one of these operators. An optional expression, that does not adhere to rule 1, may exist on the left of the operator.</li> </ol> <p>The following examples illustrate these rules.</p> <pre> struct time {     char hour;     char min;     char sec; } time;  int i0, i1;  BS WRITE time.sec BS WRITE time.sec + i0          /* Incorrect, mem type not unique */ BS WRITE time.sec == i0         /* Correct */ BS WRITE time.sec != i0*i1      /* Correct */ BS WRITE time.sec &amp;&amp; (ACC==5 &amp;&amp; i1!=i0) /* Correct */ </pre>

Error Number	Message and Description
186	<b>Invalid Item Number</b> The number specified in a delete watchpoint or delete breakpoint command does not identify a valid item. Make sure that the breakpoint or watchpoint number is valid.
187	<b>Access Out Of Bounds (Address)</b> An invalid memory location was accessed. This error occurs when a memory location that is not mapped is accessed.
188	<b>Invalid Or Out Of Range Number Base</b> The number base used in a watch expression was invalid. Only base 10 and base 16 numbers may be specified here. Other bases are not allowed.
189	<b>Address Value Out Of Range</b> The specified address is not in the range 0x000000 to 0xFFFFFFFF. Addresses outside this range are invalid.
190	<b>Exit-Address Required For PA-Range</b> An ending address must be specified for address range used for the performance analyzer. dScope for Windows cannot derive the ending address given only the starting address.
191	<b>PA-Range Overlaps An Existing PA-Range</b> The specified address range for the performance analyzer overlaps an existing performance analyzer range. Address ranges profiled by the performance analyzer may not overlay either partially or completely.
192	<b>Unknown Environment Variable</b> The specified variable is not defined. Variables you may use with the SET command are: F0, F1, F2, F3, F4, F5, F6, F7, F8, F9, F11, F12, and SRC. Refer to "SET" on page 248 for more information.
193	<b>Can't Redefine An Activated Function</b> A dScope for Windows user function that has been invoked and is still running may not be redefined. Wait for completion of the function before attempting to redefine it. If the function is a signal function, you may kill the function and then redefine it. Refer to "KILL" on page 225 for more information.
194	<b>Access Violation At &lt;Address&gt;</b> A memory access violation was detected at the specified address. This error occurs when the target program accesses memory in a fashion which is inconsistent with the memory map settings. This may happen in programs that erroneously access memory or when the memory map commands are incorrect or incomplete. Refer to "MAP" on page 234.
195	<b>Redefinition</b> An attempt was made to define a symbol that was previously defined using the DEFINE command.
196	<b>Log File Already Active</b> A LOG or SLOG command was issued and dScope attempted to open a file that was already open.
197	<b>Can't Create (Or Append To) Logfile</b> A LOG or SLOG command was issued and dScope attempted to open a file that could not be created. This may occur if the filename specifies a directory or a read-only file.
198	<b>Too Many Operands</b> When using the in-line assembler, an instruction was specified with too many operands. For example, the NOP instruction does not have any operands.



Error Number	Message and Description
199	<p><b>Number Of Operands Does Not Match Instruction</b></p> <p>When using the in-line assembler, the number of operands specified for an assembly instruction does not match the number of requested operands. For example, the <b>CALL</b> instruction requires exactly one operand. The following instruction:</p> <pre>CALL 0x1000, DPTR</pre> <p>generates this error.</p>
200	<p><b>Illegal Type Override</b></p> <p>dScope for Windows detected an illegal type override. This error occurs when a type override for an assembly instruction is invalid. For example:</p> <pre>VOID DATA 0x20</pre> <p>and</p> <pre>PTR XDATA 0x20</pre> <p>are not valid because <b>VOID</b> and <b>PTR</b> are not allowed in type overrides. Valid types are: <b>BIT</b>, <b>BYTE</b>, <b>WORD</b>, and <b>DWORD</b>.</p>
201	<p><b>Illegal Operand: '/Register' Or '#Register'</b></p> <p>An illegal register operand has been detected in an in-line assembler instruction. Register expressions may not be prefixed by '/' (not bit) or '#' (immediate).</p>
202	<p><b>Register: Illegal Type- Or Space Override</b></p> <p>A register operand cannot be preceded by a memory space or memory type override.</p>
203	<p><b>Instruction Does Not Match Cpu-Type</b></p> <p>dScope for Windows detected an assembler instruction that is not available on the current CPU. This is typically the result of an <b>LJMP</b> or <b>LCALL</b> instruction on the 80C751 and compatible derivatives.</p>
204	<p><b>Branch Target Out Of Range</b></p> <p>The branch target address given in a jmp or call instruction is out of range. The target for conditional jumps must be within +127 or -128 bytes relative to the current execution point.</p>
205	<p><b>Illegal Register Operand</b></p> <p>An invalid register operand was detected in an in-line assembler instruction. This occurs, for example, in the following instruction:</p> <pre>MOV A, R11</pre> <p>where only registers R0 to R7 are allowed for the second operand.</p>
206	<p><b>Invalid Short Value</b></p> <p>A short constant value is out of range. The value specified must be 1, 2, or 4. This error occurs in response to invalid constants used with the following 251 instructions:</p> <pre>INC Rn, #ShortConst DEC Rn, #ShortConst</pre>
207	<p><b>Invalid Instruction Operand</b></p> <p>An operand specified in an in-line assembler instruction did not match the required operand type. This happens, for example, when an immediate operand is specified in the place of a register operand.</p>

Error Number	Message and Description
<b>208</b>	<b>RESTRICTED VERSION: Code Size Limit Exceeded</b> An attempt was made to load a target program that exceeded the code size limits of dScope for Windows. This error occurs only with the limited version of dScope.  <i><b>NOTE:</b> Consult your distributor for information on how to get the unrestricted version of dScope for Windows.</i>
<b>209</b>	<b>Illegal Von Neumann Map Command</b> The address range in a von Neumann map command must not cross a 64K boundary. Additionally, the address range may not reside in the code segment (0xFF0000 to 0xFFFFF).
<b>210</b>	<b>Too Many Items</b> An attempt was made to define more than 255 performance analyzer ranges.
<b>211</b>	<b>Access To Non Existing SFR (0xnxxx)</b> An attempt was made to access a nonexistent SFR. This type of checking is performed only on those derivatives that do not allow access to nonexistent SFRs.
<b>212</b>	<b>Command Not Supported In Target Mode</b> The specified command is not supported in target mode when using the MON51 or MON251 CPU drivers. When using these drivers, features such as mapping memory and the performance analyzer are not available.
<b>213</b>	<b>Invalid Break Address</b> The specified breakpoint address is not valid. Typically, this error is caused in target mode when a break address overlaps the address range of an existing breakpoint.
<b>214</b>	<b>Unsupported Breakpoint Type</b> The type of breakpoint specified is not supported by the target driver. The MON251.DLL, for example, does not support <b>Access</b> type breakpoints.
<b>215</b>	<b>Premature End Of File</b> This error signals an error with an include file. Typically, this is caused by an include file that ends in the middle of a function definition. Function definitions may not cross file boundaries.



# Appendix A. CPU Driver Files

This appendix lists the CPU Driver Files (DLLs) that are provided with dScope for Windows. You must load a driver before you can debug any code with dScope. Refer to “Loading a CPU Driver DLL” on page 14 for more information about loading CPU drivers.

## Purpose of CPU Drivers

dScope for Windows is a general purpose debugger for the MCS<sup>®</sup> 51 and MCS<sup>®</sup> 251 microcontroller families. CPU drivers are provided to accommodate each particular 8051 or 251 derivative. The drivers are provided in the form of Dynamic Link Libraries (DLLs) which are loaded by dScope at run-time. Each DLL contains chip-specific and peripheral-specific configuration and simulation information.

The various members of the 8051 and 251 families typically differentiate in the peripherals integrated in the chip. To allow dScope the greatest flexibility, the simulation definitions of device specific peripherals are placed in separate CPU drivers (DLLs). This guarantees that as new microcontrollers are introduced, support can easily be expanded to support them.

With over 200 derivatives of the 8051 and 251, it is not possible to have a CPU driver file for each derivative. For instance, it is not possible to effectively simulate I<sup>2</sup>C Bus hardware. Nonetheless, in most cases, it is enough to simulate most of a given chip so that software may be written that can conditionally accommodate limitations in testing methodology.

Generally accepted C language programming techniques encourage modular software design. For instance, if your design uses a chip that is not completely supported with a CPU driver, you can easily test one portion (one or more modules) of your code with one driver, and another portion of your code with a different driver.

Another effective technique used by advanced developers to test code intended for chips not yet released, is to use the function definition capability of dScope to write specific peripheral device routines. In that way when specific programs or data locations are reached, or specified conditions achieved, a function is called to provide the appropriate response, activity, or result.

If no specific driver is loaded, a standard 8051 is simulated. However, none of the on-chip peripherals, like timers, ports, or A/D converters, are active or have

any significance. Using or initializing control or Special Function Registers (SFR) has no effect. The names of the I/O ports are not recognized and communication with the outside world is not possible.

## Peripheral Registers (VTREG)

The simulation DLL drivers define a number of VTREGs (pin symbols) which simulate I/O ports or special chip functions. The following table lists and describes many of the VTREG symbols declared by each of the following CPU driver DLLs.

VTREG	Description
<b>AINx</b>	An analog input pin on the chip. Your target program may read values you write to <b>AINx</b> VTREGs.
<b>PORTx</b>	A group of I/O pins for a port on the chip. For example, <b>PORT1</b> refers to all 8 or 16 pins of P1. These registers allow you to simulate port I/O.
<b>SxIN</b>	The input buffer of serial interface <b>x</b> . You may write 8-bit or 9-bit values to <b>SxIN</b> . These are read by your target program. You may read <b>SxIN</b> to determine when the input buffer is ready for another character. The value 0xFFFF signals that the previous value is completely processed and a new value may be written.
<b>SxOUT</b>	The output buffer of serial interface <b>x</b> . dScope copies 8-bit or 9-bit values (as programmed) to the <b>SxOUT</b> VTREG.
<b>SxTIME</b>	Defines the baudrate timing of the serial interface <b>x</b> . When <b>SxTIME</b> is 1, dScope simulates the timing of the serial interface using the programmed baudrate. When <b>SxTIME</b> is 0 (the default value), the programmed baudrate timing is ignored and serial transmission time is instantaneous.
<b>XTAL</b>	Defines the crystal frequency of the simulated CPU. All timings are derived from the value assigned to this VTREG.

A

### NOTE

*You may use the VTREGs to simulate external input and output including interfacing to internal peripherals like interrupts and timers. For example, if you toggle bit 2 of **PORT3** (on the 8051 drivers), the CPU driver simulates external interrupt 0.*

## CPU Driver List

dScope provides you with a number of CPU drivers for each platform. They are listed here by CPU type and are described in detail on the pages that follow.

### **dScope 51 contains the following CPU drivers:**

- 80320.DLL
- 80410.DLL
- 8051.DLL
- 8051FX.DLL
- 80515.DLL
- 80515A.DLL
- 80517.DLL
- 80517A.DLL
- 8052.DLL
- 80552.DLL
- 80751.DLL
- 80781.DLL
- MON51.DLL
- RISM51.DLL

### **dScope 251 contains the following CPU drivers:**

- 80251A1.DLL
- 80251G1.DLL
- 80251S.DLL
- MON251.DLL
- RISM251.DLL

**dScope 166 contains the following CPU drivers:**

- 80166.DLL
- 80167.DLL
- MON166.DLL

Each CPU driver description lists the derivatives supported by the driver, the hardware components supported and excluded, and the VTREG symbols available.

## 80166.DLL

80166.DLL contains the driver for the 167 and ST10 chips. This DLL simulates the logical and timing behavior of the following hardware components:

- Capture/Compare (CAPCOM) Unit (Timer 0/1)
- General Purpose Timer (GPT1) Unit (Timer 2/3/4)
- General Purpose Timer (GPT2) Unit (Timer 5/6)
- A/D Converter (10)
- Serial Channels (2)
- Watchdog Timer
- Parallel Ports (6)
- Interrupt System
- Peripheral Event Controller (PEC)
- Power Saving Modes (Idle, Power-down)

80166.DLL defines the following VTREG symbols for the on-chip peripheral registers:

VTREG (Pin Symbol)	Description
<b>AIN0</b>	Analog input line AIN0 (floating-point value)
<b>AIN1</b>	Analog input line AIN1 (floating-point value)
<b>AIN2</b>	Analog input line AIN2 (floating-point value)
<b>AIN3</b>	Analog input line AIN3 (floating-point value)
<b>AIN4</b>	Analog input line AIN4 (floating-point value)
<b>AIN5</b>	Analog input line AIN5 (floating-point value)
<b>AIN6</b>	Analog input line AIN6 (floating-point value)
<b>AIN7</b>	Analog input line AIN7 (floating-point value)
<b>AIN8</b>	Analog input line AIN8 (floating-point value)
<b>AIN9</b>	Analog input line AIN9 (floating-point value)
<b>BUSACT</b>	BUSACT# line (1 bit). This bus configuration pin is necessary for calculating the execution time of a program. You must invoke the <b>RESET</b> command after changing the value of <b>BUSACT</b> .

VTREG (Pin Symbol)	Description										
<b>EBC</b>	External Bus Configuration register after reset (2 bits). This register is copied into the <b>BUSCON0</b> SFR at RESET. <b>EBC</b> may have one of the following values: <table> <tr> <th>Value</th><th>Description</th></tr> <tr> <td>0</td><td>8-bit data bus, non-multiplexed</td></tr> <tr> <td>1</td><td>8-bit data bus, multiplexed</td></tr> <tr> <td>2</td><td>16-bit data bus, non-multiplexed</td></tr> <tr> <td>3</td><td>16-bit data bus, multiplexed</td></tr> </table> <p>Bus configuration pins are necessary for calculating the execution time of a program. You must invoke the <b>RESET</b> command after changing the value of <b>EBC</b>.</p>	Value	Description	0	8-bit data bus, non-multiplexed	1	8-bit data bus, multiplexed	2	16-bit data bus, non-multiplexed	3	16-bit data bus, multiplexed
Value	Description										
0	8-bit data bus, non-multiplexed										
1	8-bit data bus, multiplexed										
2	16-bit data bus, non-multiplexed										
3	16-bit data bus, multiplexed										
<b>PORT0</b>	Digital I/O lines of PORT 0 (16-bit)										
<b>PORT1</b>	Digital I/O lines of PORT 1 (16-bit)										
<b>PORT2</b>	Digital I/O lines of PORT 2 (16-bit)										
<b>PORT3</b>	Digital I/O lines of PORT 3 (16-bit)										
<b>PORT4</b>	Digital I/O lines of PORT 4 (2-bit)										
<b>PORT5</b>	Digital/analog I/O lines of PORT 5 (10-bit)										
<b>S0IN</b>	Serial input for SERIAL CHANNEL 0 (9 bits)										
<b>S0OUT</b>	Serial output for SERIAL CHANNEL 0 (9 bits)										
<b>S0TIME</b>	Serial timing enable for SERIAL CHANNEL 0										
<b>S1IN</b>	Serial input for SERIAL CHANNEL 1 (9 bits)										
<b>S1OUT</b>	Serial output for SERIAL CHANNEL 1 (9 bits)										
<b>S1TIME</b>	Serial timing enable for SERIAL CHANNEL 1										
<b>XTAL</b>	Oscillator frequency										

## 80167.DLL

80167.DLL contains the driver for the C167, C165, C163, and C161 chips. This DLL simulates the logical and timing behavior of the following hardware components:

- Capture/Compare (CAPCOM) Unit (Timer 0/1)
- General Purpose Timer (GPT1) Unit (Timer 2/3/4)
- General Purpose Timer (GPT2) Unit (Timer 5/6)
- A/D Converter (16)
- Serial Channels (2)
- Watchdog Timer
- Parallel Ports (9)
- Interrupt System
- Peripheral Event Controller (PEC)
- Power Saving Modes (Idle, Power-down)

---

### NOTE

*The CAN interface and the synchronous serial interface of the C167 are not supported.*

---

80167.DLL defines the following VTREG symbols for the on-chip peripheral registers:

CPU-pin Symbol	Description
AIN0	Analog input line AIN0 (floating-point value)
AIN1	Analog input line AIN1 (floating-point value)
AIN2	Analog input line AIN2 (floating-point value)
AIN3	Analog input line AIN3 (floating-point value)
AIN4	Analog input line AIN4 (floating-point value)
AIN5	Analog input line AIN5 (floating-point value)
AIN6	Analog input line AIN6 (floating-point value)
AIN7	Analog input line AIN7 (floating-point value)
AIN8	Analog input line AIN8 (floating-point value)
AIN9	Analog input line AIN9 (floating-point value)
AIN10	Analog input line AIN10 (floating-point value)

CPU-pin Symbol	Description										
<b>AIN11</b>	Analog input line AIN11 (floating-point value)										
<b>AIN12</b>	Analog input line AIN12 (floating-point value)										
<b>AIN13</b>	Analog input line AIN13 (floating-point value)										
<b>AIN14</b>	Analog input line AIN14 (floating-point value)										
<b>AIN15</b>	Analog input line AIN15 (floating-point value)										
<b>EA</b>	Status of the EA pin (1 bit). This configuration pin is necessary for calculating the execution time of a program. You must invoke the <b>RESET</b> command after changing the value of <b>EA</b> .										
<b>EBC</b>	<p>External Bus Configuration after reset (2 bits). <b>EBC</b> may have one of the following values:</p> <table> <tr> <th>Value</th><th>Description</th></tr> <tr> <td><b>0</b></td><td>8-bit data bus, non-multiplexed</td></tr> <tr> <td><b>1</b></td><td>8-bit data bus, multiplexed</td></tr> <tr> <td><b>2</b></td><td>16-bit data bus, non-multiplexed</td></tr> <tr> <td><b>3</b></td><td>16-bit data bus, multiplexed</td></tr> </table> <p>Bus configuration pins are necessary for calculating the execution time of a program. Bit 0 of <b>EBC</b> represents EBC0. Bit 1 of <b>EBC</b> represents EBC1. When <b>EBC</b> is 3, EBC0 and EBC1 are both set. You must invoke the <b>RESET</b> command after changing the value of <b>EBC</b>.</p>	Value	Description	<b>0</b>	8-bit data bus, non-multiplexed	<b>1</b>	8-bit data bus, multiplexed	<b>2</b>	16-bit data bus, non-multiplexed	<b>3</b>	16-bit data bus, multiplexed
Value	Description										
<b>0</b>	8-bit data bus, non-multiplexed										
<b>1</b>	8-bit data bus, multiplexed										
<b>2</b>	16-bit data bus, non-multiplexed										
<b>3</b>	16-bit data bus, multiplexed										
<b>PORT0H</b>	Digital I/O lines of PORT 0H (8-bit)										
<b>PORT0L</b>	Digital I/O lines of PORT 0L (8-bit)										
<b>PORT1H</b>	Digital I/O lines of PORT 1H (8-bit)										
<b>PORT1L</b>	Digital I/O lines of PORT 1L (8-bit)										
<b>PORT2</b>	Digital I/O lines of PORT 2 (16-bit)										
<b>PORT3</b>	Digital I/O lines of PORT 3 (16-bit)										
<b>PORT4</b>	Digital I/O lines of PORT 4 (8-bit)										
<b>PORT5</b>	Digital/analog Input Lines of PORT 5 (16-bit)										
<b>PORT6</b>	Digital I/O lines of PORT 6 (8-bit)										
<b>PORT7</b>	Digital I/O lines of PORT 7 (8-bit)										
<b>PORT8</b>	Digital I/O lines of PORT 8 (8-bit)										
<b>S0IN</b>	Serial input for SERIAL CHANNEL 0 (9 bits)										
<b>S0OUT</b>	Serial output for SERIAL CHANNEL 0 (9 bits)										
<b>S0TIME</b>	Serial timing enable for SERIAL CHANNEL 0										
<b>S1IN</b>	Serial input for SERIAL CHANNEL 1 (9 bits)										
<b>S1OUT</b>	Serial output for SERIAL CHANNEL 1 (9 bits)										
<b>S1TIME</b>	Serial timing enable for SERIAL CHANNEL 1										
<b>XTAL</b>	Oscillator frequency										



## 80251A1.DLL

80251A1.DLL contains the driver for the 8xC251A1 device from Temic. This DLL simulates the logical and timing behavior of the following hardware components:

- Timer 0
- Timer 1
- Pulse Measurement Unit
- Event and Waveform Controller (PCA & Enhanced PCA Mode)
- Serial Interface and Internal Baudrate Generator
- A/D Converter
- Interrupt System
- Power Saving Modes (Idle, Power-down)

80251A1.DLL defines the following VTREG symbols for the on-chip peripheral registers:

CPU-pin Symbol	Description
<b>PORT0</b>	Digital I/O lines of PORT 0 (8-bit)
<b>PORT1</b>	Digital I/O lines of PORT 1 (8-bit)
<b>PORT2</b>	Digital I/O lines of PORT 2 (8-bit)
<b>PORT3</b>	Digital I/O lines of PORT 3 (8-bit)
<b>AIN0</b>	Analog input line AIN0 (floating-point value)
<b>AIN1</b>	Analog input line AIN1 (floating-point value)
<b>AIN2</b>	Analog input line AIN2 (floating-point value)
<b>AIN3</b>	Analog input line AIN3 (floating-point value)
<b>SIN</b>	Serial input for SERIAL CHANNEL 0 (9-bit)
<b>SOUT</b>	Serial output for SERIAL CHANNEL 0 (9-bit)
<b>STIME</b>	Serial timing enable
<b>VAGND</b>	Analog reference voltage VAGND (floating-point value)
<b>VAREF</b>	Analog reference voltage VAREF (floating-point value)
<b>XTAL</b>	Oscillator frequency

## 80251G1.DLL

80251G1.DLL contains the driver for the 8xC251G1 device from Temic. This DLL simulates the logical and timing behavior of the following hardware components:

- Timer 0
- Timer 1
- Timer 2 with Reload/Capture Register
- Event and Waveform Controller (PCA & Enhanced PCA Mode)
- Serial Interface and Internal Baudrate Generator
- Synchronous Serial Link Controller
- Interrupt System
- Watchdog
- Power Saving Modes (Idle, Power-down)

---

### **NOTE**

*Timer 2 in Clock-Out Mode is not supported.*

---

80251G1.DLL defines the following VTREG symbols for the on-chip peripheral registers:

CPU-pin Symbol	Description
<b>PORT0</b>	Digital I/O lines of PORT 0 (8-bit)
<b>PORT1</b>	Digital I/O lines of PORT 1 (8-bit)
<b>PORT2</b>	Digital I/O lines of PORT 2 (8-bit)
<b>PORT3</b>	Digital I/O lines of PORT 3 (8-bit)
<b>SIN</b>	Serial input for SERIAL CHANNEL 0 (9-bit)
<b>SOUT</b>	Serial output for SERIAL CHANNEL 0 (9-bit)
<b>STIME</b>	Serial timing enable
<b>XTAL</b>	Oscillator frequency

## 80251S.DLL

80251S.DLL contains the driver for the 8xC251SA, 8xC251SB, 8xC251SQ, and 8xC251SP chips. This DLL simulates the logical and timing behavior of the following hardware components:

- Timer 0
- Timer 1
- Timer 2 with Reload/Capture Register
- PCA Timer/Counter with five 16-bit Capture/Compare Modules
- Serial Interface
- Interrupt System
- Watchdog Timer
- Power Saving Modes (Idle, Power-down)

---

### **NOTE**

*Timer 2 in Clock-Out Mode is not supported.*

---

80251S.DLL defines the following VTREG symbols for the on-chip peripheral registers:

CPU-pin Symbol	Description
<b>PORT0</b>	Digital I/O lines of PORT 0 (8-bit)
<b>PORT1</b>	Digital I/O lines of PORT 1 (8-bit)
<b>PORT2</b>	Digital I/O lines of PORT 2 (8-bit)
<b>PORT3</b>	Digital I/O lines of PORT 3 (8-bit)
<b>SIN</b>	Serial input for SERIAL CHANNEL 0 (9-bit)
<b>SOUT</b>	Serial output for SERIAL CHANNEL 0 (9-bit)
<b>STIME</b>	Serial timing enable
<b>XTAL</b>	Oscillator frequency

## 80320.DLL

80320.DLL contains the driver for the 80C320, 80C520, and other similar chips. This DLL simulates the logical and timing behavior of the following hardware components:

- Timer 0
- Timer 1
- Timer 2 with Reload/Capture Register
- Watchdog
- Two Serial Interfaces
- Interrupt System
- Power Saving Modes (Idle, Power-down)

80320.DLL defines the following VTREG symbols for the on-chip peripheral registers:

CPU-pin Symbol	Description
<b>PORT0</b>	Digital I/O lines of PORT 0 (8-bit)
<b>PORT1</b>	Digital I/O lines of PORT 1 (8-bit)
<b>PORT2</b>	Digital I/O lines of PORT 2 (8-bit)
<b>PORT3</b>	Digital I/O lines of PORT 3 (8-bit)
<b>S0IN</b>	Serial input for SERIAL CHANNEL 0 (9-bit)
<b>S0OUT</b>	Serial output for SERIAL CHANNEL 0 (9-bit)
<b>S1IN</b>	Serial input for SERIAL CHANNEL 1 (9-bit)
<b>S1OUT</b>	Serial output for SERIAL CHANNEL 1 (9-bit)
<b>S0TIME</b>	Serial timing enable for SERIAL CHANNEL 0
<b>S1TIME</b>	Serial timing enable for SERIAL CHANNEL 1
<b>XTAL</b>	Oscillator frequency

## 80410.DLL

80410.DLL contains the driver for the 80CL410 and other similar chips. This DLL simulates the logical and timing behavior of the following hardware components:

- Timer 0
- Timer 1
- Interrupt System
- Power Saving Modes (Idle, Power-down)

The following hardware components are not simulated:

- Serial Interface (I<sup>2</sup>C Bus)

80410.DLL defines the following VTREG symbols for the on-chip peripheral registers:

CPU-pin Symbol	Description
<b>PORT0</b>	Digital I/O lines of PORT 0 (8-bit)
<b>PORT1</b>	Digital I/O lines of PORT 1 (8-bit)
<b>PORT2</b>	Digital I/O lines of PORT 2 (8-bit)
<b>PORT3</b>	Digital I/O lines of PORT 3 (8-bit)
<b>XTAL</b>	Oscillator frequency

## 8051.DLL

8051.DLL contains the driver for the 8051, 8031, 80C51, 80C31, 80C52T2, and other similar chips. This DLL simulates the logical and timing behavior of the following hardware components:

- Timer 0
- Timer 1
- Serial Interface
- Interrupt System
- Power Saving Modes (Idle, Power-down)

8051.DLL defines the following VTREG symbols for the on-chip peripheral registers:

CPU-pin Symbol	Description
<b>PORT0</b>	Digital I/O lines of PORT 0 (8-bit)
<b>PORT1</b>	Digital I/O lines of PORT 1 (8-bit)
<b>PORT2</b>	Digital I/O lines of PORT 2 (8-bit)
<b>PORT3</b>	Digital I/O lines of PORT 3 (8-bit)
<b>SIN</b>	Serial input for SERIAL CHANNEL 0 (9-bit)
<b>SOUT</b>	Serial output for SERIAL CHANNEL 0 (9-bit)
<b>STIME</b>	Serial timing enable
<b>XTAL</b>	Oscillator frequency

## 8051FX.DLL

8051FX.DLL contains the driver for the 8051FA, 8051FB, 8051FC, and other similar chips. This DLL simulates the logical and timing behavior of the following hardware components:

- Timer 0
- Timer 1
- Timer 2 with Reload/Capture Register
- PCA Timer/Counter with five 16-bit Capture/Compare Modules
- Serial Interface
- Interrupt System
- Power Saving Modes (Idle, Power-down)

---

### **NOTE**

*Timer 2 in Clock-Out Mode, is not supported.*

---

8051FX.DLL defines the following VTREG symbols for the on-chip peripheral registers:

CPU-pin Symbol	Description
<b>PORT0</b>	Digital I/O lines of PORT 0 (8-bit)
<b>PORT1</b>	Digital I/O lines of PORT 1 (8-bit)
<b>PORT2</b>	Digital I/O lines of PORT 2 (8-bit)
<b>PORT3</b>	Digital I/O lines of PORT 3 (8-bit)
<b>SIN</b>	Serial input for SERIAL CHANNEL 0 (9-bit)
<b>SOUT</b>	Serial output for SERIAL CHANNEL 0 (9-bit)
<b>STIME</b>	Serial timing enable
<b>XTAL</b>	Oscillator frequency

## 80515.DLL

80515.DLL contains drivers for 80515, 80512, and other similar chips. It simulates the logical and timing behavior of the following hardware components:

- Timer 0
- Timer 1
- Timer 2 with Reload/Capture Register
- A/D Converter
- Watchdog
- Serial Interface
- Interrupt System
- Power Saving Modes (Idle, Power-down)

80515.DLL defines the following VTREG symbols for the on-chip peripheral registers:

CPU-pin Symbol	Description
<b>AIN0</b>	Analog input line AIN0 (floating-point value)
<b>AIN1</b>	Analog input line AIN1 (floating-point value)
<b>AIN2</b>	Analog input line AIN2 (floating-point value)
<b>AIN3</b>	Analog input line AIN3 (floating-point value)
<b>AIN4</b>	Analog input line AIN4 (floating-point value)
<b>AIN5</b>	Analog input line AIN5 (floating-point value)
<b>AIN6</b>	Analog input line AIN6 (floating-point value)
<b>AIN7</b>	Analog input line AIN7 (floating-point value)
<b>PORT0</b>	Digital I/O lines of PORT 0 (8-bit)
<b>PORT1</b>	Digital I/O lines of PORT 1 (8-bit)
<b>PORT2</b>	Digital I/O lines of PORT 2 (8-bit)
<b>PORT3</b>	Digital I/O lines of PORT 3 (8-bit)
<b>PORT4</b>	Digital I/O lines of PORT 4 (8-bit)
<b>PORT5</b>	Digital I/O lines of PORT 5 (8-bit)
<b>SIN</b>	Serial input for SERIAL CHANNEL 0 (9-bit)
<b>SOUT</b>	Serial output for SERIAL CHANNEL 0 (9-bit)
<b>STIME</b>	Serial timing enable
<b>VAGND</b>	Analog reference voltage VAGND (floating-point value)
<b>VAREF</b>	Analog reference voltage VAREF (floating-point value)
<b>XTAL</b>	Oscillator frequency



## 80515A.DLL

80515A.DLL contains the driver for the 80C515A other similar chips. This DLL simulates the logical and timing behavior of the following hardware components:

- Timer 0
- Timer 1
- Timer 2 with Reload/Capture Register
- A/D Converter
- Watchdog
- Serial Interface
- Interrupt System
- Power Saving Modes (Idle, Power-down)

80515A.DLL defines the following VTREG symbols for the on-chip peripheral registers:

CPU-pin Symbol	Description
<b>AIN0</b>	Analog input line AIN0 (floating-point value)
<b>AIN1</b>	Analog input line AIN1 (floating-point value)
<b>AIN2</b>	Analog input line AIN2 (floating-point value)
<b>AIN3</b>	Analog input line AIN3 (floating-point value)
<b>AIN4</b>	Analog input line AIN4 (floating-point value)
<b>AIN5</b>	Analog input line AIN5 (floating-point value)
<b>AIN6</b>	Analog input line AIN6 (floating-point value)
<b>AIN7</b>	Analog input line AIN7 (floating-point value)
<b>PORT0</b>	Digital I/O lines of PORT 0 (8-bit)
<b>PORT1</b>	Digital I/O lines of PORT 1 (8-bit)
<b>PORT2</b>	Digital I/O lines of PORT 2 (8-bit)
<b>PORT3</b>	Digital I/O lines of PORT 3 (8-bit)
<b>PORT4</b>	Digital I/O lines of PORT 4 (8-bit)
<b>PORT5</b>	Digital I/O lines of PORT 5 (8-bit)
<b>SIN</b>	Serial input for SERIAL CHANNEL 0 (9-bit)
<b>SOUT</b>	Serial output for SERIAL CHANNEL 0 (9-bit)
<b>STIME</b>	Serial timing enable
<b>VAGND</b>	Analog reference voltage VAGND (floating-point value)
<b>VAREF</b>	Analog reference voltage VAREF (floating-point value)
<b>XTAL</b>	Oscillator frequency

## 80517.DLL

80517.DLL contains the driver for the 80C517 other similar chips. This DLL simulates the logical and timing behavior of the following hardware components:

- Timer 0
- Timer 1
- Timer 2 with Reload/Capture Register
- A/D Converter
- Watchdog
- Two Serial Interfaces
- Arithmetic Unit
- Eight Data Pointers
- Interrupt System
- Power Saving Modes (Idle, Power-down)

The following hardware components are not simulated:

- Exact timing of the Arithmetic Unit
- Digital Input Lines of Port 7 and Port 8
- Oscillator Watchdog

80517.DLL defines the following VTREG symbols for the on-chip peripheral registers:

CPU-pin Symbol	Description
<b>AIN0</b>	Analog input line AIN0 (floating-point value)
<b>AIN1</b>	Analog input line AIN1 (floating-point value)
<b>AIN2</b>	Analog input line AIN2 (floating-point value)
<b>AIN3</b>	Analog input line AIN3 (floating-point value)
<b>AIN4</b>	Analog input line AIN4 (floating-point value)
<b>AIN5</b>	Analog input line AIN5 (floating-point value)
<b>AIN6</b>	Analog input line AIN6 (floating-point value)
<b>AIN7</b>	Analog input line AIN7 (floating-point value)
<b>AIN8</b>	Analog input line AIN8 (floating-point value)
<b>AIN9</b>	Analog input line AIN9 (floating-point value)

CPU-pin Symbol	Description
<b>AIN10</b>	Analog input line AIN10 (floating-point value)
<b>AIN11</b>	Analog input line AIN11 (floating-point value)
<b>PORT0</b>	Digital I/O lines of PORT 0 (8-bit)
<b>PORT1</b>	Digital I/O lines of PORT 1 (8-bit)
<b>PORT2</b>	Digital I/O lines of PORT 2 (8-bit)
<b>PORT3</b>	Digital I/O lines of PORT 3 (8-bit)
<b>PORT4</b>	Digital I/O lines of PORT 4 (8-bit)
<b>PORT5</b>	Digital I/O lines of PORT 5 (8-bit)
<b>PORT6</b>	Digital I/O lines of PORT 6 (8-bit)
<b>PORT7</b>	Digital I/O lines of PORT 7 (8-bit)
<b>PORT8</b>	Digital I/O lines of PORT 8 (8-bit)
<b>S0IN</b>	Serial input for SERIAL CHANNEL 0 (9-bit)
<b>S0OUT</b>	Serial output for SERIAL CHANNEL 0 (9-bit)
<b>S1IN</b>	Serial input for SERIAL CHANNEL 1 (9-bit)
<b>S1OUT</b>	Serial output for SERIAL CHANNEL 1 (9-bit)
<b>STIME</b>	Serial timing enable
<b>VAGND</b>	Analog reference voltage VAGND (floating-point value)
<b>VAREF</b>	Analog reference voltage VAREF (floating-point value)
<b>XTAL</b>	Oscillator frequency

## 80517A.DLL

80517A.DLL contains the driver for the 80C517A other similar chips. This DLL simulates the logical and timing behavior of the following hardware components:

- Timer 0
- Timer 1
- Timer 2 with Reload/Capture Register
- A/D Converter
- Watchdog
- Two Serial Interfaces
- Arithmetic Unit
- Eight Data Pointers
- Interrupt System
- Power Saving Modes (Idle, Power-down)

The following hardware components are not simulated:

- Exact timing of the Arithmetic Unit
- Digital Input Lines of Port 7 and Port 8
- Oscillator Watchdog

80517A.DLL defines the following VTREG symbols for the on-chip peripheral registers:

CPU-pin Symbol	Description
<b>AIN0</b>	Analog input line AIN0 (floating-point value)
<b>AIN1</b>	Analog input line AIN1 (floating-point value)
<b>AIN2</b>	Analog input line AIN2 (floating-point value)
<b>AIN3</b>	Analog input line AIN3 (floating-point value)
<b>AIN4</b>	Analog input line AIN4 (floating-point value)
<b>AIN5</b>	Analog input line AIN5 (floating-point value)
<b>AIN6</b>	Analog input line AIN6 (floating-point value)
<b>AIN7</b>	Analog input line AIN7 (floating-point value)
<b>AIN8</b>	Analog input line AIN8 (floating-point value)
<b>AIN9</b>	Analog input line AIN9 (floating-point value)

CPU-pin Symbol	Description
<b>AIN10</b>	Analog input line AIN10 (floating-point value)
<b>AIN11</b>	Analog input line AIN11 (floating-point value)
<b>PORT0</b>	Digital I/O lines of PORT 0 (8-bit)
<b>PORT1</b>	Digital I/O lines of PORT 1 (8-bit)
<b>PORT2</b>	Digital I/O lines of PORT 2 (8-bit)
<b>PORT3</b>	Digital I/O lines of PORT 3 (8-bit)
<b>PORT4</b>	Digital I/O lines of PORT 4 (8-bit)
<b>PORT5</b>	Digital I/O lines of PORT 5 (8-bit)
<b>PORT6</b>	Digital I/O lines of PORT 6 (8-bit)
<b>PORT7</b>	Digital I/O lines of PORT 7 (8-bit)
<b>PORT8</b>	Digital I/O lines of PORT 8 (8-bit)
<b>S0IN</b>	Serial input for SERIAL CHANNEL 0 (9-bit)
<b>S0OUT</b>	Serial output for SERIAL CHANNEL 0 (9-bit)
<b>S1IN</b>	Serial input for SERIAL CHANNEL 1 (9-bit)
<b>S1OUT</b>	Serial output for SERIAL CHANNEL 1 (9-bit)
<b>STIME</b>	Serial timing enable
<b>VAGND</b>	Analog reference voltage VAGND (floating-point value)
<b>VAREF</b>	Analog reference voltage VAREF (floating-point value)
<b>XTAL</b>	Oscillator frequency

## 8052.DLL

8052.DLL contains the driver for the 8052, 8032, 80C52, 80C32, and other similar chips. This DLL simulates the logical and timing behavior of the following hardware components:

- Timer 0
- Timer 1
- Timer 2 with Reload/Capture Register
- Serial Interface
- Interrupt System
- Power Saving Modes (Idle, Power-down)

8052.DLL defines the following VTREG symbols for the on-chip peripheral registers:

CPU-pin Symbol	Description
<b>PORT0</b>	Digital I/O lines of PORT 0 (8-bit)
<b>PORT1</b>	Digital I/O lines of PORT 1 (8-bit)
<b>PORT2</b>	Digital I/O lines of PORT 2 (8-bit)
<b>PORT3</b>	Digital I/O lines of PORT 3 (8-bit)
<b>SIN</b>	Serial input for SERIAL CHANNEL 0 (9-bit)
<b>SOUT</b>	Serial output for SERIAL CHANNEL 0 (9-bit)
<b>STIME</b>	Serial timing enable
<b>XTAL</b>	Oscillator frequency

## 80552.DLL

80552.DLL contains the driver for the 80552 other similar chips. This DLL simulates the logical and timing behavior of the following hardware components:

- Timer 0
- Timer 1
- Timer 2 with Reload/Capture Register
- A/D Converter
- Watchdog
- PWM Outputs
- Serial Interface 0
- Interrupt System

The following hardware components are not simulated:

- Serial Interface 1 (I<sup>2</sup>C Bus)

80552.DLL defines the following VTREG symbols for the on-chip peripheral registers:

CPU-pln Symbol	Description
<b>AIN0</b>	Analog input line AIN0 (floating-point value)
<b>AIN1</b>	Analog input line AIN1 (floating-point value)
<b>AIN2</b>	Analog input line AIN2 (floating-point value)
<b>AIN3</b>	Analog input line AIN3 (floating-point value)
<b>AIN4</b>	Analog input line AIN4 (floating-point value)
<b>AIN5</b>	Analog input line AIN5 (floating-point value)
<b>AIN6</b>	Analog input line AIN6 (floating-point value)
<b>AIN7</b>	Analog input line AIN7 (floating-point value)
<b>PORT0</b>	Digital I/O lines of PORT 0 (8-bit)
<b>PORT1</b>	Digital I/O lines of PORT 1 (8-bit)
<b>PORT2</b>	Digital I/O lines of PORT 2 (8-bit)
<b>PORT3</b>	Digital I/O lines of PORT 3 (8-bit)
<b>PORT4</b>	Digital I/O lines of PORT 4 (8-bit)
<b>PORT5</b>	Digital I/O lines of PORT 5 (8-bit)
<b>SIN</b>	Serial input for SERIAL CHANNEL 0 (9-bit)
<b>SOUT</b>	Serial output for SERIAL CHANNEL 0 (9-bit)



CPU-pin Symbol	Description
<b>STIME</b>	Serial timing enable
<b>VAGND</b>	Analog reference voltage VAGND (floating-point value)
<b>VAREF</b>	Analog reference voltage VAREF (floating-point value)
<b>XTAL</b>	Oscillator frequency



## 80751.DLL

80751.DLL contains the driver for the 80C750, 80C751, 80C752, and other similar chips. This DLL simulates the logical and timing behavior of the following hardware components:

- Timer 0
- Timer 1
- Interrupt System

The following hardware components are not simulated:

- Serial Interface (I<sup>2</sup>C Bus)

80751.DLL defines the following VTREG symbols for the on-chip peripheral registers:

CPU-pin Symbol	Description
<b>PORT0</b>	Digital I/O lines of PORT 0 (8-bit)
<b>PORT1</b>	Digital I/O lines of PORT 1 (8-bit)
<b>PORT2</b>	Digital I/O lines of PORT 2 (8-bit)
<b>XTAL</b>	Oscillator frequency

## 80781.DLL

80781.DLL contains the driver for the 80CL781 and other similar chips. This DLL simulates the logical and timing behavior of the following hardware components:

- Timer 0
- Timer 1
- Timer 2 with Reload/Capture Register
- Serial Interface 0
- Interrupt System

The following hardware components are not simulated:

- Serial Interface 1 (I<sup>2</sup>C Bus)

80781.DLL defines the following VTREG symbols for the on-chip peripheral registers:

CPU-pin Symbol	Description
<b>PORT0</b>	Digital I/O lines of PORT 0 (8-bit)
<b>PORT1</b>	Digital I/O lines of PORT 1 (8-bit)
<b>PORT2</b>	Digital I/O lines of PORT 2 (8-bit)
<b>PORT3</b>	Digital I/O lines of PORT 3 (8-bit)
<b>SIN</b>	Serial input for SERIAL CHANNEL 0 (9-bit)
<b>SOUT</b>	Serial output for SERIAL CHANNEL 0 (9-bit)
<b>STIME</b>	Serial timing enable
<b>XTAL</b>	Oscillator frequency

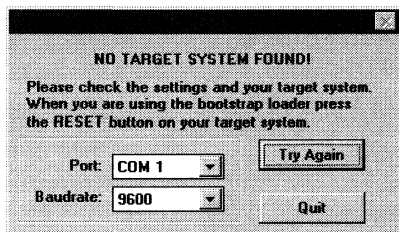
## MON166.DLL

dScope can interface to your target hardware using the MON166.DLL CPU driver. This driver uses the serial port of your PC and a serial port on your target to perform real-time debugging on hardware using the Siemens 166, 167, and other compatible devices.

### NOTE

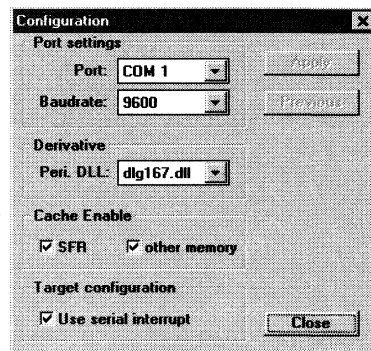
*When you debug using target hardware and a monitor program, dScope's title bar changes from dScope to tScope. tScope is the name of the target debugger built into dScope.*

Serial interface parameters such as COM port and baudrate may be specified in a configuration dialog box after loading the driver. When a bootstrap loader is detected in the target system, dScope automatically downloads the monitor program.



When you load MON166.DLL, dScope immediately attempts to communicate with the target using the serial port settings from the previous session. If communication is not established within a few seconds, dScope prompts you to check the settings and try again.

The MON166 DLL supports the on-chip peripherals of the 166 and 167 using different DLLs for the dialog boxes. The DLG166.DLL and DLG167.DLL files are selected in the Configuration dialog box available from the Peripherals menu.



The following controls are available in the Configuration dialog box.

Control	Description
<b>Port</b>	This control selects the COM port to use for communication.
<b>Baudrate</b>	This control selects the baudrate at which dScope attempts to communicate with the monitor running on your target hardware.
<b>Peripheral DLL</b>	This control selects the DLL that displays dialog boxes for the on-chip peripherals of the CPU in your target system. The MON166 DLL only provides basic communication functions to the target system. Changes to the peripheral DLL require that you exit dScope and reload the MON166 DLL.
<b>Cache Enable</b>	This control improve dScope performance during target debugging by caching certain memory areas in memory. The SFR check box enables caching of memory in the range 0F000h to 0FFFFh. The other memory check box caches the remainder of the 16 Megabyte address range. If you need to see the actual value of port pins, timers, or memory-mapped I/O devices, disable these options.
<b>Use serial interrupt</b>	This check box, when checked, lets you use the Stop button in the Debug window or <b>Ctrl+C</b> or <b>Esc</b> in the Command window to halt program execution without pressing the RESET button on your target hardware. When this feature is enabled, the serial port interrupt is no longer available in your target program. In addition, your target program may not reset the global interrupt enable flag (IE) in the PSW.

## Limitations of MON166

MON166 has several restrictions that are not imposed on the simulator.

### A

- **Memory Map**

Memory mapping for target hardware is done using hardware components like PALs and decoders. It is not possible to display or change this information using the MAP command or the Memory Map dialog box.

- **Performance Analyzer**

The performance analyzer is not supported when debugging with the monitor.

- **Call Stack**

The call stack analyzer is not supported when debugging with the monitor.

- **Code Coverage**

Code coverage is not supported when debugging with the monitor.

- **Breakpoint Options**

All breakpoints are maintained and handled by the monitor program running on your target hardware. When memory access or conditional breakpoints are used, the monitor single steps through the application and checks between

each instruction for a breakpoint. Program execution is at least 1000 times slower than real-time.

## Resources Used by MON166

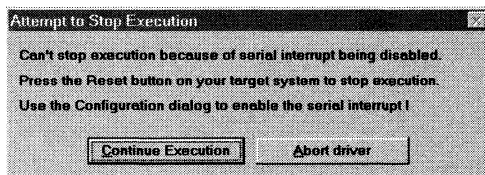
MON166 uses very few resources of your target hardware. There are basically no restrictions on the memory your target program can use. MON166 does, however, use the NMI trap for breakpoints and one of the serial interface traps. Therefore, it is necessary to reserve these areas for the monitor when you link your program. For example:

```
L166 MYPROG.OBJ RESERVE (08h-0Bh, 0ACh-0AFh, 0B8h-0BBh)
```

---

### NOTE

*When debugging using the MON166 monitor, dScope normally requires that you halt program execution on the target before exiting. To exit dScope without halting the target program, press **Esc** in the Command window to display the following dialog box.*



*Click the Abort driver button to abort the MON166 driver and exit dScope.*

---

**A**

## MON251.DLL

dScope can interface to your target hardware using the MON251.DLL CPU driver. This driver uses the serial port of your PC and a serial port on your target to perform real-time debugging on hardware using the Intel 8xC251Sx, 8x930Ax USB, 8x930Hx USB, and Temic 8xC251A1 and 8xC251G1 devices.

---

### NOTE

*When you debug using target hardware and a monitor program, dScope's title bar changes from dScope to tScope. tScope is the name of the target debugger built into dScope.*

---

Serial interface parameters such as COM port and baudrate may be specified in a configuration dialog box after loading the driver. When you load MON251.DLL, dScope immediately attempts to communicate with the target using the serial port settings from the previous session. If communication is not established within a few seconds, dScope prompts you to check the settings and try again.

## Limitations of MON251

Debugging using a monitor has several restrictions that are not imposed on the simulator. Specifically, the following features are not supported.



### A

- **Memory Map**  
Memory mapping for target hardware is done using hardware components like PALs and decoders. It is not possible to display or change this information using the MAP command or the Memory Map dialog box.
- **Performance Analyzer**  
The performance analyzer is not supported when debugging with the monitor.
- **Call Stack**  
The call stack analyzer is not supported when debugging with the monitor.
- **Code Coverage**  
The code coverage analyzer is not supported when debugging with the monitor.
- **Breakpoint Options**  
All breakpoints are maintained and handled by the monitor program running on your target hardware. When memory access or conditional breakpoints are used, the monitor single steps through the application and checks between

each instruction for a breakpoint. Program execution is at least 1000 times slower than real-time.

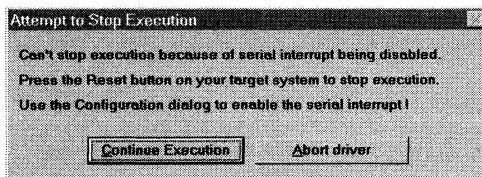
## Resources Used by MON251

MON251 uses very few resources of your target hardware. There are basically no restrictions on the memory your target program can use. MON251 may, however, use an interrupt for serial communication. Therefore, it is necessary to reserve the interrupt vector for the monitor when you link your program.

---

### NOTE

*When debugging using the MON251 monitor, dScope normally requires that you halt program execution on the target before exiting. To exit dScope without halting the target program, press **Esc** in the Command window to display the following dialog box.*



*Click the Abort driver button to abort the MON251 driver and exit dScope.*

---

## MON51.DLL

dScope can interface to your 8051 target hardware using the MON51.DLL CPU driver. This driver uses the serial port of your PC and a serial port on your target to perform real-time debugging on hardware using any 8051 compatible device.

---

### NOTE

*When you debug using target hardware and a monitor program, dScope's title bar changes from dScope to tScope. tScope is the name of the target debugger built into dScope.*

---

Serial interface parameters such as COM port and baudrate may be specified in a configuration dialog box after loading the driver. When you load MON51.DLL, dScope immediately attempts to communicate with the target using the serial port settings from the previous session. If communication is not established within a few seconds, dScope prompts you to check the settings and try again.

## Limitations of MON51

Debugging using a monitor has several restrictions that are not imposed on the simulator. Specifically, the following features are not supported.

- **Memory Map**  
Memory mapping for target hardware is done using hardware components like PALs and decoders. It is not possible to display or change this information using the MAP command or the Memory Map dialog box.
- **Performance Analyzer**  
The performance analyzer is not supported when debugging with the monitor.
- **Call Stack**  
The call stack analyzer is not supported when debugging with the monitor.
- **Code Coverage**  
The code coverage analyzer is not supported when debugging with the monitor.
- **Breakpoint Options**  
All breakpoints are maintained and handled by the monitor program running on your target hardware. When memory access or conditional breakpoints are used, the monitor single steps through the application and checks between each instruction for a breakpoint. Program execution is at least 1000 times slower than real-time.



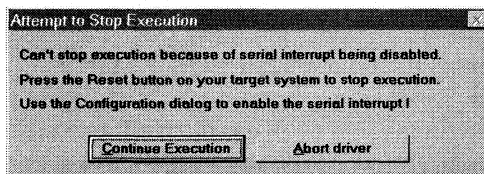
## Resources Used by MON51

MON51 uses very few resources of your target hardware. There are basically no restrictions on the memory your target program can use. MON51 may, however, use an interrupt for serial communication. Therefore, is it necessary to reserve the interrupt vector for the monitor when you link your program.

---

### NOTE

*When debugging using the MON51 monitor, dScope normally requires that you halt program execution on the target before exiting. To exit dScope without halting the target program, press **Esc** in the Command window to display the following dialog box.*



*Click the Abort driver button to abort the MON51 driver and exit dScope.*

---

## RISM251.DLL

dScope can interface to your 251 and 930 USB target hardware using the Intel Reduced Instruction Set Monitor (RISM251.DLL) CPU driver. This driver uses the serial port of your PC and a serial port on your target to perform real-time debugging on hardware using any 251-compatible device.

---

### NOTE

*When you debug using target hardware and a monitor program, dScope's title bar changes from dScope to tScope. tScope is the name of the target debugger built into dScope.*

---

Serial interface parameters such as COM port and baudrate may be specified in a configuration dialog box after loading the driver. When you load RISM251.DLL, dScope immediately attempts to communicate with the target using the serial port settings from the previous session. If communication is not established within a few seconds, dScope prompts you to check the settings and try again.

---

### NOTE

*We recommend that you use MON251 to achieve maximum performance and ease of use when target debugging.*

---

## A

### Limitations of RISM251

Debugging using a monitor has several restrictions that are not imposed on the simulator. Specifically, the following features are not supported by RISM251.

- **Memory Map**  
Memory mapping for target hardware is done using hardware components like PALs and decoders. It is not possible to display or change this information using the MAP command or the Memory Map dialog box.
- **Performance Analyzer**  
The performance analyzer is not supported when debugging with the monitor.
- **Call Stack**  
The call stack analyzer is not supported when debugging with the monitor.
- **Code Coverage**  
The code coverage analyzer is not supported when debugging with the monitor.

- **PStep**

Single stepping over a function call is not supported when debugging with RISM251.

- **Breakpoint Options**

All breakpoints are maintained and handled by the monitor program running on your target hardware. When memory access or conditional breakpoints are used, the monitor single steps through the application and checks between each instruction for a breakpoint. Program execution is at least 1000 times slower than real-time.

## Resources Used by RISM251

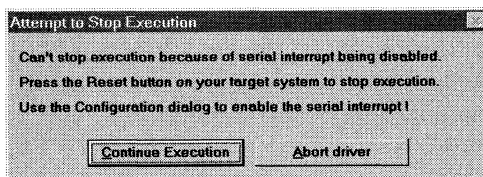
You should check the documentation for RISM251 to determine the memory space and stack used by the monitor. To successfully use RISM251, you must make several changes to your target program:

- Modify the startup code so that it doesn't clear the internal data memory.
- Reserve the memory used by RISM251 so that your target program does not overwrite it.
- Reserve the interrupt vector used by RISM251 for serial communication.

---

### NOTE

*When debugging using the RISM251 monitor, dScope normally requires that you halt program execution on the target before exiting. To exit dScope without halting the target program, press **Esc** in the Command window to display the following dialog box.*



*Click the **Abort driver** button to abort the RISM251 driver and exit dScope.*

---

## RISM51.DLL

dScope can interface to your 8051 target hardware using the Intel Reduced Instruction Set Monitor (RISM51.DLL) CPU driver. This driver uses the serial port of your PC and a serial port on your target to perform real-time debugging on hardware using any 8051 compatible device.

---

### NOTE

*When you debug using target hardware and a monitor program, dScope's title bar changes from dScope to tScope. tScope is the name of the target debugger built into dScope.*

---

Serial interface parameters such as COM port and baudrate may be specified in a configuration dialog box after loading the driver. When you load RISM51.DLL, dScope immediately attempts to communicate with the target using the serial port settings from the previous session. If communication is not established within a few seconds, dScope prompts you to check the settings and try again.

---

### NOTE

*We recommend that you use MON51 to achieve maximum performance and ease of use when target debugging.*

---

## Limitations of RISM51

Debugging using a monitor has several restrictions that are not imposed on the simulator. Specifically, the following features are not supported by RISM51.

- **Memory Map**  
Memory mapping for target hardware is done using hardware components like PALs and decoders. It is not possible to display or change this information using the MAP command or the Memory Map dialog box.
- **Performance Analyzer**  
The performance analyzer is not supported when debugging with the monitor.
- **Call Stack**  
The call stack analyzer is not supported when debugging with the monitor.
- **Code Coverage**  
The code coverage analyzer is not supported when debugging with the monitor.

- **PStep**  
Single stepping over a function call is not supported when debugging with RISM51.
- **Breakpoint Options**  
All breakpoints are maintained and handled by the monitor program running on your target hardware. When memory access or conditional breakpoints are used, the monitor single steps through the application and checks between each instruction for a breakpoint. Program execution is at least 1000 times slower than real-time.

## Resources Used by RISM51

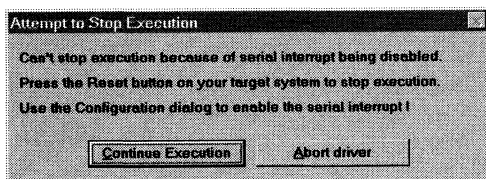
You should check the documentation for RISM51 to determine the memory space and stack used by the monitor. To successfully use RISM51, you must make several changes to your target program:

- Modify the startup code so that it doesn't clear the internal data memory.
- Reserve the memory used by RISM51 so that your target program does not overwrite it.
- Reserve the interrupt vector used by RISM51 for serial communication.

---

### NOTE

*When debugging using the RISM51 monitor, dScope normally requires that you halt program execution on the target before exiting. To exit dScope without halting the target program, press **Esc** in the Command window to display the following dialog box.*



*Click the **Abort driver** button to abort the RISM51 driver and exit dScope.*

---



## Appendix B. Improving Debugger Performance

dScope is a powerful high-level language debugger that supports source-level, symbolic debugging. It provides you with a rich set of features to aid in complex debugging tasks.

The following items may significantly impact dScope's performance. Reducing or eliminating any of them will improve dScope simulation speed.

- **Trace Recording:** When trace recording is enabled, dScope records all registers for each instruction that is executed and stores them in a circular buffer which you can view. If you do not need to use the trace buffer, disable this feature.
- **Updating the Watch Window:** As your target program runs, dScope automatically updates the contents of the Watch window. If you do not need the constant update of information, disable this feature. Note that this is not an issue if you have no watchpoints or if the Watch window is not open.
- **Updating the Memory Window:** As your target program runs, dScope automatically updates the contents of the Memory window. If you do not need the constant update of information, disable this feature.
- **Conditional Breakpoints:** Conditional breakpoints are evaluated after each assembly instruction executes. Reducing the number of conditional breakpoints or disabling conditional breakpoints that are unnecessary has a dramatic effect on the simulation speed.





## Appendix C. Debugging Bank Switching Programs

dScope lets you load and debug your MCS<sup>®</sup> 51 and MCS<sup>®</sup> 251 bank switching applications. You use the **LOAD** command to load a bank switching application.

Bank switching programs may use up to 32 code banks of 64 Kbytes each. Banks are loaded into different segments as shown in the following table. The common code loads in all active banks and is always available.

Segment	Bank
0x80	0
0x81	1
0x82	2
0x83	3
0x84	4
0x85	5
0x86	6
0x87	7
0x88	8
0x89	9
0x8A	10
0x8B	11
0x8C	12
0x8D	13
0x8E	14
0x8F	15

Segment	Bank
0x90	16
0x91	17
0x92	18
0x93	19
0x94	20
0x95	21
0x96	22
0x97	23
0x98	24
0x99	25
0x9A	26
0x9B	27
0x9C	28
0x9D	29
0x9E	30
0x9F	31

### NOTE

*dScope automatically allocates segments required to store the code for each bank. You do not need to specify the memory map before loading the bank switching program.*

The following list describes some of the restrictions and features of debugging a code banking application using dScope.

- dScope extracts bank switch function symbols from the object file to determine the bank switch code address entries. The bank switch function symbols must be named **?B\_SWITCH $nn$**  where  **$nn$**  is the bank number. This

is the naming convention used by C51, A51, and BL51 when creating bank switching applications. When dScope executes an address which denotes a bank switch, it switches to the segment containing the target code bank.

- dScope requires that banked applications are created with V3.5 of the BL51 Code Banking Linker/Locator. dScope does not handle bank switching properly for older versions of BL51.
- If an address breakpoint is set in the common area, that breakpoint is set in the common area of every bank. An address breakpoint that is set in the banked area is not duplicated in the other code banks.
- Addresses in different banks can be qualified using the double backslash notation. For example, `\3\modx1\funcx1` denotes **funcx1** in module **modx1** in bank **3**. This is a fully qualified address which may be used whenever an address parameter for a command is required.
- The Debug window changes the output of the address in mixed or assembly mode to reflect the code bank number.

# Index

-- post-decrement operator .....	106
-- pre-decrement operator .....	106
- subtraction operator .....	106
- unary minus operator .....	106
! logical negation operator .....	106
!= inequality operator .....	107
\$ system variable .....	89
% modulo operator .....	106
%= assignment operator .....	107
& address operator .....	106
& bitwise AND operator .....	107
&& logical AND operator .....	107
&= assignment operator .....	107
* multiplication operator .....	106
* pointer operator .....	106
*= assignment operator .....	107
+ addition operator .....	106
+ unary plus operator .....	106
++ post-increment operator .....	106
++ pre-increment operator .....	106
+= assignment operator .....	107
, comma operator .....	108
. bit address operator .....	106
. structure member operator .....	106
/ division operator .....	106
/= assignment operator .....	107
< less than operator .....	107
<< left shift operator .....	107
<=< assignment operator .....	107
<= less or equal operator .....	107
= assignment operator .....	107
== equality operator .....	107
> greater than operator .....	107
-> structure pointer operator .....	106
>= greater or equal operator .....	107
>> right shift operator .....	107
>>= assignment operator .....	107
? : conditional operator .....	108
[] array index operator .....	106
^ bitwise XOR operator .....	107
^= assignment operator .....	107
_break_ system variable .....	89,136
_framesize_ system variable .....	89
_iip_ system variable .....	89
_mode_ system variable .....	89
bitwise OR operator .....	107
= assignment operator .....	107
logical OR operator .....	107
~ one's complement operator .....	106
80166.DLL .....	299
80167.DLL .....	301
80251A1.DLL .....	303
80251G1.DLL .....	304
80251S.DLL .....	305
80320.DLL .....	306
80410.DLL .....	307
8051.DLL .....	308
80515.DLL .....	310
80515A.DLL .....	311
80517.DLL .....	312
80517A.DLL .....	314
8051FX.DLL .....	309
8052.DLL .....	316
80552.DLL .....	317
80751.DLL .....	319
80781.DLL .....	320
<b>A</b>	
A166 assembler programs	
Preparing for dScope .....	9
A251 assembler programs	
Preparing for dScope .....	7
A51 assembler programs	
Preparing for dScope .....	4
Access breakpoints .....	138,206
Addition operator, + .....	106
Additional commands indicator .....	39
Additional items, document	
conventions .....	v
Address expressions .....	108
Address operator, & .....	106
ANSI C .....	283
Arithmetic operators .....	106
Array index operator, [] .....	106
Arrays in watchpoints .....	160
ASM .....	197

Assembling programs for  
 dScope ..... 3  
 ASSIGN ..... 198  
 Assignment operator  
   %= ..... 107  
   &= ..... 107  
   \*= ..... 107  
   += ..... 107  
   /= ..... 107  
   <<= ..... 107  
   = 107  
   >>= ..... 107  
   ^= ..... 107  
   |= ..... 107  
 Assignment operators ..... 107

## B

Bank switching programs ..... 335  
 Banked OMF-51 object file ..... 226  
 BFUNC ..... 211, 212  
 Binary constants ..... 86  
 BIT ..... 104, 209, 219  
   Function description ..... 267  
   Predefined function ..... 266  
 Bit address operator, ..... 106  
 Bit addresses ..... 101  
 BIT memory prefix ..... 102  
 Bitwise AND operator, & ..... 107  
 Bitwise operators ..... 107  
 Bitwise OR operator, | ..... 107  
 Bitwise XOR operator, ^ ..... 107  
 Bold capital text, use of ..... v  
 Braces, use of ..... v  
 break ..... 261  
 BREAKDISABLE ..... 200  
 BREAKENABLE ..... 201  
 BREAKKILL ..... 202  
 BREAKLIST ..... 203  
 Breakpoint Commands ..... 195  
   BREAKDISABLE ..... 200  
   BREAKENABLE ..... 201  
   BREAKKILL ..... 202  
   BREAKLIST ..... 203  
   BREAKSET ..... 204  
 Breakpoints  
   Access ..... 138, 206  
   Conditional ..... 137, 205  
   Disabling ..... 150

Enabling ..... 150  
 Execution ..... 137, 205  
 Memory access ..... 206  
 Notes ..... 139  
 Overview ..... 135  
 Removing ..... 33, 149  
 Setting ..... 33, 140  
 Types ..... 137  
   Access ..... 138  
   Conditional ..... 137  
   Execution ..... 137  
   Using ..... 135  
   Viewing ..... 147  
 Breakpoints dialog box ..... 78  
 BREAKSET ..... 204  
 BYTE ..... 104, 209, 219  
   Function description ..... 267  
   Predefined function ..... 266

## C

C166 compiler programs  
   Preparing for dScope ..... 10  
 C251 compiler programs  
   Preparing for dScope ..... 8  
 C51 compiler programs  
   Preparing for dScope ..... 5  
 Call stack window ..... 63  
 case ..... 261  
 Chaining commands ..... 41  
 Changing crystal frequency ..... 91  
 Changing oscillator frequency ..... 91  
 Changing window colors ..... 70  
 Changing window fonts ..... 69  
 CHAR ..... 104, 209, 219  
   Function description ..... 267  
   Predefined function ..... 266  
 Character constant escape  
   sequence ..... 87  
 Character constants ..... 87  
 Choices, document conventions ..... v  
 Classes of symbols ..... 96  
 CLEAR ..... 234  
 Closing dScope ..... 16  
 Code coverage window ..... 65  
 CODE memory prefix ..... 102  
 Color setup dialog box ..... 69  
 Colors of windows ..... 70  
 Comma operator, ..... 108

Command files .....	43
Command line editing .....	40
Command window .....	38
Command buffers .....	39
Command chaining .....	41
Command files .....	43
Command line editing .....	40
Comment lines .....	41
Recalling command lines .....	39
Reviewing command output .....	40
Status bar .....	38
Syntax generator .....	42
Command-line include files .....	14
Commands .....	193,196
Comments .....	41
Compiling programs for dScope .....	3
Conditional breakpoints .....	137,205
Conditional operator, ?: .....	108
CONST memory prefix .....	102
Constant Expressions .....	86
Constants .....	86
Binary .....	86
Character .....	87
Decimal .....	86
Floating-point .....	87
HEX .....	86
Octal .....	86
String .....	88
continue .....	261
Courier typeface, use of .....	v
CPU driver DLL .....	226,228
CPU driver files .....	295
CPU driver symbols .....	90
CPU drivers .....	
166 .....	298
251 .....	297
8051 .....	297
List .....	297
Purpose .....	295
CPU pin register .....	<i>See</i> VTREG
Creating toolbox buttons .....	67
Crystal frequency .....	91
Ctrl+C .....	208
cycles system variable .....	90

## D

DATA memory prefix .....	102
Data types .....	104,209,219

DEBUG .....	3,238
A166 assembler directive .....	230
A251 assembler directive .....	230
A51 assembler directive .....	230
C166 assembler directive .....	230
C251 assembler directive .....	230
C51 assembler directive .....	230
PL/M-51 assembler directive .....	230
Debug symbol information .....	95
Debug window .....	27
Assembling in-line code .....	34
Breakpoints .....	33
Dialog bar .....	29
Displaying source modules .....	33
Logging to a file .....	36
Menu .....	30
Operations .....	32
Removing breakpoints .....	33
Scrolling .....	32
Searching for text .....	34
Setting breakpoints .....	33
Status bar .....	28
Toolbar .....	29
Trace recording .....	34
Debug window color items .....	71
Debugger performance .....	333
Decimal constants .....	86
DEFINE .....	209
DEFINE BUTTON .....	210
Defining watchpoints .....	156
DEFSYM .....	211,212
DIR .....	211
Disabling breakpoints .....	150
DISPLAY .....	216
Displayed text, document .....	
conventions .....	v
Division operator, / .....	106
DLG166.DLL .....	321
DLG167.DLL .....	321
DLL driver files .....	295
DLL for CPU driver .....	226,228
DLL for target driver .....	226,229
do .....	261
Document conventions .....	v
DOS .....	218
double .....	
Function description .....	268
Predefined function .....	266

Double brackets, use of ..... v  
 Dragging symbols ..... 60  
 dScope commands ..... 193  
 dScope Functions ..... 261,283  
 DWORD ..... 104,209,219  
     Function description ..... 268  
     Predefined function ..... 266

## E

EBIT memory prefix ..... 102  
 EDATA memory prefix ..... 102  
 Editing the command line ..... 40  
 Ellipses, use of ..... v  
 Ellipses, vertical, use of ..... v  
 else ..... 261  
 Enabling breakpoints ..... 150  
 Ending a debugging session ..... 16  
 ENTER ..... 219  
 Equality operator, == ..... 107  
 Error Messages ..... 285  
 Esc ..... 220  
 Escape sequence ..... 87  
 EVALUATE ..... 221  
 Examples of expressions ..... 111  
 EXEC ..... 234  
     Function description ..... 269  
     Predefined function ..... 266  
 Executing code ..... 115  
 Execution breakpoints ..... 137,205  
 EXIT ..... 222  
 Exiting dScope ..... 16  
 Exiting the dScope debugger ..... 16  
 Expression components ..... 85  
     Bit addresses ..... 85,101  
     Constants ..... 85,86  
     CPU driver symbols ..... 90  
     Line numbers ..... 85,100  
     Memory spaces ..... 85,102  
     Operators ..... 85,105  
     Program variables ..... 85,94  
     SFRs ..... 90  
     Special function registers ..... 90  
     Symbols ..... 85,94  
     System variables ..... 86,88  
     Type specifications ..... 86,104  
     Variables ..... 85,94  
     VTREGs ..... 91  
 Expression examples ..... 111

Expressions ..... 85

## F

F1 ..... 248  
 F11 ..... 248  
 F12 ..... 248  
 F2 ..... 248  
 F3 ..... 248  
 F4 ..... 248  
 F5 ..... 248  
 F6 ..... 248  
 F7 ..... 248  
 F8 ..... 248  
 F9 ..... 248  
 File menu commands ..... 20  
 Filename, document conventions ..... v  
 FLOAT ..... 104,209,219  
     Function description ..... 269  
     Predefined function ..... 266  
 Floating-point constants ..... 87  
 Fonts used in windows ..... 69  
 Fully qualified symbols ..... 97  
 FUNC ..... 211,212  
 Function Classes ..... 264  
     Predefined functions ..... 264  
     Signal functions ..... 264  
     User functions ..... 264  
 Functions  
     Classes ..... 264  
     Creating ..... 262  
     In dScope ..... 261  
     Invoking ..... 264  
     Loading ..... 263  
     Predefined ..... 266  
     Signal ..... 279  
     User ..... 276

## G

General commands ..... 195  
     ASSIGN ..... 198  
     DEFINE BUTTON ..... 210  
     DIR ..... 211  
     DOS ..... 218  
     EXIT ..... 222  
     INCLUDE ..... 224  
     KILL ..... 225  
     LOAD ..... 226

LOG .....	233
MODE .....	237
RESET .....	244
SAVE .....	245
SCOPE .....	246
SET .....	248
SETMODULE .....	251
SIGNAL .....	255
SLOG .....	256
getfloat	
Function description .....	270
Predefined function .....	266
getint	
Function description .....	270
Predefined function .....	266
getlong	
Function description .....	270
Predefined function .....	266
GO .....	223
goto .....	261
Greater or equal operator, >= .....	107
Greater than operator, > .....	107

## H

Hardware requirements .....	2
HCONST memory prefix .....	102
Help for toolbar buttons .....	26
Help menu commands .....	24
HEX constants .....	86
HEX files .....	227,232
HEX386 files .....	227,232

## I

I/O ports .....	91
IDATA memory prefix .....	102
if261	
Improving debugger	
performance .....	333
INCLUDE .....	224
Include files .....	14
Inequality operator, != .....	107
Inline assembler dialog dialog	
box .....	76
Insert mode indicator .....	39
INT .....	104,209,219
Function description .....	271
Predefined function .....	266

Intel HEX files .....	227,232
Intel HEX386 files .....	227,232
Intel OMF-251 object file .....	227
Intel OMF-51 object file .....	226
Italicized text, use of .....	v
itrace system variable .....	90

## K

Key names, document	
conventions .....	v
KILL .....	225

## L

Left shift operator, << .....	107
Less or equal operator, <= .....	107
Less than operator, < .....	107
LINE .....	211,212,238
Line numbers .....	60,100
Linking programs for dScope .....	3
Literal .....	97
Literal symbols .....	97
LOAD .....	226
Loading a banked OMF-51	
object file .....	226
Loading a CPU driver DLL .....	14,226,228
Loading a program .....	15
Loading a target driver DLL .....	226,229
Loading an Intel HEX file .....	227,232
Loading an Intel HEX386 file .....	227,232
Loading an object file .....	230
Loading an OMF-166 object file .....	227,230
Loading an OMF-251 object file .....	227,230
Loading an OMF-51 object file .....	226,230
Loading the dScope debugger .....	11
Local symbols .....	59
LOG .....	233
Logical AND operator, && .....	107
Logical negation operator, ! .....	106
Logical operators .....	107
Logical OR operator,    .....	107
LONG .....	104,209,219
Function description .....	271
Predefined function .....	266

## M

Main window .....	19
-------------------	----

- 
- |                                  |         |                                     |     |
|----------------------------------|---------|-------------------------------------|-----|
| MAP.....                         | 234     | Limitations.....                    | 322 |
| Mask characters.....             | 57      | Resources.....                      | 323 |
| Memory access breakpoints.....   | 206     | MON251.DLL.....                     | 324 |
| Memory commands.....             | 194     | Limitations.....                    | 324 |
| ASM.....                         | 197     | Resources.....                      | 325 |
| DEFINE.....                      | 209     | MON51.DLL.....                      | 326 |
| DISPLAY.....                     | 216     | Limitations.....                    | 326 |
| ENTER.....                       | 219     | Resources.....                      | 327 |
| EVALUATE.....                    | 221     | MOVX @Ri instruction.....           | 236 |
| MAP.....                         | 234     | Multiplication operator, *.....     | 106 |
| MAP CLEAR.....                   | 234     |                                     |     |
| MAP EXEC.....                    | 234     | <b>N</b>                            |     |
| MAP READ.....                    | 234     | Naming conventions for symbols..... | 95  |
| MAP VNM.....                     | 234     | Non-qualified symbols.....          | 98  |
| MAP WRITE.....                   | 234     |                                     |     |
| OBJECT.....                      | 238     | <b>O</b>                            |     |
| UNASSEMBLE.....                  | 258     | OBJECT.....                         | 238 |
| Memory location watchpoints..... | 163     | Object file                         |     |
| Memory map access                |         | OMF-166.....                        | 227 |
| Execute.....                     | 74      | OMF-251.....                        | 227 |
| Read.....                        | 74      | OMF-51.....                         | 226 |
| von Neumann.....                 | 75      | OBJECTEXTEND                        |     |
| Write.....                       | 74      | C51 assembler directive.....        | 230 |
| Memory map dialog box.....       | 72      | Octal constants.....                | 86  |
| Controls.....                    | 74      | OMF-166 object file.....            | 227 |
| Overview.....                    | 72      | OMF-251 object file.....            | 227 |
| Memory space                     |         | OMF-51 object file.....             | 226 |
| BIT.....                         | 102     | Omitted text, document              |     |
| CODE.....                        | 102     | conventions.....                    | v   |
| CONST.....                       | 102     | One's complement operator, ~.....   | 106 |
| DATA.....                        | 102     | Operators.....                      | 105 |
| EBIT.....                        | 102     | Arithmetic.....                     | 106 |
| EDATA.....                       | 102     | Assignment.....                     | 107 |
| HCONST.....                      | 102     | Bitwise.....                        | 107 |
| IDATA.....                       | 102     | Logical.....                        | 107 |
| VTREG.....                       | 102     | Other.....                          | 108 |
| XDATA.....                       | 102     | Postfix.....                        | 106 |
| Memory spaces.....               | 102     | Primary.....                        | 105 |
| Memory window.....               | 54      | Relational.....                     | 107 |
| memset                           |         | Unary.....                          | 106 |
| Function description.....        | 271     | Optional items, document            |     |
| Predefined function.....         | 266     | conventions.....                    | v   |
| Menu bar.....                    | 19      | Oscillator frequency.....           | 91  |
| MODE.....                        | 237     | OSTEP.....                          | 239 |
| MODULE.....                      | 211,212 |                                     |     |
| Module names.....                | 94      |                                     |     |
| Modulo operator, %.....          | 106     |                                     |     |
| MON166.DLL.....                  | 321     |                                     |     |



**P**

PDATA mapping .....	236
Performance analyzer .....	240
Performance analyzer setup	
dialog box .....	83
Performance analyzer window .....	49
Adding an address range .....	50
Commands menu .....	50
Defining an address range .....	50
Removing an address range .....	50
Valid address ranges .....	52
Performance of dScope .....	333
Peripheral menu commands .....	24
peripheral names .....	188
Peripheral registers .....	296
PL/M-51 .....	251
PL/M-51 compiler programs	
Preparing for dScope .....	6
Pointer operator, * .....	106
Pointers in watchpoints .....	161
Ports .....	91
Post-decrement operator, -- .....	106
Postfix operators .....	106
Post-increment operator, ++ .....	106
Pre-decrement operator, -- .....	106
Predefined functions .....	266
bit .....	266,267
byte .....	266,267
char .....	266,267
double .....	266,268
dword .....	266,268
exec .....	266,269
float .....	266,269
getfloat .....	266,270
getint .....	266,270
getlong .....	266,270
int .....	266,271
long .....	266,271
memset .....	266,271
printf .....	266,272
ptr .....	266,272
rand .....	266,273
real .....	266,273
twatch .....	266,274
uchar .....	266,274
uint .....	266,275
ulong .....	266,275
word .....	266,275

Pre-increment operator, ++ .....	106
Preparing programs for dScope .....	3
Primary operators .....	105
Printed text, document	
conventions .....	v
printf	
Function description .....	272
Predefined function .....	266
Program Commands .....	194
Ctrl+C .....	208
Esc .....	220
GO .....	223
OSTEP .....	239
Performance analyzer .....	240
PSTEP .....	243
TSTEP .....	257
Program counter system variable .....	89
PSTEP .....	243
PTR .....	104,219
Function description .....	272
Predefined function .....	266
PUBLIC .....	211,212
Public symbols .....	59
Purpose of CPU drivers .....	295

**Q**

Qualified symbols .....	97
-------------------------	----

**R**

radix system variable .....	90
rand	
Function description .....	273
Predefined function .....	266
READ .....	234
REAL .....	104,209,219
Function description .....	273
Predefined function .....	266
Register window .....	45
251 .....	46
8051 .....	45
Changing register values .....	46
MCS® 251 .....	46
MCS® 51 .....	45
Relational operators .....	107
Removing breakpoints .....	149
Removing toolbox buttons .....	68
Removing watchpoints .....	158

Requirements .....	2
RESET .....	244
Resetting a program .....	117
Restarting a program .....	117
Right shift operator, >> .....	107
RISM251.DLL .....	328
Limitations .....	328
Resources .....	329
RISM51.DLL .....	330
Limitations .....	330
Resources .....	331
RS-232 ports .....	93
Running code .....	115

## S

Sans serif typeface, use of .....	v
SAVE .....	245
SCOPE .....	246
Scroll lock .....	39
Scroll lock indicator .....	39
Serial ports .....	93
Serial window .....	47
SET .....	230,244,248
SETMODULE .....	251
Setting breakpoints .....	140
Setting watchpoints .....	156
Setup menu commands .....	22
SFRs .....	90
SIGNAL .....	211,212,255
Signal functions .....	279
Simulating I/O ports .....	91
Simulating serial ports .....	93
Single-step .....	243,257
Single-stepping .....	118
Single-stepping into functions .....	119
Single-stepping out of a function .....	119
Single-stepping over functions .....	118
size of operator .....	106
SLOG .....	256
Software requirements .....	2
Source code considerations .....	3
Special function registers .....	90
SRC .....	248
Starting a program .....	115
Starting dScope .....	11
Starting the dScope debugger .....	11
Stepping into a function .....	119
Stepping out of a function .....	119

Stepping over a function .....	118
Stopping a program .....	116
String constants .....	88
Structure member operator, .....	106
Structure pointer operator, -> .....	106
Structures in watchpoints .....	160
Subtraction operator, - .....	106
switch .....	261
Symbol browser window .....	56
Display mode .....	56
Filter options .....	57
Mask options .....	57
Options .....	57
Symbol expressions .....	94
Symbol types .....	104,209,219

## Symbols

Classification .....	96
CPU driver .....	90
Fully qualified .....	97
Information .....	95
Literals .....	97
Module names .....	94
Naming conventions .....	95
Non-qualified .....	98
Qualified names .....	97
SFRs .....	90
Special function registers .....	90
System variables .....	88
VTREGs .....	91
Syntax generator .....	39,42
System requirements .....	2
System variables .....	88
\$ 89 .....	
_break_ .....	89,136
_framesize_ .....	89
_iip_ .....	89
_mode_ .....	89
cycles .....	90
itrace .....	90
Program counter .....	89
radix .....	90

## T

Target driver DLL .....	226,229
Toolbar .....	25
Toolbar buttons .....	
Call stack window .....	26
Code coverage window .....	26

Command window.....	26
CPU driver.....	26
Debug window .....	26
Help .....	26
Memory window .....	26
Open object file .....	26
Performance analyzer	
window .....	26
Register window.....	26
Reset .....	26
Serial window.....	26
Symbol browser window.....	26
Toolbox window.....	26
Watch window.....	26
Toolbox window .....	67
Adding buttons .....	67
Removing buttons.....	68
tScope.....	229
TSTEP .....	257
Tutorial .....	165
twatch	
Function description .....	274
Predefined function .....	266
Type specifications.....	104,209,219
BIT.....	104
BYTE.....	104
CHAR.....	104
DWORD.....	104
FLOAT .....	104
INT .....	104
LONG.....	104
PTR.....	104
REAL.....	104
UCHAR .....	104
UINT .....	104
ULONG .....	104
WORD.....	104
Types of breakpoints .....	137
Types of symbols .....	104,209,219

## U

UCHAR.....	104,209,219
Function description .....	274
Predefined function .....	266
UFUNC .....	211,212
UINT .....	104,209,219
Function description .....	275
Predefined function .....	266

ULONG .....	104,209,219
Function description.....	275
Predefined function .....	266
Unary minus operator, - .....	106
Unary operators .....	106
Unary plus operator, + .....	106
UNASSEMBLE.....	258
Unions in watchpoints .....	160
User functions .....	276
User interface.....	17
Using a CPU driver.....	295
Using breakpoints .....	135
Using include files .....	14
Using the MOVX @Ri	
instruction .....	236
Using watchpoints.....	153

## V

Variable expressions .....	94
Variables, document conventions .....	v
Vertical bar, use of.....	v
View menu commands.....	21
Viewing breakpoints .....	147
Viewing watchpoints .....	155
VNM .....	234
von Neumann memory.....	234
VTREG .....	211,212,296
VTREG memory prefix .....	102
VTREGs.....	91

## W

Watch window .....	44
WATCHKILL.....	259
Watchpoint commands .....	194
WATCHKILL.....	259
WATCHSET .....	260
Watchpoints .....	44
Arrays.....	160
Defining .....	156
Global variables .....	159
Local variables .....	159
Memory locations .....	163
Notes .....	154
Pointers.....	161
Program variables .....	159
Removing .....	158
Setting .....	156

---

Structures .....	160	Function description .....	275
Unions .....	160	Predefined function .....	266
Using .....	153	WRITE .....	234
Viewing .....	155		
Watchpoints dialog box .....	81	<b>X</b>	
WATCHSET .....	260		
while .....	261		
WORD .....	104,209,219	XDATA memory prefix .....	102