

```
{  
    FILE* sfile;  
    int count = 0;  
  
    sfile = fopen("file1",  
    "r");  
  
    if( sfile == NULL)  
    {  
        return -1;  
    }  
  
    while (1)  
    {  
        char c;  
        c = fgetc(  
        if(c == EOF)  
        {  
            break;  
        }  
        else  
        {  
            count++;  
        }  
    }  
    return count;  
}
```

68K/ColdFire v10.0

CrossView Pro Debugger User's Manual

A publication of
Altium BV
Documentation Department
Copyright © 2003 Altium BV

All rights reserved. Reproduction in whole or part is prohibited
without the written consent of the copyright owner.

TASKING is a brand name of Altium Limited.

The following trademarks are acknowledged:

FLEXlm is a registered trademark of Globetrotter Software, Inc.
Motorola is a trademark of Motorola, Inc.
MS-DOS and Windows are registered trademarks of Microsoft Corporation.
IBM is a trademark of International Business Machines Corp.
SUN is a trademark of Sun Microsystems, Inc.
UNIX is a registered trademark of X/Open Company, Ltd.

All other trademarks are property of their respective owners.

Data subject to alteration without notice.

<http://www.tasking.com>
<http://www.altium.com>

The information in this document has been carefully reviewed and is believed to be accurate and reliable. However, Altium assumes no liabilities for inaccuracies in this document. Furthermore, the delivery of this information does not convey to the recipient any license to use or copy the software or documentation, except as provided in an executed license agreement covering the software and documentation.

Altium reserves the right to change specifications embodied in this document without prior notice.

CONTENTS

TABLE OF CONTENTS



CONTENTS

OVERVIEW **1-1**

1.1	Introduction	1-3
1.2	CrossView Pro's Features	1-3
1.3	Source Level Debugging	1-8
1.4	How CrossView Pro Works	1-10
1.5	Program Development	1-12
1.6	Getting Started	1-17
1.6.1	Before Starting	1-17
1.6.2	Setting Up the Execution Environment	1-17
1.6.3	Starting CrossView Pro	1-18
1.6.3.1	CrossView Pro Target Settings	1-19
1.6.3.2	Configuring CrossView Pro	1-21
1.6.3.3	Loading Symbolic Debug Information	1-22
1.6.4	Executing an Application	1-24
1.6.5	Debugging an Application	1-27
1.6.6	CrossView Pro Output	1-29
1.6.7	Exiting CrossView Pro	1-30
1.6.8	What You May Have Done Wrong	1-31
1.6.9	Building Your Executable	1-32
1.6.9.1	Using EDE	1-32

SOFTWARE INSTALLATION **2-1**

2.1	Introduction	2-3
2.2	Note about Filenames	2-3
2.3	Configuring the X Windows Motif Environment ...	2-3
2.4	Using X Resources	2-4

COMMAND LANGUAGE **3-1**

3.1	Introduction	3-3
3.2	CrossView Pro Expressions	3-3
3.3	Constants	3-4
3.4	Variables	3-7
3.5	Formatting Expressions	3-13

3.6	Operators	3-17
3.7	Special Expressions	3-18
3.8	Conditional Evaluation	3-19
3.9	Functions	3-20
3.10	Case Sensitivity	3-21

USING CROSSVIEW PRO **4-1**

4.1	Introduction	4-3
4.2	Using the CrossView Pro Interface	4-3
4.3	Starting CrossView Pro	4-4
4.4	Startup Options	4-5
4.4.1	What You May Have Done Wrong	4-9
4.5	The CrossView Pro Desktop	4-11
4.5.1	Menus	4-13
4.5.1.1	Local Popup Menus	4-14
4.5.2	Window Operation	4-14
4.5.3	Dialog Boxes	4-16
4.5.4	Customizing CrossView Pro	4-17
4.5.5	CrossView Pro Messages	4-19
4.6	CrossView Pro Windows	4-20
4.6.1	Command Window	4-21
4.6.2	Source Window	4-23
4.6.3	Register Window	4-26
4.6.4	Memory Window	4-27
4.6.5	Data Window	4-29
4.6.6	Stack Window	4-32
4.6.7	Trace Window	4-33
4.6.8	Terminal Window	4-34
4.6.9	Data Analysis Window	4-36
4.6.10	Pop-Up Windows	4-37
4.7	Control Operations for CrossView Pro	4-38
4.7.1	Echoing Commands	4-38
4.7.2	Mouse/Menu/Command Equivalents	4-38
4.8	Using the On-line Help	4-39

4.8.1	Accessing On-line Help	4-39
4.8.2	Using MS-Windows Help	4-39

CONTROLLING PROGRAM EXECUTION **5-1**

5.1	Source Positioning	5-3
5.1.1	Changing the Viewing Position	5-4
5.1.2	Changing the Execution Position	5-5
5.1.3	Synchronizing the Execution and Viewing Positions	5-7
5.2	Controlling Program Execution	5-8
5.2.1	Starting the Program	5-8
5.2.2	Halting and Continuing Execution	5-9
5.2.3	Single-Step Execution	5-9
5.2.4	Stepping through at the Machine Level	5-12
5.3	Notes About Program Execution	5-14
5.4	Calling a Function	5-14
5.5	Searching through the Source Window	5-15
5.5.1	Searching for a Function	5-15
5.5.2	Searching for a String	5-16
5.5.3	Jumping to a Source Line	5-17

ACCESSING CODE AND DATA **6-1**

6.1	Introduction	6-3
6.2	Accessing Variables	6-3
6.2.1	Viewing Variables, Structures and Arrays	6-3
6.2.2	Changing Variables	6-7
6.2.3	The I Command	6-8
6.3	Expressions	6-10
6.3.1	Evaluating Expressions	6-10
6.3.2	Monitoring Expressions	6-11
6.3.3	Formatting Data	6-13
6.3.4	Displaying Memory	6-14
6.3.5	Displaying Memory Addresses	6-16
6.4	Displaying Disassembled Instructions	6-17



- 6.4.1 Intermixed Source and Disassembly 6-18
- 6.5 The Stack 6-19
- 6.5.1 How the Stack is Organized 6-19
- 6.5.2 The Stack Window 6-20
- 6.5.3 Listing Locals and Parameters of a Function 6-22
- 6.5.4 Low-level Viewing the Stack 6-22
- 6.6 Trace Window 6-23
- 6.6.1 Trace Window Setup 6-23
- 6.7 Register Window 6-25
- 6.7.1 Register Window Setup 6-25
- 6.7.2 Editing Registers 6-26

BREAKPOINTS AND ASSERTIONS **7-1**

- 7.1 Introduction to Breakpoints 7-3
 - 7.1.1 Code Breakpoints 7-3
 - 7.1.2 Data Breakpoints 7-7
 - 7.1.3 Listing Breakpoints 7-8
- 7.2 Setting Breakpoints 7-8
 - 7.2.1 Data Breakpoints over a Range of Addresses 7-11
 - 7.2.2 Temporary Breakpoints 7-12
 - 7.2.3 Breakpoint Names 7-13
 - 7.2.4 Setting the Count 7-14
 - 7.2.5 Sequence Breakpoints 7-15
- 7.3 Deleting Breakpoints 7-16
- 7.4 Enabling/Disabling Breakpoints 7-17
- 7.5 Breakpoint Commands 7-18
 - 7.5.1 Attaching Conditionals to a Breakpoint 7-21
 - 7.5.2 Attaching Macros to a Breakpoint 7-21
 - 7.5.3 Attaching Strings to a Breakpoint 7-22
- 7.6 Suppressing Breakpoint Messages 7-22
- 7.7 Up-level Breakpoints 7-22
- 7.8 Patches 7-25
 - 7.8.1 Patching Code out of a Program 7-25
 - 7.8.2 Patching Code into a Program 7-26

7.8.3	Replacing Code in a Program	7-26
7.9	Diagnostic Output and Statistical Information	7-27
7.10	Assertions	7-28
7.10.1	Assertion Mode	7-28
7.10.2	Defining an Assertion	7-29
7.10.3	Editing an Assertion	7-31
7.10.4	Activating and Suspending Assertions	7-31
7.10.5	Deleting Assertions	7-32
7.10.6	Using Assertions	7-33
7.10.7	Gathering Statistics with Assertions	7-35

DEFINING AND USING MACROS **8-1**

8.1	CrossView Pro Macros	8-3
8.2	Defining Macros	8-3
8.2.1	Listing Macros	8-5
8.2.2	Redefining a Macro	8-5
8.2.3	Saving Macro Definitions to a File	8-6
8.2.4	Loading Macro Definitions from a File	8-7
8.2.5	Deleting Macros	8-8
8.3	Macro Parameters	8-9
8.4	Redefining Existing CrossView Pro Commands	8-10
8.5	Using the Toolbox	8-11
8.5.1	Opening the Toolbox	8-11
8.5.2	Connecting Macros to the Toolbox	8-11
8.5.3	Removing a Macro Connection	8-12

COMMAND RECORDING & PLAYBACK **9-1**

9.1	Recording Commands	9-3
9.1.1	Entering Comments	9-4
9.1.2	Suspend Recording	9-5
9.1.3	Resume Recording	9-5
9.1.4	Check Recording Status	9-6
9.1.5	Close File for Recording	9-6

9.1.6	Command Recording Example	9-7
9.2	Playing Back Command Files	9-8
9.2.1	Setting the Type of Playback	9-9
9.2.2	Calling Other Playback Files	9-9
9.2.3	Quitting Playback Mode	9-10
9.3	Command Line Batch Processing	9-10
9.4	Logging	9-12
9.4.1	Setting up Logging	9-13
9.4.2	Recording Commands and Logging Screen Output	9-15
9.4.3	Command Window Log File Example	9-15
9.4.4	Suspending and Resuming Output Log	9-15
9.4.5	Closing the Output Log File	9-17
9.5	Startup Options	9-18
9.6	CrossView Pro Command History Mechanism	9-19

I/O SIMULATION **10-1**

10.1	Introduction	10-3
10.2	I/O Streams	10-3
10.2.1	Setting Up File I/O Streams	10-4
10.2.2	Redirecting I/O Streams	10-6
10.3	File System Simulation	10-7
10.3.1	File System Simulation Libraries	10-8
10.4	Debug Instrument I/O	10-9
10.5	The Terminal Window	10-10
10.5.1	Terminal Window Keyboard Mappings	10-10

SPECIAL FEATURES **11-1**

11.1	Transparency Mode	11-3
11.2	RTOS Aware Debugging	11-4
11.3	Coverage	11-6
11.4	Profiling	11-8
11.5	Data Analysis	11-11
11.5.1	Supplied Data Analysis Window Scripts	11-13

11.5.2	Syntax of CrossView eXtension Language (CXL) . .	11-19
11.6	Background Mode	11-28
11.6.1	Configuration	11-28
11.6.2	Manual Refresh	11-29
11.6.3	Entering Background Mode	11-30
11.6.4	Leaving Background Mode	11-31
11.6.5	The Stack in Background Mode	11-32
11.6.6	Local and Global Variables	11-32
11.6.7	Refresh Limitation	11-32
11.6.8	Assertions	11-33

DEBUGGING NOTES **12-1**

12.1	Debugging Assembly Language	12-3
12.2	Debugging Multiple Programs	12-3

COMMAND REFERENCE **13-1**

13.1	Conventions Used in this Chapter	13-3
13.2	Commands: Summary	13-4
13.2.1	Viewing Commands	13-4
13.2.2	Data Monitoring	13-5
13.2.3	Data Analysis	13-7
13.2.4	Execution Control Commands	13-8
13.2.5	Record & Playback	13-11
13.2.6	Macros	13-12
13.2.7	Input/Output Simulation	13-12
13.2.8	File System Simulation	13-13
13.2.9	Target System Control	13-13
13.2.10	Save and Restore Target State	13-14
13.2.11	Help Commands	13-14
13.2.12	Search Commands	13-14
13.3	Commands: Detailed Descriptions	13-15

ERROR MESSAGES **14-1**

14.1	What this Chapter Covers	14-3
14.2	Error Messages	14-3

GLOSSARY **15-1**

15.1	What this Chapter Covers	15-3
15.2	Glossary Terms	15-3

INTERPROCESS COMMUNICATION **A-1**

1	COM Interface	A-3
1.1	Introduction	A-3
1.2	Using the COM Object Interface	A-3
1.2.1	Run-Time Environment	A-3
1.2.2	Command Line Options	A-3
1.2.3	Startup Directory	A-4
1.3	COM Interfaces	A-5
1.3.1	Activating the COM object	A-5
1.3.2	Methods	A-6
1.3.3	Implementation Details	A-7
1.4	Events	A-8
1.5	COM Examples	A-12
1.5.1	Python Examples	A-12
1.5.2	Visual Basic Examples	A-16
1.5.3	WORD Examples	A-17
1.5.4	Excerpt of the MIDL Definition	A-19
2	DDE Server Interface	A-20
2.1	Introduction	A-20
2.2	DDE Items and Topics	A-20
2.3	DDE Events	A-27
2.3.1	Packet Format	A-27
2.4	CrossView Pro DDE Specific Options and Commands	A-28
2.4.1	Command Line Options	A-28

2.4.2	Commands	A-28
2.5	Examples	A-29
2.5.1	Evaluating an Expression	A-29
2.5.2	Reading Target Memory	A-30
2.5.3	Writing Into Target Memory	A-31
2.5.4	Requesting Current File and Line Number	A-32
2.5.5	Using CrossView Pro as Pure Server	A-32

REGISTER MANAGER

B-1

1	Introduction	B-3
2	Invocation	B-3
3	Syntax of a Register File	B-4
4	SFR Base Address Register Special Variables	B-5
5	Fixed Register Set	B-6
6	Derivatives	B-7

SOUND SUPPORT (MS-Windows)

C-1

SIMULATOR

Sim-1

1	Introduction	Sim-3
2	Supported Features	Sim-3
2.1	Mapping Memory	Sim-3
2.2	Simulating I/O via I/O Port Address Blocks and Devices	Sim-4
2.3	Setting I/O Device Attributes	Sim-14
3	Restrictions	Sim-15
4	Simulator Commands	Sim-16

SmartMON ROM MONITOR

Rom-1

1	Introduction	Rom-3
1.1	Overview	Rom-3
1.2	SmartMON's Debugging Features	Rom-5

1.2.1	Initialize and Download	Rom-5
1.2.2	Stepping, Executing, and Halting	Rom-6
1.2.3	Setting Breakpoints	Rom-6
1.2.4	Full Disassembler	Rom-6
1.2.5	Displaying and Setting Memory and Registers	Rom-7
1.2.6	Tracing	Rom-7
1.2.7	Diagnostic Capabilities	Rom-7
1.2.8	System Calls	Rom-8
1.3	SmartMON Distribution Contents	Rom-8
2	Using SmartMON	Rom-9
2.1	Overview	Rom-9
2.2	SmartMON's Resource Requirements	Rom-10
2.3	SmartMON's Use of Interrupts and Traps	Rom-10
2.4	The Three Operational Modes of SmartMON	Rom-13
2.5	How SmartMON Sets Breakpoints	Rom-14
2.5.1	Setting Breakpoints on RAM Code Without Trace Mode Active	Rom-15
2.5.2	Instruction Breakpoints on ROM Code	Rom-15
2.5.3	Data Breakpoints	Rom-15
2.5.4	Complex Breakpoints	Rom-16
2.6	SmartMON's Tracing Features	Rom-16
2.6.1	Trace Points	Rom-17
2.6.2	Trace Buffer Operation	Rom-17
2.7	Single Stepping and Step-out-of-range	Rom-17
2.8	The Six Different Submodes of Execution Mode	Rom-18
2.9	How SmartMON Processes I/O	Rom-19
2.9.1	Interrupt Driven I/O	Rom-20
2.9.2	Polled I/O	Rom-23
2.9.3	Character Buffering	Rom-25
2.9.4	I/O System Calls	Rom-26
2.10	How SmartMON is Initialized	Rom-26
2.11	Run-time Notes	Rom-27
2.11.1	Stacks	Rom-27
2.11.2	Interrupt Service Routines	Rom-27
2.11.3	Downloading an ISR for Debugging	Rom-29

2.11.4	System Control	Rom-30
3	Target Interface Package	Rom-31
3.1	What is the TIP?	Rom-31
3.2	TIP Module #1: usreq.68k	Rom-32
3.2.1	Values Required by SmartMON	Rom-33
3.2.2	More Information on the usreq.68k Labels	Rom-34
3.3	TIP Module #2: rmain.68k	Rom-37
3.3.1	Stacks	Rom-38
3.4	RM_INIT Call	Rom-38
3.5	ROMM_GO System Call	Rom-42
3.6	Creating Your Own rmain.68k	Rom-43
3.7	TIP Module #3: io_drv.68k	Rom-44
3.8	portinit Call	Rom-44
3.9	Serial Port Interrupt Service Routine	Rom-45
3.10	TX_CHAR	Rom-45
3.11	RX_CHAR	Rom-46
3.12	How to Create Your Own io_drv.68k	Rom-46
3.12.1	Serial Port Polled I/O	Rom-47
3.12.2	TX_CHAR Using Polled I/O	Rom-47
3.12.3	RX_CHAR Using Polled I/O	Rom-48
3.12.4	Creating a Polled I/O io_drv.68k	Rom-48
3.13	TIP Modules #4 and #5: sysstp.68k and sys_go.68k	Rom-48
3.13.1	sys_go	Rom-48
3.13.2	sys_stop	Rom-49
3.14	TIP Module #6: diag_tbl.68k	Rom-49
4	Building SmartMON	Rom-50
4.1	Overview of the Build Process	Rom-50
4.1.1	Preparing the Build Environment	Rom-51
4.1.2	Assembling the TIP	Rom-51
4.1.3	Linking and Locating the Object Modules	Rom-52
4.2	Formatting	Rom-53
4.2.1	Programming the PROMs	Rom-54
4.3	Notes on Building Applications for SmartMON	Rom-54
4.3.1	Step 1: Modify pmain.68k	Rom-55
4.3.2	Step 2: Build the Demo Object Modules	Rom-57

4.4 Starting-up SmartMON with CrossView Pro Rom-58

4.5 Troubleshooting Rom-59

4.5.1 Locating the TIP Rom-60

4.5.2 Programming EPROMS Rom-61

4.6 Starting SmartMON with a Terminal or
Terminal Emulator Rom-62

5 SmartMON Command Language Rom-63

5.1 Overview Rom-63

5.2 Control Characters Rom-64

5.3 Operation Modes Rom-64

5.3.1 Command Mode Rom-65

5.3.2 Download Mode Rom-65

5.3.3 Execution Mode Rom-65

5.4 Command Descriptions Rom-66

6 System Calls Rom-111

6.1 Introduction Rom-111

7 Diagnostics Rom-124

7.1 SmartMON Diagnostics Rom-124

7.1.1 Overview Rom-124

7.1.2 RAM Tests Rom-124

7.2 User Diagnostics Rom-135

7.2.1 Overview Rom-135

7.2.2 How to Write a User Diagnostic Rom-138

7.2.3 Linking Diagnostics with SmartMON Rom-140

7.2.4 Downloading and Running User Diagnostics Rom-141

7.2.5 How SmartMON Processes UD Commands Rom-141

7.2.6 Installing RAM Based Diagnostics Rom-141

7.2.7 Running a Test Rom-142

BACKGROUND DEBUG MODE **Bdm-1**

1 Introduction Bdm-3

2 Background Debug Mode as a CrossView Pro
Execution Environment Bdm-3

2.1 Additional Software Contents Bdm-4

3	BDM Installation	Bdm-5
3.1	Hardware Installation	Bdm-5
3.2	Software Installation	Bdm-6
3.3	Configuration Options	Bdm-7
3.4	Target Environment Setup	Bdm-7
4	BDM Command Interface (Emulator Mode)	Bdm-9
4.1	Operation Modes	Bdm-9
4.2	Command Descriptions	Bdm-10
5	Troubleshooting	Bdm-31
5.1	Unable to Open Driver from OpenDriver	Bdm-31
5.2	Open Failed from CrossView Pro	Bdm-31
5.3	Unexpected Responses	Bdm-31
6	Other Considerations	Bdm-32

INDEX



CONTENTS

MANUAL PURPOSE AND STRUCTURE

PURPOSE

This manual is aimed at users of the CrossView Pro debugger for the 68K/ColdFire family. It assumes that you are familiar with programming the 68K/ColdFire.

MANUAL STRUCTURE

Related Publications
Conventions Used In This Manual

CHAPTERS

1. Overview
Highlights specific CrossView Pro features and capabilities, and shows how to compile code for debugging.
2. Software Installation
Describes how to install CrossView Pro on your system.
3. Command Language
Details the syntax of CrossView Pro's command language.
4. Using CrossView Pro
Describes the basic methods of invoking, operating, and exiting CrossView Pro.
5. Controlling Program Execution
Describes the various means of program execution.
6. Accessing Code and Data
Describes how to view and edit the variables in your source program.
7. Breakpoints and Assertions
Describes breakpoints and assertions.

8. Defining and Using Macros

Describes how to simplify a complicated procedure by creating a "shorthand" macro which can be used to execute any sequence of CrossView Pro or C language commands and expressions.

9. Command Recording & Playback

Describes the record and playback functions of CrossView Pro.

10. I/O Simulation

Describes how to simulate your input and output using File System Simulation (FSS), File I/O (FIO) or Debug Instrument I/O (DIO).

11. Special Features

Describes special features of CrossView Pro, such as the Transparency Mode, RTOS Aware Debugging, Coverage, Profiling and the Background Mode.

12. Debugging Notes

Contains some notes about debugging in special situations.

13. Command Reference

An alphabetical list of all CrossView Pro commands. Consult this chapter for specifics and the exact syntax of any CrossView Pro command.

14. Error Messages

Contains CrossView Pro error messages and gives advice for correcting them.

15. Glossary

Defines the most common terms used in embedded systems debugging.

APPENDICES

- A. Interprocess Communication
Contains a description of the COM interface and the DDE interface.
- B. Register Manager
Contains a description of the register manager **rm68k**.
- C. Sound Support (MS-Windows)
Describes how to add sound to CrossView Pro events under MS-Windows.

ADDENDUM

Simulator Mode

Contains information specific to Simulator Mode.

SmartMON ROM Monitor

Contains a description of the ROM Monitor.

Background Debug Mode

Contains a description of the Background Debug Mode.

INDEX

RELATED PUBLICATIONS

- The C Programming Language (second edition) by B. Kernighan and D. Ritchie (1988, Prentice Hall)
- ANSI X3.159-1989 standard [ANSI]
- ISO/IEC 9899:1999(E), Programming languages – C [ISO/IEC]
- 68K/ColdFire C Compiler/Assembler User's Manual [TASKING, MA001-022-00-00]
- 68K/ColdFire C Compiler/Assembler Reference Manual [TASKING, MA001-020-00-00]

CONVENTIONS USED IN THIS MANUAL

Notation for syntax

The following notation is used to describe the syntax of command line input:

bold	Type this part of the syntax literally.
<i>italics</i>	Substitute the italic word by an instance. For example: <i>filename</i> means: type the name of a file in place of the word <i>filename</i> .
{ }	Encloses a list from which you must choose an item.
[]	Encloses items that are optional.
	Separates items in a list. Read it as OR.
...	You can repeat the preceding item zero or more times.

For example

command [*option*]... *filename*

This line could be written in plain English as: execute the command *command* with the optional options *option* and with the file *filename*.

Illustrations

The following illustrations are used in this manual:



This is a note. It gives you extra information.



This is a warning. Read the information carefully.



This illustration indicates actions you can perform with the mouse.



This illustration indicates keyboard input.



This illustration can be read as “See also”. It contains a reference to another command, option or section.

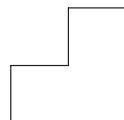
CHAPTER

1

OVERVIEW



TASKING



1

CHAPTER

1.1 INTRODUCTION

This chapter highlights many of the features and capabilities of CrossView Pro, including an Introduction to Source Level Debugging and the Embedded Development Environment.

This chapter also contains the section *Getting Started*, which shows you how to compile a program to work with the debugger.

1.2 CROSSVIEW PRO'S FEATURES

CrossView Pro is TASKING's high-level language debugger. CrossView Pro is a real-time, source-level debugger that lets you debug embedded microprocessor systems at your highest level of productivity. Its powerful capabilities include:

- Multi-Window Graphical User Interface
- C and Assembly level debugging
- C Expression Evaluation including Function Calls
- Breakpoints (both hardware and software)
- Probe Points
- Assertions (software data breakpoints)
- C-trace, Instruction Trace
- I/O Simulation (IOS)
- Data Monitoring
- Single Stepping
- Coverage
- Profiling
- Macros
- Flexible Record & Playback Facilities
- Real-Time Kernel Support
- On-line context sensitive Help
- Documentation

Multi-Window Interface

This interface uses your host's native windowing system, so that you already know how to open, close and resize windows. With windows you can keep track of information concerning registers, the stack, and variables. CrossView Pro automatically updates each window whenever execution stops.

You have great freedom in designing a suitable display. You can hide and resize the various windows if you choose.

Statement Evaluation

You can enter C expressions, CrossView Pro commands or any combination of the two for CrossView Pro to evaluate. You may also call functions defined in your source code from the command line. Expression evaluation is an ideal way to test subroutines by passing them sample values and checking the results.

Breakpoints

Breakpoints halt program execution and return control to you. There are several types of breakpoints: code, data, instruction count, cycle count, timer and sequence.

Code breakpoints let you halt the program at critical junctures of program execution and observe values of important variables.

You may place data breakpoints to determine when memory addresses are read from, written to, or both. With data breakpoints, you can easily track the use and misuse of variables.

An instruction count breakpoint halts the program after a specified number of instructions have been executed; a cycle count breakpoint stops the program after a number of CPU cycles; a timer breakpoint stops the program after a number of micro seconds or ticks and sequence breakpoints stop the program when a number of breakpoints are hit in a specified sequence.

Data breakpoints, instruction count breakpoints, cycle count breakpoints and timer breakpoints are not available for all execution environments, please check the Addendum.

Probe Point Breakpoints

A breakpoint can be treated as a probe point. When a probe point breakpoint is hit, the associated commands are executed and program execution is continued. Probe points are used with File I/O simulation and sequence breakpoints.

Assertions

A powerful assertion mechanism lets you catch hard-to-find errors. An assertion is a command, or series of commands, executed after every line of source code. You may use assertions to test for all sorts of error conditions throughout the entire length of your program.

C-Trace

CrossView Pro has a separate window that displays the most recently executed C statements or machine instructions. This feature uses the execution environment's trace buffer along with symbolic information generated during compilation. This feature is depending on the execution environment.

I/O Simulation (IOS)

With I/O simulation you can debug programs before the actual input and output devices are present. CrossView Pro can read input data from the keyboard or a file, or can send output to a window or a file. You can view the data in several formats, including hexadecimal and character. You can have an unlimited number of simulated I/O ports, which can be associated with the screen and displayed in windows.

Data Monitoring

You may place variables and expressions in the Data window, where CrossView Pro updates their values when execution stops.

Single Stepping

With CrossView Pro, you can single step through your code at source level or at assembly level, into or over procedure calls. Running your program one line at a time lets you check variables and program flow.

Coverage

When a command such as StepInto or Continue executes the application, CrossView Pro traces all memory access, i.e. memory read, memory write and instruction fetch. Through *code coverage* you can find executed and non-executed areas of the application program. Areas of unexecuted code may exist because of programming errors or because of unnecessary code. It may be that your program input, your test set, is incomplete; It does not cover all paths in the program. *Data coverage* allows you to verify which memory locations, i.e. which variables, are accessed during program execution. Additionally, you can see stack and heap usage. The availability of this feature depends on the execution environment.

Profiling

Profiling allows you to perform timing analysis on your software. Two forms of profiling are implemented in CrossView Pro.

Function profiling, also called cumulative profiling, gives you timing information about a particular function or set of functions. CrossView Pro shows: the number of times a function is called, the time spent in the function, the percentage of time spent in the function, and the minimum/maximum/average time spent in the function. The timing results include the time spent in functions called by the profiled function.

Code range profiling presents timing information about a consecutive range of program instructions. CrossView Pro displays the time consumed by each line (source or disassembly) in the Source Window. Next to this, the Profile Report dialog shows the time spent in each function. The timing results do not include the time consumed in functions called by the profiled function.

The availability of profiling depends on the execution environment. Function profiling can be supported if the execution environment provides a clock that starts and stops whenever execution starts and stops. Code range profiling heavily relies on special profiling features in the execution environment. Normally code range profiling is only supported by instruction set simulators.

Macros

Macros let you store and recall complex commands and expressions with a minimal number of keystrokes. You can store macros in a "toolbox", making it possible to execute complex functions with the touch of a mouse button. You can also place macros in command lists of breakpoints and assertions. You can use flow control statements within macros, and macros can call other macros, allowing you to construct arbitrarily complex sequences. Macros can accept multiple parameters, be saved and loaded from files and can even rename existing CrossView Pro commands.

Record & Playback

At any time, you can record the commands you type, and optionally their output, to a file. You can also play back files of commands all at once or in a single-step playback mode. These functions are helpful for setting up standardized debugging tests or to save results for later study or comparison.

Kernel Support

CrossView Pro supports RTOS (Real-Time Operating System) aware debugging for various kernels. Since each kernel is different, the RTOS aware features are not implemented in the CrossView Pro executable, but in a library that will be loaded at run-time by CrossView Pro. The amount of windows and dialogs and their contents is kernel dependent.

On-Line Help

When you click on a **Help** button or when you press the **F1** function key in an active window, the CrossView Pro help system opens at the appropriate section. From this point, you can also access the rest of the help system.

Documentation

CrossView Pro has a comprehensive set of documentation for both new and experienced users. The manual includes an installation guide, description of debugging with CrossView Pro, error messages, and a command reference section. The documentation tries to cover a wide range of expertise, by making few assumptions about the technical experience of the reader.

1.3 SOURCE LEVEL DEBUGGING

CrossView Pro is a source level debugger. *Source level* means that debugging works on the actual C code or assembly code. CrossView Pro can deal with global and local variables that are both statically and dynamically allocated variables. Therefore, it can deal with compiled addresses of variables that move around the stack. CrossView Pro knows the compiler's addressing conventions for variables of any type.

The Debugging Environment

All debugging configurations follow a similar pattern. There is a *host* system where the debugger runs, and a *target* system (usually an execution environment), where the program being debugged runs. There may also be a *probe* that can plug into the actual hardware of the embedded system being designed.

CrossView Pro provides a high-level interface between you, the user, working at the host system and a program running at the target system (execution environment). This means that you may issue commands that refer directly to the variables, source files, and line numbers as they appear in the source program. You can do this because CrossView Pro uses symbol information generated during compilation to translate the high-level commands that you type into a series of low level instructions that the target system understands. Using Generic Debug Instrument (GDI) calls towards a shared library for the simulator, or using a connection between the host and target, CrossView Pro finds out information about the state of the target program and then tells the target to perform the requested actions.

A host-target arrangement can perform functions beyond the reach of traditional software-based debuggers. Since the target contains the actual chip, CrossView Pro can observe its operations without interfering. The existence of CrossView Pro and the host is invisible to the target program. This means that the program under debug runs exactly the same as the final program will in a real embedded system (except for real-time situations like timings).

With CrossView Pro, you may also take advantage of any advanced capabilities of your target hardware through *emulator mode* (transparency mode). In transparency mode you can communicate with the target as if the host system were a terminal directly connected to the target. You can enter and leave transparency mode freely without restarting the debugger or the target system. CrossView Pro therefore does not interfere with the normal operation of the target hardware. Thus the debugger is a powerful accessory to the machine-level debugging that you might do with the target system alone. The transparency mode is not available for all execution environments.

1.4 HOW CROSSVIEW PRO WORKS

Although it is not necessary to know how CrossView Pro performs its debugging, you may be curious how CrossView Pro works.

Whenever you enter a debugger command, CrossView Pro obtains information from or controls the execution environment by sending appropriate commands over the host-target link. A typical session may go something like this:

1. Highlight `initval` and click on the **Show Expression** button in the Source Window.

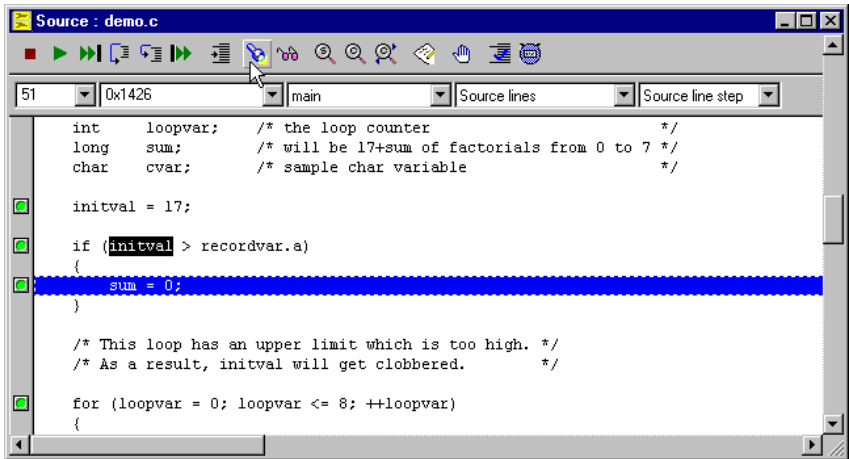


Figure 1-1: Inspect a variable

2. CrossView Pro converts this action into a command. Depending on preferences you have set, the variable is shown in the Data Window or the Expression Evaluation dialog is shown.
3. CrossView Pro consults the symbol table to deduce the type and address of `initval`. Suppose `initval` is a variable of type `int` which lies at absolute location 100.
4. The debugger forms a command asking the target system to read two bytes starting at address 100 (the size of an `int` equals 2).
5. CrossView Pro then transmits the command to the target system and receives the response.

6. CrossView Pro interprets the response, and for example determines that `initval` equals 17.
7. CrossView Pro then displays `initval=17` since it knows `initval`'s type.

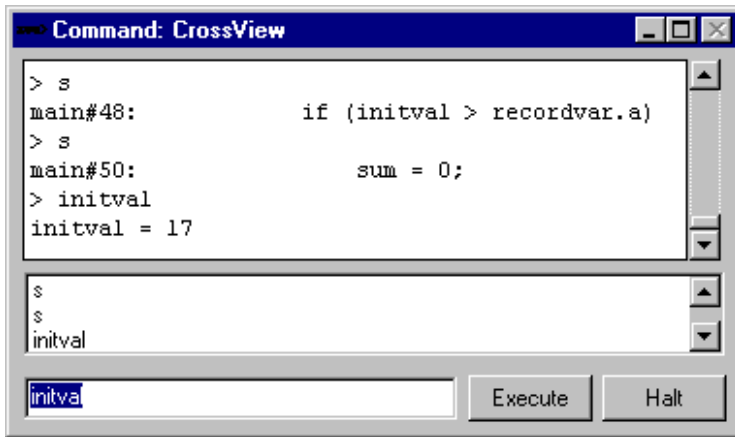


Figure 1-2: CrossView Pro Command Output

This is a simplified example, many CrossView Pro commands require several complex transactions, but all take place without you being aware of them.

1.5 PROGRAM DEVELOPMENT

The CrossView Pro debugger is part of a toolchain that provides an environment for modular program development and debugging. Figure 1-3 shows the structure of the toolchain.

Apart from the debugger the toolchain contains the following elements:

Compiler

The compiler translates C source into machine instructions for the target microprocessor. The input is one or more source programs. The C language implemented conforms to the ANSI C standard ANSI/ISO 9899-1990.

Compiler output is an object module suitable for linking with other modules. These object modules can also be catalogued in a library using the librarian utility. The compiler has optional listings which show interleaved source and generated machine instructions, along with cross-reference listings.

Run-time Library

The 68K/ColdFire toolset includes full run-time libraries: math functions, memory allocation functions, standard I/O functions, string manipulation functions, and floating point routines.

Assembler

The 68K/ColdFire toolset includes a macro assembler. The source format is manufacturer-compatible. That is, existing manufacturer-compatible assembly code is easily reassembled using the TASKING assembler. Minor changes may be needed if the assembled modules are to be invoked by compiled modules.

The input to the assembler is one or more source programs. The output is a corresponding number of object modules suitable for linking to other modules. The object modules can be catalogued in a library. Assembler object modules are compatible with C compiler object modules. Source, cross-reference, and symbol table listings are available from the assembler.

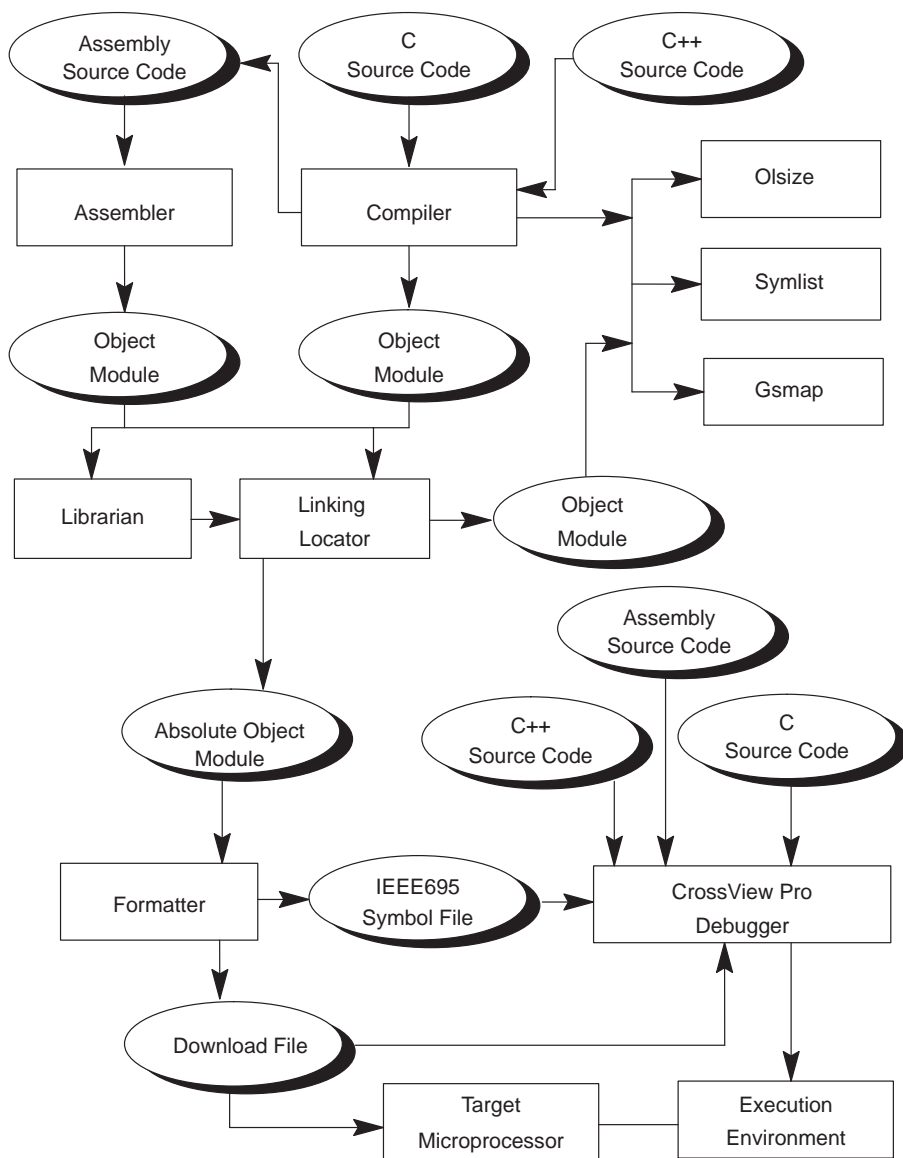


Figure 1-3: Toolchain development flow

Utilities

The TASKING compiler and assembler software includes a full set of utilities. These tools increase programming productivity by reducing the time spent on repetitive software building tasks. A brief description of the utilities is given below

- **Linking Locator**

The Linking Locator integrates the results of separate compilations and assemblies into a single absolute module. This is done in three separate steps, any or all of which can be performed in a single invocation of the linking locator. The first step, called “linking”, consists of combining separate object modules into a composite module by resolving references. Usually these object modules are produced by the assembler and/or compiler, but pre-linked object modules may be also used as input. The linking locator searches libraries to satisfy any unresolved references in the module it is constructing.

The second (optional) step, called “ROM processing”, consists of building initialization segments used to initialize read-write data. All ROM-based systems must execute code to initialize their read-write data, since the initial values cannot be maintained in RAM (random-access memory), and read-write data cannot be allocated in ROM (read only memory). This data could be initialized by large numbers of assignment statements, but it is more convenient and efficient to employ ROM processing instead. Unlike the read-write data, the initialization segment is suitable for placement in ROM. The initial data values are copied from ROM to RAM at the time of initialization by the library routine `rcopy`.

The final step, called “locating”, consists of assigning absolute target-memory locations to relocatable segments and resolving address references. The linking locator gives you complete control over placement of all code and data, but it also has the capacity; to automatically locate collections of segments in bounded areas of the target memory. The output is an object module with absolute addresses substituted where appropriate. A completely located module contains all the information necessary to load and execute the code on the target microprocessor. The linking locator can resolve the problem of storing a program into a fragmented memory space consisting of ROM, RAM and I/O mapped device addresses.

- **Formatter**

The formatter converts the contents of an absolute object module into one of the industry standard formats, in either an ASCII hex or a binary format. The formats provide for loading of object text, that is, code and data, into memory of the target processor using a loader. The loader is generally provided by an emulator or other instrumentation system, or by a ROM-resident monitor program. The formatter offers many different formats in order to be compatible with a wide range of loaders.

The input is a module from the linking locator and the output is a formatted load file. The formats may also be used as input to a PROM burner to program read-only memory.

- **Librarian**

The librarian is a tool for managing libraries of program modules at the pre-link or post-link phase of development. The librarian creates, maintains, and selectively lists library index files. A library index file is a text file defining an indexing structure which describes a collection of object modules. It consists of a series of index entries, one for each object module. The librarian's input is taken from the library and/or object modules named on the command line or through options specified on the command line. The object modules named on the command line or in a file are added to the library. Libraries simplify the task of linking modules, since the linking locator can automatically search libraries for required modules.

- **Global Symbol Mapper**

The global mapper (**gsmmap**) displays global symbols either alphabetically or by address. Gsmmap can be used before or after linking or locating to list external names and the definitions of global symbols. The gsmmap listing shows an absolute address (after locating), length, class, and alignment for each segment.

- **Object Size List Utility**

The object size list utility (**olsize**) lists the total number of words of code, data, and constant data in an object module.

- **Symbol List Utility**

The symbol list utility (**symlist**) produces a listing of all global and local symbols. When the debugger option, (**-d**), is used in compilation or assembly, target locations for source lines of input code are included in the listing. The input may be any combination of unlinked object modules, linked object modules, and absolute modules. The symlist listing is composed of three parts: a table of executable line numbers and code addresses, a listing of all symbols and their attributes, and an alphabetical list of all symbols with pointers to each symbol's definition and attributes.

1.6 GETTING STARTED

1.6.1 BEFORE STARTING

Before using CrossView Pro, there are several things that you must do:

- Install the CrossView Pro software. Directions for your particular system are found in the *Software Installation* chapter.
- Configure your execution environment as described in the *Execution Environment* addendum.
- Compile the program that you want to debug. A brief description of this process is outlined in the section *Building Your Executable* later in this chapter.

For the purpose of getting you started quickly, we have supplied you with a demo program that you can debug. The demo program is `demo.abs`.

1.6.2 SETTING UP THE EXECUTION ENVIRONMENT

The following text only applies to ROM monitor and emulator versions of CrossView Pro.

In order for the host and execution environment to communicate, a proper connection must exist between the two machines. Here are some important considerations:

- Use the correct kind of RS-232 cable. Note there are at least two types of cables, *null modem* and *direct*. Consult the execution environment's manual for the correct type.
- Make sure the execution environment is configured to communicate with the host at the baud rate that CrossView Pro expects. Usually, the baud rate is 9600, but this is not always the case.
- Use the correct ports on both the execution environment and host. Many machines have two ports. If you use a different port on the host than the default (COM1 for PC), you will have to use a special startup switch, **-D**. See the startup options of the *Using CrossView Pro* chapter.
- See the addendum for details on the connection to the execution environment.

1.6.3 STARTING CROSSVIEW PRO

To invoke CrossView Pro, simply double-click on its icon. CrossView Pro starts up and opens the command window, source window and other windows.

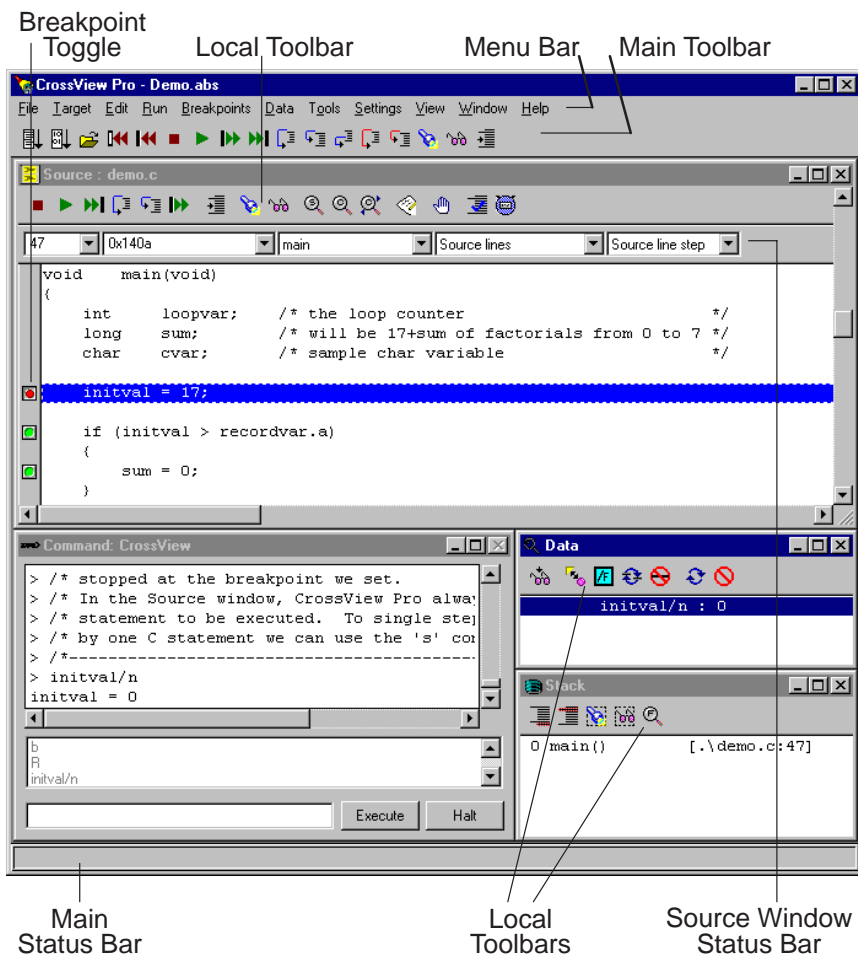


Figure 1-4: Command Window

CrossView Pro can be passed the name of an execution (*.abs) file. This can be done from a command line, but the native windowing system often provides alternatives. Usually this involves dragging the program to be debugged onto the CrossView Pro executable from the Windows Explorer for Windows 95/98/XP/NT/2000, and dropping it there or associating CrossView Pro to be the application to start when double-clicking an .abs icon. CrossView Pro will start and load the symbol information from that file.

1.6.3.1 CROSSVIEW PRO TARGET SETTINGS

You can specify specific CrossView Pro startup settings in the Target Settings dialog.

To open the Target Settings dialog:

- From the **Target** menu, select **Settings...**

The Target Settings dialog box appears as shown in figure 1-5.

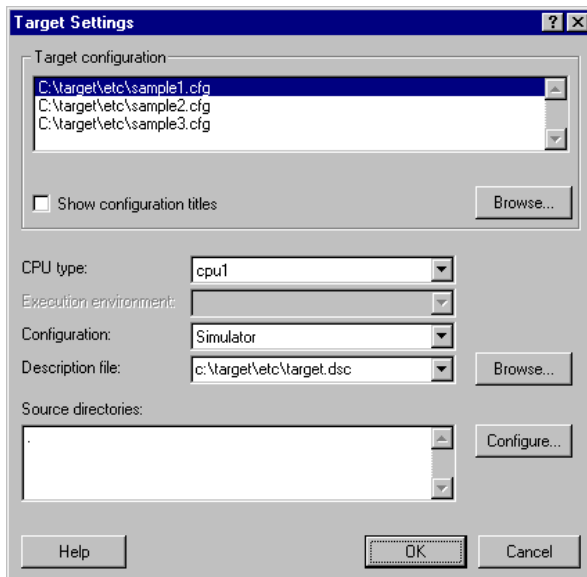


Figure 1-5: CrossView Pro Target Settings

You can set the following items in this dialog:

- Select a target configuration file (*.cfg) containing some target specific configuration items. This file is optional. See the text below for more information.
- Select the CPU type.
- Specify the source directories for CrossView Pro. Click on the **Configure...** button to change the list of source directories.

Target Configuration

The available targets are described by the target configuration files (*.cfg in the etc subdirectory). The target configuration files are text files and can be edited with any text editor.

Empty lines, lines consisting of only white space are allowed. Comment starts at an exclamation-sign (!) and ends at the end of the line.

An information line has the following synopsis:

[! comment] *field*: *field-value*

field one of the keywords described below

field-value the value assigned to the field

comment optional comment

The *fields* listed in the configuration file are:

Field	Description
title	The full name of the configuration. This name will be displayed in the Target configuration field of the Target Settings dialog.
cpu_type	The name of the CPU. You can specify multiple CPU types separated by white space.
debug_instrument_module	The name of the Debug Instrument (using GDI) used for debugging.
radm	The name of the Debug Instrument (using KDI) used for RTOS aware debugging. (optional).

Field	Description
transparent_cmd	The terminal emulator program command line to use when entering transparency mode via the View Transparent Mode menu item (ROM monitor only).
BDM_DelayFactor	(0-453556) set the timing delay factory for communications to the BDM port (BDM 68K only).



Notes:

- Fields not required for the target can be omitted.
- CrossView Pro searches for the *.cfg files in the current directory and in the etc directory.

1.6.3.2 CONFIGURING CROSSVIEW PRO

You may have to configure CrossView Pro to talk to the emulator or ROM monitor. If you have a simulator version this step is not needed and the associated menu item is grayed. To configure CrossView Pro:

- From the **Target** menu, select **Communication Setup...**
The Communication Setup dialog box appears as shown in figure 1-6.
- Adjust the communication parameters (baud rate and I/O port) to match your hardware configuration.
- Close the dialog box by clicking on the **OK** button.
- The settings in this dialog (and other dialogs) will be saved on exiting CrossView Pro, when the **Save desktop and target settings** check box in the **Save** tab of the Options dialog is set. This dialog always appears on exiting CrossView Pro.

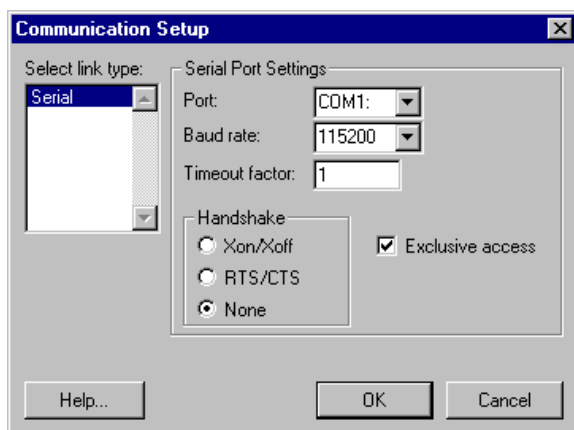


Figure 1-6: Setting up CrossView Pro Communications

1.6.3.3 LOADING SYMBOLIC DEBUG INFORMATION

You must tell CrossView Pro which program that you want to debug. To do this:

- From the **File** menu, select **Load Symbolic Debug Info...**
The Load Symbolic Debug Info dialog box appears, as shown in figure 1-7.
- Type in the path and file name of the program that you want to debug, or click on the **Browse...** button to bring up a file selection dialog box. In our example we are using `demo.abs`. Note that in most cases you will want to set the code bias field to `0x0000`.
- Set the **Download image too** check box by clicking on it, if you want to download the image of your absolute object file to the target. You can decide to postpone downloading to the target. In that case you can select **Download Application...** from the **File** menu any time afterwards.
- Set the **Reset target system** check box if you want to reset the target system to its initial state. You can decide to postpone resetting the target. In that case you can select **Reset Target System** from the **Run** menu afterwards.

- Set the **Goto main** check box if you want to execute the startup code. This automatically enables the **Reset application** check box. You can decide to postpone going to the main function. In that case you can execute a high-level single step afterwards.
- Set the **Use memory definition file** check box if you want CrossView Pro to process an application specific memory definition file before a new application file is loaded and/or downloaded to the target. CrossView Pro uses such a file to determine how much memory must be allocated from the system and how logical addresses are mapped to physical addresses.
- When you click on the **Communication setup...** button (if available), the Communication Setup dialog box appears as shown in figure 1-6. With the **Target Settings...** button you can open the Target Settings dialog. Please check the information in these dialogs before downloading an application.
- When you click on the **Load** button, the program's symbol file will be loaded into the debugger and, if you have set the **Download image too** check box, the image of your absolute object file will be downloaded.
- Clicking on **Cancel** ignores all actions.

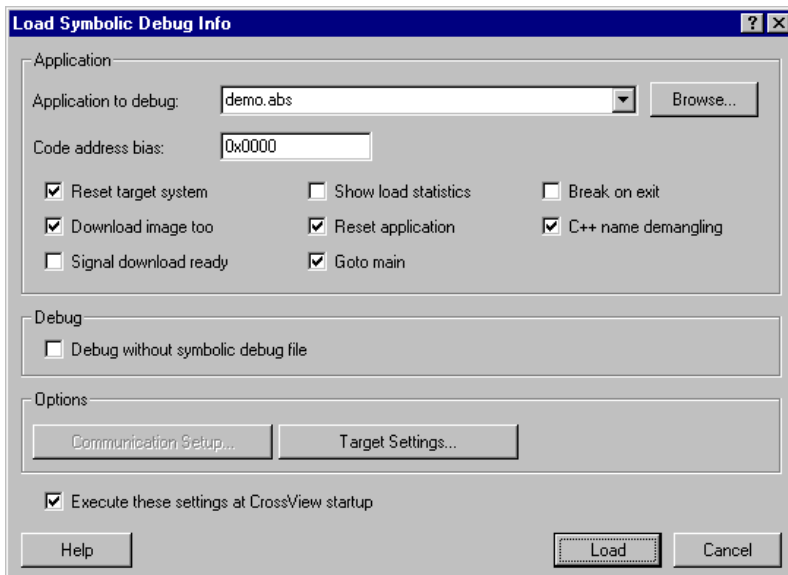


Figure 1-7: Loading Symbolic Debug Information

CrossView Pro remembers all previously saved settings. In this case, the Load Symbolic Debug Info dialog already contains the previously saved configuration, so you only have to click the **Load** button to perform your actions.

Compare Application

You can use the **File | Compare Application...** dialog to check if a file matches the downloaded application. This can be useful when your program has changed some of your code.

1.6.4 EXECUTING AN APPLICATION

To view your source while debugging, the Source Window must be open. To open this window,

- From the **View** menu, select **Source | Source lines**

Before starting execution you have to reset the target system to its initial state. The program counter, stack pointer and any other registers must be set to their initial value. The easiest way to do this is:

- Set the **Reset target system** check box and the **Goto main** check box in the Load Symbolic Debug Info dialog box. (See the previous section) **Goto main** automatically enables the **Reset application** check box.



Depending on your execution environment a target system reset may have undesired side effects. For this reason, the target system reset is executed before the code is downloaded to the target.

If you have not checked these items:

- From the **Run** menu, select **Reset Target System**
- From the **Run** menu, select **Reset Application**
- Execute a high-level single step (either into or over) using the toolbar in the Source Window (or **F11/F10**).

The first single step executes the startup code and stops at the first line of code in `main()`. You should see your program's source code.

Another way of getting there is:

- Set a breakpoint at the entry of `main()` by clicking on a breakpoint toggle at the left side of the text in the Source Window. See figure 1-8.

- Start the application with **Run | Reset Application** and **Run | Run**.

To set a breakpoint you can:

- Click on a breakpoint toggle (as shown in figure 1-8) to set or to remove a breakpoint. A green colored toggle shows that no breakpoint is set. A red colored toggle shows that a breakpoint is installed. An orange colored toggle shows that an installed breakpoint is disabled.

Due to compiler optimizations it is possible that a C statement does not translate in any executable code. In this case you cannot set a breakpoint at such a C statement. No breakpoint toggle is shown in this case.

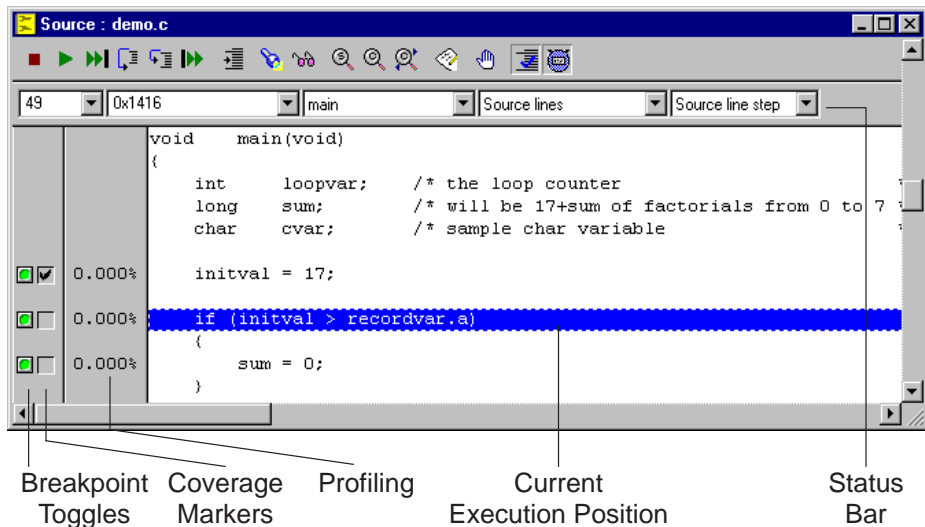


Figure 1-8: Getting Control

Now it is time to execute your program:

- From the **Run** menu, select **Run**

In the Source Window the current execution position (the statement at the address identified by the current value of the program counter) is highlighted in blue. As a result, when execution stops, the line you set a breakpoint on is highlighted. You can now single step through your program using the **Step Into** and **Step Over** buttons in the Source Window. Or you may choose to execute the rest of the program (or at least until the next breakpoint) with the **Run** button.

At any point you can interrupt the emulator and regain control by clicking on the **Halt** button in either the Source Window or the Command Window.

For more information on executing a program, see the chapter *Controlling Program Execution*.

1.6.5 DEBUGGING AN APPLICATION

When debugging your application you probably want to see the calling sequence of your program, and inspect the contents of variables and data structures used within your program.

To see the calling sequence of your program the Stack Window must be open. The stack window shows the functions that are currently on the stack. To open the stack window,

- From the **View** menu, select **Stack**

To see the value of the local variables of a function,

- From the **View** menu, select **Data | Watch Locals Window**

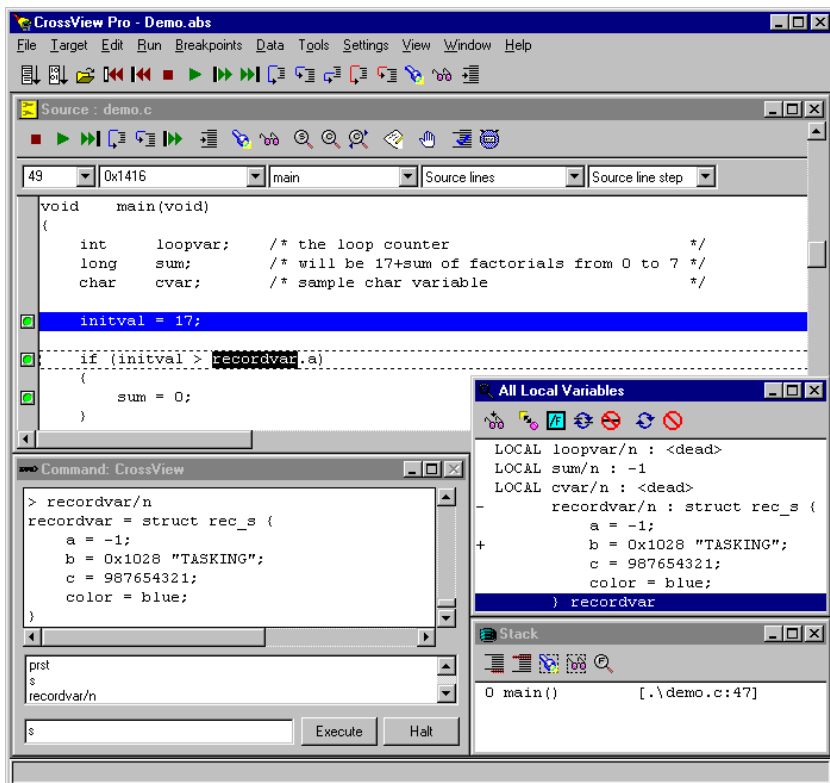


Figure 1-9: Watch variables

To inspect the value of global variables and data structures,

- Double-click on the variable name in the Source Window.

Depending on preferences you have set, the variable is shown in the Data Window as shown in figure 1-9 or the dialog displayed in figure 1-10 is shown.

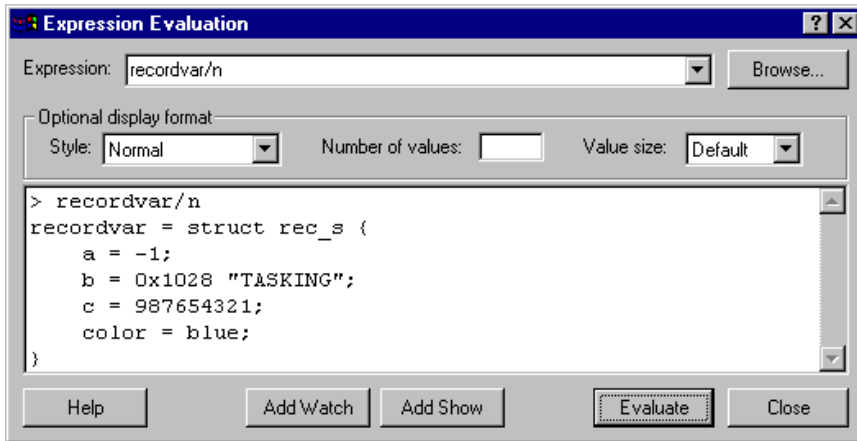


Figure 1-10: Expression evaluation

Pointers, structures and arrays displayed in the data window have a compact and expanded form. The compact form for a structure is just `<struct>`, while the expanded form shows all the fields. The compact form of a pointer is the value of the pointer, while the expanded form shows the pointed-to object. The compact form is indicated by putting a '+' at the start of the display. (i.e., the object is expandable), while a '-' indicates the expanded form (i.e., the object is contractible). Nesting is supported, so structures within structures can likewise be expanded, ad infinitum.

To expand a pointer or a structure:

- Click on the '+' in the Data Window

1.6.6 CROSSVIEW PRO OUTPUT

Nearly every CrossView Pro command can be given using the graphical user interface. These commands and the debugger's response is logged in the Command Output Window which is the upper part of the Command Window. Alternatively, CrossView Pro commands can be entered directly (without using the menu system) in the command edit field of the command window.

To open the Command Window:

- From the **View** menu, select **Command | CrossView**

Figure 1-11 shows an example of the Command Window. Commands can be typed into the command edit field (bottom field) or selected from the command history list (middle field) and edited then executed. The top field is referred to as the Command Output Window. Each command, echoed from the command edit field, is displayed with a '>' prefix. CrossView's response to the command is displayed below the command.

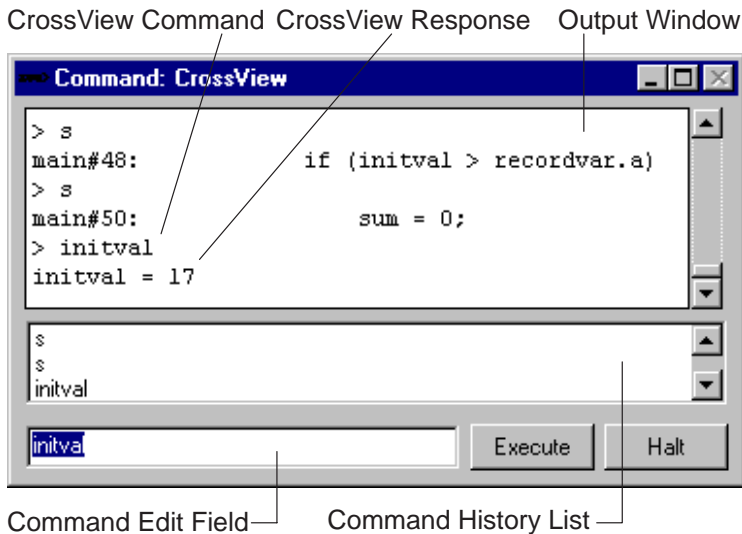


Figure 1-11: CrossView Pro Command Output

You can choose to clear the command edit field after executing a command. From the **File** menu, select **Options...** and select the **Desktop** tab. Enable the **Clear command line after executing command** check box. You can use the **clear** command to clear the Output Window.

1.6.7 EXITING CROSSVIEW PRO

To quit a debugging session:

- From the **File** menu, select **Exit** or close the Command Window.
- In the Options dialog that appears, select in the **Save** tab the options you want to be saved for another debug session.
- Click on the **Exit** button in the Options dialog.

If you selected one or more items in the Options dialog, your settings will be saved in the initialization file `xvw.ini`. This file is located in the startup directory.

Workspace files

If you have set the **Save desktop and target settings** check box in the **Save** tab, CrossView Pro will create a workspace file (`.cws`) for each debugged or loaded application. The settings will be restored in a following debug session. If CrossView Pro cannot find a workspace file for a loaded application it uses the default workspace file `xvw.cws` in the `etc` directory.

A CrossView Pro workspace file contains:

- Window positions and sizes
- Local toolbars status
- Main toolbar configuration
- Monitored variables in Data windows
- Memory window settings
- Terminal window settings
- Coverage and profiling display settings in the Source window
- Color settings

1.6.8 WHAT YOU MAY HAVE DONE WRONG

Most problems in starting up CrossView Pro for a debugging session stem from improperly setting up the execution environment or from an improper connection between the host computer and the execution environment. Some targets will require you to enter transparency mode to set the execution environment for a debugging session. Check the notes for your particular execution environment.

Here are some other common problems:

- Specifying the wrong device name when invoking the debugger.
- Specifying a baud rate different from the one the execution environment is configured to expect.
- Not supplying power to the execution environment or an attached probe.
- Using the wrong kind of RS-232 cable.
- Plugging the cable into an incorrect port on the execution environment or host. Some target machines and hosts have several ports.
- Installation of a device driver or resident application that uses the same communications port on the host system.
- The port may already be in use by another user on some UNIX hosts, or being allocated by a login process.

1.6.9 BUILDING YOUR EXECUTABLE

The subdirectory `xvw` in the `examples` subdirectory contains a demo program for the 68K/ColdFire toolchain.

In order to debug your programs, you will have to compile, assemble, link and locate them for debugging using the TASKING 68K/ColdFire tools. You can use EDE, the Embedded Development Environment (which uses a project file and a makefile) or you can call the makefile from the command line.

1.6.9.1 USING EDE

EDE stands for "Embedded Development Environment" and is the Windows oriented Integrated Development Environment you can use with your TASKING toolchain to design your application.

To use EDE on the demo program, located in the subdirectory `xvw` in the `examples` subdirectory of the 68k product tree, follow the steps below.

A detailed description of the process creating the sample program `demo.abs` is described below. This procedure is outlined as a guide for you to build your own executables for debugging.



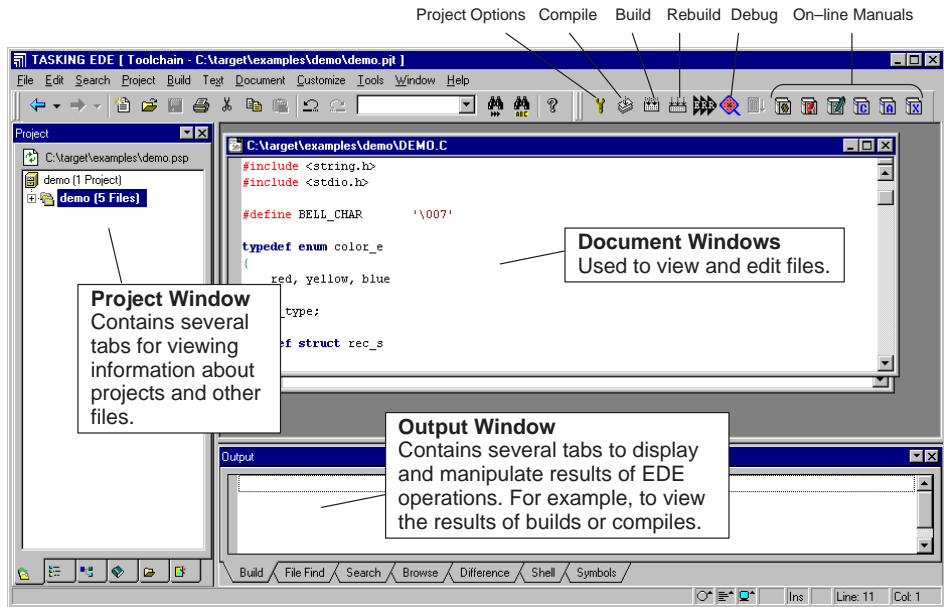
The dialog boxes shown in this manual serve as an example. They may slightly differ from the ones in your product.

How to Start EDE

You can launch EDE by double-clicking on the EDE shortcut on your desktop.



The EDE screen provides you with a menu bar, a toolbar (command buttons) and one or more windows (for example, for source files), a status bar and numerous dialog boxes.



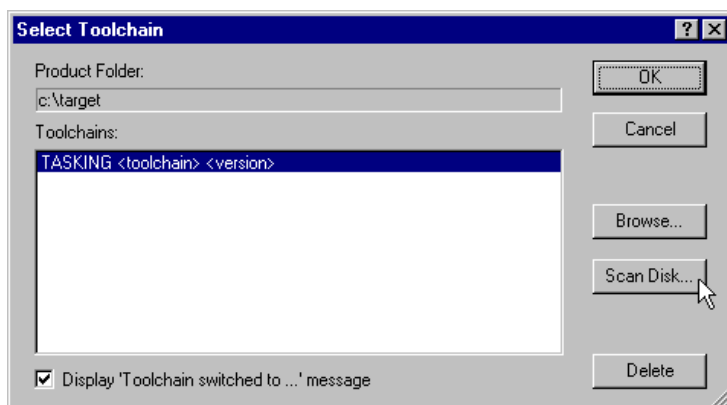
How to Select a Toolchain

EDE supports all the TASKING toolchains. When you first start EDE, the correct toolchain of the product you purchased is selected and displayed in the title of the EDE desktop window.

If you selected the wrong toolchain or if you want to change toolchains do the following:

1. From the **Project** menu, select **Select Toolchain...**

The Select Toolchain dialog appears.



2. Select the toolchain you want. You can do this by clicking on a toolchain in the **Toolchains** list box and click **OK**.

If no toolchains are present, use the **Browse...** or **Scan Disk...** button to search for a toolchain directory. Use the **Browse...** button if you know the installation directory of another TASKING product. Use the **Scan Disk...** button to search for all TASKING products present on a specific drive. Then return to step 2.

How to Open an Existing Project

Follow these steps to open an existing project:

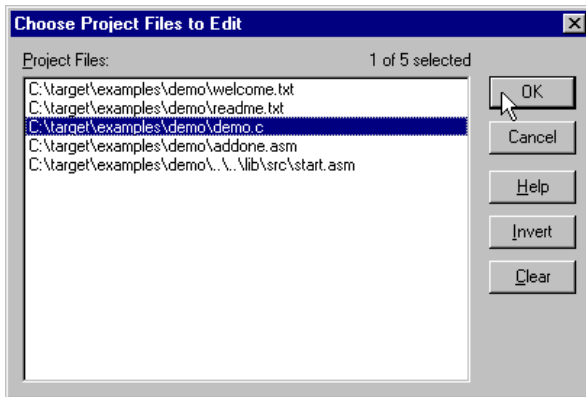
1. From the **Project** menu, select **Set Current ->**.
2. Select the project file to open. For the demo program select the file `demo.pjt`, located in the subdirectory `xvw` in the `examples` subdirectory of the 68K/ColdFire product tree. If you have used the defaults, the file `demo.pjt` is in the directory `c:\...\c68k version\examples\xvw`.

How to Load/Open Files

The next two steps are not needed for the demo program because the file `demo.c` is already open. To load the file you want to look at:

1. From the **Project** menu, select **Load Files...**

The Choose Project Files to Edit dialog appears.



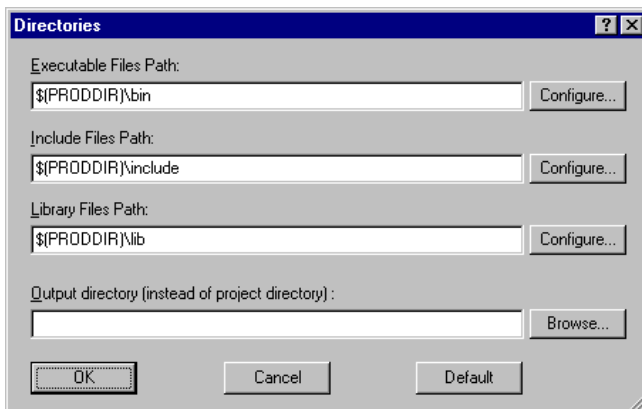
2. Choose the file(s) you want to open by clicking on it. You can select multiple files by pressing the **<Ctrl>** or **<Shift>** key while you click on a file. With the **<Ctrl>** key you can make single selections and with the **<Shift>** key you can select everything from the first selected file to the file you click on. Then click **OK**.

This launches the file(s) so you can edit it (them).

Check the directory paths

1. From the **Project** menu, select **Directories...**

The Directories dialog appears.



2. Check the directory paths for programs, include files and libraries. You can add your own directories here, separated by semicolons.

3. Click **OK**.

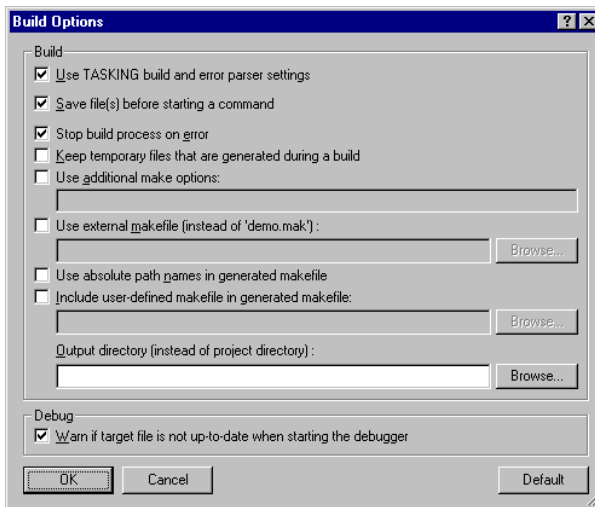
How to Build the Demo Application

The next step is to compile the file(s) together with its dependent files so you can debug the application.

Steps 1 and 2 are optional. Follow these steps if you want to specify additional build options such as to stop the build process on errors and to keep temporary files that are generated during a build.

1. From the **Build** menu, select **Options...**

The Build Options dialog appears.



2. Make your changes and press the **OK** button.
3. From the **Build** menu, select **Scan All Dependencies**.
4. Click on the **Execute 'Make' command** button. The following button is the execute Make button which is located in the toolbar.



If there are any unsaved files, EDE will ask you in a separate dialog if you want to save them before starting the build.

How to View the Results of a Build

Once the files have been processed you can inspect the generated messages in the **Build** tab.

How to Start the CrossView Pro Debugger

Once the files have been compiled, assembled, linked, located and formatted they can be executed by CrossView Pro.

To execute CrossView Pro:

1. Click on the **Debug application** button. The following button is the Debug application button which is located in the toolbar.



CrossView Pro is launched. CrossView Pro will automatically download the compiled file for debugging.

How to Start a New Project

When you first use EDE you need to setup a project space and add a new project:

1. From the **File** menu, select **New Project Space...**

The Create a New Project Space dialog appears.

2. Give your project space a name and then click **OK**.

The Project Properties dialog box appears.

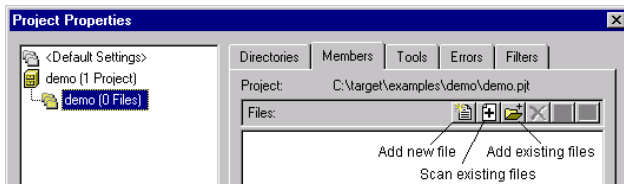
3. Click on the **Add new project to project space** button.

The Add New Project to Project Space dialog appears.

4. Give your project a name and then click **OK**.

The Project Properties dialog box then appears for you to identify the files to be added.

5. Add all the files you want to be part of your project. Then press the OK button. To add files, use one of the 3 methods described below.



- If you do not have any source files yet, click on the **Add new file to project** button in the Project Properties dialog. Enter a new filename and click **OK**.
- To add existing files to a project by specifying a file pattern click on the **Scan existing files into project** button in the Project Properties dialog. Select the directory that contains the files you want to add to your project. Enter one or more file patterns separated by semicolons. The button next to the **Pattern** field contains some predefined patterns. Next click **OK**.
- To add existing files to a project by selecting individual files click on the **Add existing files to project** button in the Project Properties dialog. Select the directory that contains the files you want to add to your project. Add the applicable files by double-clicking on them or by selecting them and pressing the **Open** button.

The new project is now open.

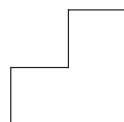
6. From the **Project** menu, select **Load Files...** to open the files you want on your EDE desktop.

EDE automatically creates a makefile for the project. EDE updates the makefile every time you modify your project.

CHAPTER

2

SOFTWARE INSTALLATION



2

CHAPTER

2.1 INTRODUCTION

This chapter describes additional notes for running the CrossView Pro debugger under the X Windows environment on UNIX.

Installation of the TASKING CrossView Pro debugger is part of the installation of the TASKING 68K/ColdFire product, which is described in chapter *Installation Guide* of the *68K/ColdFire Getting Started Manual*.

2.2 NOTE ABOUT FILENAMES

Members of the CrossView Pro family of debuggers use the following name convention for their executables:

xfw68

2.3 CONFIGURING THE X WINDOWS MOTIF ENVIRONMENT

To run the Motif version of CrossView Pro on a Sun, you must define the environment variable **LD_LIBRARY_PATH** to where the library file `libMrm.a` resides. For example:

```
LD_LIBRARY_PATH=/usr/dt/lib  
export LD_LIBRARY_PATH
```

CrossView Pro uses a binary resource file for appearance-related specifications for windows, menus, dialog boxes, and strings to be accessed at run-time. The name of the resource file has the same name as the executable but with `.uid` extension. Be sure that the `.uid` file is present in one of the following directories:

1. the current directory
2. the directory specified by the **UIDPATH** environment variable

The environment variable **UIDPATH** specifies the path used by Motif to locate the resource (`.uid`) file. If not set, it is set to a default value. The resource file is installed in the same directory as the associated executable. So, you should set **UIDPATH** as follows (Bourne shell syntax):

```
UIDPATH=path_to_uid/%U  
export UIDPATH
```

Replace *path_to_uid* by the path to the directory in which the resource file is installed. The %U is required.

For more details refer to `MrmOpenHierarchy` in the *OSF/Motif Programmer's Reference* manual.

2.4 USING X RESOURCES

X toolkit resources specify GUI object (widget) attributes. Resources are specified in either the `.Xdefaults` file or in application class-specific files.

The `.Xdefaults` file is (typically) loaded into the X server at the start of the session. Any changes take effect only in a new session, or after using **xrdb**. Alternatively, application class resource files may be used. Application resource files have the same name as the executable CrossView Pro version they refer to (first letter NOT capitalized). Application resource files must be present either in the directory specified by the **HOME** environment variable, or in the `app-defaults` directory. The `app-defaults` directory is typically located under `/usr/lib/X11`.

X recognizes various environment variables for specifying paths to the application resource files. For more information, consult the chapter on X resources in *O'Reilly's X Toolkit Intrinsic Programming Manual* and your system documentation.

The X resource specification allows either global (loosely) bound specifications (`*foreground: black`) or per-widget instance specifications (`*button.foreground: black`).

The following list shows the relevant widgets used by the Motif version of CrossView Pro:

Windows:

TOP-LEVEL	– XmMainWindow	=> XmDrawingArea
CHILD	– XmScrolledWindow	=> XmDrawingArea

Dialog:

MODAL	– XmBulletinBoard
MODELESS	– XmBulletinBoard

Menu:

MENUBAR	- XmMenuShell
PULLDOWN	- XmCascadeButton

Controls:

CHECKBOX	- XmToggleButton
RADIOBUTTON	- XmToggleButton
TEXT	- XmLabel
EDIT	- XmText
LISTBOX	- XmScrolledWindow => XmList
SCROLLBAR	- XmScrollBar
PUSHBUTTON	- XmPushButton
LISTBUTTON	- XmText & XmArrowButton & XmScrolledWindow => XmList
LISTEDIT	- XmText & XmArrowButton & XmScrolledWindow => XmList
GROUPBOX	- XmFrame => XmLabel
ICON	- XmLable with pixmap
FILESELECTION	- XmFileSelectionBox
ERRORPOPUP	- XmMessageBox

CrossView Pro repaints its windows in the default color as specified with the Motif widget resource settings. It is possible to overrule this behavior with a resource setting like: `*XmDrawingArea.background: blue`.

CrossView Pro uses a non proportional font in all of its windows. The font size is selected using the "Desktop Setup dialog". You can use the "font" resource (`*fontList` on Motif) to select the font to be displayed in the menubar and dialogs, it won't affect the font displayed in the CrossView Pro windows.

The CrossView Pro stack and data windows are implemented using a `XmScrolledWindow` widget on Motif.

The following list show the contents of an example `app-defaults` file intended for Motif environments. Of course you may adjust the colors and font to your preferences. Sample `app-defaults` files are delivered with the product in the `etc` directory (`app_def.mwm` for Motif).

```
*fontList:                7x13bold
*foreground:              black
*XmMainWindow.background: white
```

```

*XmScrolledWindow*background:    white
*XmDrawingArea.background:       white
*XmBulletinBoard.background:     DarkSeaGreen
*XmToggleButton*background:     gray
*XmLabel*background:             gray
*XmText*background:              white
*XmScrollBar*background:         gray
*XmPushButton*background:        gray
*XmFrame*background:             SeaGreen
*XmArrowButton*background:       gray
*XmForm.background:              SeaGreen
*XmMenuShell*background:         DarkSeaGreen
*XmCascadeButton*background:     SeaGreen

```

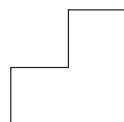
If you encounter any problems due to incorrect resource settings, like invisible text caused by identical text and background color, clear the `RESOURCE_MANAGER`. Use the following procedure to clear the `RESOURCE_MANAGER`:

1. Save a copy of the `.Xdefaults` file located in your home directory.
2. Install an empty `.Xdefaults` file.
3. Execute **xrdb -all .Xdefaults** to actually clear the `RESOURCE_MANAGER` property.
4. Restart CrossView Pro and check if windows and dialogs are displayed correctly.
5. Now you add the saved resources (one by one) back into the `.Xdefaults` file and execute **xrdb** to install them in the server. Restart CrossView Pro and check the influence of the new resource settings. Adapt your saved resources when necessary.

CHAPTER

3

COMMAND LANGUAGE



3

CHAPTER

3.1 INTRODUCTION

The syntax and semantics of CrossView Pro's command language is discussed here. This language is mainly used to enter textual commands in the command edit field of the Command Window. The mouse and menus allow you to access most actions without knowing the command language, although the command language is more powerful. The command language is also used when evaluating expressions and in commands associated with assertions, breakpoints and macros. For information about specific CrossView Pro commands, refer to Chapter 13, *Command Reference*.

3.2 CROSSVIEW PRO EXPRESSIONS

There are several methods that you can use to input an expression into CrossView Pro:

It is possible to display both monitored and unmonitored expressions in the Data Window. Monitored expressions are updated after every halt in execution. Unmonitored expressions are just one-shot inspections of the expressions value. Refer to section 4.6, *CrossView Pro Windows* for a detailed description of the Data Window.

To evaluate a simple expression:



Double click on a variable in the Source window. The result of the expression appears in the data window. Alternatively, depending on the preferences you set in the Data Display Setup dialog, the expression appears in the Evaluate Expression dialog. Click the **Add Watch** or **Add Show** button to display the result of the expression in the Data Window. Click the **Evaluate** button to display the result of the expression in the output field of the Evaluate Expression dialog.

To evaluate a complex expression:



From the **Data** menu, select **Evaluate Expression...** and type in any C expression in the Evaluate Expression dialog box. Optionally select a display format. Click the **Evaluate** button.



Type the expression into the command edit field of the Command Window followed by a return or click the **Execute** button.

Expressions can be any length in most windows and dialog boxes; CrossView Pro provides a horizontal scroll bar if an expression exceeds the visible length of the entry field.

In CrossView Pro, C expressions may consist of a combination of numeric constants, character constants, strings, variables, register names, C operators, function names, function calls, typecasts and some CrossView Pro-specific symbols. Each of these is described in the next sections.

Evaluation Precision

CrossView Pro evaluates expressions using the same data types and associated precision as used by the target architecture when evaluating the same expression.

3.3 CONSTANTS

CrossView Pro, like C, supports integer, floating point and character constants.

Integers

Integers are numbers without decimal points. For example, CrossView Pro will treat the following as integers:

5 9 23

The following number, however, are not treated as integers:

5.1 9.27 0.23

Negative integers, if they appear as the first item on a line, must have parentheses around the number:

(-5) * 4

This is to prevent confusion with CrossView Pro's own - (minus sign) command.

In addition, CrossView Pro supports standard C octal, hexadecimal and binary notation. You can specify a hexadecimal constant using a leading **0x** or a trailing **H** (or **h**). The first character must be a decimal digit, so it may be necessary to prefix a hexadecimal number with the '0' character. The hexadecimal representation for decimal 16 is:

0x10 or 10H

For the hexadecimal digits **a** through **f** you can use either upper or lower case. The following are all correct hexadecimal representations for decimal 43981:

0xabcd 0xABCD 0abCdH 0AbcDh

You can specify a binary constant using a trailing **B** or **Y** (or **b** or **y**). The following are all binary representations for decimal 5:

0101b 101Y 00000101B

You can specify an octal constant using a leading '0'. The octal representation for 8 decimal is:

010

You can use an **L** to indicate a **long integer constant**. For example, CrossView Pro will recognize the following as long integers:

0L 57L 0xffL



CrossView Pro uses the same ANSI C integral type promotion scheme as the C compiler.

Floating Point

A floating point number requires a decimal point and at least one digit before the decimal point. The following are valid examples of floating point numbers:

12.34 5.6 7.89

Exponential notation, such as 1.234e01, is not allowed. The following are *not* valid floating point numbers:

.02 1.234e01 5

As with integers, bracket a negative number with parentheses:

(-54.321)

Expressions combining integers and floating point numbers will evaluate to floating point values:

2.2 * 2
4.4



Character

Character constants are single characters or special constants that follow the C syntax for special characters. Examples of valid character constants include:

```
'm'  'x'  '\n'
```

Character constants must be a single byte and are delimited by ' ' (single quotation marks). For instance:

```
$mychar='m'
```

Remember not to confuse character constants with *strings*. A character constant is a single byte, in this example, the ASCII value of m.

Strings

Strings are delimited by " " (double quotation marks). In C all strings end with a null (zero) character. Strings are referenced by pointer, not by value. This is standard C practice. In CrossView Pro, you may assign a string literal to a variable which is of type char* (pointer to character):

```
$ystring = "name"
```

CrossView Pro supports the standard C character constants shown below:

Code	ASCII	Hex	Function
\b	BS	08	Backspace
\f	FF	0C	Formfeed
\n	NL (LF)	0A	Newline
\r	CR	0D	Carriage return
\t	HT	09	Horizontal tab
\\	\	5C	Back slash
\?	?	3F	Question mark
\'	'	27	Single quote
\"	"	22	Double quote
\ooo			3-digit octal number
\xhhh			hexadecimal number

Table 3-1: C character codes

Trigraph sequences are not supported.

3.4 VARIABLES

CrossView Pro lets you use variables in the C expressions you type. You may reference two classes of variables: variables defined in the source code and *special variables*.

Variables defined in your source code fall into two categories: *local variables* and *global variables*.

Storage Classes

Variables may be of any C storage class. The size of each class is target dependent. Consult the C Compiler/Assembler User's Manual for specific sizes.

You may cast variables from one class to another:

```
(long) $mychar
```

Local Variables

You define local variables within a function; their values are maintained on the stack or in registers. When the program exits the function, you lose local variable values. This means that you can only reference local variables when their function is active on the stack.

Local variables of type `static` retain values between calls. Therefore, you can reference `static` variables beyond their functions, but only if their function is active on the stack.

CrossView Pro knows whether the compiler has allocated a local variable on the stack or directly in a register and whether the register is currently on the stack. The compiler may move some local variables into registers when optimizing code.

If a part of your source code looks like this:

```
x = 5;  
y = x;
```

and you stopped the program after the assignment to `x`, and set `x` to another value, this may not prevent the second statement from setting `y` to 5 due to "constant folding" optimizations performed by the compiler.

Global Variables

Global variables are defined outside every function and are not local to any function. Global (non-static) variables are accessible at any point during program execution, after the system startup code has been executed.

Global variables can be defined `static` in a module. These variables can only be accessed when a function in this module is active on the stack, or when that file is in the Source Window using the **e** command.

Specifying Variables in C expressions

The following table specifies how CrossView Pro treats different variables in C expressions. The left column is the variable's syntax in the expression, the right column is the CrossView Pro semantics.

Variable Syntax	CrossView Pro Behavior
<i>variable</i>	CrossView Pro performs a scope search starting at the current viewing position and proceeding outwards. The debugger first checks locals, local statics and parameters, followed by statics and globals explicitly declared in the current file. Finally, globals in other files are checked.
<i>function#variable</i>	CrossView Pro searches for the first instance of <i>function</i> . If found, the debugger uses the frame's address to perform a scope search for <i>variable</i> . Variables are available only if the specified function is active. That is, the stack frame for that function can be found on the run-time stack.
<i>number#variable</i>	The frame at stack level <i>number</i> is used by the debugger for the scope search. The current function is always at stack level 0. This format is very useful if you are debugging a recursive function and there are multiple instances of a variable on the stack.
<i>:variable</i>	CrossView Pro searches for a global variable named either <i>variable</i> or <i>_variable</i> , in that order.
<i>\$variable</i>	CrossView Pro searches the list of special variables for <i>\$variable</i> .

Table 3-2: Variables in C expressions

Variables and Scoping Rules

A variable is in scope at any point in the program if it is visible to the C source code. For instance, if you have a local variable `initval` declared in `main()`, and then step (or move the viewing position) into `factorial`, `initval` will be out of scope. You can still find the value of `initval` by typing:

```
main#initval
```

In this case CrossView Pro will search the stack for the function `main()`, then look outwards from that function for the first occurrence of `initval` in scope and report its value. Note that `main()` must be active, that is, program execution must have passed through `main()` and not yet returned, in order for `initval` to have a value.

You can also use the **Browse...** button in the Expression Evaluation dialog box. This dialog box appears when you click the **New Expression** button in the toolbar or select **Evaluate Expression...** from the **Data** menu.

Special Variables

CrossView Pro maintains a set of variables that are separate from those defined in your program being debugged. These special variables reside in memory on the host computer, not on the target system. They contain the values of the target processor's registers, information about the debugger's status, and user-defined values. Special variables are case insensitive. Use the **opt** command to display and set these variables (without using the '\$'-sign).

The following is a list of the reserved special variables for CrossView Pro:

Reserved Variable	Description
\$ARG(<i>n</i>)	Contains the value of the <i>n</i> th int-sized argument of the current function. Allows access to arguments of variable argument list functions without knowing the name of the argument.
\$FILE	Contains the name of the file that holds the current viewing position.
\$IN(<i>function</i>)	Contains the value 1 if the current pc is inside the specified <i>function</i> , otherwise 0.
\$LINE	Contains the line number of the current viewing position. This variable is often used in assertions to monitor program flow.

Reserved Variable	Description
\$PROCEDURE	Contains the name of the procedure at the current viewing position.
\$ASMHEX	Contains a string "ON" or "OFF". The value "ON" specifies that the disassembled code as displayed in the assembly window will display hexadecimal opcodes. Default is "OFF".
\$AUTOSRC	Contains a string "ON" or "OFF". The value "ON" specifies that the debugger will automatically switch between the source window and the assembly window display depending on the presence of symbolic debug information at the current location. The value "OFF" prevents the automatic window switching. Default is "OFF".
\$CPU	Contains a string indicating the current CPU type.
\$FP	Contains the value of the frame pointer.
\$MIXEDASM	Contains a string "ON" or "OFF". The value "ON" specifies that the disassembled code as displayed in the assembly window will be intermixed with the corresponding source lines. The value "OFF" suppresses this intermixing. Default is "ON".
\$MORE	Contains a string "ON" or "OFF". The value "ON" specifies that the more output pager is enabled. The value "OFF" disables the more output pager. Default is "ON".
\$PC	Contains the value of the program counter.
\$ <i>register</i>	Contains the value of the specified <i>register</i> .
\$SP	Contains the value of the stack pointer.
\$SYMBOLS	Contains a string "ON" or "OFF" indicating if local symbols and symbolic addresses (e.g. <code>main:56+0x4</code>) or absolute addresses are present in disassembly. Default is "ON".
\$SRCLINENRS	Contains a string "ON" or "OFF". The value "ON" specifies that line numbers should be printed in the source window. The value "OFF" suppresses printing of line numbers. Default is "OFF".
\$SRCMERGE LIMIT	Contains the value for the source merge limit in the assembly window, the number of source lines to be intermixed in the assembly window. Value 0 indicates that there is no limit. Default is 0.

Table 3-3: Reserved special variables

Registers

You can reference registers and special function registers (SFRs) directly. The format is *\$register*. For instance, type:

\$D0 = 0x12345678 *Set value of register D0 to 0x12345678*

\$SR *Inspect value of status register*

For CrossView Pro, a fixed set of registers is always available. Additional SFRs can be added by using the utility **rm68**. See appendix B, *Register Manager*, for more information about **rm68**.

You can configure which (and in which order) registers must appear in the register window in the Register Window Setup dialog (**Settings | Register Window Setup...**).

It is possible to request the address of an SFR by using the address operator **&**.

&\$sp
Location of \$SP is reg [SP]
Operand for '**&**' incorrect

&\$my_sfr
0x578218

In addition to the standard register special variables, CrossView Pro supplies the special variables: **\$sp** (the stack pointer), **\$pc** (the program counter) and **\$fp** (the current frame pointer).

The values of Reserved special variables cannot be changed interactively (i.e., on the CrossView Pro command line).

User-defined Special Variables

During a debugging session, you may need some new variables for your own debugging purposes, such as counting the number of times you encounter a breakpoint. CrossView Pro allows you to create and use your own special variables for this purpose. CrossView Pro does not allocate space for these variables in target memory; it maintains them on the host computer.

The names of these variables, which must begin with a **\$** (dollar sign), are defined when they are first used. For instance:

\$count = 5

defines a variable named `$count` of type `int` with a value of 5. Special variables are of the same type as the last expression they were assigned. For example:

```
$name="john"
```

then:

```
$name=3*4
```

creates a special variable `$name` of type `(char *)`. The second statement creates a special symbol `$name` and assigns it the value of 12 of type `int`.

Special variables are just like any other variables, except you cannot meaningfully take the address of them. CrossView Pro allows as a default 26 user-defined special variables. You can change this limit with the **-s** option at startup, or by selecting the **Options...** menu item from the **File** menu and choosing the **Initialization** tab.



See the startup options in Chapter 4, *Using CrossView Pro*.

3.5 FORMATTING EXPRESSIONS

By default, CrossView Pro displays the value of an expression using the appropriate format for the type of expression. CrossView Pro follows several simple rules for displaying variables:

- The defaults are: addresses appear in hexadecimal format, characters as ASCII and integers as decimal.
- There are four possible formats to show one integer value: **decimal**, **hexadecimal**, **octal**, and **ASCII**.
- There are two different formats to display one floating point value: **decimal real** and **hexadecimal**. If the absolute value is either too big or too small (with too many non-significant zeroes), the debugger automatically converts the format to one with fixed decimal point and exponent.
- **ASCII** is the only format to display a string. Note that you can opt for the array format. Unpredictable characters are output as `\xbb`, where **bb** is a hexadecimal value. Control characters are output as `^C`.
- All the values in an array appear in the same format. You are free to select this format from the available options.
- If All the values of a structure appear in the same format. You are free to select this format from the available options.



You can determine in which format a variable is displayed. Once the format has been selected, however, you must enter values or change values in the appropriate format. When editing is finished, the debugger interprets all values in terms of the currently selected formats.

You may, however, tell CrossView Pro to display an expression in a particular format other than the default format. The format code follows the variable, in one of two ways:



The simplest method of specifying display formats is from the Evaluate Expression dialog box. To access this dialog box:

- From the **Data** menu, select **Evaluate Expression...**



In the Command Window, you can use several **format codes** shown in the next table to specify the variable display. The format codes can be entered as:

variable/format



to display the variable in format *format*, or:

```
variable@format
```

to display the variable’s address in format *format*.

The structure of the formatting code is:

```
[count] style [size]
```

Count is the number of times to apply the format style *style*. *Size* indicates the number of bytes to be formatted. Both *count* and *size* must be numbers, although you may use **c** (char), **s** (short), **i** (int), and **l** (long) as shorthand for *size*. Legal integer format sizes are 1, 2, and 4; legal float format sizes are 4 and 8.



Be sure not to confuse CrossView Pro format codes with C character codes, e.g. \a. CrossView Pro uses a forward slash / not a backward slash \.

Style	Description
a	Print the specified number of characters of the character array; any positive <i>size</i> is OK. Use the expression’s value as the address of the first byte.
c	Print a character; any positive <i>size</i> is OK; default <i>size</i> is sizeof(char).
D	Print in decimal; needs NO <i>size</i> specifier; <i>size</i> is sizeof(long).
d	Print in decimal; can have a <i>size</i> specifier; default <i>size</i> is sizeof(expression).
E	Print in “e” floating point notation; needs NO <i>size</i> specifier; default <i>size</i> is sizeof(double).
e	Print in “e” floating point notation; the <i>size</i> specifier can be sizeof(float) or sizeof(double); default <i>size</i> is sizeof(expression).
F	Print in “f” floating point notation; needs NO <i>size</i> specifier; default <i>size</i> is sizeof(double).
f	Print in “f” floating point notation; the <i>size</i> specifier can be sizeof(float) or sizeof(double); default <i>size</i> is sizeof(expression).
G	Print in “g” floating point notation; needs NO <i>size</i> specifier; default <i>size</i> is sizeof(double).

Style	Description
g	Print in “g” floating point notation; the <i>size</i> specifier can be sizeof(float) or sizeof(double); default <i>size</i> is sizeof(expression).
l	Print the function, source line, and disassembled instruction at the address.
i	Print the disassembled instruction at address.
n	Print in the “natural” format, based on type; use it for printing variables that have the same name as an CrossView Pro command.
O	Print in octal; needs NO <i>size</i> specifier; <i>size</i> is sizeof(long).
o	Print in octal; can have a <i>size</i> specifier; default <i>size</i> is sizeof(expression).
P	Print the name of the function at the address.
p	Print the names of the file, function, and source line at the address.
s	Print the specified number of characters of the string, using the expression’s value as the address of a pointer to the first byte. Equivalent to <i>*expression/a</i> . If no <i>size</i> is specified the entire string, pointed to by expression, is printed (till nil-character).
t	Display the type of the indicated variable or function.
U	Print in unsigned decimal; needs NO <i>size</i> specifier; <i>size</i> is sizeof(long).
u	Print in unsigned decimal; can have a <i>size</i> specifier; default <i>size</i> is sizeof(expression).
X	Print in hexadecimal; needs NO <i>size</i> specifier; <i>size</i> is sizeof(long).
x	Print in hexadecimal; can have a <i>size</i> specifier; default <i>size</i> is sizeof(expression).

Table 3-4: Format style codes

For example, typing:

```
initval/4xs
```

displays four, hexadecimal two-byte memory locations starting at the address of `initval`.

The following piece of C-code can be accessed in CrossView Pro using the string format codes:

```
char  text[] = "Sample\n";
char *ptext = text;
```

text *What is the address of this char array*
 text = 0x8200

text/a *Print it as a string*
 text = "Sample^J"

ptext *What is the contents of this pointer*
 string = 0x8200

ptext/s *Print it as a string*
 string = "Sample^J"

&ptext *Where does ptext itself reside*
 0x8210

With format codes, you may view the contents of memory addresses on the screen. For instance, to dump the contents of an absolute memory address range, you must think of the address being a pointer. To show (dump) the memory contents you use the C language indirection operator `*`. Example:

***0x4000/2x4**
 0x4000 = 0x00DB0208 0x5A055498

This command displays in hexadecimal two long words at memory location 0x4000 and beyond. Instead of using the size specifier in the display format, you can force the address to be a pointer to unsigned long by casting the value:

***(unsigned long *)0x4000/2x**
 0x4000 = 0x00DB0208 0x5A055498

To view the first four elements of the array `table` from the `demo.c` program, type:

table/4d4
 table = 1 1 2 6

This command displays in decimal the first four 4-byte values beginning at the address of the array `table`.

3.6 OPERATORS

Standard C Operators

CrossView Pro supports the standard C operators in the ANSI defined order of precedence. The order of precedence determines which operators execute first.

The semicolon character (;) separates commands on the same line. In this way, you may type multiple commands on a single line. Comments delimited by /* and */ are allowed; CrossView Pro simply ignores them.

Order of Precedence (in descending order)
() [] -> .
! ~ ++ -- + - * & (type) sizeof
* // %
+ -
<< >>
< <= > >=
== !=
&
^
&&
? : = += -= *= /= %= &= ^= = <<= >>=

Table 3-5: Order of precedence of standard C operators

The *, - and + operators appear twice since they exist as both unary and binary operators and unary operators have higher precedence than binary.



Division is represented by // (two slashes) not / (one slash). This is to avoid confusion with CrossView Pro's format specifier syntax.

Using Addresses

To specify an address, you may use the `&` operator. To determine the address of `initval`, type:

```
&initval
```

If you try to use the `&` operator on a local variable in a register, CrossView Pro issues an error message and tells you which register holds the variable.

3.7 SPECIAL EXPRESSIONS

String Commands

Whenever CrossView Pro encounters an expression consisting solely of a string by itself, it simply echoes the string. For example:

```
"hello, world\n"
hello, world
```

Use this technique to place helpful debugging messages on breakpoints. For example, setting the following breakpoint:

```
60 b {"now in for loop\n"; sum; C }
```

this cause CrossView Pro to echo the message `now in for loop`, to display the value of `sum` in the Command Window, and to continue when line 60 is encountered. You can also enter this breakpoint and the associated commands via the Breakpoints dialog box, which you can open by selecting the **Breakpoints...** menu item from the **Breakpoints** menu.

The Period Operand

As a shorthand, CrossView Pro supports a special operand, period `.`, that stands for the value of the last expression CrossView Pro calculated. For instance, in the following example, the period in the second command equals the value 11, which is the result of the previous expression:

```
5 + 6
11
4 * .
44
```

The period operand assumes the same size and format implied by the specifier used to view the previous item. Thus if you look at a long as a char, a subsequent '.' is considered to be one byte. Use this technique to alter specified pieces of a larger data item, such as the second highest byte of a long, without altering the rest of the long. The period operand may be used in any context valid for other variables.



'.' is the *name* of a location. When you use it, it is dereferenced like any other name. If you want the address of something that is 30 bytes farther on in memory, do not type `+.30` as this takes the contents of dot and adds 30 to it. Type instead `&+.30` which adds 30 to the address of the period operand.

3.8 CONDITIONAL EVALUATION

CrossView Pro supports the `if` construct. Use this construct in breakpoints and assertions to alter program flow conditionally. For example, if you reset the following breakpoint:

```
60 b {if (sum<=5931){C}{sum}}
```

CrossView Pro compares the value of `sum` with 5931 when the program stops at line 60. If `sum` is less than or equal to 5931, CrossView Pro continues. Otherwise, CrossView Pro displays the value of `sum` with 5931 when the program stops at line 60.

You can also use the `exp1 ? exp2 : exp3` C ternary operator for conditional expressions. For example:

```
$myvar = (5 > 2) ? 1 : -1
```

assigns the value 1 to `myvar`.

3.9 FUNCTIONS

In CrossView Pro expressions, you can include functions defined in the program's code.



Command line function calls are only allowed in an application that uses a reentrant stack frame. When a static stack frame is used, the locator overlays stack frames based on the application's call graph. A command line function call may conflict with the calling sequence in the call graph and corrupt the stack when executed.



You can call functions through the Call a Function dialog box. Note that only the results of the function call are shown. You cannot enter expressions in this field. If you want to use the results of the function call in an expression, then type the expression into the Evaluate Expression dialog box or type in the command into the Command Window (described in the keyboard method below).

- From the **Run** menu, select **Call a Function...**
- List all functions by clicking the **Browse...** button.
- You can place parameters in the **Parameters** field of the Call a Function dialog box, separated by commas, but without the usual parentheses or select from the drop-down history list.

The Command Window receives the results of the function call.



Type in the expression containing a function call directly into the Command Window.

To execute a function on the target type the function name and the arguments as you would do in your C program. For example,

```
do_sub(2, 1)    or:    a = do_add(3,4)
```

3.10 CASE SENSITIVITY

The absolute file supplies the case sensitivity information for variable names. It is initially case *sensitive* for the C language. You may toggle case sensitivity by:



From the **Edit** menu, select **Search String...** to view the Search String dialog box. This dialog contains the **Case Sensitive** check box.



Typing the (capital) **Z** command in the Command Window.

CHAPTER

4

USING CROSSVIEW PRO



4

CHAPTER

4.1 INTRODUCTION

This chapter and the following 8 chapters give you a comprehensive picture of CrossView Pro's features. In order to address the broadest range of expertise, the contents range from introductory examples to the more technical aspects and techniques of debugging with CrossView Pro. While it is not necessary for you to read the chapters straight through, you may find it especially helpful to do so. All of the examples are from the sample program `demo.c` which comes with CrossView Pro. For a complete description of the commands presented in this chapter, consult the *Command Reference* chapter.

Each CrossView Pro command introduced in the text has a matching box summarizing its syntax and semantics. The command description follows these general rules:

Items in **bold** font are the actual CrossView Pro commands: **save**, **set**. Items in *italics* are names for the things you should type: *filename*, *commands*. In addition, the | symbol means *or*. For instance, **screen** | *filename* means you can use the word "screen" or a filename in the syntax.

4.2 USING THE CROSSVIEW PRO INTERFACE

This manual uses the word "Windows" to generically refer to the host computer system's windowing system. On IBM-PCs and compatibles, this is equivalent to Microsoft Windows (95/98/XP, NT or 2000). On UNIX workstations, this refers to the X Window System. Generally, this manual makes no distinctions between the various windowing systems unless needed to clarify the discussion.

This manual assumes you possess a basic familiarity with Windows software. For this reason, discussion focuses on how CrossView Pro works, rather than how to use the Window interface. For more information on your Windows system, consult the Windows documentation provided with your host system.

You can execute most CrossView Pro commands using either mouse or textual commands. Mouse commands are executed by means of buttons and pull-down menus in each of the separate CrossView Pro windows. Text commands are typed at the prompt in the Command Window. In most cases, there is no difference in functionality between mouse and text equivalents.

This manual discusses both methods of performing CrossView Pro functions. For a quick-reference guide to all CrossView Pro commands, refer to the *Command Reference* chapter.

4.3 STARTING CROSSVIEW PRO

Once an absolute file has been made it can be executed by CrossView Pro. There are several ways to invoke CrossView Pro.

From EDE

To start CrossView Pro from EDE (the Embedded Development Environment), click on the **Debug application** button. The following button is the Debug application button which is located in the toolbar.



From the desktop

With MS-Windows you can start CrossView Pro through the **Start** menu. Or in the Windows Explorer you can double-click on an absolute file if the .abs extension is associated with the CrossView Pro executable.



On the PC, CrossView Pro is a Microsoft Windows application. As such, you must invoke it from the Windows environment.

From the command line

To begin the debugging session, type the name of the CrossView Pro debugger and optionally the name of the target program (absolute file).

```
xfw68 [absolute-file] [option]...
```

4.4 STARTUP OPTIONS

CrossView Pro allows you to specify several options when you invoke the program. Type these startup options (or switches as they are sometimes called) after the optional basename of the application. The basename can also contain a path specification. In this case, CrossView Pro sets its current directory to the specified path. A minus sign proceeds each option; the options can appear in any order.

Note that some versions of CrossView Pro have different startup options and procedures than the ones described here. Please consult the Addendum (at the end of this manual), for precise information about starting up CrossView Pro with your target hardware.

From EDE

You can select the execution environment, setup communication parameters, specify record and playback files and set some maximum values via the **CrossView Pro** entry of the **Project | Project Options...** dialog.

From CrossView Pro

You can set many of CrossView Pro's options by using the dialog boxes called by the **Target | Settings...** and **File | Options...** menu items. You can save the options in the `xvw.ini` file and they are automatically used upon startup.

In Windows 95/98/XP, Windows NT 4.0 or Windows 2000 (or higher), add startup options to the program's property sheet:

- Right-click on the CrossView Pro shortcut icon, shown in your program installation folder.
- Select **Properties**. The Program Item Properties dialog box appears.
- Enter the startup options after the executable's name in the Target field of the shortcut.



Use menus to set options. After setting the options in the menus and selecting the appropriate options in the Save Options dialog on exit, CrossView Pro saves the settings in the file `xvw.ini` for future debug sessions.

To start up CrossView Pro type:

xfw68

When your execution environment itself has a human-oriented ASCII interface, you can use transparency mode with the **-T** option. In transparency mode you can configure the execution environment's memory. Check the Addendum, the hardware-specific section of this manual. In-circuit emulators generally require you to map the address space, allocating memory ranges to the execution environment and/or the target system. Fortunately, this generally does not mean you need to learn your emulator's command set, just a rote sequence of startup commands. When your CrossView Pro version does not support transparency mode, you do not need to configure the memory, and the **-T** option is not needed.

If your target system supports serial communication and if the target system is connected to a port other than the default port (see Chapter 1, *Overview*, to determine the default port for your host), you can use the **-D** option to specify the port name. The default baud rate is 9600. You may use the **-D** option to specify the baud rate if the execution environment is not the same as the default. For example:

```
xfw68 -D rs232,com2,19200
```

instructs CrossView Pro to use the COM2 port at 19200 baud. See your execution environment in the Addendum of this manual for specific communication information.

When you specify a startup option in CrossView Pro, the option overrules the corresponding value in the current `xvw.ini` file.

There are many different options you can invoke when starting up CrossView Pro. The listing below gives an overview of all startup options.

There are several startup options having to do with the recording and playing back of CrossView Pro command files. See also Chapter 9, *Command Recording & Playback*.

Startup Option	Description
-a <i>number</i>	Sets the maximum number of assertions (the default is 100).
-b <i>number</i>	Sets the maximum number of code breakpoints (the default is 200).
-c <i>number</i>	Sets the maximum number of instruction trace for the trace buffer (the default is 32).
-C <i>cpu</i>	Forces CPU type selection. This option also determines which register file (<i>regcpu.dat</i>) will be used. The default is 68000.
-D <i>device_type,opt1[,opt2]</i>	Selects a device and specifies device specific options, such as communication port and baud rate. The allowed combinations for your execution environment are described in the manual addendum for that specific execution environment. The following combinations are possible:
-D rs232,port,speed	Select RS-232 communication. <i>port</i> For PC this is COM1, COM2, COM3 or COM4. A colon should not be added. For UNIX this is the full path of the RS-232 device driver (e.g., <i>/dev/tty01</i>). By default CrossView Pro uses the first RS-232 port. <i>speed</i> This is the baud rate used for the specified <i>port</i> . The default is 9600.
-D parallel,port	Select parallel communication. <i>port</i> For PC this is LPT1 or LPT2. Do not add a colon. For UNIX this is the full path of the parallel device driver. By default CrossView Pro uses the first parallel port.
-D tcp,host,port	Select TCP/IP communication. On UNIX the standard TCP/IP implementation is used. On MS-Windows the <i>WINSOCK.DLL</i> implementation is used. <i>host</i> The name of the host to be accessed via TCP/IP. <i>port</i> The port number on <i>host</i> to be accessed.

Startup Option	Description
-D dev,device-file	Use a UNIX device driver as communication channel. For RS-232 devices use the -D rs232 option, described above. <i>device-file</i> The full path of the UNIX device file.
-D isa,io-port,address	Select communication channel to an (E)ISA interface card in the PC. <i>io-port</i> PC I/O port number or I/O channel used for accessing the (E)ISA card. <i>address</i> The memory address used to access the (E)ISA card.
-f file	Read command line options from <i>file</i> .
--fss_root_dir="path"	Specify root directory for File System Simulation.
-G path	Specify startup directory for CrossView Pro.
-i	Has CrossView Pro download the image of the absolute object file.
-L file	Keeps a log of CrossView-to-target communications in a <i>file</i> . Not available for all execution environments.
-n address	Informs CrossView Pro that the program was loaded into memory at an <i>address</i> other than zero.
--orti=file	Specify the name of an OSEK/ORTI file for RTOS aware debugging.
-p file	Starts playing back commands from <i>file</i> .
-P file	Starts playing back commands from <i>file</i> with commands single step.
-r file	Starts recording commands in <i>file</i> .
-R file	Starts recording screen output in <i>file</i> .
--radm=file	Same as the <code>radm</code> field in the target configuration file: specify the name of the Debug Instrument (using KDI) used for RTOS aware debugging.
-s number	Sets the maximum <i>number</i> of special variables (variables independent of the program that CrossView Pro provides for your use). The default is 26.

Startup Option	Description
-sd <i>directory</i> [<i>;directory</i>]...	Specifies the directories CrossView Pro should search for source files. Relative paths are allowed. When the x command is used to load a new symbol file, the current directory is set to the directory containing the symbol file and CrossView Pro now searches for source files relative to this directory. Directories must be separated by semicolons.
-tcfg <i>file</i>	Specify a target configuration <i>file</i> . This overrides the filename specified in <i>xvw.ini</i> . See section <i>CrossView Pro Target Settings</i> in the <i>Overview</i> chapter.
--timeout = <i>n_seconds</i>	Start CrossView Pro command line batch operation mode and terminate after <i>n_seconds</i> .
-T [<i>file</i>]	Starts CrossView in transparency mode if present; if <i>file</i> is given, commands in <i>file</i> are sent to the execution environment.

Table 4-1: CrossView Pro Startup Options

4.4.1 WHAT YOU MAY HAVE DONE WRONG

Most problems in starting up CrossView Pro for a debugging session stem from improperly setting up the execution environment or from an improper connection between the host computer and the execution environment. Some execution environments require you to enter transparency mode to set the execution environment for a debugging session. Check the notes for your particular execution environment and the Addendum of this manual.

Here are some other common problems:

- Specifying the wrong device name when invoking the debugger.
- Specifying a baud rate different from the one the execution environment is configured to expect.
- Not supplying power to the execution environment or an attached probe.
- Using the wrong kind of communication cable.
- Plugging the cable into an incorrect port. Some target machines have several ports.

- Installation of a device driver or resident applications that use the same communications port on the host system.
- The port is already in use by another user or login process on some UNIX hosts.
- Specifying no or an invalid cpu type with the **-C** option.

4.5 THE CROSSVIEW PRO DESKTOP

The CrossView Pro desktop is the screen background in which all windows, icons and dialog boxes appear (see figure 4-1). Under some windowing systems, the desktop is itself a window that does not contain all other CrossView Pro windows.

The desktop always has the Command Window opened or iconized.

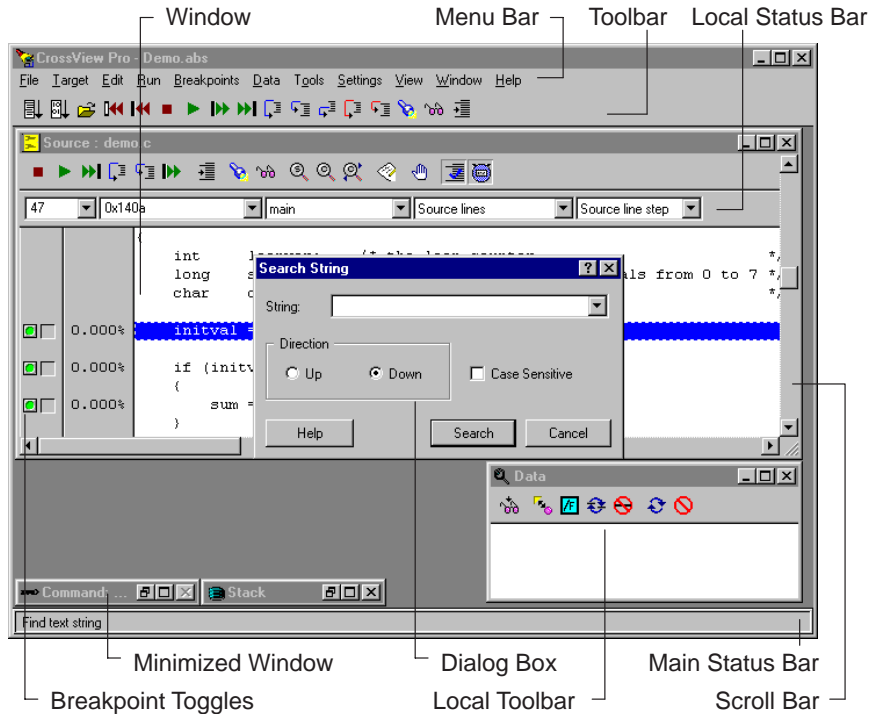


Figure 4-1: CrossView Pro Desktop

At the top of the desktop is the **Menu Bar**, which contains the menus applicable to the currently active window. Below the menu bar is the main **Toolbar**, from which you can execute commands to control program execution as button functions. Except for the Command Window, the desktop can contain other windows as well.

Along the bottom of the desktop there is a **Main Status Bar**. The status bar displays messages such as short “help messages” when you move the cursor over any button in any CrossView Pro window.

Menus

Each CrossView Pro window may have a menu associated with it. Under Microsoft Windows, the active window's menu is displayed in the menu bar of the desktop.

Depending on your execution environment some menu items are always grayed out. For example, **Communication Setup** is grayed out if your target is an instruction set simulator.

Windows

The debugger supports two types of windows: *primary windows* and *dialog boxes*. Dialog boxes are the windows you access from a primary window. For the remainder of this manual, the term “window” denotes a primary window.

This manual also uses the term *pop-up window*. A pop-up window is a primary window that contains supplemental information such as on-line help.

CrossView Pro Windows are used to display information and to get user input through either buttons, commands typed in input fields, or menu selections. Windows may be moved around the desktop, sized, or iconized. All windows can be opened from the **View** menu. The section on *CrossView Pro Windows* provides more detail about each window.



A window is considered opened even if it is iconized (under Microsoft Windows, this is called minimized). A window is considered closed if it does not exist on the desktop in any form.

Dialog Boxes

Certain menu items or push buttons may call up a dialog box to complete an action, display information, or get additional data. No other actions can be performed until the dialog box is closed.

4.5.1 MENUS

Each window in CrossView Pro uses the menu as shown in figure 4-2. The method of selection of a menu item varies depending on the windowing system being used. See your Windowing System's manual for details of how to do this.

Each window has a hidden control menu (the icon on the top-left of the window), to manipulate the window. The menu **C**lose command in the control menu closes the current window. Your implementation of the windowing system may have additional features. See your documentation for further details.

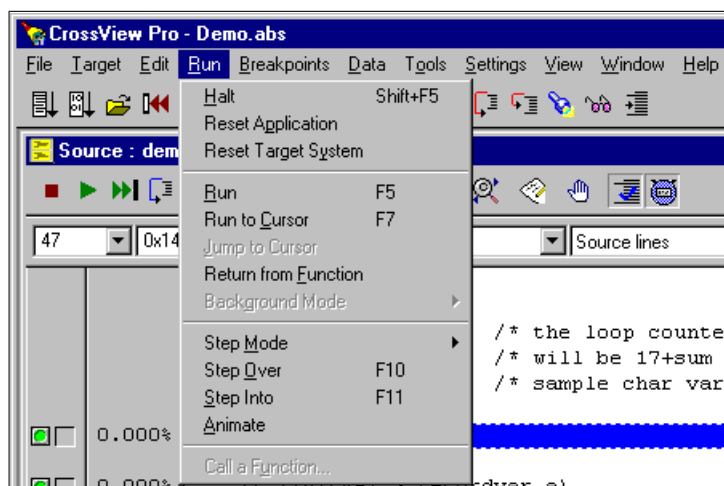


Figure 4-2: CrossView Pro Menus

4.5.1.1 LOCAL POPUP MENUS

On MS-Windows environments CrossView Pro supports local popup menus. Local popup menus are invoked by clicking the right mouse button. The menu contents is context sensitive. If the mouse pointer is on top of the global (main) toolbar the Configure Toolbar dialog is shown. If the mouse pointer is located in the MDI window (task window or background) the View Menu is shown which allows you to open new windows.

Within the Source Window four different local popup menus may appear. If the cursor is within the display area of the window the Run Menu is shown. The Run Menu contains commands associated with program execution. If your cursor is at a breakpoint indicator, the Breakpoints dialog is shown. If the cursor is on a code coverage marker then the local popup menu contains commands to move the cursor to the next or previous block of (not)covered statements. If your cursor is in the profile column you can change the format of the timing figures. All other windows have their own local popup menu. The exception to the rule is the command window which does not have a local popup. See figure 4-3 for an example of the local popup menu of the Memory Window.

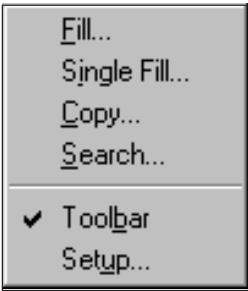


Figure 4-3: CrossView Pro Local Popup Menu (Memory Window)

4.5.2 WINDOW OPERATION

Windows can be opened, made active, and closed.

Opening Windows

The **View** menu of the menu bar lists all windows. Selecting a window name from this list causes the window to open up. Selecting a window that is already open brings that window to the front.

Selecting a Window

At any one time, a particular window is *active*. Most operations act (by default) on the active window. The active window is distinguished by highlighting the title bar. Only one window may be active at a time. There are several ways to select a window (that is, make a window active).

- Open the window from the **View** menu. If the window is already open it will be brought to the front.
- Click on the window's border (or on any portion of the window in some windowing systems). It will be brought to the front.
- Select the window name from the **Window** menu. The window will be made active and is brought to the front. (This option is available under Microsoft Windows only).

Closing a Window

Windows are closed by selecting **Close** from the **Control** menu, or by clicking a **Close** button, as shown in figure 4-4. Selecting this item from the Command Window will exit CrossView Pro.

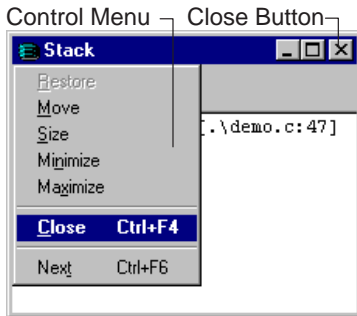


Figure 4-4: Closing a Window

4.5.3 DIALOG BOXES

The debugger uses dialog boxes to acquire information needed to complete a requested operation. The debugger also uses dialog boxes to display information. If a button or menu item displays an ellipsis (...) after its name, then there is an associated dialog box.

For example, the dialog box shown in figure 4-5 searches for a string. This dialog box uses a list edit field to enter a search string, radio buttons to select the search direction, a check box to specify case sensitivity and push buttons to allow certain functions to be performed.

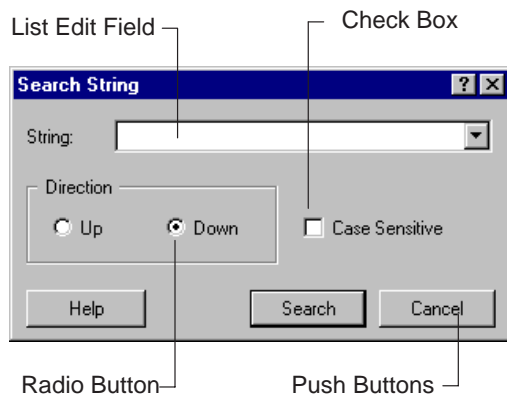


Figure 4-5: Dialog Box

4.5.4 CUSTOMIZING CROSSVIEW PRO

You can customize CrossView Pro's visual appearance and operative parameters to best suit your debugging environment.

Changing the Visual Appearance

Windows can be organized by resizing and moving them around the desktop (see your Windowing System's manual for details on how to do this). All windows under Microsoft Windows have an additional **Window** menu item. This menu allows the user to arrange all opened windows in a tiled or cascaded format. In the tiled format, selected by **Window | Tile**, all windows become the same size. All windows are the visible, the same size and do not overlap. In the cascaded format, selected by **Window | Cascade**, all open windows are changed to the same size and overlapped in a cascade with a constant vertical and horizontal offset. Iconized (minimized) windows can be automatically rearranged by selecting **Arrange Icons** from the **Window** menu.

See the section *Using X Resources* in the chapter *Software Installation* for details on changing the visual appearance of CrossView Pro under X Windows.

Changing Operative Parameters

You can adjust the operative parameters for CrossView Pro using the various menus in CrossView Pro.

In the **Target** menu you will find:

- **Settings:** Allows you to specify the execution environment and the CPU type, and the source directories for CrossView Pro. The values are processed at CrossView Pro startup before executing commands entered in the Command Window or before the target is accessed as a result of opening a window. So, first edit this dialog when you start CrossView Pro. If you have not loaded a symbol file yet, you do not have to restart CrossView Pro.
- **Communication Setup:** Allows you to set parameters for communication between CrossView Pro and your target board.

In the **File | Options...** dialog you will find:

- **Initialization:** Allows you to specify the maximum number of breakpoints, assertions, special variables, C-trace instructions, command history lines, command output lines, emulator output lines. All values are processed at CrossView Pro startup, except for C-trace. Changing the maximum number of C-trace instructions has an immediate effect on the Trace window.
- **Desktop:** Allows you to specify color settings for the execution position in the Source Window and the colors used in the Memory Window to show how a memory location has been accessed by the application program. You can also specify font sizes to be used in output windows.
- **Toolbar:** Allows you to configure the main toolbar to your personal preferences.

In the **Tools** menu you will find:

- **Record, Playback, and Log:** Allow you to set command recording and playback options.
- **Toolbox Setup, and Macro Definitions:** Allow you to define macros, and assign them to a push button in the Toolbox.

In the **Data** menu you will find:

- **Data Display Setup:** Allows you to specify how CrossView Pro displays data. This dialog also determines if the Expression Evaluation dialog box must be bypassed or not.

In the **Settings** menu you will find:

- **Source Window Setup:** Allows you to specify the step mode, symbolic disassembly, automatically switching between source lines and disassembly source to be displayed in the Source Window and display code coverage information.
- **Register Window Setup:** Allows you to specify the registers that appear in the Register Window. And you can set the display format to hexadecimal or decimal.
- **Memory Window Setup:** Allows you to specify the mode and size of the data and the number of data rows and columns to be shown in the Memory Window. It also allows you to automatically refresh the Memory Window and to display data coverage information.
- **Data Analysis Window Setup:** Allows you to configure the graph display of a Data Analysis Window.

- **I/O Simulation Setup:** Allows you to specify the I/O streams to be used in the Terminal Windows.
- **Terminal Window Setup:** Allows you to specify the input and output format of a Terminal Window. You can map linefeeds to carriage-return linefeeds, wrap at the end of a line, specify buffered input or specify that the window must be cleared at system reset and program reset. You can also log the input and output data to a file.
- **Background Mode Setup:** Allows you to specify which windows to automatically refresh when running in background mode. This feature is only available if it is supported by your execution environment.

Saving Changes on Exit

If you find yourself using a particular configuration, you may want to save your configuration when you exit CrossView Pro:

- From the **File** menu, select **Exit** or close the Command Window.
- In the **Save** tab of the Options dialog that appears, select the options you want to be saved for another debug session.
- Click on the **Exit** button in the Options dialog.

CrossView Pro exits. If you selected one or more items in the **Save** tab of the Options dialog your settings are saved in the initialization file `xvw.ini`. This file is in the startup directory.

4.5.5 CROSSVIEW PRO MESSAGES

CrossView Pro communicates with you in a variety of ways. The command window displays the results of commands. Important messages, such as errors, appear in dialog boxes that pop up.

4.6 CROSSVIEW PRO WINDOWS

The two prominent windows used in CrossView Pro are the Command Window and the Source Window. From the Command Window you can type CrossView Pro and emulator commands, and gain access to all other windows. You can accomplish most global operations from either the menu bar or the Command Window. Only from the Command Window can you accomplish Single step playback. When you close the Command Window, you exit CrossView Pro.

The Source Window focuses on the program being debugged. This window controls most of the commonly-used execution operations, such as breakpoints and searching functions.

Available Windows

You can open all CrossView Pro windows (except for the Data Analysis windows) from the **View** menu by selecting the name of the window. Selecting a window in this case brings the window to front and makes it the active window. Available windows are:

- **Command Window:** Supports two modes: CrossView or Emulator. Displays all CrossView Pro commands and responses or Emulator commands and responses.
- **Source Window:** Controls the execution of the program and displays the source file or disassembly.
- **Register Window:** Displays the current state of the processor's registers.
- **Memory Window:** Displays target memory and allows you to change it.
- **Data Window:** Displays the values of data that are being monitored.
- **Data Analysis Window:** Graphically displays signal data for analysis.
- **Stack Window:** Displays the application's stack trace.
- **Trace Window:** Displays the most recently executed lines.
- **Terminal Windows:** Can be used for I/O simulation of an application.

Improving CrossView Pro Performance

CrossView Pro updates every window that is open (except for the Data Analysis windows), even if it is iconized (minimized). Keeping a window up to date usually involves extra communication with the emulator, slowing CrossView Pro down. For instance, if the Register Window is open, CrossView Pro asks the emulator to dump the contents of all displayed registers after each single step. Thus it is a good idea to keep only those windows open that you need.

4.6.1 COMMAND WINDOW

The Command Window allows you to:

- Enter CrossView Pro and emulator commands from the keyboard.
- View a history of CrossView Pro commands or emulator commands.
- View the result of CrossView Pro commands or emulator commands.
- Execute playback files (in single step mode).

From the **View** menu you can specify if you want the Command Window to be a CrossView Pro Command Window or an Emulator Command Window. This way you can specify whether CrossView Pro interprets commands or they go directly to the emulator.

Figure 4-6. shows the Command Window. You can type commands into the command edit field (bottom field) or select them from the command history list (middle field), edit and execute them. The command history field displays previously entered commands. You can select and execute one or more commands. The command history list provides you with a clear, comfortable way to re-execute specific commands or sequences of commands by preserving them in a scrollable list.

You can switch between the history list and the command edit field by hitting the **<Tab>** key. Hitting the **<Esc>** key (escape) returns you to an empty edit field.

The top field is the Command Output Window or the Emulator Output Window, depending on the type of Command Window you choose. Each command, echoed from the command edit field, appears with a '**>**' prefix. CrossView Pro displays its response (or the emulator's response if the window is an Emulator Command Window) to the command immediately following the command. You can use the **clear** command to clear this window.

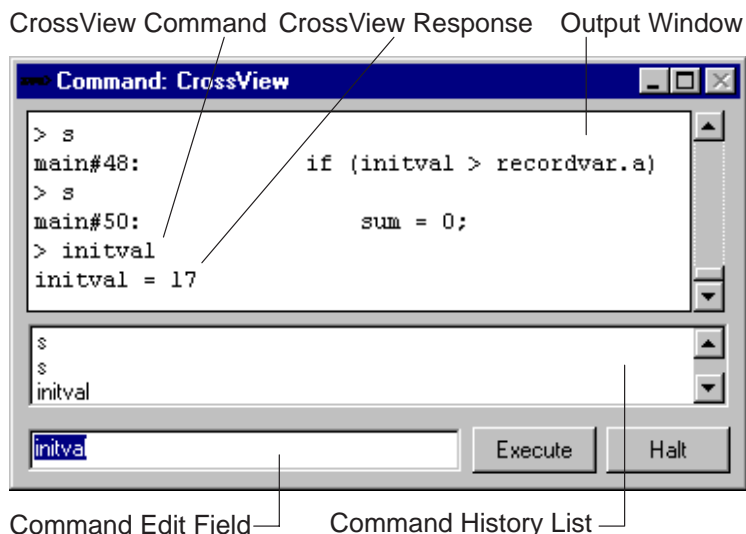


Figure 4-6: CrossView Pro Command Window

The Command Window also has two push buttons that provide rapid access to frequently used actions. The **Execute** button executes the current command (or sequence of commands if more than one command is selected). Note that the **<Enter>** or **<Return>** key is equivalent. Use the **Halt** button to interrupt commands executing in continuous mode, or to stop the emulator.

The Command Window maintains a history of recently executed commands. To re-perform previously executed commands simply double-click on it or select the command(s) from the command history list in the Command Window and press the **Execute** button. By hitting the **<Tab>** key, it is also possible to select one or more entries. Hitting **<Tab>** or **<Esc>** will return you to the command edit field.



The maximum number of lines saved to the CrossView Pro command buffer list is set during debugger startup. The default is 100 lines. To change the default select **Options...** from the **File** menu and select the **Initialization** tab. This number can also be modified via a startup option.

4.6.2 SOURCE WINDOW

The Source Window offers most of the debugging functions you will need on a regular basis. It allows you to:

- View the source file (source lines, disassembly or both).
- Set and clear assertions (not in Toolbar).
- Set and clear breakpoints.
- Monitor and inspect variables.
- Search for strings, functions, lines, addresses.
- Control execution.
- Call functions (not in Toolbar) and evaluate expressions.
- View code coverage information.
- View profiling/timing information.

An example of the source window is shown in figure 4-7.

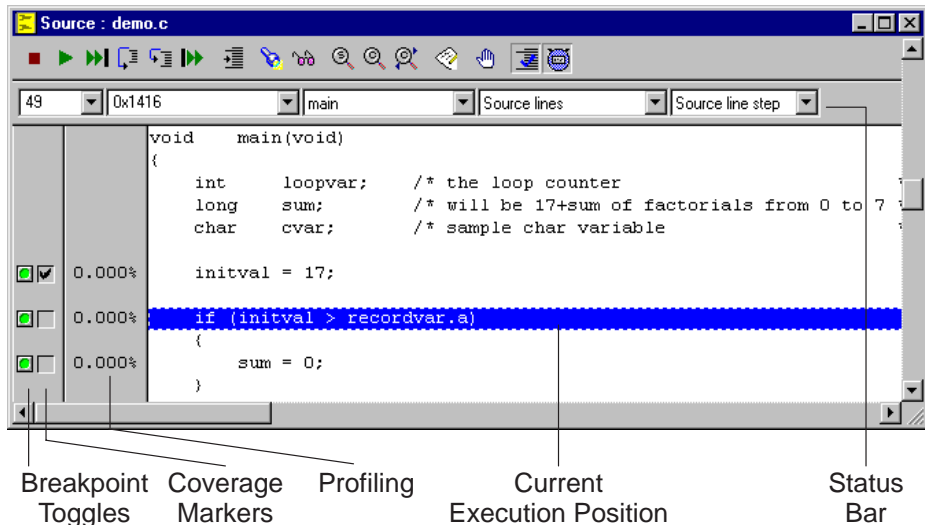


Figure 4-7: CrossView Pro Source Window

You can specify the step mode, symbolic disassembly and source lines / disassembly with the Source Window Setup dialog box (**Settings | Source Window Setup...**) or with **Run | Step Mode**. Alternatively, you can use the drop-down menus in the Source Window's status bar.



The default step modes are:

Source lines Window:	Source line step
Disassembly Window:	Instruction step
Source and Disassembly Window:	mode of previous window!
(assumes the step mode of the previous Source Window setting)	

The location of the cursor is also the viewing position. The line number and address of the viewing position, appears at the top-left position of the Source Window. This does NOT represent the current execution position (\$pc). The current execution position appears in reverse or blue color. The cursor appears as a dotted line.

On MS-Windows the so-called "quick watch" feature is supported. When you position the mouse cursor over a variable or a function, a bubble help box appears showing the value of the variable or the type information of the function respectively.

A green colored toggle shows that no breakpoint is set. A red colored toggle indicates an installed breakpoint. An orange colored toggle indicates an installed but disabled breakpoint. If code coverage is enabled, coverage markers appear to the right of the breakpoint toggles. If a checkmark appears next to a line, it has been executed. If no checkmark appears next to a line, it has not been executed.

The Source Window provides a local Toolbar containing the following buttons, nearly all of which are shortcuts (using selected text) to operations that you can perform via the menu bar:



Stop program or command



Run or continue execution (same as F5)



Run to cursor (same as F7)



Step (over function calls)












Step (into function calls)



Restart application



Find program counter (PC)

-  Show selected source expression
-  Watch selected source expression
-  Find symbol
-  Search for a text string
-  Repeat search for text string
-  Edit current source file
-  Edit breakpoint at cursor
-  Display code coverage
-  Display profiling

You can toggle the appearance of this local toolbar by selecting **Local Toolbars | Source** from the **View** menu.

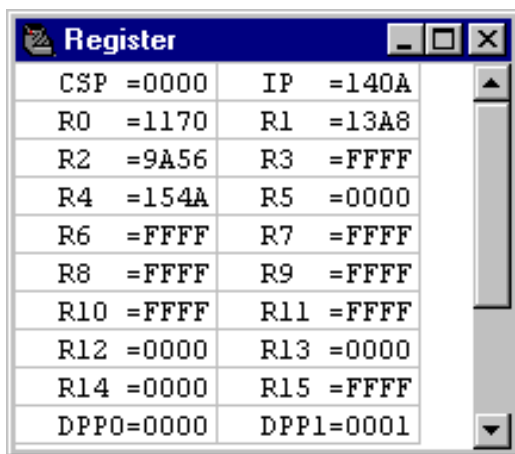
Edit Source

To edit the current source file in the Source Window, select **Edit | Edit Source** or press the **Edit Source** button. On MS-Windows the Codewright editor will be called with the filename and line number of the file that is currently in the debugger. on UNIX systems the **xvwedit** program will be called with the filename and line number of the file that is currently in the debugger.

The **xvwedit** program is a shell script. You can adapt it to your specific requirements.

4.6.3 REGISTER WINDOW

Figure 4-8 shows the Register Window. This window allows you to view and edit register contents.



Register	
CSP =0000	IP =140A
R0 =1170	R1 =13A8
R2 =9A56	R3 =FFFF
R4 =154A	R5 =0000
R6 =FFFF	R7 =FFFF
R8 =FFFF	R9 =FFFF
R10 =FFFF	R11 =FFFF
R12 =0000	R13 =0000
R14 =0000	R15 =FFFF
DPP0=0000	DPP1=0001

Figure 4-8: CrossView Pro Register Window



Note that the contents of the Register Window for your particular target may be different from the one shown in figure 4-8.

You can specify which register set definition appears in the Register Window with the Register Window Setup dialog box (**Settings | Register Window Setup...**). In this dialog you can also specify the display format of values in the Register Window: hexadecimal or decimal.

CrossView Pro supports multiple Register Windows. Register Windows either have the title "Register" or "Register – *register set name*". The "Register" title indicates the default register set.

In-situ editing allows you to change the registers contents directly by clicking on the corresponding cell.

4.6.4 MEMORY WINDOW

The Memory Window is shown in figure 4-9. This window allows you to view and edit the target memory.

Depending on the setting of the **Automatically refresh** check box in the Memory Window Setup dialog, CrossView Pro updates the displayed values every time the program is stopped or only updates the values by user request. For example, by pressing the **Update Memory Window** button located on the toolbar.

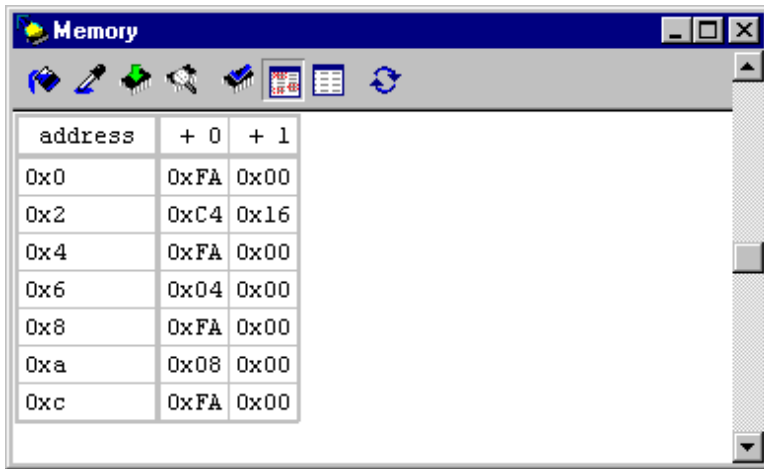


Figure 4-9: CrossView Pro Memory Window

To edit the target memory, click on a memory cell and type a new value. To display another memory region: click on an address cell and type a new address. CrossView Pro accepts input in symbolic format, so you can enter expressions instead of just values.

CrossView Pro supports multiple instances of the Memory Window. If your target supports multiple memory spaces, the Memory Window supports them all. Refer to the section about memory space keywords to become familiar with the memory space keywords and associated syntax your target system uses.

You can specify the way data appears in the Memory Window by opening the Memory Window Setup dialog. From the **Settings** menu, select **Memory Window Setup...** to open this dialog. The memory contents can appear in many formats including ASCII character, hexadecimal, decimal, signed, unsigned, and floating point formats. You can specify the size of the memory window. You specify the number of memory cells that appear within the window. The number of cells is fixed in the sense that if you re-size the window the number of cells does not change.

Besides the current value of memory locations, the Memory Window also displays whether memory locations have been accessed during program execution. This is called 'data coverage'. An application program may read from, write to, or fetch an instruction from a memory location. Of course all combinations may be legal. Although writing data to a memory location from which an instruction has been fetched is suspicious. All types of access, read, write, fetch or combinations of these, can be shown using different foreground and background colors. The color combination used to show "rwx" access are specified in the Desktop Setup dialog. Change the background color if instructions are fetched from a memory location, and change the foreground color to show read and write access.

You can display data coverage information in the Memory Window by clicking on the **Coverage** button in the Memory Window or by setting the **Display data/code coverage** check box in the Memory Window Setup dialog.

The Memory Window has the ability to highlight memory cells of which the contents have been changed. Click on the **Highlight Value Changes** button in the Memory Window to see the changed cells. With the **Freeze Highlight Reference Values** button you can enter a new reference point for highlighting. All the cells that have been changed since that reference point are highlighted.

The Memory Window provides a local Toolbar containing the following buttons:



Fill memory



Fill single memory address



Copy memory



Find memory



Display data coverage



Highlight changed values



Set highlighted values as reference



Refresh memory window

You can toggle the appearance of this local toolbar by selecting **Local Toolbars | Memory** from the **View** menu.

4.6.5 DATA WINDOW

The Data Window is shown in figure 4-10. This window allows you to show the value of monitored expressions and variables.

The Data Window updates the values shown every time the program stops, and after an **o** command.

It is possible to display both monitored and unmonitored data expressions in the Data Window. CrossView Pro monitors and updates "WATCH" expressions after every halt in execution, and marks them with the text "WATCH" at the start of the display line in the Data Window. "SHOW" expressions, on the other hand, are one-shot inspections of an expression's value, and CrossView Pro does not update them until you click on the **Update Selected Data Item** button or **Update Old Data Items** button. When a "SHOW" expressions is no longer actual, it is marked with the word "OLD".

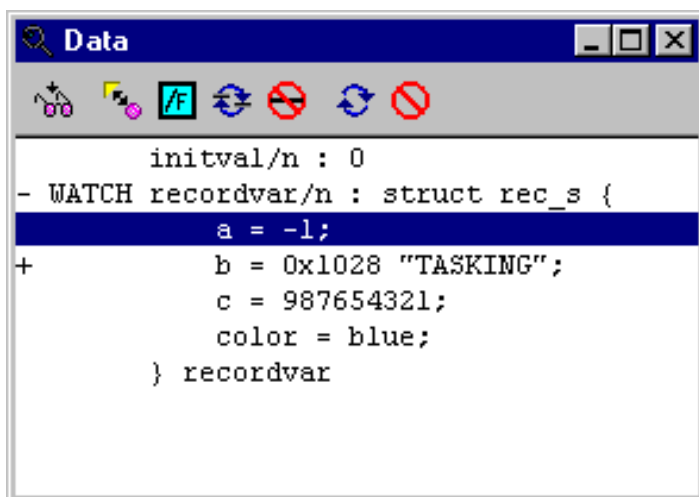


Figure 4-10: CrossView Pro Data Window

To set the default display format of the data shown, select the proper format in the **Data | Data Display Setup...** dialog.

To inspect the value of global variables and data structures, double-click on the variable name in the Source Window. Depending on preferences you set in the Data Display Setup dialog, the variable appears immediately in the Data Window, see figure 4-10, or the Expression Evaluation dialog appears first.

In-situ editing allows you to change the contents of everything in this window by clicking the value you want to change.

If you have set the **Display addresses** check box in the Data Display Setup dialog box the addresses of the variables are also shown.

Pointers, structures and arrays displayed in the data window have a compact and expanded form. The compact form for a structure is just `<struct>`, while the expanded form shows all the fields. The compact form of a pointer is the value of the pointer, while the expanded form shows the pointed-to object. Indicate the compact form by putting a '+' at the start of the display. (i.e., the object is expandable), and indicate the expanded form with (i.e., the object is contractible). Nesting is supported, so you can expand structures within structures ad infinitum.

To expand a pointer, structure or an array, double-click on the '+' in the Data Window.

The Data Window provides a local Toolbar containing the following buttons:



Show or watch a new expression



Toggle watch attribute of selected item "on" or "off"



Reformat selected item



Update selected data item



Delete selected data item



Update old data items



Delete old data items

You can toggle the appearance of this local toolbar by selecting **Local Toolbars | Data** from the **View** menu.

The auto-watch locals feature may be activated or deactivated. When active, a selected Data Window becomes the "auto-watch" window, and all local variables from the current top-of-stack frame appear in that Data Window. The text "LOCAL" appears at the start of the display for variables displayed in this manner. As the execution position changes, the auto-watch window deletes and adds locals as necessary, so that the locals on the current top-of stack frame always appear.

To see the value of the local variables of a function, Select **Data | Watch Locals Window** from the **View** menu.

CrossView Pro supports multiple Data Windows. Data Windows either have the title "Data Window #n" or "All Local Variables". The "All Local Variables" title indicates the auto-watch window if it exists (as explained above).

4.6.6 STACK WINDOW

The *stack* records the return addresses of all functions the application has called, and CrossView Pro can use this information to reconstruct the path to the current execution position. The Stack Window, shown in figure 4-11, displays the function calls on the stack with the values of the parameters passed to them in an easily accessible and understandable form.

The Stack Window can help you assess program execution and allows you to view parameter values. The stack window allows you to:

- View the stack trace which includes information about function names, parameter values, source line numbers and stack level.
- Easily switch to the call statement of a stack level by clicking on it once.
- Set temporary and permanent breakpoints at any level of the stack, by double-clicking on the desired level.

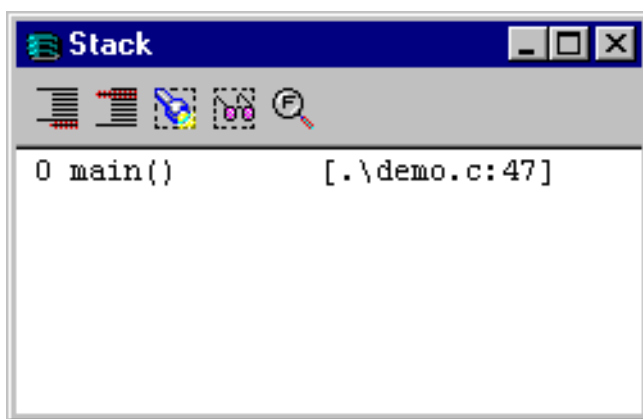







Figure 4-11: CrossView Pro Stack Window with Toolbar

The Stack Window provides a local Toolbar containing the following buttons:

-  Set stack breakpoint after call to function
-  Set stack breakpoint at function entry point
-  Show local variables in selected stack frame
-  Watch local variables in selected stack frame
-  Find call site

You can toggle the appearance of this local toolbar by selecting the **Local Toolbars | Stack** from the **View** menu.

4.6.7 TRACE WINDOW

The Trace Window, shown in figure 4-12, allows you to:

- Display the most recently executed lines of code.

CrossView Pro automatically updates the Trace Window each time you halt execution, as long as the window is open, allowing you to check the progress and flow of your program throughout the debugging session.

The Trace Window is only supported if your execution environment supports the trace facility.

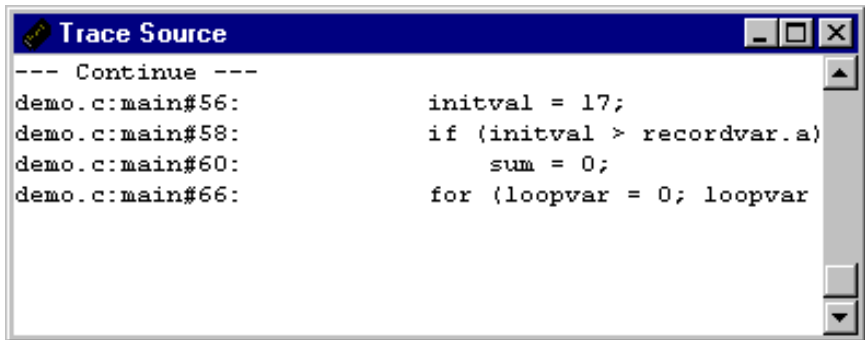


Figure 4-12: CrossView Pro Trace Window

4.6.8 TERMINAL WINDOW

The Terminal Windows, shown in figure 4-13, let you observe and test the input and output of your program.

The CrossView Pro Terminal windows provide an interface to exchange data with the application on the target. This I/O facility can be implemented in various ways. Using standard I/O stream function calls like `printf()` in your source, you can test I/O to and from the target system or simulator.

The File System Simulation feature redirects I/O to a Terminal Window if the filename `FSS_window:window_name` is used in the "open" call, *window_name* is the name of a Terminal Window.

A terminal window can be connected to multiple I/O streams of various types. For example, streams 0, 1 and 2 can be mapped to one terminal window. An I/O stream, however, can be mapped to one terminal window only. Each terminal window must have a unique name.

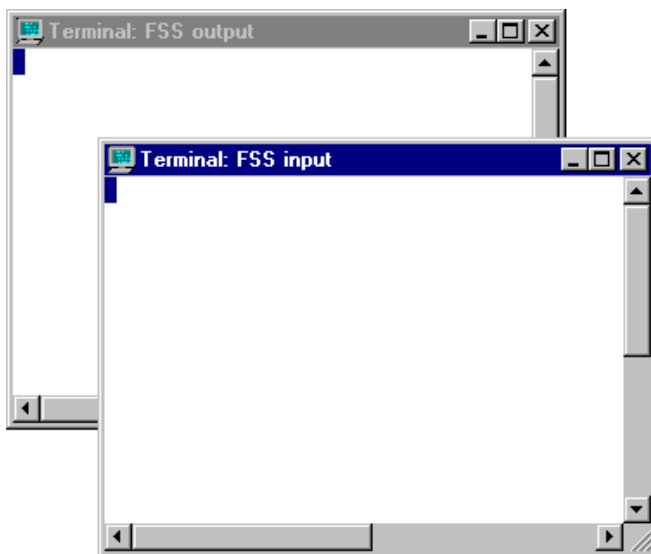


Figure 4-13: CrossView Pro Terminal Windows

You can specify the characteristics of the Terminal Window by opening the Terminal Window Setup dialog. From the **Settings** menu, select **Terminal Window Setup...** to open this dialog, or click with the right mouse button in the Terminal Window to bring up a popup menu and select **Setup...**. You can specify the input and output format of the terminal window. The input format can be a VT100-like terminal. The output format can be a VT100 terminal, display control codes, decimal, octal or hexadecimal. You can map linefeeds to carriage-return linefeeds, wrap at the end of a line, specify buffered input or specify that the window must be cleared at system reset and program reset. You can also log the input and output data to a file.

The default size of a terminal window is 24 lines of 80 characters. Everything that scrolls outside this window is lost. The visual window size can be smaller (scroll-bars are shown). You can specify another size in the Terminal Window Setup dialog.

Each terminal window has a local popup menu, which you can activate by clicking the right mouse button.



Figure 4-14: Terminal Window Local Popup Menu

Reset clears the contents of the terminal window and it also clears all attributes set with escape sequences. A **Clear** just clears the contents of a terminal window. **Reverse** changes the foreground and background colors and **Local echo** enables echoing back of typed characters in a terminal window. **Setup...** opens the Terminal Window Setup dialog.

You can connect an I/O stream to a terminal window in the **Connections** tab of the **Settings | I/O Simulation Setup...** dialog box.

4.6.9 DATA ANALYSIS WINDOW

CrossView Pro incorporates an advanced signal analysis interface designed to enable developers to monitor signal data more critically and thoroughly. This feature is useful when developing signal processing software for application areas such as communication, wireless and image processing.

Contrary to the other CrossView Pro windows the Data Analysis window (as shown in figure 4-15) is not opened from the View menu, but is opened as result of processing a data analysis script (or from the Settings menu). Most other CrossView Pro windows are updated whenever the target application stops execution due to, for example, a breakpoint. The Data Analysis window is only updated on user request. This is done because a large set of data is shown in the Data Analysis window and this set of data must be available and complete at the time the window is updated. Therefore, the user normally constructs a complex breakpoint to trigger the update of the Data Analysis window.

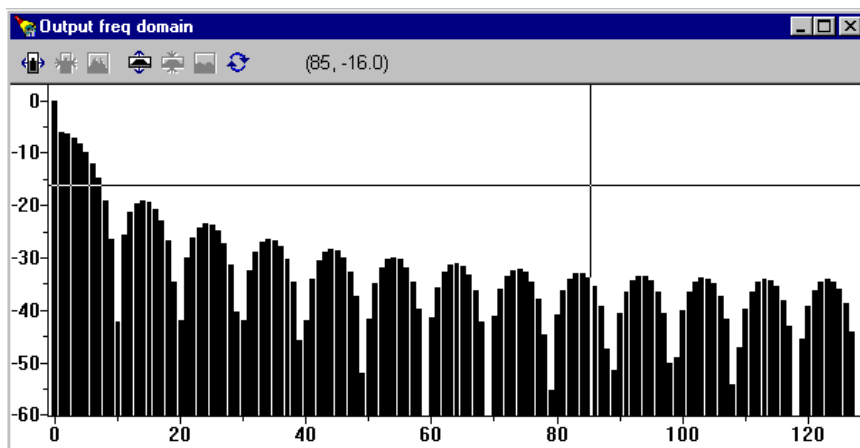


Figure 4-15: CrossView Pro Data Analysis Window

The Data Analysis Window provides a local Toolbar containing the following buttons:



Zoom in horizontally



Zoom out horizontally



Unzoom horizontally to normal (show all collected data)



Zoom in vertically



Zoom out vertically



Unzoom vertically to normal (show all collected data)



Update Data Analysis window

The graph displayed in the Data Analysis window is constructed by processing a CXL script. Refer to the CXL syntax specification in section 11.5.2, *Syntax of CrossView eXtension Language (CXL)*, for details. TASKING provides scripts for standard signal analysis such as FFT. However, the programmer can write CXL scripts and process the data in the format he desires.



See section 11.5, *Data Analysis*, for more details on data analysis.

4.6.10 POP-UP WINDOWS

Finally, two more windows can appear in certain situations:

Help Window: Activated with function key **F1** or when a **Help** button is pressed inside a dialog.

Toolbox: This window contains user defined buttons.

4.7 CONTROL OPERATIONS FOR CROSSVIEW PRO

All control operations can take place in any CrossView Pro Window. You can select and save startup options. You can record and play back playback files. You can define macros and assign them a button in the toolbox (allowing you to configure up 16 buttons).

4.7.1 ECHOING COMMANDS

The Command Window echoes every command given to CrossView Pro. CrossView Pro translates most button actions and menu selections into the CrossView Pro keyboard command equivalents. The Command Window echoes the equivalent commands just as if you had typed them there.

4.7.2 MOUSE/MENU/COMMAND EQUIVALENTS

Actions in CrossView Pro are performed by using keyboard commands typed into the Command Window, selecting a menu item, by clicking on a push button and sometimes by direct manipulation of objects with the mouse. Many actions can be accomplished several ways. For instance there are three different ways to set a breakpoint. You can:

1. Use the ***line b*** command in the command entry field.
2. Click on a breakpoint toggle in the Source Window.
3. From the **Breakpoints** menu, select **Breakpoints...** to open up the Breakpoints dialog box.

4.8 USING THE ON-LINE HELP

CrossView Pro has an extensive on-line help system to aid you. Help topics cover all CrossView Pro Windows, commands, and dialog boxes.

4.8.1 ACCESSING ON-LINE HELP

You can access help in several ways:

1. Click the **Help** button on a dialog box

Opens the help system with information about how to perform the task or about the meaning of the dialog.

2. Click on the question mark in the upper right corner of a dialog, then click the element in the dialog you want help on.

A yellow box briefly explains the element you asked help on.

3. Select the **Help | Help** menu item or press the **F1**-key.

Opens the help system with information about the active window.

4. Hover the mouse pointer over a toolbar button.

A yellow box shows the title of the button. A more complete description is shown in the status bar at the bottom of the screen.

4.8.2 USING MS-WINDOWS HELP

You enter help at a topic that explains the current window or dialog. By clicking on links, you can follow different paths. To return to your starting point click the **Back** button or open the **Options | Display History Window** and click on the node that you want to return to.

The **Contents** tab displays a list of main subjects. The **Index** tab displays a list of keywords that relate to certain topics. When you click the **Find** tab, you can search for a string pattern.

To save time, you can iconize the Help Window and maximize it when necessary.



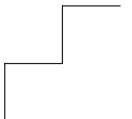
CHAPTER

5

CONTROLLING PROGRAM EXECUTION



TASKING



5

CHAPTER

5.1 SOURCE POSITIONING

When you have the Source Window open and it displays a source file, there are two points of reference to keep in mind: the execution position and the viewing position.

The **execution position** refers to the line of source at the Program Counter address. This line is always the next statement or instruction to be executed. When you load a file into the Source Window, CrossView Pro automatically displays the portion of the source code that contains the execution position.

The **viewing position** (also called 'cursor') is the line currently being examined in the displayed source file. Since many Source Window operations act on this line, you can think of the viewing position as the 'current line'. For instance, if you set a breakpoint without specifying a line number, CrossView Pro sets the breakpoint at the line marked by the viewing position. Please note that it is the viewing position that appears to the left of the Source Window (NOT the execution position!).

The execution position and the viewing position refer to the same line when a source file is first loaded into the Source Window. You can then change the viewing position, if you wish.

The execution position and the viewing position appear different to distinguish them from the rest of the source code. The execution position line appears in the execution position highlight colors, while the viewing position appears as a broken-line frame, also called the **cursor**. Note that a line containing a breakpoint appears in the breakpoint highlight colors.

A combination of execution position, cursor and breakpoint (all of which are potentially active on the same line) appear accordingly.

5.1.1 CHANGING THE VIEWING POSITION

When a program is active the viewing position is always visible in the Source Window. You can change the viewing position to move throughout the source file. Usually, whenever the execution position changes, the viewing position automatically follows suit. But you may easily change the viewing position without affecting the execution position.



To change the viewing position use any of the following possibilities:

- Use the vertical scroll bar to move a line or a page at a time. The view point stays on the same line until it is no longer visible. It then stays on the first or last line of the display, depending on the direction of scrolling.
- Click on the desired, unmarked source line.
- From the **Edit** menu, select **Find Line...** to specify to which particular line you wish to move.

In the upper-left corner of the Source Window, there are two text fields. These fields show the line number of the current viewing position and the address of the first machine instruction for that line. CrossView Pro updates the **Line** and **Address** values each time the viewing position changes.



You can change the viewing position to the first executable line of a particular function with the **e** command. For instance:

```
e main
```

will make the first executable line of `main()` the current viewing position and display it in the Source window. You may also use the stack depth as an argument, if you place it before the **e**:

```
1 e
```

This will change the viewing position to stack depth 1, that is, the line that called the current function.

FUNCTION: Change the viewing position.

COMMAND: *stack e*
 e function

To change the viewing position to a specified address, you can use the **ei** command. This command is useful for viewing some code in the assembly window, without changing the program counter, since the execution position is not changed.

FUNCTION: Change the viewing position to address.

COMMAND: *address* **ei**

5.1.2 CHANGING THE EXECUTION POSITION

There may be times when you want to start or resume execution at a different line than the one marked by the current execution position.

Exercise caution when changing the execution position. Often each line of C source code compiles into several machine language instructions. Moving the program counter to a new address in the middle of a series of related assembly instructions is sometimes risky. Moreover, even though you change the program counter, registers and variables may not have the expected values if you bypass parts of the code.



In the Source Window you can change the execution position to the viewing position with the menu entry **Run | Jump to Cursor**. This menu entry is disabled in Source file window mode to prevent problems by skidding pieces of C code which are required to be executed. See also the **g** and **gi** commands below.



When the program halts, you can change the execution position with the **g** command in the Command Window. The **g** command moves the execution position, but does not continue the program. To resume execution from your new execution position, use the **C** command.

Although risky, the **g** command does have its uses, especially in conjunction with breakpoints to patch code. Refer to the *Breakpoints and Assertions* chapter for more information.

For example, to change the execution position from the current line, 54, to line 62, enter:

g 62

When you resume execution in this program, it is from line 62 instead of line 54.

FUNCTION: Change the execution position to a specified C source line

COMMAND: **g** *line_number*

You can also change the execution position to a specified address directly, although the same warnings apply. To do so, use the **gi** command. For instance:

0x800 gi

FUNCTION: Change the execution position to address.

COMMAND: *address* **gi**

Of course, moving the program counter (**gi** command) is even more potentially reckless than using the **g** command. Use both with caution especially when debugging a program which has instructions re-ordered due to optimizations.

To determine the address of a line of source, use the **P** command:

80 P
80:(0x1486): sum = sum + 1;

The hexadecimal number in parentheses is the instruction address for line 80.

FUNCTION: Print a source line and its instruction address.

COMMAND: *line_number* **P**

5.1.3 SYNCHRONIZING THE EXECUTION AND VIEWING POSITIONS

Each time you stop execution, the position of the program counter (PC) is visible in the source window. However, it may disappear from the window when scrolling through the source or when you loaded a new program. To find the program counter again:



Click on the **Find PC** button in the Source Window or select **Find PC** from the **Edit** menu.



From the Command Window, use the **L** command.

The L Command

The **L** command is shorthand for **oe**. It synchronizes the viewing and the execution positions, adjusting the viewing position if the two are different. The **L** command never affects the execution position. The **L** command is useful if you have changed your viewing position and do not remember where your execution position is.

FUNCTION: Synchronize viewing and execution position.

COMMAND: **L**

5.2 CONTROLLING PROGRAM EXECUTION

Using the mouse in the Source Window, you can direct the execution of your source programs. Among your options are:

- Starting execution from the first instruction or from the current execution position.
- Manually stopping execution whenever you want.
- Executing the program a single line at a time.
- Executing functions by calling them directly.

5.2.1 STARTING THE PROGRAM

To restart a program from its first instruction:



Click on the Restart program button in the Source Window.

or:

- From the **Run** menu, select **Reset Application**
- From the **Run** menu, select **Run**, or click on the **Run/Continue** button.



Type the **R** command from the Command Window.



This is NOT a target system reset. Refer to the **rst** command for information about side effects that may be introduced due to a target system reset.

After restarting a program, you can stop execution only by a breakpoint, an assertion or a halt operation from the user.

FUNCTION: Reset program; run program.

COMMAND: **R**

5.2.2 HALTING AND CONTINUING EXECUTION

To stop or continue execution:



Click on the **Halt** button in the Source Window to stop execution. Click on the **Run/Continue** button to resume execution.



From the **Run** menu select **Halt** to stop execution. Select the **Run** menu item to resume execution.



Use the **C** command or function key F5 to resume execution.

When you halt the program, all the active windows update automatically to reflect the program's current status. For instance, if you have any expressions monitored in the Data Window, their current value appears.

Note that when you use any of the above methods to stop the program, CrossView Pro halts at the machine instruction that was on when interrupted. While this is a convenient way to stop the program, it is hardly an accurate one — you may stop execution in the middle of a C source statement.

To stop a program at a precise line of C source code, set a breakpoint. For more about breakpoints see the *Breakpoints and Assertions* chapter.

When continuing, CrossView Pro resumes execution as if the program had never stopped.

FUNCTION: Continue execution from the current execution position.

COMMAND: **C**

5.2.3 SINGLE-STEP EXECUTION

When the program stops, you can continue execution, or you can step through it one line or instruction at a time. This is called **single-step execution**.

Single-stepping is a valuable tool for debugging your programs. The effect is to watch your programs run in stop motion. You can observe the values of variables, registers, and the stack at a precise point in a program's execution. You can catch many potential bugs by watching a program run line by line.

When you single step, CrossView Pro normally executes one line of your source and advances to the next sequential line of the program. If you single step to a line that contains a function call, however, you have two options: step into the function or step over the function call.

Source Single-Step Into

There are several methods you can use to single step into:



Click on the **Step Into** button in the Source Window or select **Step Into** from the **Run** menu.



Press function key **F8** or type the **s** command in the Command Window. You have the option of setting the number of lines you want to execute. For example, to execute 2 lines of the program, type: **2 s**.

FUNCTION:	Step through a program one source line at a time.
COMMAND:	<i>number s</i>

Stepping Into Functions

Stepping into a function means that CrossView Pro enters the function and executes its prologue machine instructions, halting at the first C statement. When you reach the end of the function, CrossView Pro brings you back to the line after the function call and continues with the flow of the program. The debugger changes the source code file displayed in the Source Window, if necessary.



If you accidentally step into a function that you meant to step over, you can select **Return from Function** from the **Run** menu to escape quickly.

For example, suppose you are at line 59 of a file, which contains a call to the function `factorial()`:

```
main#59:    table[ loopvar ] = factorial(loopvar)
```

By performing one **Step Into** action, you can step into the source code for `factorial()`. Your Execution and viewing position change to:

```
factorial#103:  char locvar = 'x';
```

CrossView Pro shows you the current function and line number and the C source code for the current execution position.

Source Single-Step Over

To step *over* a statement or a function call:



Click on the **Step Over** button in the Source Window or select the **Step Over** from the **Run** menu.



Press function key **F10** or enter the **S** command in the Command Window. You have the option of setting the number of lines you want the debugger to execute. For example, to execute three lines of source, single stepping over functions, enter: **3 S**.

FUNCTION:	Single step, but treat function calls as single statements.
COMMAND:	<i>number S</i>

Stepping over Functions

Stepping over a function means that CrossView Pro treats function calls as a single statements and advances to the next line in the source. This is a useful operation if a function has already been debugged or if you do not want to take the time to step through a function line by line.

For example, suppose you reach line 59 in `demo.c`, which calls the function `factorial()`, as in the example above. If you give a **Step Over** command, the execution position moves to line 60 of the source code in the `main()` function immediately, without entering the source code for `factorial()`. CrossView Pro has executed the function call as a single statement.

If you try to step over a function that contains a breakpoint or that calls another function with a breakpoint, CrossView Pro halts at that breakpoint. Once execution stops, the step over command is complete. Therefore, if you resume execution by clicking on the **Run** button or with the C command, you do not regain control at the entrance to the function with the breakpoint. You can either single step through the rest of the function, or select the **Run | Return from Function** menu item to return to the line after the point of entry.

5.2.4 STEPPING THROUGH AT THE MACHINE LEVEL

While single stepping through code at the source level is informative, you might need a lower level approach. CrossView Pro can step through a program at the assembly language instruction level.

While more time-consuming than a source level step-through, an instruction level step-through allows you to examine how your code has been compiled. As you advance through the assembly instructions, notice how CrossView Pro translates data addresses to variable names, and correlates branch addresses to points in the source code. This makes it much easier to follow the source at the instruction level.



The default step modes are:

Source lines Window:	Source line step
Disassembly Window:	Instruction step
Source and Disassembly Window:	mode of previous window!
(assumes the step mode of the previous Source Window setting)	



Mouse and menu actions:

- The **Step Into** and **Step Over** buttons, and **Run | Step Over** and **Run | Step Into** menus can be set to step by instructions by selecting **Run | Step Mode | Instruction step** from the menu bar.
- To change back to stepping by source lines, select **Run | Step Mode | Source line step**.
- Another way to set the step mode is to select the **Source line step** or **Instruction step** radio button in the **Settings | Source Window Setup** dialog box.



To control this function from the Command Window, use the **Si** and **si** commands. The **Si** and the **si** commands are analogous to the **S** and **s** commands, **Si** will treat function calls (more precisely, jump to subroutine instructions) as single statements, while **si** will enter the function.

FUNCTION: Single step at instruction level. Step into functions.

COMMAND: *number si*

FUNCTION: Single step at instruction level. Step over functions.

COMMAND: *number Si*

As an example of stepping through instruction level code, restart the program. Then select **Run | Step Mode | Instruction step**. Once it stops at the breakpoint you installed, advance execution one assembly language instruction at a time by using the **Step Over** and **Step Into** buttons. Or give the **Si** or **si** commands.

CrossView Pro will display disassembly of the next machine instruction that forms part of the C code in the Command Output Window:

```
main#47+0x4:      disassembled instruction
```

Different types of targets, of course, have different assembly code, so debugging at the assembly level is hardware dependent.

Notice that a single C statement is usually compiled into several, sometimes many, machine instructions.



CrossView Pro supports debugging on machine instruction level using the Intermixed or Assembly mode of the Source Window.

5.3 NOTES ABOUT PROGRAM EXECUTION

If you stop the program in a module without debug symbols, then an **S** or **s** command attempts to step to a module with symbols. CrossView Pro does this by searching the run-time stack for a return address in a module with symbols, then setting a temporary breakpoint there, and running. This process relies on two assumptions: that the stack layout is uniform, and that each function eventually returns. In the unlikely event that these assumptions are violated, the program may run away when you attempt to single step.

5.4 CALLING A FUNCTION

You can execute a function by calling it directly, without waiting for the program to run to the function's position in the code. CrossView Pro gives you the capability of passing input parameters to the function and allows you to examine the return value after the function executes.

After you manually call a function, CrossView Pro returns to the current execution position and restores the values of all your registers to the state before calling the function.

You can call a function by selecting a menu item with a mouse or by use of a command in the Command Window.



From the **Run** menu, select **Call a Function...** to view the Call a Function dialog box.

The Call a Function dialog box contains two fields, in each field there are drop-down history lists. In the first, you write the name of the function. Alternatively, click on the **Browse...** button in the dialog box, which opens the Function Lists dialog box where you can search for global and local functions.

The second field allows you to enter all the values for the input parameters to this function. The syntax for the parameters is exactly the same as used in the source code. That is, each parameter is input as an expression and separated by commas. You do not need to enter any parentheses or semicolons, and you can use names of variables and constants in the expressions.

The return value appears in the Command Output Window.



To call a function from the Command Window, simply enter the name of the function along with the appropriate parameter values. For example, the following demonstrates calling the function `factorial()`:

```
factorial(3)  
6
```

You can also use functions in expressions, as in:

```
factorial(3) - 2  
4
```



Command line function calls are only allowed in an application that uses a reentrant stack frame. When a static stack frame is used, the locator overlays stack frames based on the application's call graph. A command line function call may conflict with the calling sequence in the call graph and corrupt the static stack frame when executed.

5.5 SEARCHING THROUGH THE SOURCE WINDOW

CrossView Pro can search for addresses and functions in the entire application and for line numbers, and strings in the current source file. A string search starts from the current viewing position and "wraps around" the end (or begin) of the current source file. The string search ends when a matching string is found or when it returns to the starting point.

5.5.1 SEARCHING FOR A FUNCTION

There are several ways to find a function:



Using the mouse:

- From the **Edit** menu, select **Find Symbol...** to open the Find Symbol dialog box. Select the function you are looking for.
- Click on the **Find Symbol** button in the Source Window to open the Find Symbol dialog box.
- Select a function in the Stack Window (double-click) to show the line that called it.



From the Command Window, you can either specify **c** followed by the function name, or a stack position followed by **c**. For example:

```
e main      Find the function main().
1 e         Find the line that called the current function.
```

CrossView Pro searches through all the relevant source code files to find the one containing the body of the function. The part of the file containing the function appears in the Source Window.

5.5.2 SEARCHING FOR A STRING

CrossView Pro allows you to search for a particular string in the current source file. CrossView Pro searches the Source Window from the current viewing position. If it finds the string, it moves the viewing position to the corresponding line. This does not affect the execution position.

To find a string:



Open the Search String dialog box by clicking on the **Find Text String** button, or select **Search String...** from the **Edit** menu. Click on the Case Sensitive check box to turn case sensitivity on or off.

You can also highlight a text fragment in the source code and click on the **Find Next Text String** button to find that fragment again.



In the Command Window, use the **/** or **?** commands. The **/** command searches forwards and the **?** command searches backwards. For example, to find the string `initval`, enter:

```
/initval Search forward for the string "initval"
```

CrossView Pro's searches "wrap around" beyond the top or bottom of the file if necessary.

FUNCTION:	Search forward for a string.
COMMAND:	<code>/ string</code>
FUNCTION:	Search backward for a string.
COMMAND:	<code>? string</code>

If no string is supplied to the / or ? command, or if you hit carriage return, or press the function key **F3** or select the **Search Next String** from the **Edit** menu item, CrossView Pro searches again for the last string requested.

5.5.3 JUMPING TO A SOURCE LINE

As mentioned earlier in the *Changing the Viewing Position* section, you can use the scroll bar to scroll through the source code or use the arrow keys or the + and - keys. To find a specific line, you can use one of several methods:



From the **Edit** menu, select **Find Line...** to open the Find Line dialog box.

After you enter a line number (or select one from the history list) in this dialog box and click on the **Find** button, CrossView Pro will change the viewing position to the indicated line number. At the first use, the Find Line dialog box contains no line number, but on subsequent invocations it will show the line number you entered before.



Enter the line number on the command line.



PROGRAM EXECUTION

CHAPTER

6

ACCESSING CODE AND DATA



6

CHAPTER

6.1 INTRODUCTION

This chapter discusses topics related to viewing and editing the variables in your source program and execution environment, including accessing variables and registers, viewing and modifying the data space, using monitors, viewing the source file, and disassembling code.

6.2 ACCESSING VARIABLES

This section describes how to view and edit your program variables using the debugger. You can monitor data so that every time you stop the program, CrossView Pro updates the current value.

The Data Window displays the values of variables and expressions. As long as the this window is open, CrossView Pro automatically updates the display for each monitored variable and expression each time the program stops.



Uninitialized variables will not have meaningful values when you first start the debugger, since your program's startup code has not been executed. Also note that global data is initialized at load time. Re-running a program may produce unexpected results. To guarantee that global data is initialized properly, download the program again.

6.2.1 VIEWING VARIABLES, STRUCTURES AND ARRAYS

You may view variable values, and change them, from the Source Window and the Command Window. CrossView Pro returns the variable in the format *var_name = value* in the Command Window.

It is possible to display both monitored and unmonitored expressions in the Data Window. After every halt in execution, CrossView Pro updates monitored expressions. Unmonitored expressions are just one-shot inspections of the expressions value. Refer to section 4.6, *CrossView Pro Windows* for a detailed description of the Data Window.

To set the default display format of the data shown, select the proper format in the **Data | Data Display Setup...** dialog.

To show the contents of a variable or to show the type information of a function:



Position the mouse cursor over a variable or a function in the Source Window. A bubble help box appears showing the value of the variable or the type information of the function, respectively.

To evaluate a simple expression:



Double-click on a variable in the Source window. The result of the expression is shown in the Data Window. Alternatively, depending on the preferences you set in the Data Display Setup dialog, the expression appears in the Evaluate Expression dialog. Click the **Add Watch** or **Add Show** button to display the result of the expression in the Data Window. Click the **Evaluate** button to display the result of the expression in the output field of the Evaluate Expression dialog.

To evaluate a complex expression:



From the **Data** menu, select **Evaluate Expression...** and type in any C expression in the Evaluate Expression dialog box. Optionally select a display format. Click the **Evaluate** button.



Type the expression into the command edit field of the Command Window followed by a return or click the **Execute** button.

For example, to find the value of `initval` in `demo.c` type:

```
initval
```

and CrossView Pro will display:

```
initval = 17
```

FUNCTION: Display the value of a variable.

COMMAND: *variable's_name*



For variables having the same name as an CrossView Pro command, use `/n` as format style code.



Any expression that can be typed into the Command Window can also be typed in the **Expression** field of the Expression Evaluation dialog box. Throughout this discussion, expressions can be typed in either location, depending on what is convenient.

Viewing Structures

You can also view structures.

By using any of the methods described above, you can print out the entire structure. For example:

recordvar

and CrossView Pro prints out the structure of `recordvar` and values of `recordvar`'s fields in correct C notation:

```
recordvar = struct rec_s {
    a = -1;
    b = 0x1028 "TASKING";
    c = 987654321;
    color = blue;
} recordvar
```

Displaying Individual Fields

Similarly, you can instruct the debugger to print the value of an individual field.

In the Source Window, highlight `recordvar.color` and click the **Show Expression** button. Or, in the **Expression** edit field of the Expression Evaluation dialog box or in the Command Window, type the structure name followed by a period and the field name. For instance, to see the field `color` for the structure `recordvar`, enter:

```
recordvar.color Command
color = blue    Output
```

Note that CrossView Pro returns the value in the form *field_name = value*. CrossView Pro also displays enumerated types correctly.



Variables will not have meaningful values when you first start CrossView Pro, since your program's startup code has not been executed.

Displaying the Address of an Array

If you enter the name of an array in the Expression Evaluation dialog box or in the Command Window, the debugger returns its address. For instance, to find the address for the array `table`, select `table` from the browse list in the dialog box or type the name in the Command Window:

table	<i>Command</i>
table = 0x200	<i>Output</i>

Note that CrossView Pro returns the address in the form *array_name = address*.

The debugger can also display the address and value of an individual element of an array. Enter the name of the array and the number of the element in brackets. For instance, to find the address and value of the third element of array `table`, enter:

table[3]	<i>Command</i>
0x20C = 0	<i>Output</i>

Note that CrossView Pro returns the information in the form *address = value*.

Displaying Character Pointers and Character Arrays

The following piece of C code can be accessed in CrossView Pro using the string format codes:

```
char text[] = "Sample\n";
char *ptext = text;
```

text	<i>What is the address of this char array</i>
text = 0x8200	

text/a	<i>Print it as a string</i>
text = "Sample^J"	

ptext	<i>What is the contents of this pointer</i>
string = 0x8200	

ptext/s	<i>Print it as a string</i>
string = "Sample^J"	

&ptext	<i>Where does ptext itself reside</i>
0x8210	

Sizing Structures

With structured variables, it is especially useful to know the size of a variable.

In the Command Window, you can determine the size of a variable with the `sizeof()` function. For instance, to determine the size of the structure `recordvar`, enter:

```
sizeof(recordvar)  
24
```

6.2.2 CHANGING VARIABLES

With CrossView Pro, you can not only view your variables, but change them. This function allows you to easily test your code by single-stepping through the program and assigning sample values to your variables. For instance, to set the variable `initval` to 100, enter:

```
initval=100
```

and CrossView Pro confirms `initval`'s new value:

```
initval = 100
```

Note that CrossView Pro returns the values of variables with the syntax: *var_name = value*, with any right-hand side expression evaluated to a single value.

Changing variables in the Data Window



To change a variable in the Data Window, follow these steps:

- In the Data Window, double-click on the variable you wish to edit. In-situ editing will be activated.
- Specify the new value in the edit control and hit the **Enter** key.

When in-situ editing is active, you can use the **Tab** key to move the edit field to the next variable value or use the **Shift+Tab** key combination to move the edit control to the previous variable.



Assigning Structures

CrossView Pro also allows you to assign whole structures to one another.

You can use a simple equation to assign the structures. For instance, to assign `statrec` to `recordvar`, enter:

```
statrec = recordvar
```

6.2.3 THE l COMMAND

CrossView Pro’s windows contain a great deal of information about the current debugging session. Occasionally, however, you have a few closed windows, or wish the information to appear in the Command window (for instance, when you are recording output). Using the **l** (list) command, you can find out all sorts of things about the current state of the debugger and have the information appear in the Command window.

Arguments of the l Command	
a assertions	k kernel state data
b breakpoints	m memory map (of application code sections)
d directory	p procedures (functions)
f files (modules)	r registers
g globals	s special variables

For configurations that support real-time kernels the **l k** command can have additional arguments. See the description of the **l** command in the *Command Reference* for details.

You may for example view the contents of the registers:

```
l r
```

Or the list of procedures (that is, functions):

```
l p
```

a complete list of global variables:

```
l g
```

The **lf** command (list files) also shows the address where CrossView Pro placed the first procedure in the module. If the module is a data module then the address reflects the first item's placement.

With all of these **l** commands you can specify a string:

l g record

and CrossView Pro searches the globals for a match with the same initial characters; in this case global variables that begin with `record`.

6.3 EXPRESSIONS

6.3.1 EVALUATING EXPRESSIONS

CrossView Pro expressions use standard C syntax, semantics, and allow special variables. You can calculate and show the values of expressions in CrossView Pro by using a variety of methods:

It is possible to display both monitored and unmonitored expressions in the Data Window. CrossView Pro updates monitored expressions after every halt in execution. Unmonitored expressions are just one-shot inspections of the expressions value. Refer to section 4.6, *CrossView Pro Windows* for a detailed description of the Data Window.

To evaluate a simple expression:



Double-click on a variable in the Source window. The result of the expression appears in the data window. Alternatively, depending on the preferences you set in the Data Display Setup dialog, the expression appears in the Evaluate Expression dialog. Click the **Add Watch** or **Add Show** button to display the result of the expression in the Data Window. Click the **Evaluate** button to display the result of the expression in the output field of the Evaluate Expression dialog.

To evaluate a complex expression:



From the **Data** menu, select **Evaluate Expression...** and type in any C expression in the Evaluate Expression dialog box. Optionally select a display format. Click the **Evaluate** button.

CrossView Pro calculates the result and displays the value in the appropriate format. For details about expression formats see the section *Formatting Expressions* in the chapter *CrossView Pro Command Language*.



Type the expression in the Command Window.

Expressions can contain variable names as arguments. For instance, if the variable `initval` has a value of 17 and you enter:

```
initval * 2
```

CrossView Pro displays:

34

The expression can contain names of variables, constants, function calls with parameters, and so forth; anything that you can write directly at the Command Window, you can use in the Evaluate Expression dialog box. For more information on expressions and the CrossView Pro command language, refer to the section *CrossView Pro Expressions* in the *Command Language* chapter.

The Dot Operand

Using the dot shorthand “.” can save you some typing. The dot stands for the last value CrossView Pro displayed. For instance:

```
initval  
initval = 17
```

Now you can use the value 17 in another expression by typing:

```
. * 2  
34
```

The value is the result of the new expression.

Naturally, using the dot operand saves you from retyping complex expressions.

6.3.2 MONITORING EXPRESSIONS

CrossView Pro allows you to monitor any variable or expression. Monitoring means that the debugger evaluates a particular expression and displays the result each time the program stops. If you are in window mode, CrossView Pro displays the values of the monitored variables and expressions in the Data window.

Monitor Set Up

To set up a monitor you can:



From the **Data** menu, select **Evaluate Expression...** or double-click on a variable in the Source Window, or click on the **Watch Expression** button to view the Expression Evaluation dialog box. From this dialog box, you can enter an expression and monitor (watch) its value in the Data Window. You can skip the Expression Evaluation dialog if you activate the **Bypass Expression Evaluation dialog** check box in the Data Display Setup dialog.

Alternatively, click on the **New Expression** button in the Data Window.



The Data Window must be open to display the result. Otherwise CrossView Pro does not monitor the expression. Therefore, CrossView Pro opens the Data Window automatically when you choose to show or watch an expression.



Type the **m expression** command in the Command Window.

To place the variable `initval` in the Data window type:

```
m initval
```

`initval` remains in the Data window. You may run the program, step through it, and the display updates continually. Even if you are not in window mode, CrossView Pro still displays the value of `initval` after every CrossView command.

FUNCTION: Monitor an expression or variable.

COMMAND: **m expression**

Similarly, if you want twice the value of `initval` you could type:

```
m initval*2
```

And the expression `initval*2` is monitored.



Monitored expressions are evaluated exactly as if you had typed them in from the command line; therefore, if you are monitoring a variable, say `R`, identical to an CrossView Pro command, use the `/n` format, in this example `R/n`.

Monitor Delete

To remove a monitored expression you can:



Select the item in the Data Window and click on the **Delete Selected Data Item** button from the Data Window, or select **Data | Delete | Item**.

To remove all expressions from the Data Window, select **Data | Delete | All**.



Type the **number m d** command in the Command Window.

To remove `initval` from your Data Window #1, type the number of the expression (first item of the Data Window has number 0) and **m d** (monitor delete):

0 m d

and CrossView Pro removes `initval` (in this case, assuming it is the first variable listed in the window) from the Data Window.

FUNCTION: Remove an expression from the Data Window

COMMAND: *number* **m d**

Since local variables have no meaning beyond their range, CrossView Pro issues error messages if you try to evaluate local variables beyond their scope. Some variables also become invisible when the program call another function. For instance, if you are in `main()`, monitoring `sum`, and `main()` calls `factorial()`, the unqualified name `sum` is no longer visible inside `factorial()`. You can get around this problem, however, by monitoring `main#sum` instead.

6.3.3 FORMATTING DATA

When you display a particular variable, CrossView Pro displays it in the format the symbolic debug information defines for it. You may, however, easily specify another format using dialogues or keyboard commands. See the section *Formatting Expressions* in the chapter *CrossView Pro Command Language*.

Examples

To print the value of `initval` in hexadecimal format, enter

initval/x

Be sure not to confuse CrossView Pro format codes with C character codes. CrossView Pro uses a / (forward slash) not a \ (backward slash).

Don't worry about trying to memorize the list, you probably won't have occasion to use all these formats. Notice, however, that the `/t` format code give information about a particular value. For instance, if you wanted to find out what the type of `initval` is, type:

```
initval/t
global long initval
```

You can also take more low-level actions, such as finding out which function contains the hexadecimal address `0x100`.

```
0x100/P
main
```

CrossView Pro tells you that address `0x100` is in the function `main()`.

6.3.4 DISPLAYING MEMORY

CrossView Pro supports several methods to display memory contents. The Memory Window provides a very user-friendly yet powerful way to display the raw contents of the target memory.

Refer to section 4.6.4 for a description of the Memory Window.

Format codes also give you control over the number and size of multiple pieces of data to display beginning at a particular address. The debugger accepts format codes in the following form:

```
[count] style [size]
```

Count is the number of times to apply the format style *style*. *Size* indicates the number of bytes to be formatted. Both *count* and *size* must be numbers, although you may use **c** (char), **s** (short), **i** (int), and **l** (long) as shorthand for *size*. Legal integer format sizes are 1, 2, and 4; legal float format sizes are 4 and 8.

For instance:

```
initval/4xs
```

displays four, hexadecimal two-byte memory locations starting at the address of `initval`.

With format codes, you may view the contents of memory addresses on the screen. For instance, to dump the contents of an absolute memory address range, you must think of the address being a pointer. To show the memory contents you use the C language indirection operator `*`. Example:

```
*0x4000/2x4
0x4000 = 0x00DB0208 0x5A055498
```

This command displays in hexadecimal two long words at memory location 0x4000 and beyond. Instead of using the size specifier in the display format, you can force the address to be a pointer to unsigned long by casting the value:

```
*(unsigned long *)0x4000/2x
0x4000 = 0x00DB0208 0x5A055498
```

To view the first four elements of the array `table` from the `demo.c` program, type:

```
table/4d4
table = 1      1      2      6
```

This command displays in decimal the first four 4-byte values beginning at the address of the array `table`.

By typing the a space followed by a carriage return you can advance and see the succeeding values in the same format:

```
[Enter]
0x11 = 24  120  720  5040
```

You may recognize that the array `table` contains the factorials for the integers 0 through 7.

Displaying memory in this way is particularly effective when you have two-dimensional arrays. In this case you can display each row by specifying the appropriate count. For instance, if `myarr` is defined as `int myarr[5][8]`:

```
myarr/8ds
```

displays the values for the eight elements in the first row of `myarr`. Typing the carriage return repeatedly then display subsequent rows in the same format.

To scroll back in memory, type the ^ (caret) sign:

```
^
0x9 = 1 1 2 6
```

FUNCTION: Display value(s) at previous memory location.

COMMAND: ^

6.3.5 **DISPLAYING MEMORY ADDRESSES**

The **f** command lets you specify in which notation CrossView Pro displays memory addresses. It takes the same arguments as the `printf()` function in C.

FUNCTION: Specify memory address notation.

COMMAND: **f** [*printf-style-format*]

For instance, if you wish to display all memory addresses in octal, type:

```
f "%o"
```

Now all addresses appear in octal. To return to the default hexadecimal, type:

```
f "%x"
```

Using the **f** command without an argument also returns to hexadecimal address display.

6.4 DISPLAYING DISASSEMBLED INSTRUCTIONS

To show disassembled instructions:



From the **View** menu, select **Source | Disassembly** to open the Disassembly Source Window.



Use the `/i` format switch to display disassembled code in the Command Window.

By using an address and the `/i` format it is possible to display disassembled code at any point. Suppose you wish to see how the `factorial()` function has been compiled. One method would be to examine the instructions displayed as you single step through a program at the assembly language level. There is however a quicker method that does not require you to execute the instructions. Type:

```
factorial/10i
```

This command displays the first ten assembly language instructions of `factorial()`. Remember that in C a function's name is also its address. Thus `factorial` is the address of the function `factorial()`.

Note that CrossView Pro keeps track of variable and function names for you in the disassembled code. You can also disassemble from the current execution position by using the program counter:

```
$pc/5i
```

This command disassembles five assembly language instructions from the current execution line.

You can display disassembled code for any function:

```
main#56/7i
```

disassembles seven instructions from line 56.

See also the **ei** command for displaying disassembly in a window.

Labels in Disassembly

To show labels in disassembly:



From the **Settings** menu, select the **Source Window Setup...** to open the Source Window Setup dialog box and enable the **Symbolic disassembly** check box.



Turn the `$symbols` special variable "ON" by typing the following command in the Command Window:

```
opt symbols=on
```

6.4.1 INTERMIXED SOURCE AND DISASSEMBLY

To show intermixed source and disassembly:



From the **View** menu, select **Source | Source and Disassembly** to open the Source and Disassembly Window.



Use the `/I` format switch to display intermixed C and disassembled code in the Command Window.

The `/I` format works exactly as the `/i` format, except CrossView Pro intermixes the pseudo-assembly listing with the original C source. This feature is often helpful in displaying long portions of code.

Auto Switch between Source and Disassembly

To automatically switch between source and disassembly window depending on the presence of symbols:



From the **Settings** menu, select the **Source Window Setup...** to open the Source Window Setup dialog box.

Enable the **Show assembly when SDI is missing** check box.



Turn the `$autosrc` special variable "ON" by typing the following command in the Command Window:

```
opt autosrc=on
```

6.5 THE STACK

During debugging, you frequently find yourself lost or unable to pinpoint your location through a series of function calls. The **stack** helps you with the problem by recording the return addresses of all functions you have passed through. CrossView Pro can use this information to reconstruct the path to your current location.

The following diagram shows the structure of the stack.

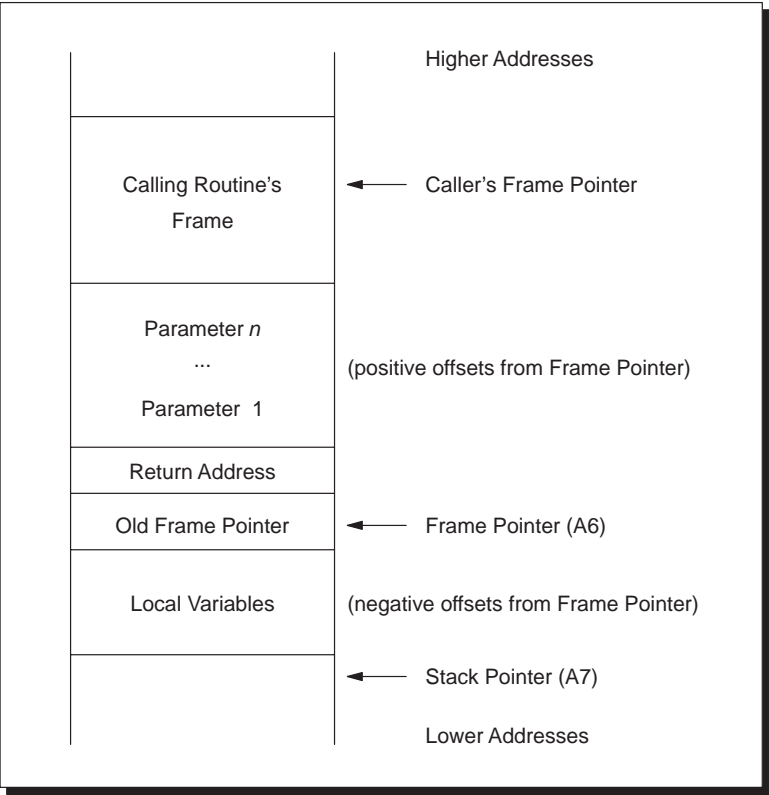


Figure 6-1: Stack frame layout

6.5.1 HOW THE STACK IS ORGANIZED

Whenever your program calls a function, the calling function pushes the arguments (in reverse order) onto the stack.

The function's prologue creates its own stack frame, by reserving enough room on the stack for any local variables and temporary data storage, and saves the return address after jumping to the function's code. When the function is finished, the epilogue code performs the above steps in reverse order: it dismantles the local frame and jumps to the return address.

The compiler may optimize some of the function frames away. The **-do** (disable all optimizations that interfere with debugging) compiler switch ensures that functions always have frames on the stack, thus allowing stack traceback to work.

6.5.2 THE STACK WINDOW

The Stack Window shows the current contents of the stack after the program has been stopped. This window helps you assess program execution and allows you to view program values. You can also set breakpoints for different stack levels from this window, as described in the chapter *Breakpoints and Assertions*.

The Stack Window displays the following information for each stack level:

- The name of the function that was called
- All parameters specified to the function
- The line number in the source code from which the function was called

Each stack level shown in the Stack Window is displayed with its level number first. The levels are numbered sequentially from zero. That is, the lowest/last pushed level in the function call graph is always assigned zero.

When you first see stack information, the lowest level appears against a darker background than the other lines in the window. The marked line in the Stack Window is the **selected stack level**, meaning that this line is selected for window operations. You can change the selected stack level by clicking on a different line.

Checking the Stack from the Command Window

The stack information is also accessible from the Command Window with the **t** and **T** commands. The **t** command reconstructs the program's calling path. For instance, if you stepped into the function `factorial()` and issue a **t** (trace) command:

```
t
```

CrossView Pro displays:

```
0 factorial(num=0) [demo.c:105]
1 main() [demo.c:59]
```

The numbers to the left indicate the depth of each function on the stack. The function at the zero stack level is your current function. CrossView Pro tells you the line number where the function was called ([demo.c:line_nr]) and the value of the argument passed (num=value). With this information it is fairly easy to reconstruct your calling path, and see what parameter values your functions have received.

FUNCTION: Trace stack to reconstruct program's calling path.

COMMAND: **t**

There is a slight variation on the **t** command called the **T** command. The two are identical, except that the **T** command also displays the local variables for each function. For instance:

```
T
0 factorial(num=0) [demo.c:105]
    locvar = 'x'
1 main() [demo.c:59]
    loopvar = 0
    sum = 0
    cvar = '\xff'
```

FUNCTION: Trace stack and display local variables.

COMMAND: **T**

6.5.3 LISTING LOCALS AND PARAMETERS OF A FUNCTION

As mentioned in the previous section, CrossView Pro displays all parameters of a function. You can view the local variables and parameters of any single function active on the stack. To do this:



Follow these steps:

- Open up the Expression Evaluation dialog box by clicking on the **New Expression** button from the toolbar or selecting **Evaluate Expression...** from the **Data** menu.
- Click on the **Browse...** button.



In the Command Window, use the **l** (lowercase L) command.

For example, assuming you are still in `factorial()`, issue an **l** command:

```
l factorial
num = 0
locvar = 'x'
```

You can accomplish the same task by specifying the stack depth instead of a function name:

```
l 0
```

6.5.4 LOW-LEVEL VIEWING THE STACK

You can directly view the contents of the stack. Although CrossView Pro provides several high level methods of tracing functions on the stack, you can view its contents directly with the frame pointer special variable, `$fp`. For instance, the command:

```
$fp[0]/4x1
```

displays the four one-byte values in hexadecimal to which the frame pointer points. Notice that the stack frame is not really an array, but by pretending it is, you can display the memory much as you did with the `table` array. Refer to the *Accessing Variables* section in this chapter for more information.

6.6 TRACE WINDOW



C level trace is not available for all execution environments. Please check the Addendum for details.

The Trace Window displays the most recently executed lines of code each time program execution stops. CrossView Pro automatically updates the Trace Window each time execution halts, as long as the window is open.

For each executed line of code, the Trace Window displays:

- The name of the source file
- The name of the function
- The line number and corresponding source code
- The window shows all the code executed since the the last time the program halted.

6.6.1 TRACE WINDOW SETUP

The Trace Window's only function is to display the contents of the emulator's/ simulator's trace buffer. The only operation you can perform in this window that directly affects the contents is to set the maximum number of instructions in the display.



To set the displaying limit, select the **Initialization** tab in the **File | Options...** dialog. You can change the maximum number of C-Trace machine instructions to fetch from the execution environment's trace buffer and the maximum number of trace output lines in the Trace Window.



To view the most recently executed source statements from the Command Window, use the **ct** command preceded by the number of machine instructions you want to list. For example, to view the last source lines corresponding to the last ten machine instructions, enter:

```
10 ct
```

FUNCTION: Display in the Command window the most recently executed C statements.

COMMAND: *number* **ct**



To activate the source level trace window:



From the **View** menu, select **Trace | Source Level** to view the Trace Source Window.



You can view the last machine instructions executed with the **ct i** command. For example:

15 ct i

displays the last 15 machine instructions in disassembled form in the Command Window.

FUNCTION:	Display the most recently executed machine instructions.
COMMAND:	<i>number</i> ct i

To activate the instruction level trace window:



From the **View** menu, select **Trace | Instruction Level** to view the Trace Instructions Window.



You can view a raw trace with the **ct r** command. For example:

20 ct r

displays the last 20 trace frames in the Command Window.

FUNCTION:	Display a raw trace.
COMMAND:	<i>number</i> ct r

To activate the raw trace window:



From the **View** menu, select **Trace | Raw** to view the Trace Raw Window.

6.7 REGISTER WINDOW

The Registers Window shows you the values of internal registers on your target processor.

You can create multiple Register Windows and each Registers Window contains the names and contents of all currently selected registers in the selected register set definition. Values are displayed in hexadecimal format. As long as the window is open, the debugger automatically updates the values when the program stops.



To show the list of current registers and their contents in the Command Window, enter the list registers command (**lr**).

CrossView Pro also supplies the following special variables:

\$sp	stack pointer
\$pc	program counter
\$fp	current frame pointer

for all targets. For more information, refer to the *Command Language* chapter.

6.7.1 REGISTER WINDOW SETUP

You can configure which register set definition with which (and in which order) registers must be displayed in the Register Window; using the **Settings | Register Window Setup...** dialog. Since you can have more than one Register Window, the last active Register Window will be configured when you select this menu item.



To configure a Register Window follow these steps:

- Select a Register Window.
- From the **Settings** menu, select **Register Window Setup...** to view the Register Window Setup dialog box.

The dialog will show the active register set definition and the list of available and selected registers for this particular register set definition.

- You can create a new register set definition by entering an unique register set definition name in the **Name** edit field and using the **Add** button.

- You can delete a register set definition by selecting an item from the defined register set definition list and using the **Delete** button. Note that when you delete a register set definition, any Register Window displaying a deleted register set will be closed.
- You can select a register set definition by selecting an item from the defined register set definition list. The list of available and selected registers will be updated according to the configuration of the selected register set definition.

Once you have selected a register set definition, follow these steps to configure this register set definition:

- You can add registers to the list of selected registers by selecting registers from the list of available registers by highlighting those registers in the left list box and using the **Add->** or **Add All** button or by double-clicking on the register you want to add.
- You can remove registers from the list of selected registers by highlighting those registers in the right list box and using the **Remove <-** or **Remove All** button, or by double-clicking on the register you want to remove.
- By using the **Move Up** and **Move Down** buttons you can change the display order of the selected registers in the Register Window.

CrossView Pro automatically updates all Register Windows and places the registers in each Register Window starting at the top-left position on one line, wrapping to the next line if the next register does not fit.

6.7.2 EDITING REGISTERS

CrossView Pro lets you change the contents of registers in a simple and direct manner.



Follow these steps:

- In the Register Window, click on the register value you wish to edit. In-situ editing will be activated.
- Specify the new value in the edit control and hit the **Enter** key.

If the edited value is not acceptable, the debugger will emit an error message and reset the old value.

When in-situ editing is active, you can use the **Tab** key to move the edit field to the next register value or use the **Shift+Tab** key combination to move the edit control to the previous register. Use the **Esc** key to cancel in-situ editing. When a register is not in view the contents of the Register Window will be updated automatically.



You can enter any expression in the Registers Window.

Registers which can be edited symbolically have a special marker just before the register name. You can click on this marker to activate the Assign Register Symbolically dialog.



To access registers from the Command Window, use the **\$** designation and the register name in the format:

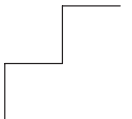
\$register = value



CODE AND DATA

CHAPTER 7

BREAKPOINTS AND ASSERTIONS



7 | CHAPTER

You can use breakpoints to stop program execution at specified locations and return control to the user. An assertion is a number of statements executed by the debugger each time the target executes a program line. Use assertions to track down bugs, the cause of which is very hard to find.

7.1 INTRODUCTION TO BREAKPOINTS

Breakpoints halt program execution and return control to you. There are several types of breakpoints: code, data, instruction count, cycle count, timer and sequence. A code breakpoint halts the program on a particular statement or instruction; a data breakpoint stops the program when a particular memory address (or range of addresses) is accessed; an instruction count breakpoint halts the program after a specified number of instructions have been executed; a cycle count breakpoint stops the program after a number of CPU cycles; a timer breakpoint stops the program after a number of micro seconds or ticks and sequence breakpoints stop the program when a number of breakpoints are hit in a specified sequence.

Data breakpoints, instruction count breakpoints, cycle count breakpoints and timer breakpoints are not available for all execution environments, please check the Addendum.

7.1.1 CODE BREAKPOINTS

A **code breakpoint** is set on a line in the code and makes the program halt exactly before that line executes. When you define a code breakpoint, you can include four elements:

- A *count*, which is the number of times the breakpoint must be encountered before it stops the program (default is 1).
- A *reset count*, which is the value assigned to the *count* after the program has stopped on a breakpoint (default is 1).
- A *name*, which is the symbolic name you can associate with a breakpoint.
- A list of commands, which will be executed when the program hits the breakpoint.

In the Source Window, a green colored toggle shows that no breakpoint is set. A red colored toggle shows that a breakpoint is installed. An orange colored toggle indicates an installed but disabled breakpoint. If coverage is enabled, coverage markers are present to the right of the breakpoint toggles. An executed line is marked and not executed lines are not marked.

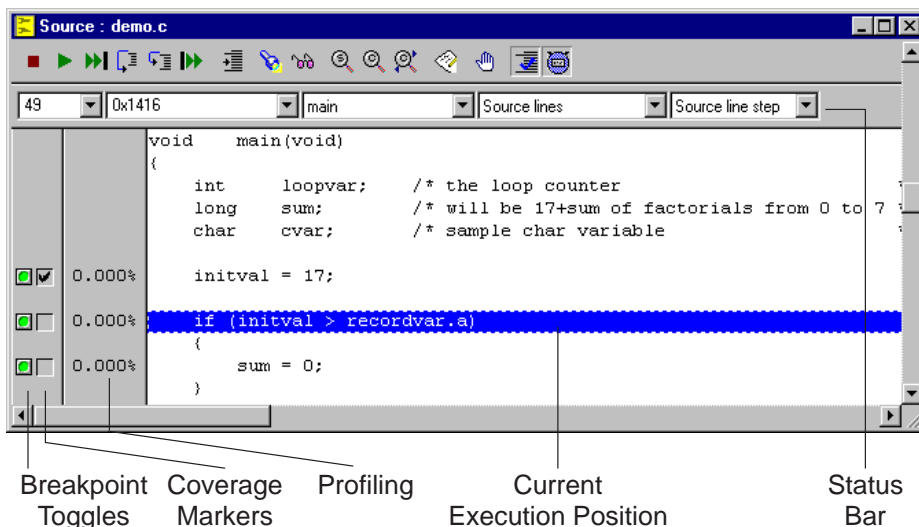


Figure 7-1: Code Breakpoint

Permanent/Temporary Code Breakpoints

Code breakpoints can be: *permanent* or *temporary*. A permanent breakpoint exists until explicitly deleted. A temporary breakpoint only exists until it stops the program once.

Probe Point Code Breakpoints

A breakpoint can be treated as a probe point. When a probe point breakpoint is hit, the associated commands are executed and program execution is continued. Probe points are used with File I/O simulation and sequence breakpoints.

How CrossView Pro Sets Code Breakpoints

CrossView Pro depends on the symbol table for information about how machine instructions map to lines of source. In general, the C compiler issues line symbols at the start of each statement or line, whichever comes first. This can lead to some surprising results. If you look carefully, you can tell on which line CrossView Pro set the breakpoint, since CrossView Pro tells you on which line the program stopped, a line that may be different from the one you expected. To find out what happens if you install a code breakpoint, use single stepping and watch the order in which the source lines print out.

Multiple Statements on a Single Source Line

If you frequently include multiple statements on a single line in your source code, you may have difficulties setting code breakpoints at certain locations. For instance, suppose you have a source line containing:

```
a = 0; b = 1
```

Suppose you want to halt execution after the assignment to `a` and before the one to `b`. A normal code breakpoint does not work here, because execution stops at the first instruction of the source line. CrossView Pro provides you with the capability of disassembling the code and inserting breakpoints at the machine level. You can use the Assembly Source Window or the Intermixed Source Window to spot the right location.

For more information on machine level breakpoints, see below.

Setting Breakpoints for Multi-line Statements

Code breakpoints have a special behavior for multiple-line statements, such as a multiple-line `if`. In an `if` clause, a line symbol is generated at the beginning of the list of conditions, and the other lines of the conditions are generally associated with the first line of the clause. In an `if-then-else` construct, the `}` character before the `else` is associated with the branch-around to the end of the statement.

Consider the following example:

```
22: if ((a == b)&&  
23:  (c == d)) {  
24:  x = 2;  
25: } else {  
26:  y = 3;  
27: }
```

If you try to set a code breakpoint at line 23, CrossView Pro sets the breakpoint on the preceding statement. If you try to set a breakpoint on line 22, CrossView Pro highlights line 23. If you set a breakpoint on line 25, it hits after the assignment to `x`, but before the jump to line 27. Notice that it is not hit unless the `if` clause is true. In other words, a breakpoint on line 25 is really a break on the `}`, not on the `else {`. The same behavior applies when the `else {` statement is on the next source line.

Breakpoints and For Loops and While Loops

The code generated for a C `'for'` statement has three parts: the initialization; the body of the loop; and the increment, test, and branch. The initialization part and the increment, test, and branch are different parts of code, but are both associated with the `'for'` statement itself. For example consider:

```
99: for (i = 0; i < 9; i++) {  
100:   myfunction(i);  
101: }
```

A breakpoint placed on line 99 will only be hit once, because it is hit at the initialization code. The code for the increment, test, and branch is associated with line 101, not 99, as you might expect.

The same applies to `'while'` loops.

Breakpoints and Emulator Mode

Upon entering emulator mode, the debugger removes any breakpoints it established in the target code. Removing breakpoints ensures that you can access unmodified target code. When emulator mode ends, CrossView Pro reestablishes breakpoints as necessary.

As long as you avoid the debugger's own breakpoint trap, you may establish arbitrary breakpoint conditions while in emulator mode. These will not be removed by CrossView Pro and thus remain active, however, after you exit emulator mode. If one of these breakpoints is hit during normal debugging, CrossView Pro will issue a message such as:

```
Stopped on breakpoint not set by debugger.
```

System Startup Code

It is possible (for example, by using the **si** command) to debug system level startup code that initializes the target environment. You should not use any global variables in CrossView Pro expressions until the data area has been initialized. CrossView Pro assertions and other CrossView Pro commands that examine C variables may deliver erroneous information or cause memory access errors if used before the C environment is established.

7.1.2 DATA BREAKPOINTS

A *data breakpoint* instructs the execution environment to watch a particular data address or address range and halt execution if the program reads from or writes to that address. Data breakpoints are a powerful feature for tracking the use, and possible misuse, of pointers, global variables and memory mapped I/O ports.



Data breakpoints are not available for all execution environments, please check the Addendum.

When setting a data breakpoint, you can specify whether the breakpoint stops the program when data is read from, written to, or both.

Data breakpoints are implemented in hardware. As a consequence, the number of allowable data breakpoints is limited by your execution environment. A simulator does not have these restrictions. Refer to the environment-specific Addendum for more information.

On the 68K/ColdFire family of microprocessors, some *skidding* may occur when you use data breakpoints. Skidding means that the execution environment executes the next several instructions after the data breakpoint stops the program. This occurs because the microprocessor executes instructions in its cache before stopping. You should know, therefore, that the data breakpoint may not stop the program at the precise line of code where the break occurred.

You may set a data breakpoint on a local variable, but only if the local variable is active. CrossView Pro notifies you when program execution passes beyond a local variable's scope, and a breakpoint set on such a variable is deleted automatically. Data breakpoints for static variables do not have this restriction.

Note that any local variables placed in registers cannot be tracked with data breakpoints. In this case, you must use an assertion. Refer to the *Assertions* section later in this chapter for more information.

7.1.3 LISTING BREAKPOINTS

To see a listing of all of the currently defined breakpoints:



From the **Breakpoints** menu, select **Breakpoints...** to view the Breakpoints dialog box.



In the Command Window, enter the **l b** or **B** commands. The list appears in the Command Window.

For example entering the **B** command can result in:

```
B
0 ena CODE main (CODE:0x78) 2/2
```

The breakpoint's number (used when deleting breakpoints) is listed first, then if it is enabled or disabled, then its type: such as **CODE** for code breakpoints and **DATA** for data breakpoints. Next, CrossView Pro lists the function and/or address, its count and reset count, and finally any attached commands enclosed by { and }.

FUNCTION: View all breakpoints in the Command window.
COMMAND: **B**

CrossView Pro decrements the count each time the breakpoint is hit. When the breakpoint's count reaches 0, CrossView Pro halts the program.

7.2 SETTING BREAKPOINTS

You may set a code or data breakpoint by:

- Using the mouse to open the Breakpoints dialog box.
- Using the mouse in the Source Window.
- Using the Stack Window.
- Using the command line in the Command Window.

When you set a new breakpoint using the mouse, without using the Breakpoint dialog box, the type is always permanent, the count 1 and the location corresponds to the current viewing position, if the Source Window is open. These variables are described in more detail below.

Setting Breakpoints from the Menu

To set a breakpoint from the menu, select **Breakpoints...** from the **Breakpoints** menu to view the Breakpoints dialog box. From this dialog box, you can define several types of breakpoints.



To set a code break point at line number # of the C source, click the **Add >** button and select **Code Breakpoint....** Click the **Break At...** button, choose a C module (for example demo.c) and click the **OK** button. Now you can enter a line number to set the breakpoint at.

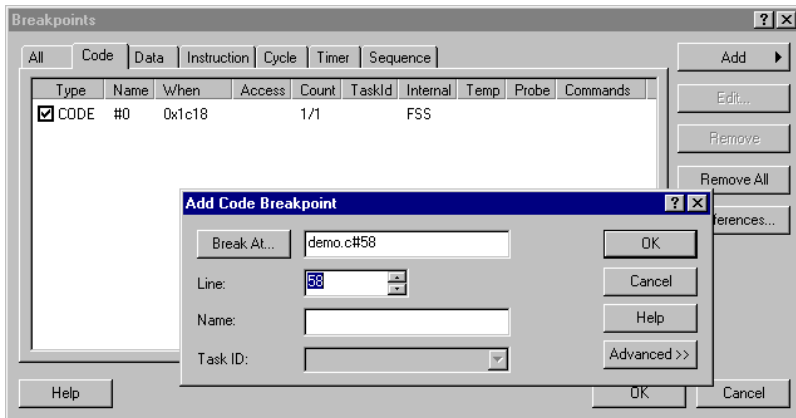


Figure 7-2: Breakpoints dialog box

The last entry of the list is always empty. Select it to start defining a new breakpoint.

Setting Breakpoints from the Source Window

You can set or remove a code breakpoint directly from the Source Window by clicking on:

- The breakpoint toggle next to the source lines in the Source Window.

To set data breakpoints use the menu as described above.

Setting Breakpoints from the Stack Window

See the section *Up-level Breakpoints* later in this chapter.

Setting Breakpoints from the Command Window

You can set a code breakpoint from the Command Window using the **break code** command or the **b** command, and set a data breakpoint using the **break data** command. Several options are available after these commands.



See the **break** command in the *Command Reference* for detailed information.

For example, the following command sets a code breakpoint at the address specified by function `main`:

```
break code main
```

To set a code breakpoint at a specific source line, you can enter a breakpoint address in the form: *filename#line* after the **break** command, or you can specify a line number, followed by the **b** command and any commands you want to attach to the breakpoint. For example, to set a code breakpoint at line 51 in your source, enter:

```
break demo.c#51
```

or

```
51 b
```

If you do not specify a line number, a breakpoint will be set at the current viewing position.

FUNCTION:	Set a code breakpoint.
COMMAND:	break [code] <i>address</i> [, <i>option</i>]...
FUNCTION:	Set a code breakpoint.
COMMAND:	[<i>line_number</i>] b [<i>commands</i>]

To set a data breakpoint, you must specify the **break data** command, followed by an address, followed by any commands you want to attach to the breakpoint. There are three types of data breakpoints:

- A data read breakpoint to see if a variable is read from (**break data address, access_type=r** command)
- A data write breakpoint to watch if a variable is written to (**break data address, access_type=w** command)
- A data read or write breakpoint to check if a variable is either read from or written to (**break data address, access_type=rw** command)

For example, to set a data breakpoint to watch the lowest byte in memory of the global variable `initval`, enter:

```
break data &initval, access_type=w
```

This command instructs CrossView Pro to set a data breakpoint that will halt execution if the program writes to the lowest byte in memory of the variable `initval`. Note that you have to specify the variable's address, otherwise the variable's value is used.

FUNCTION: Set a data breakpoint.

COMMAND: **break data** *address* [*option*]....

7.2.1 DATA BREAKPOINTS OVER A RANGE OF ADDRESSES

You can also use data breakpoints to watch a contiguous range of memory. As with standard data breakpoints, data breakpoints over a range of addresses can be set to watch for reading, writing or both. To set a data breakpoint of this type:



Using mouse and menu:

- From the **Breakpoints** menu, select **Breakpoints...** to open the Breakpoints dialog box.
- Select the data breakpoint you want to edit and click the **Edit...** button, or click the **Add >** button and select **Data Breakpoint...**
- Specify a start address and click on the **Advanced** button.

- Select one of the **Type** options: break on read, write, read or write.
- Specify an end address. The end address is part of the range.



From the Command Window:

- Type **break data** *address*, **end_addr=***end_address*, **access_type=***r* to set a data read breakpoint over a range.
- Type **break data** *address*, **end_addr=***end_address*, **access_type=***w* to set a data write breakpoint over a range.
- Type **break data** *address*, **end_addr=***end_address*, **access_type=***rw* to set a data breakpoint for both reading and writing over a range.

For example, to ensure that the program stops if any of `recordvar`'s fields are either written to or read from:

```
break data &recordvar, end_addr=(int) \
    &recordvar+sizeof(recordvar)-1, access_type=rw
```

FUNCTION: Set a data breakpoint over a range of addresses.

COMMAND: **break data** *address*, **end_addr=***end_address* [*option*]...

7.2.2 TEMPORARY BREAKPOINTS

Breakpoints can be: *permanent* or *temporary*. A breakpoint exists until it is manually deleted. A temporary breakpoint is automatically removed by CrossView Pro after it halts the program once.

To set a temporary breakpoint:



Follow these steps:

- Open the Source Window by selecting **Source | Source lines** from the **View** menu.
- Open the Breakpoints dialog by selecting **Breakpoints...** from the **Breakpoints** menu.
- Click on the **Add >** button and select **Code Breakpoint...**
- Enter an address in the **Break At** field and click on the **Advanced** button.
- Enable the **Remove when hit** check box in the **Behavior** field.

- Click on the **Continue** button in the Source Window when the program halts. This removes the temporary breakpoint at the viewing position and the program continues.
- Alternatively, scroll to the line that you want to stop at and click once (to establish a viewing position). From the **Run** menu, select **Run to Cursor** to continue execution until you reach this temporary breakpoint.



From the Command Window:

- Type **break code** *address*, **temporary=true** to set a temporary code breakpoint.
- Type the **C** command followed by a line number, to set a temporary breakpoint at a line number.

For example,

C 51

sets a temporary breakpoint at line 51 and resumes execution at the current execution position.

FUNCTION: Set a temporary code breakpoint.

COMMAND: **break code** *address*, **temporary=true** [*option*]...

7.2.3 BREAKPOINT NAMES

You can associate a symbolic name with a breakpoint. You can then use this name with the following commands: **break set** and **break delete**. Breakpoint names must be unique and cannot be a number or the word "all". Allowed characters are a-z, A-Z, 0-9 and ' _'.

To assign a name to a breakpoint:



Follow these steps:

- From the **Breakpoints** menu, select **Breakpoints...** to open the Breakpoints dialog box.
- Select a breakpoint to edit and click on the **Edit...** button.
- Alternatively, click on the **Add >** button and select a breakpoint type to create.

- Enter the breakpoint information in the first field, for example an address.
- Enter a symbolic name in the **Name** field.



Use the **name=name** option of the **break** command in the Command Window.

For example,

```
break code 0x1234, name=brk_1
```

sets a code breakpoint at address 0x1234 with the name `brk_1`.

7.2.4 SETTING THE COUNT

CrossView Pro allows you to set a breakpoint's *count*. The count defines how many times you encounter the breakpoint before it halts the program. For example, a breakpoint with a count of 3 means the program stops on the third hit. Each time the breakpoint is hit, CrossView Pro decrements the count. When the count reaches 0, CrossView Pro halts the program, and resets the count to the value of the *reset count*. The default reset count is 1.

To set a breakpoint's count,



Follow these steps:

- From the **Breakpoints** menu, select **Breakpoints...** to open the Breakpoints dialog box.
- When you add or edit a breakpoint, click on the **Advanced** button.
- Enter a breakpoint's count in the **Breakpoint count** field.
- Enter a reset count in the **Reset count** field.



From the Command Window,

- Use the **count=** argument with the **break** command to set both the current count and the reset count.
- Use the **curr_count=** and/or **reset_count=** arguments with the **break** command to set the current count and the reset count separately.

For example, suppose you have a breakpoint set at address 0x59 of your source code. The first time the program halts at address 0x59, enter:

```
break code 0x59, curr_count=3, reset_count=4
```

This command sets the breakpoint's count to 3 and the reset count to 4. You can observe a breakpoint's current count and reset count when you list the breakpoints in the Command Window with the **lb** command.

FUNCTION: Set the count and reset count for a breakpoint.

COMMAND: **break** *type address, count=count*

FUNCTION: Set the count and reset count for a breakpoint separately.

COMMAND: **break** *type address, count=count, reset_count=reset_count*

7.2.5 SEQUENCE BREAKPOINTS

A sequence breakpoint is a special kind of breakpoint. Only if other breakpoints are hit in a specified order, the sequence breakpoint itself will hit.

To hit a breakpoint without halting the program, the breakpoint in the sequence must be specified as a *Probe point*. When a probe point is hit, the associated commands are executed and program execution is continued.

When all specified probe points are passed in the logical sequence you specified, the program stops at the last breakpoint in the sequence.

To set a sequence breakpoint:



Follow these steps:

- From the **Breakpoints** menu, select **Breakpoints...** to open the Breakpoints dialog box.
- Click on the **Add >** button and select **Sequence Breakpoint...**

- Click the **Sequence...** button to open the Edit Sequence Breakpoint dialog box.
- Select a breakpoint from the **Available Breakpoints** list box and add it to the sequence with the buttons **ADD**, **AND** or **OR**. Use the **NOT** button for a breakpoint that should *not* be passed. All breakpoints you add to the list must be enabled, otherwise the sequence breakpoint itself will not hit.



From the Command Window:

- Use the **sequence** argument of the **break** command with a list of breakpoints to specify the sequence.

For example,

```
break sequence (0)(1 and 3)(2)
```

In this example, the sequence breakpoint hits when probe point 0 is hit first, then 1 and 3 are hit in any order, and finally probe point 2 is hit.

FUNCTION: Set a sequence breakpoint.

COMMAND: **break sequence** *sequence* [, *option*]...

7.3 DELETING BREAKPOINTS

You can delete a breakpoint directly from the source code, using the menu items, or through the Command Window. To see a list of active breakpoints, select **Breakpoints...** from the **Breakpoints** menu or use the **lb** command in the Command Window.

To delete a code breakpoint:



Click on the corresponding red breakpoint toggle next to the source line in the Source Window. This deletes the code breakpoint and the breakpoint toggle turns green.



You can also follow these steps:

- From the **Breakpoints** menu, select **Breakpoints...** to open the Breakpoints dialog box. This box contains a remove function.
- Select the Breakpoint from the list.

- Click the **Remove** button.



Use the **break delete** *breakpoint_number* | *name* command in the Command Window. You need to know the breakpoint's number or name for this command.

For example, to delete the breakpoint numbered 1, enter:

```
break delete 1
```

FUNCTION: Delete a breakpoint.

COMMAND: **break delete** *breakpoint_number*
break delete *breakpoint_name*

To clear all the breakpoints in the program, type:

```
break delete all
```

```
Do you want to delete all breakpoints?y
```

FUNCTION: Delete all breakpoints.

COMMAND: **break delete all**

7.4 ENABLING/DISABLING BREAKPOINTS

You can enable or disable a breakpoint directly from the source code, using the menu items, or through the Command Window. To see a list of active breakpoints, select **Breakpoints...** from the **Breakpoints** menu or use the **lb** command in the Command Window.

To enable or disable a code breakpoint:



Follow these steps:

- From the **Breakpoints** menu, select **Breakpoints...** to open the Breakpoints dialog box. This box contains an edit function.

On Windows:

- In the list of breakpoints toggle the check box in front of the breakpoint to enable or disable the breakpoint.

On UNIX:

- Select the breakpoint form the list.
- Click the **Enable** or **Disable** button to enable or disable a breakpoint.



Use the **break enable** or **break disable** command in the Command Window to enable or disable a breakpoint. You need to know the breakpoint's number or name for these commands.

For example, to disable the breakpoint numbered 1, enter:

```
break disable 1
```

FUNCTION: Disable a breakpoint.

COMMAND: **break disable** *breakpoint_number*
 break disable *breakpoint_name*

To enable the breakpoint numbered 1, enter:

```
break enable 1
```

FUNCTION: Enable a breakpoint.

COMMAND: **break enable** *breakpoint_number*
 break enable *breakpoint_name*

7.5 BREAKPOINT COMMANDS

CrossView Pro allows you to attach commands to code and data breakpoints. When execution halts at a breakpoint, CrossView Pro executes the commands. Valid commands are almost any C statements and CrossView Pro commands, giving you a very powerful tool for manipulating a debugging session. To do this:



Follow these steps:

- From the **Breakpoints** menu, select **Breakpoints...** to open the Breakpoints dialog box.

- Select an existing breakpoint from the list and click on the **Edit...** button or click on the **Add >** button and select a type of breakpoint you want to add.
- Enter the breakpoint information in the first field, for example an address.
- Click on the **Advanced** button. Note that the button is only visible when there is more information available on the breakpoint.
- Click in the **Commands edit** area.
- Type in the commands to be executed when the breakpoint is reached.



You do not need to enclose a group of commands in brackets. However, each individual command must be delimited by a semicolon.

Figure 7-3: Breakpoint Commands



Type the commands, enclosed in brackets and delimited by semicolons, after **commands=** argument of the **break** command in the Command Window.

For instance, suppose you want a program to stop at a breakpoint, display a variable's value, and resume execution all in one stroke. To perform this function, you need to attach the appropriate commands to a breakpoint. Enter:

```
break code main, commands={initval;C}
```

This places a breakpoint at address `main`. When execution stops at the breakpoint, CrossView Pro displays the value of `initval` and immediately resumes execution.



If you enable the **Probe point** check box, you can omit the **C** command. This is done automatically.

You can attach almost any valid CrossView Pro commands or C statement to breakpoints. This latitude allows you to use breakpoints in powerful ways. Later on you find out how breakpoints can create patches in your program.



CrossView Pro does not check the syntax of attached commands until the breakpoint is hit.

Data breakpoints accept command lists the same way as code breakpoints. For instance, to set a data breakpoint that monitors the lowest byte in memory of the value of `initval`, enter:

```
break data &initval, access_type=w, commands={initval; C}
```

Every time the program writes to the lowest byte in memory of the variable `initval`, this breakpoint halts the program, prints the value of `initval` and continues execution.



For more information on the use of attached commands, see the *Patches* and *Diagnostic Output and Statistical Information* sections later in this chapter.

7.5.1 ATTACHING CONDITIONALS TO A BREAKPOINT

You can pass standard C conditionals to a breakpoint.

For example:

```
break code demo.c#63, commands= {if (initval==17) {C}
{initval/n}}
```

stops the program at line 63, checks to make sure the variable `initval` is 17, and resumes execution if it is. If `initval`'s value does not equal 17, CrossView Pro prints the value, and the program remains halted.

7.5.2 ATTACHING MACROS TO A BREAKPOINT

You can attach any currently defined macro to a breakpoint in a command list. For example, suppose you define a macro named `rg` that checks the value of the variable `initval`. The command to define this macro is:

```
set rg "if (initval != 17) {initval/n} {C}"
```

If the value does not equal 17, the macro prints the value and halts the program. Otherwise, execution continues.

You can include this macro at any point by attaching it to a breakpoint. Entering:

```
break code demo.c#51, commands={rg}
break code demo.c#63, commands={rg}
```

this is a very efficient way to insert the macro with breakpoints at lines 51 and 63.



For more information on macros, refer to *Defining and Using Macros* chapter.

7.5.3 ATTACHING STRINGS TO A BREAKPOINT

You can attach strings to a breakpoint's command list. This feature is useful for placing comments and reminders within your breakpoints. Attaching a string to a breakpoint also eliminates the need for diagnostic `printf()` statements in your compiled code.

For example, you could place a breakpoint on line 49 such as:

```
49 b {"Passed line 47\n";C}
```

Whenever the breakpoint on line 49 is hit, CrossView Pro prints the string and continues execution.

7.6 SUPPRESSING BREAKPOINT MESSAGES

Whenever a breakpoint is hit, CrossView Pro displays in the Command Window, the name of the function, line number and file in which the breakpoint appears. You can suppress this information by setting breakpoint "silent" mode. In the silent mode, the current location is not printed out.

To set silent mode you can use the **Q** (for quiet) command as part of the command attached to a breakpoint definition.

Pass the **Q** command to a breakpoint first. For example:

```
51 b {Q; initval = 5}
```

stops the program on line 51, but does not print a message stating where the break occurred.

7.7 UP-LEVEL BREAKPOINTS

Up-level breakpoints are breakpoints set at the entrance and/or exit of functions. Basically, up-level breakpoints are code breakpoints that are directly connected to the current HLL stack handling.

To see the current HLL stack, open the Stack Window or enter the **t** command in the Command Window.

You can set up-level breakpoints via the Stack Window or in the Command Window. You cannot set up-level breakpoints in the Source Window:



Double-click on the function in the Stack Window to install a stack breakpoint after the function call.



You can also follow these steps:

- Click on the function in the Stack Window.
- From the **Breakpoints** menu, select either **Stack Breakpoint | After Call to Function** or **Stack Breakpoint | At Function Entry**

You have the option of setting the breakpoint before (function entry) or after (up-level) a selected function.



All breakpoints set through the Stack Window are temporary by default. To make a breakpoint permanent, select **Breakpoints...** from the **Breakpoints** menu to open the Breakpoints dialog. Select the breakpoint you want to edit and click on the **Edit...** button. Click on the **Advanced>>** button and disable the **Remove when hit** check box.



In the Command Window, use the following commands:

Command	Function	Type
bU	Sets breakpoint <i>after</i> call to function	temporary
bu	Sets breakpoint <i>after</i> call to function	permanent
bB	Sets breakpoint at <i>beginning</i> of function	temporary
bb	Sets breakpoint at <i>beginning</i> of function	permanent

For example, suppose you have accidentally single-stepped into a function called `factorial()`. If you do not want to single step through the function, an up-level breakpoint can help you. Enter:

bU

The **bU** command sets a temporary breakpoint after return of the function. Now, instead of having to single step all the way through the function, you can start continuous execution, which stops when it hits the new breakpoint at the function's return. Note that it makes no difference whether the function has several possible points of return; the up-level breakpoint works at all points of return. Note that when the function that contains the breakpoint is called from one of the functions that are located below it on the stack, the execution may be stopped before returning at the desired stack level, for example with recursive functions.

When setting up-level breakpoints from the Command Window, you can specify how deep in the stack the function's address is located. For example, if you are two functions down from the `main()` program, enter:

2 bU

This command breaks when you return to the top level of the call graph.

FUNCTION: Set a temporary breakpoint after call to function.

COMMAND: `[stack] bU [commands]`

FUNCTION: Set a permanent breakpoint after call to function.

COMMAND: `[stack] bu [commands]`

FUNCTION: Set a temporary breakpoint at function entry.

COMMAND: `[stack] bB [commands]`

FUNCTION: Set a permanent breakpoint at function entry.

COMMAND: `[stack] bb [commands]`

7.8 PATCHES

A *patch* is a means of using CrossView Pro to change the execution of your program without recompiling. Patches involve manipulating breakpoints to skip code, include code, or replace existing code with new code.

Basically, a patch is a breakpoint with certain associated commands that enable you to alter program execution. This capability is a useful debugging tool.

You can associate the commands used to patch code with a breakpoint through either the Command Window or through the Commands edit box in the Breakpoint dialog box. The examples below set breakpoints using CrossView Pro commands typed in the Command Window. You can also set breakpoints in the **Breakpoints | Breakpoints...** dialog. In this case the commands between the brackets are entered into the Command edit area.

7.8.1 PATCHING CODE OUT OF A PROGRAM

To patch code out of a program, you can set a breakpoint that changes the execution position. For instance, suppose you want to patch an infinite loop out of your source.

```
78:  while (loopvar)
79:  {
80:      sum = sum + 1;
81:  }
82:
83:  sum = sum + 5;
```

On line 78, place a breakpoint that jumps to line 83, effectively bypassing the loop. In the Command Window, enter:

```
78 b {g 83; c}
```

This creates a breakpoint on line 78 that does nothing more than move the execution position beyond the loop and issue a **C** command. Remember that the breakpoint on line 78 is hit before the **C** statement on that line executes.

7.8.2 PATCHING CODE INTO A PROGRAM

You can also patch code into a program by just including the code in the breakpoint command. For example, suppose you want to add an equation with the variable `loopvar`.

```
78:  while (loopvar)
79:  {
80:      sum = sum + 1;
81:  }
82:
83:  sum = sum + 5;
```

In the Command Window, enter:

```
78 b {loopvar = 0;C}
```

This command halts execution at line 78, adds the statement `loopvar=0` to the program, and continues execution.

7.8.3 REPLACING CODE IN A PROGRAM

Finally, you can combine the two techniques described above to replace code in a program. For instance, suppose you want to replace an infinite loop with new code.

```
78:  while (loopvar)
79:  {
80:      sum = sum + 1;
81:  }
82:
83:  sum = sum + 5;
```

In the Command Window, enter:

```
78 b {Q; if (sum<100) {sum++; g 78; C} {g 83; C}}
```

This command sets a breakpoint that halts execution (quietly) at line 78 and inserts an `if` statement into the program. If `sum` is less than 100, `sum` increments and line 78 executes again. If `sum` equals 100, CrossView Pro moves the execution position to line 83 (beyond the infinite loop) and resumes execution.

7.9 DIAGNOSTIC OUTPUT AND STATISTICAL INFORMATION

Breakpoints with attached commands allow you to report on various variables while the program executes. In the past, one inefficient method of tracking variables was to litter code with `printf()` statements. Using breakpoints makes that process unnecessary.

For instance, suppose you want to keep track of the variable `loopvar` at line 59 of a program. Install a breakpoint with the following command:

```
59 b {Q; loopvar; C}
```

The breakpoint halts the program, prints the value of `loopvar`, and resumes execution. The **Q** command suppresses the listing of where the break occurred. This breakpoint does not affect the source code and no recompilation is necessary.

Using special variables, you can also keep statistics about your program, such as how many times a line of code executes or how many times a variable is accessed.

For example, suppose you want to know how many times line 60 executes. You must define a special variable to keep track of your statistical data, and set a breakpoint to accumulate the data for you.

First, define the special variable. In the Command Window, enter:

```
$test = 0
```

This command defines the special variable `$test` and sets it to zero. For convenience, you can also set a breakpoint at the beginning of the program that initializes `$test`.

Secondly, set a breakpoint at line 60 that increments `$test` and continues execution every time the program hits line 60:

```
60 b {$test++ ; C}
```

7.10 ASSERTIONS

An *assertion* is a collection of debugger commands executed by the debugger after each program line. When you execute a program using assertions, the debugger is in *assertion mode*. Running the debugger in assertion mode is a way of executing continuous control of certain data.

Using assertions, you can have continuous control of certain data and stop program execution if any of the set conditions are fulfilled. In this respect, assertions are similar to data breakpoints. Assertions, however, are more versatile than data breakpoints. For instance, a data breakpoint can only detect when a variable is accessed. An assertion, on the other hand, can check that the variable's value falls within a certain range. Also, an assertion can monitor variables whose values are kept in registers.

The default limit for the number of assertions you can define is 16. It is possible to increase the number of assertions by selecting the **Initialization** tab in the **File | Options...** dialog box. Each individual assertion can be activated or deactivated. In addition, you can also choose to suppress all assertions by turning off the global assertion mode.

Opening the Assertions Dialog Box



From the **Breakpoints** menu, select **Assertions...**

The Assertions dialog box contains scrollable lists of all defined assertions, and provides functions for defining, activating, suspending, editing and deleting assertions.

7.10.1 ASSERTION MODE

The debugger is running in assertion mode when there is at least one active assertion. A program executing in assertion mode is actually being single-stepped very quickly, to ignore breakpoints. Because the program is single-stepping, however, it runs significantly slower than at normal speed.

An Assertion Mode Active checkbox is available that activates all marked (*) assertions. Clear this option if you want to suspend all assertions temporarily. To activate marked assertions:



Open the Assertions dialog box and activate all marked assertions by enabling the **Assertion Mode Active** check box.



In the Command Window, enter the **A** command:

- **A a** — activates assertion mode
- **A s** — suspends assertion mode
- **A** — (by itself) toggles the assertion mechanism

The Global Active state activates all assertions. Globally activating the assertion mode, however, does not change how each assertion is marked.

FUNCTION: Activate assertion mechanism.

COMMAND: **A a**

FUNCTION: Suspend assertion mechanism.

COMMAND: **A s**

FUNCTION: Toggle assertion mechanism.

COMMAND: **A**

7.10.2 DEFINING AN ASSERTION

To define or edit an assertion:



Follow these steps:

- From the **Breakpoints** menu, select **Assertions...** to open the Assertions dialog box.
- Click on the **New...** button to open a text edit dialog box as shown in figure 7-4 to type in commands.

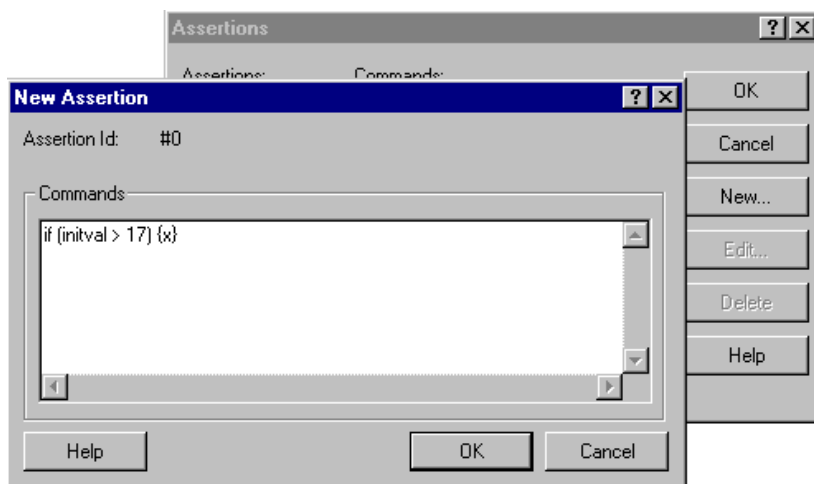


Figure 7-4: Defining Assertions



Use the **a** command followed by a list of commands.

FUNCTION: Create an assertion.

COMMAND: **a** *commands*

Assertions accept standard C statements and certain CrossView Pro commands as arguments.

An assertion usually contains a conditional. For example, suppose you want to create an assertion that watches the value of the global variable `initval` to see that its value does not exceed a certain limit. In this case, you enter in the Assertion dialog box (or into the Command Window after the **a** command):

```
if (initval > 17) {x}
```

This command creates an assertion with the condition that if `initval` exceeds 17, CrossView Pro halts the program. The `{x}` is a special assertion command that tells CrossView Pro to halt the program and return control to you.

7.10.3 EDITING AN ASSERTION

To edit the contents of an assertion:



Follow these steps:

- From the **Breakpoints** menu, select **Assertions...** to open the Assertions dialog box.
- Click on the assertion to edit.
- Click on the **Edit...** button. A text edit dialog box opens allowing you to edit the assertion. Click on **OK** or **Cancel** when finished.



You must delete the specific assertion (section 7.10.5) and define a new assertion (previous section) with the desired command.

7.10.4 ACTIVATING AND SUSPENDING ASSERTIONS

A particular assertion is either *active* or *suspended*. A suspended assertion does not execute before every line, but it retains its definition.

You may find it helpful to use activate and suspend assertion commands in conjunction with code breakpoints, since assertions tend to slow the target program. By attaching commands to a breakpoint to activate and suspend assertions, you can turn assertions on only for certain sections of code where a particular value needs checking. This method can dramatically speed up the program.



From the **Breakpoints** menu, select **Assertions...** and double-click on the assertion's number.



To activate or suspend an assertion from the Command Window, you must know the assertion's number. To see a list of assertions and their assigned numbers:

- Enter **l a**, the list assertions command, in the Command Window.

To activate an assertion:

- Enter *assertion_number* **a a** command. For example:

2 a a *activates assertion 2*

To suspend an assertion:

- Enter the *assertion_number* **a s** command. For example:

2 a s *suspends assertion 2*

FUNCTION: Activate an assertion.

COMMAND: *assertion_number* **a a**

FUNCTION: Suspend an assertion.

COMMAND: *assertion_number* **a s**

7.10.5 DELETING ASSERTIONS

Deleting an assertion removes its definition. It is important to note the difference between suspending an assertion and deleting an assertion: deleting an assertion removes its definition for good, while suspending it retains the definition but prevents its execution.



Follow these steps:

- From the **Breakpoints** menu, select **Assertions...** to open the Assertions dialog box.
- Click on the assertion to delete.
- Click the **Delete** button. Click on **OK** or **Cancel** when finished.



Follow these steps:

- List the assertion numbers with **l a** command in the Command Window.
- In the Command Window, enter the assertion number followed by the **a d** command. For example:

2 a d *Deletes assertion 2.*

FUNCTION: Delete an assertion.

COMMAND: *assertion_number* **a d**

7.10.6 USING ASSERTIONS

You can use assertions for almost any type of debugging task. For example, if you want to check the value of a global variable, `global_val`, during the execution of a certain function, `f()`. A data breakpoint or a straightforward CrossView Pro assertion does not suffice for this task since there is no way to make either method limited to that function's code range. The solution lies in creating an assertion that is active only over a specific range of lines. In this case, you could solve your problem with the following steps:

```
110: void f(void)
111: {
112:     if ( global_flag )
113:     {
114:         ++global_val;
115:     }
116:     else
117:     {
118:         global_val = g();
119:     }
120: }
```



Using the mouse and menu:

1. From the **Breakpoints** menu, select **Assertions...** to open the Assertions dialog box.
2. Click on the **New...** button.
3. Set up the assertion to check the value of `global_val`. Enter:

```
if (global_val == 17) {x}
```

This assertion halts program execution if the value of `global_val` equals 17.

4. From the **Breakpoints** menu, select **Breakpoints...** to open the Breakpoints dialog box. Click the **Add >** button and select **Code Breakpoint...**
5. We want to establish a breakpoint at line 112, the first line of the function `f()` and attach commands to the breakpoint to activate assertion mode and continue execution. Change the **Line** number to 112. Click in the **Command** edit area and enter:

A a; C

Activate the assertion and continue.

6. Create an assertion whose only function is to check that the current line number is still valid for assertion mode. To do this, use the reserved special variable `$LINE`, which contains the line number of the current execution position. In the Assertions dialog box, click on **New...** and enter:

```
if ($LINE >= 120) {A s; 1 x; C}
```

If the line number exceeds 120, the program is about to leave the function `f ()` and CrossView Pro deactivates assertion mode. Normally, the **x** command would make the program stop, but the non-zero value tells CrossView Pro to execute the rest of the commands in the list, in this case, **C** for continue.



You must enter all commands in the Command Window.

1. First set up the assertion you want:

```
a if (global_val == 17) {x}
```

2. Now set a breakpoint on the first line of the function `factorial ()` that will activate assertion mode, and continue execution:

```
110 b {A a; C}
```

3. Now create an assertion that does nothing but make sure that the current line number is still valid for assertion mode. If the line number exceeds 120, you know you have left the function `f ()` and assertion mode should be suspended.

```
a if ($LINE >= 120) {A s; 1 x; C}
```

`$LINE` is a reserved special variable that CrossView Pro maintains containing the number of the line currently executing. If it becomes equal to 120, assertion mode is turned off. Normally, the **x** would make the program stop, but the non-zero value 1 tells CrossView Pro to execute the rest of the commands in the list, in this case, **C** for continue.

In this manner you have created an assertion that is only active over a limited range of source lines.

7.10.7 GATHERING STATISTICS WITH ASSERTIONS

You can also use assertions to gather statistics about your code. For instance, you can find out how many lines of C code execute in a particular session:

```
a {$numlines++}
```

`$numlines` is a user-defined special variable that increments on each line of C code. When the program stops, type:

```
$numlines
```

and CrossView Pro gives the result. To start again, you may want to re-initialize `$numlines` to zero:

```
$numlines = 0
```

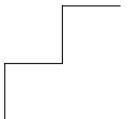
Or just set a breakpoint on the first line of code to do the same.



BREAKPOINTS AND ASSERTIONS

CHAPTER 8

DEFINING AND USING MACROS



8

CHAPTER

8.1 CROSSVIEW PRO MACROS

A *macro* is a user-created shorthand for any sequence of CrossView Pro or C commands and expressions. Macros allow you to debug more efficiently when using CrossView Pro by substituting a short string for a longer combination of words and evaluators.

You can use a macro anywhere an CrossView Pro or C expression is valid: in a breakpoint's command list, with assertions, from the keyboard, among other places. CrossView Pro also allows you to save macro definitions, so they are always available. By passing parameters to a macro, you can create powerful and flexible macros to debug your code more efficiently.

You can use macros in the Command Window, or connect them to the graphic interface in a feature called the *toolbox*. You can have this toolbox visible as a CrossView Pro window and use it to execute a macro by clicking a button. You control which macros have corresponding buttons, making the toolbox easy to adapt to different situations.

8.2 DEFINING MACROS

You can create as many macros as you want:



From the **Tools** menu, select **Macro Definitions...** to open the Macro Definitions dialog box and click on the **New...** button.

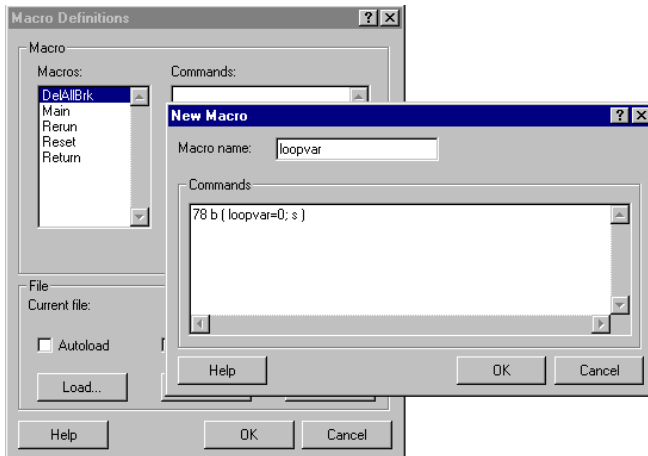


Figure 8-1: Macro Definitions



In the Command Window, use the **set** command followed by the macro's invocation name and the list of commands. Note that the list of commands must be in (double) quotation marks. For example, the command:

```
set st "e main; R"
```

creates a macro call **st** that tells CrossView Pro to change the viewing position to be the first executable line in the function `main()` and restart the program from the beginning. Each time you enter **st** in the Command Window, CrossView Pro substitutes the lengthier list of commands in the definition.

FUNCTION:	Create a macro.
COMMAND:	set <i>name</i> "commands"

Note that there is no rule that the macro definition must be shorter than the commands it represents. For instance, you could substitute **break** for the **b** command, to make CrossView Pro's command language more expressive:

```
set break "b"
```

Now instead of typing **74 b** to set a breakpoint, you can also type:

```
74 break
```

Macros defined using either the command line or the graphic interface are accessible both from the Command Window and the Toolbox.

Macros may call other macros, so it is possible to use simple macros as building blocks for more complex functionality. No macro, however, can call itself, or another macro that refers to the calling macro, since this type of action results in infinite recursion.



Because of the order in which CrossView Pro parses statements, you may not use the CrossView Pro commands **#** or **%** in a macro.

8.2.1 LISTING MACROS



From the **Tools** menu, select **Macro Definitions...** to open the Macro Definitions dialog box. This dialog box contains a scrollable list of the macros.

To see the current definition of a macro:



Follow these steps:

- From the **Tools** menu, select **Macro Definitions...** to open the Macro Definitions dialog box.
- Click on the macro that you want to see.
- The Commands box shows (a part of) the macro. If you need to see more, click on the **Edit...** button.



Type the **echo name** command in the Command Window. For instance, to see the definition for the `st` macro:

echo st	<i>Command.</i>
<code>e main; C 56</code>	<i>Output.</i>

FUNCTION: Display macro expansion.

COMMAND: **echo name**

8.2.2 REDEFINING A MACRO

If you want to change the definition of a macro:



From the **Tools** menu, select **Macro Definitions...** to open the Macro Definitions dialog box. Click on the name of the macro you want to change and click on the **Edit...** button.



In the Command Window, use the **set** command again, but enter an exclamation point after the macro name. For instance, to redefine the macro `st`, which was defined in the example above, use the command:

```
set st! "e main; C 56"
```

Now, the `st` macro changes the viewing position and restarts program execution, placing a temporary breakpoint at line 56. Be sure you do not include a space before the exclamation point. Otherwise, CrossView Pro may interpret the `!` as the C “not” operator.

8.2.3 SAVING MACRO DEFINITIONS TO A FILE

You can save all the macros you define in a debugging session in an external file. This way, you do not lose the definitions when the program ends.

To save macros to an external file:



Follow these steps:

- From the **Tools** menu, select **Macro Definitions...** to open the Macro Definitions dialog box.
- Click on the **Save as...** button. A Save Macro File dialog box opens.
- If you want to save a file previously opened, click on the **Save** button. This saves the file without opening the Save Macro File dialog box.
- Alternatively, you can use the **Autosave** check box. When **Autosave** is checked, all macros are saved in the 'current file' when you leave CrossView Pro.



Type the **`save file`** command in the Command Window. This command saves your macros to the file of your choice. For instance:

```
save macro.mac           writes all your macros to macro.mac
```

FUNCTION: Save macros to a file.

COMMAND: **save** *filename*

8.2.4 LOADING MACRO DEFINITIONS FROM A FILE

You can load saved macros anytime you want to re-use a definition. There is no limit to the number of times you can load macros.

To load a macro file:



Follow these steps:

- From the **Tools** menu, select **Macro Definitions...** to open the Macro Definitions dialog box.
- Click on the **Load...** button and select the macro file you want to load.
- Alternatively, you can use the **Autoload** check box. When **Autoload** is checked, the macros saved in the 'current file' are loaded at startup.



To reinstate your macro definitions from the Command Window, use:

< ***filename.mac***

You must load a program before you can read a macro definition file. Autoload will be ignored when the **Execute these settings at CrossView startup** check box in the Load Symbolic Debug Info dialog box is not checked.



For more information on record and playback functions, see the next chapter, *Command Recording & Playback*.

8.2.5 DELETING MACROS

To delete a specific macro:



Follow these steps:

- From the **Tools** menu, select **Macro Definitions...** to open the Macro Definitions dialog box.
- Highlight the name of the macro.
- Click on the **Delete** button. To delete all the macro definitions at the same time, click on the **Delete All** button. CrossView Pro prompts you for confirmation.



Type the **unset** command in the Command Window. For example, to remove the `st` macro, enter:

```
unset st!
```

When you are removing a macro definition in this manner, you must place an exclamation point after the macro name to prevent CrossView Pro from expanding the name to its full macro definition. To update your macro definition files, issue a **save** command after using **unset**.



You can remove *all* existing macro definitions by entering the **unset** command by itself. CrossView Pro prompts you for confirmation before deleting the macros:

```
unset  
Do you want to delete all macros?y
```

FUNCTION: Delete a macro.

COMMAND: **unset** *name!*

8.3 MACRO PARAMETERS

Macros can accept arguments. Parameters are labelled sequentially in a macro definition: \$1, \$2, \$3, etc. Note that \$0 has no meaning. When you invoke a macro with parameters, enclose the parameters with parentheses and separate them with commas.

CrossView Pro macros can accept any number of parameters, so it is possible to create very complex command shortcuts. You may use any type of parameter when defining a macro, including integers, strings, or addresses. Note, however, that you must pass the macro the correct type at invocation.

For instance, suppose you want to set a detailed breakpoint on any number of lines and a parameter is to specify each line number on which to install a breakpoint. Defining a macro named `brk`, type in the Macro Definitions dialog box:

```
$1 b {Q; initval; recordvar.a; if (initval > 1) {C}}
```

or type in the Command Window:

```
set brk "$1 b {Q; initval; recordvar.a; if (initval > 1) {C}}"
```

In this case, the argument \$1 represents the intended line number. To use the `brk` macro, type:

```
brk(72)     From the Command Window
```

CrossView Pro replaces every instance of \$1 with the value 72. For this example, that means a breakpoint is set at line 72.

8.4 REDEFINING EXISTING CROSSVIEW PRO COMMANDS

Using macros, you can even redefine an existing CrossView Pro command.

For instance, you could redefine the breakpoint command **b** to always place a breakpoint at line 72 of your source code. To do this, enter the command:

```
set b "72 b!"
```

CrossView Pro now interprets the **b** command as **72 b**.

The exclamation point in the definition is necessary to prevent infinite recursion. It tells CrossView Pro to take the command literally and to not expand it into a macro definition. For example:

```
66 b!
```

CrossView Pro interprets this command as the standard breakpoint command and places a breakpoint at line 66, despite the macro definition for **b**.



Be sure not to have any space between the command and the exclamation point. Otherwise CrossView Pro may interpret the **!** as the C *not* operator.

8.5 USING THE TOOLBOX

The CrossView Pro toolbox, shown in figure 8-2, is controlled from the **View** menu. Using the **Tools** menu, you can configure the toolbox and define the macros for it. You can resize the toolbox to the size you want.

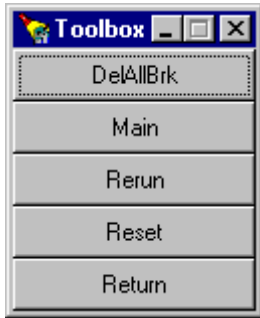


Figure 8-2: CrossView Pro Toolbox

8.5.1 OPENING THE TOOLBOX

To open the toolbox:



From the **View** menu, select **Toolbox**.



The Toolbox is a pop-up window that remains on top of the CrossView Pro Desktop while you work in other windows.

8.5.2 CONNECTING MACROS TO THE TOOLBOX

To configure the toolbox, select **Toolbox Setup...** from the **Tools** menu to view the Toolbox Setup dialog box, shown in figure 8-3. This dialog box displays the toolbox buttons and an alphabetized list of the current macro definitions.

To connect a macro to a toolbox button:



Follow these steps:

- Click on the button you wish to change
- Scroll through the macro list to highlight the desired function

- Click on the **Assign** button or press the **Enter** key

Note that double clicking on the macro name in the alphabetized list performs the third step automatically. The name of the new function appears on the selected button and the connection is performed.

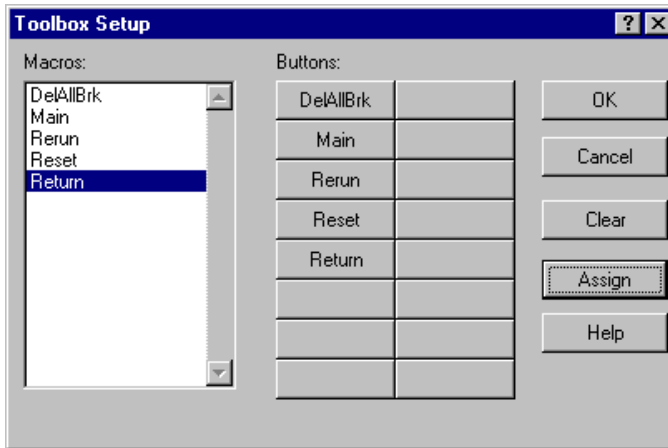


Figure 8-3: Setting Up the Toolbox



Do not assign parameterized macros to the toolbox since there is no way to pass in parameter values.

8.5.3 REMOVING A MACRO CONNECTION

To delete a macro definition from the toolbox:



Follow these steps:

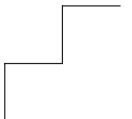
- From the **Tools** menu, select **Toolbox Setup...** to open the Toolbox Setup dialog box.
- Select the desired button.
- Click **Clear**.

This deletes the macro definition from the toolbox.

CHAPTER

9

COMMAND RECORDING & PLAYBACK



9

CHAPTER

9.1 RECORDING COMMANDS

CrossView Pro lets you save a series of CrossView Pro commands to the file of your choice. This is **record mode**. You can re-load a saved file to repeat parts of debugging tasks or replay a debugging session (up to the point where you left the last time).

Record mode means that all CrossView Pro commands from the keyboard, mouse or menu are recorded to a disk file. The debugger can read this file and execute the commands as if they were entered into the Command Window. This is called **playback mode**, see more about playback mode later in this chapter.



Record and playback modes can never be active at the same time.

You can record CrossView Pro commands and/or Emulator commands. When recording on CrossView Pro command level, all commands that you type in the Command Window, as well as the CrossView Pro command language equivalents of dialog actions and menu selections are saved in a file. When you (also) want to record commands entered in the Emulator Command Window, you can record them in a separate dialog or combine them with the CrossView Pro commands.

From the Command Window you control record mode using either the mouse or keyboard commands. To start or setup recording:



From the menu system:

- From the **Tools** menu, select **Record | CrossView...** to open the Record CrossView dialog box, or select **Record | Emulator...** to open the Record Emulator dialog box.

The Record dialog box contains an **Automatically at CrossView startup** check box. If you select this check box the debugger enters record mode at every startup.

- Enter the name of the file in the **Command file:** edit field, or click on the **Browse...** button to select an existing file. The default filename extension is `.cmd`.
- Optionally, select **Include emulator commands** in the Record CrossView dialog. In this case all recorded emulator commands are also recorded, preceded by the "o" command.
- Click on the **OK** button to save the current settings into the initialization file `xvw.ini` for following debugging sessions.
- Click on the **Start** button to start recording.



Enter the **>** command with the name of the file to start recording. For example, enter:

>session.cmd

After you invoke this command, CrossView Pro saves every executed command, whether using the mouse or manually typed into the Command Window, to the file `session.cmd`.

FUNCTION: Save CrossView Pro commands to a file.

COMMAND: **>filename**

FUNCTION: Save CrossView Pro commands to a file and force flushing.

COMMAND: **>!filename**

FUNCTION: Save CrossView Pro and emulator commands to a file.

COMMAND: **>@filename**

FUNCTION: Save emulator commands to a file.

COMMAND: **>#filename**

9.1.1 ENTERING COMMENTS

Every command, whether typed into the Command Window or the result of a mouse or menu action goes into the recording file. To add comments to a file recording CrossView Pro commands, enclose text typed in the Command Window with C comments delimiters, “/*” and “*/”. When logging emulator commands, refer to your emulator documentation for the appropriate comment characters.

9.1.2 SUSPEND RECORDING

This function acts like the pause button on a tape recorder: the recording mechanism stays in place, but suspends temporarily. CrossView Pro does not save to file any commands you enter while you suspend recording, but the file remains open and ready to accept input. To suspend recording:



From the **Tools** menu, select **Record | CrossView...** or select **Record | Emulator...** Click on the **Suspend** button.



In the Command Window, use the **>f** or **>#f** command (for “false”).

FUNCTION: Suspend recording CrossView Pro commands.

COMMAND: **>f**

FUNCTION: Suspend recording emulator commands.

COMMAND: **>#f**

9.1.3 RESUME RECORDING

This function is the counterpart of the suspend recording function. CrossView Pro resumes adding commands to the current record file. Any new command you enter appears in the file; they do not affect the commands already saved.



From the **Tools** menu, select **Record | CrossView...** or select **Record | Emulator...** Click on the **Resume** button to resume recording.



In the Command Window, use the **>t** or **>#t** command (for “true”).

FUNCTION: Resume recording CrossView Pro commands.

COMMAND: **>t**

FUNCTION: Resume recording emulator commands.

COMMAND: >#t

9.1.4 CHECK RECORDING STATUS

If at any point you do not remember whether recording is on or off, check by:



From the **Tools** menu, select **Record | CrossView...** or select **Record | Emulator...** If record mode is active, the **Stop** button is enabled. If the **Start** and **OK** buttons are enabled, record mode is off.



Enter the **>** command in the Command Window.

This command shows the status of the recording and logging mechanism. For example, if you enter **>** you might see:

```
>
Output logging is OFF
Command recording is ON
Emulator command recording is OFF
Target communication logging is OFF
```

The **>** command gives you the status for the different recording mechanisms. Output logging and target communication logging are described below.

9.1.5 CLOSE FILE FOR RECORDING

Closing a file for recording differs from suspending recording in that when you close a file, you may not add any more commands to it. If you were to start recording again using the same filename, the old commands in the file would be deleted. (Note that this does not exclude editing the file manually by some other means, since the file is saved as ASCII text.)



From the **Tools** menu, select **Record | CrossView...** or select **Record | Emulator...** Click on the **Stop** button to stop recording.



Enter the **>c** or **>#c** command to close the file.

FUNCTION: Close command recording file.

COMMAND: **>c**

FUNCTION: Close emulator command recording file.

COMMAND: **>#c**

9.1.6 COMMAND RECORDING EXAMPLE

For example, consider the following command sequence (from the Command Window):

```

>session.cmd          ----- Start Recording to File
  initval
  p 12

>f                    ----- Carriage Return
  l b                 ----- Suspend Recording
  sum

>t                    ----- Resume Recording
/* This is a comment! */
>c
  
```

This series starts with a command to record to a file named `session.cmd`. The blank line above represents a carriage return. After the last command, **c**, if you were to view this file, it contains:

```

initval
p 12
/* This is a comment! */
  
```

The saved command file contains simply the commands, without any output. Note that commands entered while recording was suspended (**l b** and **sum**) do not appear in the file. Carriage returns are not recognized as commands.

9.2 PLAYING BACK COMMAND FILES

Once you have recorded a set of CrossView Pro commands, you can play them back to recreate a debugging session or repeat often-used sequences. Running the debugger while reading commands from a file is **playback mode**.



Remember that for a file to be played back, it can only contain CrossView Pro or emulator commands. For this reason, screen output files cannot be used in playback mode. Refer to the *Recording Commands* section earlier in this chapter for more information.

As with recording, the Command Window controls playback mode. To playback a command file:



Follow these steps:

1. From the **Tools** menu, select **Playback | CrossView...** to open the CrossView Playback dialog box, or select **Playback | Emulator...** to open the Emulator Playback dialog box.



You can choose to playback either CrossView Pro commands or Emulator commands. Open the Emulator Command Window if the playback file contains commands sent directly to your emulator.

2. Type the playback filename or use the **Browse...** button to select the file. The default filename extension is `.cmd`.



In the Playback dialog box, you have two additional options: **Playback at XVW startup** and **Continuous playback**. CrossView Pro enters playback mode automatically when you start the debugger if you click on the **Playback at XVW startup** check box in the Playback dialog box. The entire playback file executes if you enable the **Continuous playback** check box.

3. Click on the **Execute** button to start the playback.



In the Command Window, use the `<` or `<< filename` command to playback CrossView Pro commands.



On the command line of CrossView Pro give the option `-T filename` to start CrossView Pro in transparency mode and playback emulator commands. This is not available for all execution environments.

9.2.1 SETTING THE TYPE OF PLAYBACK



Enable the **Continuous playback** check box in the CrossView Playback dialog box to turn on continuous play back of commands.



In the Command Window, there are two commands for the type of playback. The **<filename** command starts playback. Commands are read from a file and executed without any stop. For example:

<session.cmd *load and execute all the commands*



The **<<** command causes CrossView Pro to playback commands one at a time, similar to single-stepping through code. For example:

<<session.cmd *read a command from the file.*

Clicking the **Execute** button or pressing the Enter key executes the next command.

FUNCTION: Play back a file of CrossView Pro commands.

COMMAND: **<filename**

FUNCTION: Play back a file of CrossView Pro commands, one command at a time.

COMMAND: **<<filename**

9.2.2 CALLING OTHER PLAYBACK FILES

A playback file can call another playback file in the course of its execution.

When CrossView Pro creates a command file, it saves all commands in their textual form, whether entered by the mouse or as text. You must edit this file to use the **<** and **<<** commands.

When the debugger reaches a **<** or **<<** command in a playback file, playback execution switches to the new file, but does not return to the original file. In other words, you chain playback files but not nest them.

9.2.3 QUITTING PLAYBACK MODE

Playback mode stops automatically when CrossView Pro reaches the end of the command file. If you want to end playback mode before this point, click the **Halt** button.

9.3 COMMAND LINE BATCH PROCESSING

CrossView Pro supports command line batch file processing, but CrossView Pro will halt if a modal dialog is encountered or if the target program contains an endless loop. The command line option **--timeout=*n_seconds*** switches CrossView Pro to a different mode of operation, without the two drawbacks mentioned above.

In order to process files in batch mode you have to do the following:

1. Create a temporary directory.
2. Start CrossView Pro from this temporary directory. For Windows 95/98/XP/NT/2000 you can create a separate icon or shortcut to start CrossView Pro, which has the working directory (`Start in:`) set to the temporary directory.
3. Close all CrossView Pro windows except the Command Window.
4. Exit CrossView Pro (with **Save desktop and target settings** enabled).

You now have generated an `xvw.ini` file with minimal GUI overhead.

5. Save the `xvw.ini` file and remove the temporary directory.

For each batch run of CrossView Pro you have to do the following:

1. Create a temporary directory.
2. Copy the saved `xvw.ini` file to the temporary directory.
3. Create a command file in the temporary directory.

The following command file `session.cmd` loads the `.abs` file, downloads the code, runs the code and exits.

<code>N hello.abs</code>	<i>load the symbols</i>
<code>dn</code>	<i>download the program</i>
<code>__exit bi</code>	<i>set a breakpoint at the exit point</i>
<code>R</code>	<i>run the program</i>
<code>\$pc</code>	<i>optional: show the program counter</i>
<code>q y</code>	<i>exit CrossView Pro</i>

where `hello.c` contains

```
#include <stdio.h>

void main()
{
    printf("Hello World!\n");
}
```

4. Copy the `.abs` file to the temporary directory. This is needed because CrossView Pro changes its working directory when the **N** command is used.
5. The following line executes CrossView Pro in batch mode and waits for it to finish:

Windows 95/98/XP/NT/2000:

```
start /wait c:\c68k\bin\xfw68 --timeout=120 -tcfg sim.cfg
-p session.cmd -R session.log
```

UNIX:

```
xfw68 --timeout=120 -tcfg sim.cfg -p session.cmd -R session.log
```

This command must be issued in the temporary directory! After the execution has ended, the file `session.log` contains a transcript of the commands.

6. Save the output files and clean up (or remove) the temporary directory. This must be done because the `xvw.ini` file has been modified now. If CrossView Pro would be started again in the temporary directory, the file `session.cmd` would be executed again.

The `--timeout=n_seconds` command activates the batch operation mode of CrossView Pro. It causes CrossView Pro to terminate when the specified amount of time has elapsed, which is crucial in batch processing: if a

program does not terminate, the timeout will terminate CrossView Pro, so that the next program in the batch can be executed. CrossView Pro will also terminate in the batch mode if a modal dialog pops up, since this requires user interaction to continue. Before CrossView Pro exits, the text in the dialog will be written to the log file. A special case of this dialog is the 'End of program reached' dialog. For this reason, the line `__exit bi` has to be added to the `.cmd` file, so it is possible to do some things (for example, read registers modified by a machine code program) after the program is finished. If the breakpoint at `__exit` is absent, CrossView Pro immediately exits after having executed the **R** command, so any consecutive commands will be ignored.

9.4 LOGGING

Logging means that all output text to a particular window is saved in a file for later use. Two windows allow logging:

- Command Output Window
(upper part of the CrossView Command Window)
- Emulator Output Window
(upper part of the Emulator Command Window)

"GDI Accesses" can also be logged. This is the information transferred between CrossView Pro and the Debug Instrument (DI).

You can control logging from the **Tools** menu or from the Command Window.

You can also determine the status of each logging function:



From **Tools** menu, select **Log | Command Input/ Output...**, **Log | CrossView-Emulator I/O...** or **Log | CrossView-GDI Accesses...**

If a logging function is active, the **Stop** button is enabled. If the **Start** and **OK** buttons are enabled, logging is off.



Enter the `>>`, `>&` or `>*` command in the Command Window.

Each type of logging is described in the following section.



The Emulator Output Window is primarily a diagnostic tool. It should be used wisely, since it generates substantial amounts of output, the format of which is emulator dependent. For emulators that have an ASCII interface, the actual command/response dialogue will be displayed. For emulators with a binary interface, CrossView Pro will generate a record of function calls with their associated input and output parameters. This also applies to the GDI Accesses output logging.

9.4.1 SETTING UP LOGGING

To setup logging:



From the menu system:

- From **Tools** menu, select **Log | Command Input/ Output..., Log | CrossView-Emulator I/O...** or **Log | CrossView-GDI Accesses...** to open the appropriate dialog box.
- Type in the name of the log file or use the **Browse...** button to select a filename. The default filename extension is `.log`.

Each Log dialog box has an **Automatically at CrossView startup** check box. This check box instructs CrossView Pro to start recording the output of a particular window or information stream upon starting up of CrossView Pro.

- Click on the **OK** button to save the current settings into the initialization file `xvw.ini` for following debugging sessions.
- Click on the **Start** button to start logging.



You can open up a log file for CrossView Command Output by using the **>> *filename*** command as in:

```
>>screen.log
```

You can force flushing by using the **>>! *filename*** command as in:

```
>>!screen.log
```



You can open up a log file for Emulator Output by using the **>& *filename*** command as in:

```
>&target.log
```

You can force flushing by using the **>&!filename** command as in:

```
>&!target.log
```



You can open up a log file for GDI accesses output logging by using the **>*filename** command as in:

```
>*gdi.log
```

You can force flushing by using the **>*!filename** command as in:

```
>*!gdi.log
```

<hr/>	
FUNCTION:	Save CrossView Pro commands and command window output to a file.
COMMAND:	>>filename
<hr/>	
FUNCTION:	Force flushing of CrossView Pro commands and command window output to a file.
COMMAND:	>>!filename
<hr/>	
FUNCTION:	Log target communications.
COMMAND:	>&filename
<hr/>	
FUNCTION:	Force flushing of target communication logging.
COMMAND:	>&!filename
<hr/>	
FUNCTION:	Log GDI accesses.
COMMAND:	>*filename
<hr/>	

FUNCTION: Force flushing of GDI accesses logging.
COMMAND: *>!*filename*

9.4.2 RECORDING COMMANDS AND LOGGING SCREEN OUTPUT

It is possible to have command recording, command output logging and target communication logging on at the same time. That is, you can have one file recording just the CrossView Pro commands, and another file concurrently recording both the commands and the computer responses. Refer to the previous section for information on command record files.

Since the Command Window log file contains both your commands and the computer responses, you cannot use it in playback mode.

9.4.3 COMMAND WINDOW LOG FILE EXAMPLE

For example, if you entered the following commands:

```
>>screen.log  
initval  
l a
```

The output file, `screen.log`, contains:

```
> initval  
initval = 0  
> l a  
no assertions
```

9.4.4 SUSPENDING AND RESUMING OUTPUT LOG

You can resume and suspend the Logging process from the menu or from the Command Window:



From **Tools** menu, select **Log | Command Input/ Output..., Log | CrossView-Emulator I/O...** or **Log | CrossView-GDI Accesses...** to select the appropriate dialog box.

To suspend logging:



Click on the **Suspend** button.



In the Command Window, use the **>>f** command for suspending the logging of the Command Output Window. Type **>&f** to suspend the Emulator Output Window. Type **>*f** to suspend GDI accesses logging. After you issue this command, CrossView Pro does not save all subsequent commands and their computer responses.

To resume logging:



Click on the **Resume** button.



In the Command Window, use the **>>t** command to resume logging the Command Output Window. Type **>&t** to resume the Emulator Output Window. Type **>*t** to resume GDI accesses logging. After you issue this command, CrossView Pro saves all subsequent commands and their computer responses.

FUNCTION: Suspend output logging (logging is false).

COMMAND: **>>f**

FUNCTION: Resume output logging (logging is true).

COMMAND: **>>t**

FUNCTION: Suspend target logging (logging is false).

COMMAND: **>&f**

FUNCTION: Resume target logging (logging is true).

COMMAND: **>&t**

FUNCTION: Suspend GDI accesses logging (logging is false).

COMMAND: **>*f**

FUNCTION: Resume GDI accesses logging (logging is true).

COMMAND: **>*t**

9.4.5 CLOSING THE OUTPUT LOG FILE

To close the output file:



From **Tools** menu, select **Log | Command Input/ Output..., Log | CrossView-Emulator I/O...** or **Log | CrossView-GDI Accesses...** to select the appropriate dialog box. Click on the **Stop** button to stop logging.



Enter the **>>c** or **>&c** command in the Command Window to close the Command Output and Emulator Output log files. These commands end the recording for the currently specified output log file.

FUNCTION: Close output log file.

COMMAND: **>>c**

FUNCTION: Close target log file.

COMMAND: **>&c**

9.5 STARTUP OPTIONS

When starting up CrossView Pro you may immediately start recording or playing back files. For instance,

```
xfw68 fact -p session
```

plays back the commands in the file `session`. A **-P** option single-steps through each command, prompting you for a return after each command. You can also start recording:

```
xfw68 fact -r session
```

This command records all your commands (just like the `>` command) to the file `session`, while:

```
xfw68 fact -R session
```

logs your commands and screen output to the file `session` (just like the `>>` command).

You can also use the **Automatically at CrossView startup** option in the Record, Playback, and Log dialogs to immediately start recording, playback or logging at CrossView Pro startup.



You can also enter record and playback files via EDE. From the **Project** menu, select **Project Options...** Expand the **CrossView Pro** entry and select **Logging**. Enter your record and playback filenames.

9.6 CROSSVIEW PRO COMMAND HISTORY MECHANISM

CrossView Pro stores the command history in the list box of the Command Window.

You can select a command from the history list by clicking on it or jumping with the **<Tab>** key to the history listing and using the arrow keys.. The command appears in the edit field of the Command Window. You may edit the command if you want.

To execute the command, click on the **Execute** button.

If you do not want to edit the command, double-click on the selected command in the list box to execute the command, or hit the **<Return>** key.



RECORD & PLAYBACK

CHAPTER

10

I/O SIMULATION



TASKING



10

CHAPTER

10.1 INTRODUCTION

The CrossView Pro Terminal windows provide an interface to exchange data with the application on the target. You can use the following I/O simulation types for this purpose.

File I/O (FIO)

With File I/O you can connect actions to a probe point. Probe points are breakpoints that do not update the graphical user interface (GUI) and when they are hit, connected actions are performed and execution continues. The actions are in this case I/O actions to a file and/or a terminal window.

File System Simulation (FSS)

With FSS you can use standard stream I/O function calls like `printf()` in your source, to test I/O to and from the target system or simulator.

Debug Instrument I/O (DIO)

If you have a debug instrument that supports it, the debug instrument can perform input and output using GDI callback functions.

10.2 I/O STREAMS

You can setup I/O streams with the I/O Simulation Setup dialog. There is virtually no limit on the number of streams that can be opened or created. Each type of I/O stream (FIO, FSS, DIO) has its own numbering:

FIO 0,1,2,...,k

FSS 0,1,2,...,m

DIO 0,1,2,...,n

You can map multiple streams to one terminal window.

For File I/O you can use the **ios_** commands to open or close a FIO stream on the command line. Streams can be opened manually or are opened at the first call or operation that accesses a specified I/O stream (for Debug Instrument I/O handling). For FSS the target application can open streams with `open()` calls and close streams with `close()` calls.

Streams can be mapped to a terminal window and/or a file that is NOT the terminal log file. If a stream is mapped to a terminal window and a file the output will go to the terminal window and also to the file. In case of input the input will be read from the file. The read input will be echoed on the connected terminal window.

I/O streams opened by FSS are closed when end of program is reached or if a program reset occurs. I/O streams opened by CrossView Pro will be rewound. The windows to which the streams are mapped remain open.

In the I/O Simulation Setup dialog you can connect an I/O stream to a terminal window before the stream is opened by specifying the stream type, filename and terminal window name.

10.2.1 SETTING UP FILE I/O STREAMS

You can set up an input or output stream. For input you may specify either a file or the keyboard, for output either a file or the screen. Each stream has its own identifying number.

You can also specify the format of the stream's values. The default is character, but you may want to use hexadecimal or octal values when directing data to or from a file.

To setup a File I/O stream:



From the menu system:

- From the **Settings** menu, select **I/O Simulation Setup...** to open the I/O Simulation Setup dialog box.
- Open the **File I/O** tab to setup a File I/O stream.
- Select the **Configure...** button. This opens the File I/O Configuration dialog.
- In the **Probe point** list box, select an existing probe point or press the **New...** button to set a new probe point. The Breakpoints dialog appears.
- In the **Stream** list box, select a stream or press the **New...** button to create a new stream. Select a new stream and click **OK**.
- Enter the **Address** and **Length** (in minimum addressable units, MAU) of the memory location you want to read from or write to.
- Optionally, enable the **Use hexadecimal format** check box when you want the data to be interpreted as a hexadecimal value.

- Choose the Direction: **Input** if the stream must provide input to the application, or **Output** if the stream must be an output stream.
- Click on the **Apply** button to accept the contents and enter another configuration or click on the **OK** button to close this dialog box.



Enter the **ios_open** or **ios_wopen** command in the Command Window to open a File I/O stream.

FUNCTION: Open a File I/O stream

COMMAND: **ios_open** ["file",[mode],[r],*\$xvw_variable*]]]

FUNCTION: Open a File I/O stream and map the stream to a terminal window

COMMAND: **ios_wopen** ["terminal_window"],*\$xvw_variable*]]]



Enter the **ios_read** or **ios_write** command in the Command Window to read from or write to a File I/O stream.

FUNCTION: Read from a File I/O stream

COMMAND: **ios_read** {stream | "file"},address,number_of_maus[,**x**]

FUNCTION: Write to a File I/O stream

COMMAND: **ios_write** {stream | "file"},address,number_of_maus[,**x**]

To read 1 MAU hexadecimal value from file `mydata.dat` and store it at address `0x100`, type:

```
ios_read "mydata.dat",0x100,1,x
```

10.2.2 REDIRECTING I/O STREAMS

In the I/O Simulation Setup dialog you can connect an I/O stream to a terminal window before the stream is opened or you can redirect an existing stream to a file and/or terminal window.

To redirect an I/O stream to a file and/or terminal window:



From the menu system:

- From the **Settings** menu, select **I/O Simulation Setup...** to open the I/O Simulation Setup dialog box.
- In the **Connection** tab select the I/O stream you want to change and select the **Redirect...** button.
- In the Connection Configuration dialog enter a filename and/or a terminal window name.
- Click **OK** to accept the changes and close the dialog.



Enter the **ios_open** or **ios_wopen** command in the Command Window to open a File I/O stream.

To disconnect an I/O stream from a file and/or terminal window:



From the menu system:

- From the **Settings** menu, select **I/O Simulation Setup...** to open the I/O Simulation Setup dialog box.
- In the **Connection** tab select the I/O stream you want to change and select the **Redirect...** button.
- In the Connection Configuration dialog erase the filename and/or terminal window name.
- Click **OK** to accept the changes and close the dialog.



Enter the **ios_close** command in the Command Window to close a File I/O stream.

FUNCTION: Close a File I/O stream

COMMAND: **ios_close** {*stream* | "*file*"}

To disable/enable an I/O stream:



From the menu system:

- From the **Settings** menu, select **I/O Simulation Setup...** to open the I/O Simulation Setup dialog box.
- In the **Connection** tab clear the check box in front of the I/O stream you want to disable. Set the check box to enable the stream.

Disabling a File I/O stream means that I/O actions will not be honored. Writing is not passed to the output file, and reading does not result in new data being placed in the target buffer.

10.3 FILE SYSTEM SIMULATION

File system simulation enables the application on the target board to use system calls (such as open, read, write) that are handled by the host system file I/O services. These files can be read directly from the host system, and output can be written to a file on the host system or in a CrossView Pro window. File system simulation is available for all execution environments.

The File System Simulation feature redirects I/O to a Terminal Window if the filename `FSS_window:window_name` is used in the "open" call, *window_name* is the name of a Terminal Window.

You can specify a root directory for FSS. CrossView Pro will search for the file from the root directory downwards. You can do this in the I/O Simulation Setup dialog, by entering a directory name in the **FSS root directory** field of the **Options** tab. This setting is saved in the `xvw.ini` file. Another possibility is to set a temporary resource by specifying the command line option `--fss_root_dir="path"` on CrossView Pro startup.

You can redirect File System Simulation streams to a file or another stream. Redirection to a file can be needed when a stream is only mapped to a window and you want it to be mapped to a file also.

Redirection can be used for scripting purposes, using the **FSS** command.

```
FSS { < | > } &stream | "file"
```

For example,

```
FSS 2>&1
FSS 1<&4
FSS 4<"data.txt"
FSS 3>"data.txt"
```

The first example will redirect output of stream 2 to stream 1. The second example will retrieve input for stream 1 from stream 4. The third example will retrieve input for stream 4 from file "data.txt". The fourth example will redirect output of stream 3 to file "data.txt".

Disabling an FSS stream means in effect connecting the stream to /dev/null or NUL, causing writes to go into oblivion, and reads to return EOF.

10.3.1 FILE SYSTEM SIMULATION LIBRARIES

The low-level I/O functions such as `_open()`, `_close()`, `_read()` and `_write()` are implemented in the C library to use File System Simulation. These functions redirect high-level I/O calls such as `printf()` and `scanf()` type functions through CrossView Pro's FSS feature, allowing you to perform `stdin`, `stdout` and `stderr` I/O by just using these standard C library functions.

The libraries have been optimized to only attach the file I/O routines if the application actually uses file I/O. The default I/O streams `stdin`, `stdout` and `stderr` are opened on the fly whenever file I/O is used; this behavior is transparent to the user. It is no longer necessary to inform CrossView Pro about the use of any streams.

For more information see the section *Run-Time Library Routines* in the C Compiler/Assembler Reference Manual.

10.4 DEBUG INSTRUMENT I/O

If you have a debug instrument that supports it, the debug instrument can perform input and output using GDI callback functions. The Debug Instrument I/O (DIO) stream number is passed as parameter to these callbacks. The output can be redirected to DDE (Windows only). The first access to a DIO stream will create a new terminal window and the title of the window will be "DIO x ", where x is the number of the used stream. No new window will be created if the used stream is already mapped to a terminal window. You can use the I/O Streams Terminal Map dialog to map one or more streams to one window.

10.5 THE TERMINAL WINDOW

If you direct I/O simulation to the screen, CrossView Pro displays the output in the terminal window. Similarly, if you direct input from the keyboard; whatever you input appears in the appropriate terminal window. See section 4.6.8, *Terminal Window* for more information.

10.5.1 TERMINAL WINDOW KEYBOARD MAPPINGS

The following keyboard mappings, being both control codes and escape sequences, are supported by the VT100-like terminal mode of the terminal windows:

Key	Character Sequence and/or Decimal Value
Backspace	8d
TAB	9d
DEL	127d
ESC	27d
Insert	ESC [2 ~
Prev/Page Up	ESC [5 ~
Next/Page Down	ESC [6 ~
Arrow Up	ESC [A
Arrow Right	ESC [B
Arrow Left	ESC [C
Arrow Down	ESC [D

Table 10-1: General Keyboard Mappings

Display Control

The VT100-like terminal mode of the terminal windows comprises the following control codes and escape sequences for displaying:

ASCII Code	Decimal Value	Operation
BELL	7	Ring the bell
BS	8	Move cursor one position back
TAB	9	Move cursor to next tab stop
LF	10	Move cursor one line down
CR	13	Move cursor to start of line
ESC	27	Start escape sequence (see below)

Table 10-2: Control Codes

Escape Sequences

Escape Sequence	Operation
ESC D	Cursor one line down (scrolls if already at last line)
ESC E	Cursor one line down and to left margin (scrolls)
ESC M	Cursor one line up (scrolls if already at top line)
ESC [<i>n1</i> A	Cursor <i>n1</i> lines up
ESC [<i>n1</i> B	Cursor <i>n1</i> characters right
ESC [<i>n1</i> C	Cursor <i>n1</i> characters left
ESC [<i>n1</i> D	Cursor <i>n1</i> lines down
ESC [H	Cursor home
ESC [<i>n1</i> ; <i>n2</i> H	Move cursor to (<i>n1</i> , <i>n2</i>) with <i>n1</i> =row, <i>n2</i> =col

Table 10-3: Cursor Motion



Parameters *n1* and/or *n2* may be left out, in which case a value of 1 is assumed.

Escape Sequence	Operation
<i>ESC</i> [J	Clear screen from cursor till bottom-right
<i>ESC</i> [<i>p1</i> J	0: Clear screen from cursor till bottom-right 1: Clear screen from top-left till cursor 2: Clear entire screen
<i>ESC</i> [K	Clear line from cursor till end
<i>ESC</i> [<i>p1</i> K	0: Clear line from cursor till end 1: Clear line from begin to cursor 2: Clear entire line

Table 10-4: Erasing

For example, to clear the entire screen in the C programming language, you can enter:

```
printf( "\033[H\033[2J" );  
fflush(stdout);
```

Escape Sequence	Operation
<i>ESC</i> [<i>n1</i> @	Insert characters
<i>ESC</i> [<i>n1</i> P	Delete <i>n1</i> characters
<i>ESC</i> [<i>n1</i> L	Insert <i>n1</i> lines
<i>ESC</i> [<i>n1</i> M	Delete <i>n1</i> lines

Table 10-5: Inserting and Deleting



Parameter *n1* may be left out, in which case a value of 1 is assumed.

Escape Sequence	Operation
<i>ESC</i> [m	Turn off all attributes
<i>ESC</i> [n1 m	0: turn off all attributes 1: bold 4: underline 5: blinking 7: reverse 8: invisible 22: turn off bold 24: turn off underline 25: turn off blinking 27: turn off reverse 28: turn off invisible

Table 10-6: Character Attributes

Multiple parameters may be specified simultaneously:

ESC [n1 ; ... ; nN m

Some attributes or combinations of attributes are mapped to a regular standout mode.

Parameters may be left out, in which case a value of 0 is assumed.

Escape Sequence	Operation
<i>ESC</i> [12 l	Local echo on
<i>ESC</i> [12 h	Local echo off
<i>ESC</i> [? 7 h	Wrap around on
<i>ESC</i> [? 7 l	Wrap around off
<i>ESC</i> [? 25 h	Cursor on
<i>ESC</i> [? 25 l	Cursor off
<i>ESC</i> [? 92 l	Enquire after the window's size Response: <i>ESC</i> [? rows, columns c

Table 10-7: Miscellaneous



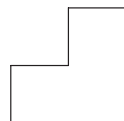
I/O SIMULATION

CHAPTER 11

SPECIAL FEATURES



TASKING



11

CHAPTER

11.1 TRANSPARENCY MODE

Transparency mode allows you to communicate directly with the execution environment. Most of the time CrossView Pro will handle all the low level communications, freeing you to concentrate on the high level C code. Depending on the type of execution environment, however, you may have to enter transparency mode to set up the execution environment when the machine is first turned on.

To enter transparency mode:



From the **View** menu, select **Command | Emulator**.

All commands entered in the Emulator Command Window are passed directly to the execution environment.

To exit transparency mode:



From the **View** menu, select **Command | CrossView**.

In CrossView Pro, you can pass a string directly to the execution environment without leaving CrossView Pro with the **o** command:

o map

This passes the command `map` directly to the execution environment, while you remain in CrossView Pro. Naturally you will have to learn your execution environment's command set to make use of the **o** command.

FUNCTION: Pass a command to the execution environment.

COMMAND: **o** *string*



Do not issue one-shot transparency commands that result in large output (or otherwise require intervention other than a carriage return to terminate output). Instead, enter transparency mode first, then issue the command.

You may also enter transparency mode upon startup with the **-T** option. See the section on startup options.

11.2 RTOS AWARE DEBUGGING

CrossView Pro supports RTOS (Real-Time Operating System) aware debugging for various kernels. Since each kernel is different, the RTOS aware features are not implemented in the CrossView Pro executable, but in a library (RADM: RTOS aware debugging module) that will be loaded at run-time by CrossView Pro. The amount of windows and dialogs and their contents is kernel dependent.

CrossView Pro for the 68K/ColdFire supports an OSEK RADM (`osek_radm.dll`) according to the OSEK standard. You have to create your own OSEK Run Time Interface (ORTI) and specify this file to CrossView Pro. CrossView Pro supports ORTI specifications v2.0 and v2.1.

EDE



From the **Projects** menu, select **Project Options...** Expand the **CrossView Pro** entry and select **RTOS Aware Debugging Module**. Select **OSEK** and specify the name of the ORTI file, or select **User Defined** and specify your RADM DLL name.

CrossView Pro

Within the CrossView Pro's Target Settings dialog (Target | Settings...), select the CrossView Pro configuration you will use by selecting a "Target configuration". These target configuration files are normal ASCII text files. The name of the shared library that contains the kernel aware code can be specified in the target configuration. The "radm" configuration item specifies the name of the shared library that contains the kernel aware code.

The syntax of a target configuration file is:

`[! comment] field : field-value`

field one of the defined keywords

field-value the value assigned to the field

comment optional comment

Empty lines, lines consisting of only white space are allowed. Comments start at an exclamation-sign (!) and end at the end of the line.

The line for the shared library that supports RTOS aware code could be:

```
radm: yourrtos.dll
```



Or you can specify the RADM filename on the CrossView Pro command line with the following option:

```
--radm=osek_radm.dll
```

You can specify the ORTI filename on the CrossView Pro command line with the following option:

```
--orti=ORT-filename
```

The OSEK RADM adds an **OSEK/ORTI** menu to CrossView Pro that has several items (each description in the notation '<text>' is represented in the syntax of the OSEK Run Time Interface file):

- *OSEK implementation name* (if reading of the ORTI file succeeded)

The OSEK implementation name is specified with <name> in the <declaration_section> of the ORTI file.

For each <declaration_spec> a sub menu item will be created with the name represented for <object>. When selecting an object item a window will appear with all objects from the <information_section> for the specified <object>. The new created window always contains the Object column and then the columns represented in the <object_decl_list> of the specified object.

- Info Messages

This menu item lists all expressions from the ORTI file that could not be evaluated. This could be an expression within the <declaration_section> represented in the <enum_value_list>. However evaluating the expressions from the objects in the <information_section> also could have problems. The problems could occur when the expression is too difficult to be evaluated or when one of the variables of the expression is not available when the symbolic debug info is loaded.

When an expression could not be evaluated it results into 'N.A.' for the specified window object entry item. When the expression could be evaluated but the enumerated type could not be found or the specified type could not be converted correctly this will result into 'n.a.' for the specified window object entry item.

So, there are two situations:

- n.a. : Expression could be evaluated but could not be converted correctly at current moment. This expression will not occur in the list when the menu item 'Info Message' is selected.
- N.A. : Expression could not be evaluated and will not change until the ORTI file is updated with a valid expression.

For the second situation you can type the expression in the command window and CrossView Pro will show a message box with the reason why the expression could not be evaluated.

- About RADM

This menu item shows the supported OSEK/ORTI version and the RADM version.

11.3 COVERAGE



You can only use this feature if it is supported by the execution environment (see the addendum).

When the application program is executed as a result of a command such as StepInto or Continue, CrossView Pro traces all memory access, i.e. memory read, memory write and instruction fetch. Through code coverage, executed and not execute areas of the application program can be found. Areas of unexecuted code may exist in case of programming errors or simply dead code which could be eliminated. Alternatively, your program input, your test set, is incomplete. It does not cover all paths in the program. Data coverage allows you to verify which memory locations, i.e. which variables, are accessed during program execution. Additionally, stack and heap usage can be shown.

To enable/disable coverage:



From the **Tools** menu, select the **Coverage** checked menu item.

When the menu item is checked, coverage is enabled. Select the menu item again to disable coverage.



Type the **ce** or **cd** command on the command line:

ce

FUNCTION: Enable coverage.
 COMMAND: **ce**

FUNCTION: Disable coverage.
 COMMAND: **cd**

Two dialogs are present to give you coverage information. The code coverage dialog shows the percentage of executed code within application, module and function scope. Code coverage information can also be displayed in the Source Window. The data coverage dialog shows the data access of HLL variables in the executed program. Data coverage can also be displayed in the Memory Window. The coverage dialogs can be opened via the **Tools** menu.

FUNCTION: List coverage information to output window or file.
 COMMAND: **covinfo** *[[all | module_or_function_name][filename]]*

You can display code coverage information in the Source Window by clicking on the **Coverage** button in the Source Window. In this case an extra column appears to the right of the breakpoint toggles (to the left of the source line). For each source code line that is executed (*covered*), the source line is marked. The not executed lines are not marked. CrossView Pro has special commands to move the cursor to the next or previous covered or uncovered line:

FUNCTION: Move cursor to next covered line.
 COMMAND: **nC**

FUNCTION: Move cursor to next uncovered line.
 COMMAND: **nU**

FUNCTION: Move cursor to previous covered line.

COMMAND: **pC**

FUNCTION: Move cursor to previous uncovered line.

COMMAND: **pU**

You can display data coverage information in the Memory Window by clicking on the **Code Coverage** button in the Memory Window. Besides the current value of memory locations, the memory window also displays whether memory locations have been accessed during program execution. An application program may read from, write to, or fetch an instruction from a memory location. Of course all combinations may be legal. Although writing data to a memory location from which an instruction has been fetched is suspicious. All types of access, read, write, fetch or combinations of these, can be shown using different foreground and background colors. The color combination used to show "rwx" access are specified in the **Desktop** tab of the **File | Options...** menu item. It is advised to change the background color if instructions are fetched from a memory location, and to change the foreground color to show read and write access.

11.4 PROFILING



You can only use this feature if it is supported by the execution environment (see the addendum).

Profiling allows you to perform timing analysis on your software. Two forms of profiling are implemented in CrossView Pro. Both forms of profiling are fully implemented in the CrossView Pro debugger. You do not have to recompile your source code to enable the profiling features.

Function profiling, also called cumulative profiling, gives timing information about a particular function or set of functions. The time spent in functions called by the function being profiled is included in the timing results. Within the Cumulative Profiling Setup dialog you select one or more functions to be profiled. The gathered profile is shown in the Cumulative Profiling Report dialog. For each function the number of calls, the minimum/maximum/average and total time spent in the function are shown. Also, the relative amount of time consumed by a function in respect to the time consumed by the application is shown.

Function profile data is gathered whenever the program is executed using the Continue command (not single stepped). Function profiling can be supported if the execution environment provides a clock that starts and stops whenever execution starts and stops. Basically function profiling is implemented by using a special type of breakpoint. Breakpoints are inserted at the function entry address and all it's return addressed. Whenever execution stops due to a profile-breakpoint hit, CrossView Pro will read the clock, update the internal profile tables, and restart execution.

To specify the functions to be profiled:



From the **Tools** menu, select **Cumulative Profiling Setup...**



Type the **cproinfo** command on the command line:

```
cproinfo add main
```

To view the profiling results:



From the **Tools** menu, select **Cumulative Profiling Report...**



Type the **cproinfo** command on the command line:

```
cproinfo
```

FUNCTION:	List cumulative profiling results to output window or file, or add or remove functions from the list of profiled functions.
-----------	---

COMMAND:	cproinfo [all [, <i>filename</i>] { add remove } <i>function</i>]
----------	--

Code range profiling presents timing information about a consecutive range of program instructions. CrossView Pro displays the time consumed by each statement, C or assembly, in the source window. The timing data can be displayed in three different formats: absolute, relative to program, and relative to function. To change the display format: position the cursor on the profile column and click the right mouse button. Select the appropriate format from the popup menu.

Next to the source window, the profile report dialog (Tools | Profiling Report...) shows the time spent in each function. The time consumed by functions called from the function being profiled is not included in the displayed time.

FUNCTION: List profile information to output window or file.

COMMAND: **proinfo** [[all | *module_or_function_name*][*filename*]]

Code range profiling data is gathered whenever the program is executed. It does not matter if the program executes due to a continue, step-over or step-into command. Code range profiling heavily relies on special profiling features in the execution environment. Normally code range profiling is only supported by instruction set simulators.

To enable/disable profiling:



From the **Tools** menu, select the **Profiling** checked menu item.

When the menu item is checked, code range profiling is enabled. Enabled means that the execution environment starts gathering profiling data. Select the menu item again to disable profiling.



Type the **pe** or **pd** command on the command line:

pe

FUNCTION: Enable profiling.

COMMAND: **pe**

FUNCTION: Disable profiling.

COMMAND: **pd**

Select the **Profiling** button in the Source Window to display profile data in the Source Window. If profiling is not enabled, this button also starts gathering of profiling data.



Normally both function and code range profiling will slow down the execution speed of the application being debugged. Therefore, switch off profiling whenever the timing information is not required.

11.5 DATA ANALYSIS

CrossView Pro incorporates an advanced signal analysis interface designed to enable developers to monitor signal data more critically and thoroughly. This feature is useful when developing signal processing software for application areas such as communication, wireless and image processing.

The Data Analysis window (as shown in figure 4-15) is used for this purpose. This window is opened as result of processing a data analysis script (CXL script) and is only updated on user request. TASKING provides scripts for standard signal analysis such as x-t plotting, x-y plotting, FFT power spectrum, FFT waterfall, combined FFT power spectrum and phase, and eye diagram. However, the programmer can write CXL scripts and process the data in the format he desires.



Refer to the CXL syntax specification in section 11.5.2, *Syntax of CrossView eXtension Language (CXL)*, for details.

Four processes are associated with the graph window:

1. Get raw data
2. Transform data
3. Generate representation
4. Draw

The *get raw data* process retrieves data from the target and stores the data at the host system in one or more CrossView Pro internal acquisition data buffers. Since these buffers reside on the host system it is possible to maintain a history of data.

The *transform data* process takes the raw data as input, processes it, and the result of the transformation, a set of (x, y) pairs, is saved in the processed data buffer associated with a window. Since the transformations are described in CXL (CrossView eXtension Language) the user can program the data transformation that is of most interest for him. For example, an FFT power spectrum would produce (*frequency*, *power*) pairs.

The *generate representation* process takes data from the processed data buffer, (x, y) pairs, as input and generates a display list. This process scales the data according to the given display window size. This process is coded in CXL. So, in addition to the scripts provided by TASKING, the user can write his own representation processes. For example, an FFT power spectrum is usually represented by a bar graph.

The *drawing engine* process takes the display list as input and produces the graph that is displayed in the Data Analysis window. The drawing engine is part of the CrossView Pro executable and cannot be configured by the user.

A clear separation between data transformation (the transform data process) and data presentation (generate representation process) has been made to increase the reusability of complex data presentation scripts.

Once the scripts are written (a number of frequently used operations are supplied), the following three steps must be made in order to display data:

1. Set the display mode for the desired window using the **graphm** command. For example,

```
graphm "demo", "show_x_t.cxl"
```

"demo" will be shown in the title bar of the window. It is also the name used to refer to the window.

2. Retrieve data from the target into a buffer using the **memget** command. For example,

```
memget ((int []) 0x0)[$i],128,$buffer
```

`$i` is the "iterator" to walk 128 times through the expression (Note: the retrieved elements are assumed to be equidistantly placed in memory) and store the results in `$buffer`.

Optionally the buffer contents can be appended to another buffer using the **bufa** command, in order to maintain a (limited) history. For example,

```
bufa $all_data,$buffer,1024
```

3. Transform the buffer contents to displayable data using the **graph** command. For example,

```
graph "demo","x_t.cxl",$buffer,0,1
```

For details of the arguments provided to `x_t.cxl`, see below. Now a used buffer can be freed using the **bufd** command (if the target data is not to be used anymore).

Steps 1. and 2. can be repeated as many times as desired. The display mode can be changed at any time by issuing a **graphm** command for the window to be changed. Using the **graphp** command, a window can be positioned anywhere on the screen.

11.5.1 SUPPLIED DATA ANALYSIS WINDOW SCRIPTS



The following scripts and commands are described for completeness. Normally, you will not use the commands directly, because they are automatically invoked when you click **OK** in the Data Analysis Window Setup dialog.

For some **graphm** scripts both x- and y-axis can be user specified. If the limits are not specified or low \geq high, then autoscaling is used.

X-T plotting

An x-t plot is the most straightforward way of displaying data. Data is taken from one buffer, each value is taken as the x value and the t value is increasing linearly. It is displayed as a graph the way it is found in the buffer (memory). The layout of the scales and the form of the graph (line, bar, dot) can be selected as shown below.

1. Generating window data pairs:

graph "win_title", "x_t.cxl", \$buffer, t_offset, t_increment

generates (t, x) pairs: ($t_offset + i * t_increment$, \$buffer[i]). The generated data is attached to the specified window.

2. Setting the display mode:

graphm "win_title", "show_x_t.cxl" [, low_x, high_x [, low_y, high_y]]

displays lines drawn between successive coordinates specified by the window data pairs.

graphm "win_title", "show_cross.cxl" [, low_x, high_x [, low_y, high_y]]

displays 'x's at the coordinates specified by the window data pairs.

graphm "win_title", "show_plus.cxl" [, low_x, high_x [, low_y, high_y]]

displays '+'s at the coordinates specified by the window data pairs.

graphm "win_title", "show_bars.cxl" [, low_x, high_x [, low_y, high_y]]

displays bars at the coordinates specified by the window data pairs. The x-coordinates are expected to be equidistant.

X-Y plotting

An x-y plot takes values from two buffers, one from each at a time. The first is interpreted as the x-value, the second as the y-value of a point to display. No further processing is done on these values. The most common display mode is 'x's or '+'s (`show_cross.cxl`, `show_plus.cxl`, see previous description) to give a scattergram. The values can also be interconnected in order (`show_x_y.cxl`) to create Lissajous-like displays.

1. Generating window data pairs:

graph "win_title", "x_y.cxl", \$x_buffer, \$y_buffer

2. Setting the display mode:

graphm "win_title", "show_x_y.cxl" [, low_x, high_x [, low_y, high_y]]

draws lines from all ($x[i]$, $y[i]$) to ($x[i+1]$, $y[i+1]$). When autoscaling is active, some space is reserved on both x- and y-axis.

FFT power spectrum

The FFT power spectrum plot takes a buffer of arbitrary size to compute the power of all frequencies present in the signal (in decibels). If the buffer size is not a power of two, it will expand its input set to the next higher power and augment it with zeroes. To handle non-recurrent data correctly, several window functions can be applied in the process. If no reference level is given the maximum level is calculated and set to be 0 dB. The usual display mode is bars, although all x-t display methods can be used. The horizontal axis is in frequency steps, the vertical axis in decibels.

1. Generating window data pairs:

graph "win_title", "fft.cxl", \$buffer, filter_index, frequency_step[,ref_level]

generates pairs ($i * frequency_step$, $\log_power[i]$). The *filter_index* specifies one of the following FFT windowing functions:

- 0 rectangular
- 1 triangular
- 2 Hanning
- 3 Blackman-Harris

ref_level is the 0 dB reference level.

2. For displaying the generated pairs, any of the x-t plotting display scripts can be used. "show_bars.cxl" is recommended.

Multi FFT power spectrum ("waterfall")

The multi FFT power spectrum displays a chronological series of FFT power spectra. This diagram is also known as FFT waterfall. The FFT power spectrum plot takes a buffer of arbitrary size and splits it up in a number of frames of size 2^{two_exp} . You can specify the overlap between successive frames. The overlap can be negative indicating gaps between successive frames. For each frame, the power (in decibels) of all frequencies present in the signal is computed.

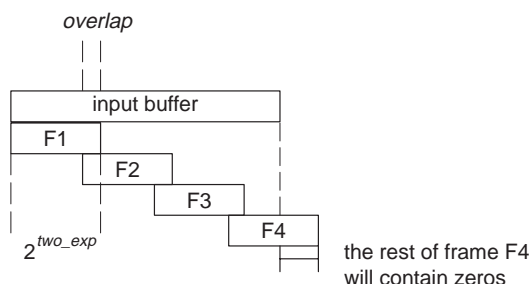
1. Generating window data pairs:

graph "win_title", "multi_fft.cxl", \$buffer, filter_index, frequency_step, two_exp[,overlap[,ref_level]]

generates pairs ($i * frequency_step$, $\log_power[i]$). The *filter_index* specifies one of the following FFT windowing functions:

- 0 rectangular
- 1 triangular
- 2 Hanning
- 3 Blackman-Harris

2^{two_exp} is the width of one single frame in number of input samples. two_exp must be a value between 2 and 14 (inclusive). If the input buffer does not contain enough samples to fill the last frame, the frame is completed with zeros.



$overlap$ is the number points shared by successive frames. When negative, a 'gap' will occur between processed points. The first sample taken from the input buffer of frame N is equal to the first sample of frame $N + 2^{two_exp} - overlap$. $overlap$ must be smaller than 2^{two_exp} .

ref_level is the 0 dB reference level.

2. For displaying the generated pairs, the display script "show_multiBars.cxl" is required.

Multi FFT power spectrum in lines

Displays the same multi FFT power spectrum, but now in lines instead of bars. Here a 3D graph is shown. The script name is show_multi_lines.cxl.

Multi FFT power spectrum in lines and grid

Displays the same multi FFT power spectrum as the multi lines spectrum.. Now each point on a curve is interconnected with a point with the same x-coordinate of the previous graph. What you see here is a 'grid' with the values. The script name is show_multi_grid.cxl.

Use of colors in Multi FFT power spectrum

For all three **graphm** scripts `show_multi_bars.cxl`, `show_multi_lines.cxl` and `show_multi_grid.cxl` an optional third parameter can be added to set the color offset value. This allows you to create a dynamic display in which the color of each curve remains the same. The color offset can range from 0 to the maximum number of colors, and the maximum number of colors is the number of curves to be plotted. When the color offset exceeds the number of colors, the modulo will be taken; if it is negative it will be set to zero. The colors selected for the curves are spread evenly over the color spectrum. The number of colors can also be set as an (optional) fourth parameter of the script.

An example of a command file for a running script can be:

```
/* INITIALIZE */
rst          /* Rerun the program when the script is executed */
$fast_mode=2 /* If on the simulator, go to fast mode          */
s            /* Step to the main() routine to allow access    */
            /* to the output[] array.                          */
memget output[$i],256,$t /* It's clear now. */
bufa $f,$t,4096 /* Construct an empty time domain history */
bufa $f,$t,4096
bufa $f,$t,4096
bufa $f,$t,4096
bufa $f,$t,4096
bufa $f,$t,4096
bufa $f,$t,4096
bufa $f,$t,4096
$color=0 /* Initialize the $color variable to track the graphs */

/* DEFINE THE TIME DOMAIN WINDOW */
graphp "Output time domain",50,25,716,295 /* set window position */
graphm "Output time domain","show_x_t.cxl" /* set draw method */
graph "Output time domain","x_t.cxl",$t,0,1
/* use the 't' buffer */

graph_clear_updates "Output time domain"
/* Set 'Output time domain' window update actions: */
graph_add_update "Output time domain",memget output[$i],256,$t
/* Get new time domain data from output[] into $t buffer */
graph_add_update "Output time domain",graph "Output time
domain","x_t.cxl",$t,0,1
/* This command recalculates and redraws the window */

/* DEFINE THE FREQUENCY DOMAIN WINDOW */
graphp "Output freq domain",50,350,716,295
/* set window position */
graphmn "Output freq domain","show_multi_grid.cxl",-120,5,($color)
/* set draw method */
graph "Output freq domain","multi_fft.cxl",$f,0,1,256
/* use the 'f' buffer */
```

```

graph_clear_updates "Output freq domain"
/* Set 'Output freq domain' window update actions: */
graph_add_update "Output freq domain", bufa $f,$t,4096
/* Add new data to buffer, max size 4096 (purging oldest) */
graph_add_update "Output freq domain", $color = ($color+1) % 16
/* 4096/256 = 16 graphs, increment color offset to follow */
graph_add_update "Output freq domain", graphmn "Output freq
domain","show_multi_grid.cxl",-120,5,($color)
/* Use the graphmn command to avoid double redraws */
/* Place $color in braces to avoid confusion with buffers */
graph_add_update "Output freq domain",graph "Output freq
domain","multi_fft.cxl",$f,0,1,256
/* This command recalculates and redraws the window */

/* PLACE COMPLEX BREAKPOINT, HAVE IT UPDATE THE GRAPHICAL DATA
WINDOWS */
main#141 bi { update! "Output time domain"; update! "Output freq
domain"; C }

/* CONTINUE RUNNING THE PROGRAM */
C

```

For passing the parameter `$color`, the command interpreter requires parentheses around it, otherwise it is interpreted as a buffer.

Combined FFT power spectrum and phase

The combined FFT power spectrum and phase plot adds a display of the phase of each component to the FFT power spectrum. The phase is normalized between -180 degrees and $+180$ degrees. To display both features of the input data a special display script must be used (`show_fft_pairs.cxl`).

1. Generating window data pairs:

`graph "win_title", "fft_pairs.cxl", $buffer,filter_index,freq_step[,ref_level]`

The *filter_index* specifies one of the following FFT windowing functions:

- 0 rectangular
- 1 triangular
- 2 Hanning
- 3 Blackman-Harris

ref_level is the 0 dB reference level.

2. Setting the display mode:

For displaying the generated display list, the display script `"show_fft_pairs.cxl"` is required.

graphm "win_title", "show_fft_pairs.cxl" [, min_power, max_power]

Eye diagram

The eye diagram is a recurrent x-t plot. The input data is not processed, but the time parameter is reset when the signal crosses the trigger level, and also after a specified interval (*wrap_limit*). After crossing *trigger_level*, retriggering is suppressed during the *trigger_hold_off* next data values. The eye diagram uses the X-t plot method and exploits the feature of suppressing the fly-back of the displayed line.

1. Generating window data pairs:

graph "win_title", "eye.cxl", \$buffer, wrap_limit [, t_increment [, t_offset [, trigger_level [, trigger_hold_off]]]]

2. Setting the display mode:

graphm "win_title", "show_x_t.cxl" [, low_x, high_x [, low_y, high_y]]

displays lines drawn between successive coordinates specified by the window data. If $x[i+1] < x[i]$ (going back in time), no line is drawn from $(x[i], y[i])$ to $(x[i+1], y[i+1])$, which can be regarded as the fly-back suppression in an oscilloscope.

11.5.2 SYNTAX OF CROSSVIEW EXTENSION LANGUAGE (CXL)

CXL has a C-like syntax, the basic differences from C are:

- No preprocessor, so no defines.
- Only "//" comments.
- No structs or unions, so the operators "." and "->" are not supported.
- No type definition
- No enums
- No switch statement.
- Blocks must not be empty ("1;" is the minimal expression).
- No 'main', all the script code is to be enclosed within a '{' and '}' pair.

- Function prototypes and function definitions can be nested, but must be preceded by the keyword "sub". They can be used anywhere in the source. Following the scope rules, a function declaration hides a previous definition when it is defined.
- No casts allowed. Casts are (like in C) performed automatically. When explicit rounding/casts are needed, you can use the `floor()` function. E.g. by `floor(x)` (chopping) or `floor(x+0.5)` (rounding off to nearest integer).
- No `? :` operator allowed.
- Single statements after a flow control statement (if, else, for, while) should *always* be between braces. For example, the usual C expression

```
if (x < 0 )
    x = 0;
```

should be written as

```
if (x < 0 )
{
    x = 0;
}
```

- Initializers in declarations are not allowed. For example,


```
int i = 1;
```

 should be written as


```
int i;
... possible other variable declarations ...
i = 1;
```
- Modifiers such as `signed`, `unsigned`, `register` and `static` are not supported.
- Floating point numbers below 1 should always be preceded by a zero. For example, the number `.15` is treated as invalid, this should be `0.15`.

Furthermore, the syntax is like the C syntax.

Example:

```
{
    sub void p(function f)
    {
        outd(f());
        outc('\n');
    }

    sub int h() { return 1; }

    {
        //This is the "main" entry point
        p(h);
        sub int h() { return 2; }
        p(h);
    }
}
```

This example would print the following output in the command window:

```
1
2
```

Base Types

CXL supports the following base types:

- char
- int
- long
- float
- double
- string (only allowed for parameters)
- function (only allowed for parameters)

Internally, char, int and long are treated the same, as are float and double. Since they are the same, types belonging to one group can be interchanged freely.

Pointer to base type is only supported for parameters not for other variables. Pointers to variables are the result of the "address-of" operator and are treated as arrays of the mentioned base type with upper-bound 1.

A return value can be of any base type. Data type `void` is also a valid return type.

Compound Types

CXL supports the following compound types:

- array of `char`
- array of `int`
- array of `long`
- array of `float`
- array of `double`

Structures, unions and type definitions are not part of the CXL syntax.

Predefined Functions

1. Mathematical functions:

```
double sin(double x);
double cos(double x);
double tan(double x);
double acos(double x);
double asin(double x);
double atan(double x);
double sinh(double x);
double cosh(double x);
double tanh(double x);
double log(double x);
double log10(double x);
double exp(double x);
double sqrt(double x);
double ceil(double x);
double floor(double x);
double fabs(double x);
double pow(double x, double y);
```

2. Functions to send output to the command window:

```
double outc(double x); -> { printf("%c", (int) x); return x; }
double outd(double x); -> { printf("%ld", (long) x); return x; }
double outf(double x); -> { printf("%f", x); return x; }
```

3. Upperbound of an array:

```
long upperbound(array a);
```

4. GUI interaction functions available when a script is passed to the **graph** command::

```
void add_point(double x, double y);
```

This function adds graph points to the acquisition buffer.

```
void printf(string format, ...);
```

The output of `printf` is written to the command window.

5. GUI interaction functions available when a script is passed to the **graphm** command::

```
void printf(string format, ...);
```

The output of `printf` is sent to the "window contents script".



This at first sight strange function name of `printf` is chosen to facilitate development and debugging **graphm** scripts using a host system C development environment. The C code can be very easily ported to CXL afterwards. The output is in fact a command of the *drawing engine* and is therefore *not* the same as a usual `printf` and no the same as `printf` in the **graph** command. Logging to the command window from a **graphm** script is not possible via `printf`.

The following drawing commands are supported:

clear

Clear drawing area. This is usually the first command issued in a drawing sequence.

graph_area *x-offset, y-offset, x-size, y-size*

```
printf("graph_area %d,%d,%d,%d/n", xo, yo, xs, ys)
```

Set graph area size. The offset determines the lower left corner of the graph area. Size is the exact number of pixels.

axis *xlow, ylow, xhigh, yhigh*

Define the axes ranges, for determining the cross-hair cursor coordinates (to be displayed in the cursor field and to be passed to the representation generator). The axes range up-to the top-right coordinate, which is excluded (reduces axis drawing maths, but mind axis lengths of 0). The axes are linear.

pen_color *color*

Set pen color. Black is the default color. The *color* can be specified by name or by RGB number in the form *red,green,blue* as decimal number for each base color for 0 to 255. E.g. 255,128,0 is orange. Valid names are:

black, red, yellow, green, blue, cyan, magenta,
dkgray, gray, ltgray, white

brush_color *color*

Color used for filling areas. Black is the default color. See **pen_color** for possible colors. The value **background** sets the brush to the current background color which is WINDOW_BACKGROUND under Windows.

filled_rectangle *x1, y1, x2, y2*

Puts a rectangle, filled with the latest set brush_color, bounded by (x1, y1) and (x2, y2) (both points inclusive). Coordinates are expressed in pixels. The origin is the lower left corner.

dot *x1, y1*

Draw pixel. Coordinates are expressed in pixels. The origin is the lower left corner.

line *x1, y1, x2, y2*

Draw line from (x1, y1) up-to and including (x2, y2). Coordinates are expressed in pixels. The origin is the lower left corner.

polygon *x1, y1, x2, y2, ... xn, yn***polyline** *x1, y1, x2, y2, ... xn, yn*

Puts a polyline, using the latest set pen_color, from line sections from the points (x1, y1) to (xn, yn), where $n \geq 3$ (which means at least 2 lines). Coordinates are in pixels.

filled_polygon *x1, y1, x2, y2, ... xn, yn*

Puts a polygon, filled with the latest set brush_color, bounded by a polygon formed by the line sections between the points (*x1*, *y1*) to (*xn*, *yn*) and back to (*x1*, *y1*), where $n \geq 3$. Coordinates are in pixels. As with **filled_rectangle**, the pen is only temporarily set to the same color as the current brush and restored when the call is finished.

filled_polygon_brush *x1, y1, x2, y2, ... xn, yn*

As with **filled_polygon**, but using a separate brush and pen, that is, using the latest brush and pen color also when they are different.

text *x, y, anchor, "text"*

Draw text with its anchor at location (*x*, *y*). The anchor is the point in the text string, which will get placed at the specified location. For example, anchor 7 specifies that the text must be placed such that the bottom-left side of the text is at the specified position. Coordinates are expressed in pixels. The origin is the lower left corner.

Anchors:

```

1-----2-----3
|           |
4         5         6
|           |
7-----8-----9

```

Text may include any characters, except a nil character. Double quote and backslash characters must be escaped by a backslash character. Text will be formatted using the current font settings. See below for the font info exchange between the window and the representation generator.

```
long get_attr(string attribute);
```

Supported attributes are:

```
"draw-area-x-size"
"draw-area-y-size"
"x-scrollbar-present"
"x-scrollbar-size"
"x-scrollbar-low"
"x-scrollbar-high"
"y-scrollbar-present"
"y-scrollbar-size"
"y-scrollbar-low"
"y-scrollbar-high"
"selection-available"
"selection-start"
"selection-end"
```

```
long get_text_attr(string attribute, string text_format, ...);
```

Supported attributes are:

```
"leading"
"ascent"
"descent"
"width"
```

6. Argument passing.

The **graph** and **graphm** commands can be given a number of arguments. These arguments are accessible as follows.

```
long n_args;
```

The number of arguments.

```
arg1..argN
```

with $N = n_args$ are added to the global scope and have type `double *` for buffers, type `string` for strings and type `double` for evaluated expressions.

Parsing the script will fail if a certain argument has not been provided. Evaluation of the script will fail if the type of the argument does not match its use.

For argument testing and argument retrieval the following functions are provided:

```
long is_string_arg(long n);
long is_double_arg(long n);
long is_buffer_arg(long n);
double get_double_arg(long n);
string get_string_arg(long n);
double *get_buffer_arg(long n);
```

Numerical arguments can be retrieved by using `get_double_arg()`. In the **graphm** command, the (x, y) pairs produced by a sequence of calls to `add_printf()` in the **graph** script are accessible via global variables `x` and `y` of data type `array of double`.

11.6 BACKGROUND MODE

Background mode is a feature for running the application under debug and CrossView Pro at the same time. This allows you to monitor the target application using CrossView Pro, while the application is running. Depending on the target hardware and/or debug instrument connected to CrossView Pro, target execution can even be real-time.

Since CrossView Pro’s monitoring of the target hardware must be non-intrusive, not all functions of the debugger are enabled while running in background mode.



You can only use this feature if it is supported by the execution environment (see the addendum).

11.6.1 CONFIGURATION

CrossView Pro can be instructed to automatically refresh one or more windows of the debugger periodically while running in background mode. You can use the Background Mode Setup dialog for specifying the desired set of windows to be refreshed.



From the **Settings** menu, select **Background Mode Setup...** to open the Background Mode Setup dialog.



A distinction has been made between updating the Source lines window and updating the Disassembly window. Updating the Disassembly window may be to time-consuming, so you may want to disable its updating in Background mode, while still keeping the Source lines window up-to-date when that is displayed on screen.



Use the **u** command to toggle the updating of windows in background mode.

FUNCTION:	Toggle update of window in background mode.
COMMAND:	<code>[interval] u [d k r cd ck cr s a mem t]</code>

The following windows can be updated in background mode:

d (Data), **k** (Stack), **r** (Register),
s (Source), **a** (Assembly), **mem** (Memory), **t** (Trace)

Initially only the data window will be updated. CrossView Pro repeatedly looks at the execution environment to react on changes. It pseudo-simultaneously looks for user commands from the keyboard (or from the playback file), and periodically it updates the windows.

If all windows would be updated the update frequency would drop. That is why you can toggle a switch for each window. To toggle the updating of the register window, you can type:

```
xvw% u r
```

If the switch for a window is 'on', it will be updated, otherwise it will be skipped.

You can also specify a new update interval.

Without arguments, CrossView Pro displays all windows updated periodically plus the update interval.



Notice that simulated I/O is done through 'invisible' breakpoints, and these must be handled inside the loop. Hence, if updating the windows takes a lot of time (many monitor commands), it will also slow down simulated I/O.

11.6.2 MANUAL REFRESH

If you have windows which you do not want to refresh periodically, you can disable them in the Background Mode Setup dialog's refresh list, and refresh these windows manually.



From the **View** menu, select **Background Mode** and select one of the refresh options.



Use the **ubgw** command.

FUNCTION: Update the appropriate window when the target runs in the background.

COMMAND: **ubgw** [**s** | **a** | **k** | **r** | **d** | **mem** | **t** | **all**]



Section *Refresh Limitation* in this chapter.

11.6.3 ENTERING BACKGROUND MODE

To run a program in background mode:



From the **Run** menu, select **Background Mode | Run in Background**



Type the **CB** command on the command line.

FUNCTION: Run a program in background mode.

COMMAND: [*count*] **CB** [*linenumber*]

This will start the application under debug to run continuously (as with the **C** command), and switch CrossView Pro from Halted to Background Mode. *count* is assigned to the breakpoint at the current execution position as the number of times to hit this breakpoint before execution to stop. *linenumber* specifies the source line to place a temporary breakpoint.

The mouse pointer changes to an arrow with a small watch face underneath. This indicates that CrossView Pro is now in background mode. Some commands are treated a little different in this mode, because they can otherwise influence the running program badly. Commands that need information from the stack (like **bU**, **bu**, **bb** or **bbB**) are not allowed because that information is not reliable. Other commands require great care, for example the **o** command.

For example if you type the **g** while in background mode you will see:

```
xvw% g 56
```

```
Command "g" is not allowed while the emulator is
running in background.
```

11.6.4 LEAVING BACKGROUND MODE

You can leave Background Mode in three ways:

1. Stop the target immediately:



From the **Run** menu, select **Background Mode | Halt Target**



Enter the **st** command:

```
xvw% st
```

2. Let CrossView wait for the target to stop:



From the **Run** menu, select **Background Mode | Wait for Target to Stop**



To wait for a breakpoint, you can use the **wt** command:

```
xvw% wt
```

3. A program running in background mode also stops when it encounters a breakpoint.

FUNCTION: Stop a program in background mode.

COMMAND: **st**

The **wt** command behaves just as if you have typed the **C** command. CrossView Pro returns with a prompt, after the program hits a breakpoint. However, there is an interesting difference with the **C** command. If you push the **Halt** button, it returns with the background prompt. The program that runs in the execution environment continues without interruption.

FUNCTION: Wait for the running process to stop

COMMAND: **wt**

11.6.5 THE STACK IN BACKGROUND MODE

While the execution environment runs in background, CrossView Pro does not allow the use of information that comes from the stack. The reason is that the running program must be stopped in order to get consistent information from the stack. Stopping (and afterwards continuing) the program conflicts with the "real-time" nature of the background mode.

If there is a need for it, you can make a macro that performs the desired operations.

11.6.6 LOCAL AND GLOBAL VARIABLES

In background mode you can continuously monitor variables. However, realize that local variables (in CrossView Pro variables are called 'local' if they reside on the stack) cannot be monitored. Instead you will see "unknown name". Global variables have a fixed address, so CrossView Pro knows where to get their contents from.

If you are very anxious to see local variables you can first get an address and then use that address to monitor the contents. For example:

```
$adr_sum = &sum
m *(adr_sum)/x4
```

In this example `sum` is a long (4 bytes). You must be sure that `sum` remains at that address while the program is running.



The values you get this way are only valid under specific conditions. Local variables from the function `main` normally meet these conditions.

11.6.7 REFRESH LIMITATION

While running the application in the background mode, the automatic refresh functionality may not be able to keep up with all the debugging information produced by the running target. Typically, the collected information will be correctly displayed and automatically updated in the current open views and no information will be lost. You might lose the debugging information when scrolling these views during the background mode. The reason is that either CrossView Pro does not run fast enough or the communication with the target hardware is not handled fast enough by the operating system.

The information that cannot be processed by CrossView Pro within the specified update interval, is displayed as either '<unknown>' or dashes. The way the lost information is displayed depends on the internal communication level within CrossView Pro where the information is lost. Information lost during communication with the target hardware is displayed as '<unknown>'. Information lost by CrossView Pro while processing and interpreting this information, is displayed as dashes.

On the next automatic or manual update, all debugging information in the currently open views is automatically updated. All visible '<unknown>' values and dashes are replaced with their actual values as produced by the running target.

11.6.8 ASSERTIONS

CrossView Pro automatically suspends assertions with the **CB** command.

CHAPTER

12

DEBUGGING NOTES



TASKING



12

CHAPTER

Here are a few notes about debugging in special situations:

12.1 DEBUGGING ASSEMBLY LANGUAGE

You may debug assembly language programs or modules much as you do C source. The **s**, **S** and **si** commands single step through the assembly source. You may place code breakpoints on assembly language instructions with the **bi** command.

For additional information on debugging assembly code, see \$autosrc, \$mixedasm and \$symbols in the *Reserved Special Variables* table in section 3.4.

There is a restriction on debugging assembly language code:

- Assembly language subroutines cannot be called from the command line.

12.2 DEBUGGING MULTIPLE PROGRAMS

You probably have only one linked and located absolute object file that describes the whole system load. However, for various reasons, you may want to build your system load by linking and locating into several files. The debugger can handle the symbols from only one load module in one absolute object file at a time. Consequently, if there are several absolute files or several load modules within one absolute file, you will have to change the context from one to another explicitly. Use the **N** command or the Load Symbolic Debug Info dialog to load the appropriate symbols. This does not disturb the state of the target system.

You can also download the image part of another absolute object file (using the **dn** command), without leaving the debugger.

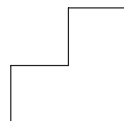
CHAPTER

13

COMMAND REFERENCE



TASKING



13

CHAPTER

This chapter contains a summary of all CrossView Pro commands, followed by a complete description of each command.

13.1 CONVENTIONS USED IN THIS CHAPTER

Each CrossView Pro command has a particular syntax, that is, the form it must take for CrossView Pro to recognize it. To help you learn the syntax of each command, this chapter uses a special notation to describe the syntax of each command. Consider the following example:

ios_read {*stream* | "*file*"},*address,number_of_maus*[**x**]

Command items in **bold** font are the actual command keywords typed from the keyboard. In the example above, **ios_read** is in bold font since you must type it exactly as shown.

Items in *italics* are names of the command part. Here *stream* is in italics, since you must substitute the appropriate value for stream. The Description section for each command describes what kinds of values should be substituted for italicized terms.

Expressions in [brackets] are optional items you may include in a particular command. In this example, **x** is not necessary for the **ios_read** command to work. Usually if you omit an optional expression, CrossView Pro uses a default value.

The | symbol means *or*. For instance, {*stream* | "*file*" } means a stream number or a filename between double-quotes (but not both) can be used in the command.

13.2 COMMANDS: SUMMARY

13.2.1 VIEWING COMMANDS

- ^[*format*]** Display contents of preceding memory location.
- exp*** Print value of expression using */n* format.
- exp @ format*** Print address of expression *exp* in format *format*.
- exp/format*** Print value of expression *exp* in format *format*.
- line*** Move viewing position to line *line*.
- clear** Clear the Command Output Window.
- number ct*** Display a source-level trace corresponding to the last number of machine instructions executed.
- number ct i*** Display a disassembled assembly-level trace corresponding to the last number of machine instructions executed.
- number ct r*** Display a raw trace corresponding to the last number of trace frames.
- e [*func* | *file*]** Enter function *func* or file *file* or view current viewing position.
- stack e** Enter function using stack address.
- [*addr*] ei** View current viewing position or view instruction at address *addr*.
- f ["*printf-style-format*"]** Change default address display format.
- gus {on | off}** Suppress or reactivate CrossView Pro window updating.
- L** Synchronize the viewing position with the execution position. Print current file, function and line number.

- l** {**a**|**b**|**d**|**f**|**g**|**k**|**l**|**L**|**m**|**p**|**r**|**s**|**S**} [*string*]
 List **a**ssertions, **b**reakpoints, **d**irectories, **f**iles, **g**lobals, **k**ernel state data, **l**abels (on module scope), all **L**abels, **m**emory map (of application code sections), **p**rocedures, **r**egisters, **s**pecial variables, **S**ymbol tables. If given, only those starting with *string*.
- l** [*func*] List all parameters and locals of function *func*. Without a function, this command lists all parameters and locals of the current function in view.
- l** *stack* List all parameters and locals of function at depth *stack*.
- nC** Move viewing position to next covered line.
- nU** Move viewing position to next uncovered line.
- opt** [*option* [= *value*]]
 List or set option value. Without an argument, list all option values.
- [*line*] **P** [*exp*] Print *exp* lines of source starting at line *line*, include machine addresses.
- [*line*] **p** [*exp*] Print *exp* lines of source starting at line *line*.
- pC** Move viewing position to previous covered line.
- pU** Move viewing position to previous uncovered line.
- [*exp*] **T** Trace the stack for *exp* number of levels, list local variables.
- [*exp*] **t** Trace the stack for *exp* number of levels, printing active functions and parameters passed.
- td** Disable tracing.
- te** Enable tracing.

13.2.2 DATA MONITORING

- cd** Disable, turn off, gathering of coverage data.
- ce** Enable, turn on, gathering of coverage data.

- covinfo** `[[all | module_or_function_name][,filename]]`
List coverage info.
- cproinfo** `[all[,filename] | {add | remove }function]`
List cumulative profiling info or add or remove functions from the list of profiled functions.
- dis** `address [, {address|#count} [,i]]`
Disassemble a range of memory.
- dump** `address [, {address|#count} [,style [width]] [,filename [,a]]]`
Dump a memory range.
- M** Display list of monitored expressions in the Command window.
- m** `exp` Monitor the expression `exp`.
- `num` **m d** Remove monitored expression labeled `num`.
- `addr_start` **mcp** `addr_end, addr_dest`
Memory copy.
- `addr` **mF** `exp[,exp]...`
Single fill memory address `addr` with expressions.
- `addr_start` **mf** `addr_end, exp[,exp]...`
Fill memory address range with expressions and repeat the pattern until the end address of the memory region is reached.
- `addr_start` **ms** `addr_end, exp[,exp]...`
Search memory address range for a given pattern.
- pd** Disable, turn off, profiling.
- pe** Enable, turn on, profiling.
- proinfo** `[[all | module_or_function_name][,filename]]`
List profiling info.

13.2.3 DATA ANALYSIS

bufa *target_buffer_name,added_buffer_name[,size_limit]*
Add the contents of buffer *added_buffer_name* to buffer *target_buffer_name*.

bufd *buffer_name*
Discard the specified buffer.

graph *"window","script"[,arg]...*
Create Data Analysis window and execute CXL script.

graphm *"window","script"[,arg]...*
Set the representation script for the window specified.

graphmn *"window","script"[,arg]...*
Similar to the **graphm** command, but without an update of the graph window.

graphp *"window",left_top_x,left_top_y,width,height*
Position the named window at the specified screen coordinates.

graph_add_update *"window",command*
Add *command* to the sequence of update commands for the specified window.

graph_clear_updates *"window"*
Clear the update commands associated with the specified window.

graph_close *"window"*
Close the specified window.

graph_debug *expression*
Enable the "graphical data window debugging mode", showing all communication between the scripts and the windows in the command window.

memget *expr,count,buffername*
Retrieve symbolically specified data from the target system and store the data in the acquisition buffer.

rawmemget *address,type,count,buffername [,interleave]*
Retrieve data from the target system and store the data in the acquisition buffer.

update "*window*"

Update the window specified.

13.2.4 EXECUTION CONTROL COMMANDS

A [**a** | **s**] Toggle state of assertion mechanism.

a *cmds* Create a new assertion with the command list *cmds*.

exp **a** {**a** | **d** | **s**}
 Activate, delete, suspend assertion *exp*.

B List all breakpoints.

[*line*] **b** [*cmds*]
 Set breakpoint at source line *line*, and associate command list *cmds* with breakpoint.

[*stack*] **bb** [*cmds*]
 Set temporary breakpoint at beginning of function at stack level *stack* and associate command list *cmds*.

[*stack*] **bb** [*cmds*]
 Set breakpoint at beginning of function at stack level *stack* and associate command list *cmds*.

[*number*] **bc** [*count*] [*reset_count*]
 Set breakpoint *count* and *reset_count* for breakpoint with number *number*.

count **bcyc** [*cmds*]
 Set temporary breakpoint after the specified cycle *count* and associate command list *cmds*.

count **bcyc** [*cmds*]
 Set breakpoint after the specified cycle *count* and associate command list *cmds*.

exp **bD** {**r** | **w** | **b**} *exp2* [*cmds*]
 Set a data range breakpoint (between addresses *exp* and *exp2*) read (**r**), write (**w**) or both read and write (**b**), and associate command list *cmds*.

exp **bd** {*r* | *w* | *b*} [*cmds*]

Set a data breakpoint, read (*r*), write (*w*) or both read and write (*b*) at address *exp*, and associate command list *cmds*.

num **bdis** Disable code breakpoint.

num **bena** Enable code breakpoint.

[*addr*] **bi** [*cmds*]

Set temporary breakpoint at machine instruction and associate command list *cmds*.

[*addr*] **bi** [*cmds*]

Set breakpoint at machine instruction and associate command list *cmds*.

count **binst** [*cmds*]

Set temporary breakpoint after *count* machine instructions and associate command list *cmds*.

count **binst** [*cmds*]

Set breakpoint after *count* machine instructions and associate command list *cmds*.

break [*type*] *where* [, *option*]...

Universal breakpoint command. Several *types* of breakpoints are available. The meaning of *where* depends on the selected *type*. Breakpoint *options* must be separated by commas.

time **btim** [*cmds*]

Set temporary breakpoint after *time* number of seconds and associate command list *cmds*.

time **btim** [*cmds*]

Set breakpoint after *time* number of seconds and associate command list *cmds*.

[*stack*] **bu** [*cmds*]

Set a temporary up-level breakpoint at stack level *stack* and associate command list *cmds*.

[*stack*] **bu** [*cmds*]

Set up-level breakpoint at stack level *stack* and associate command list *cmds*.

<i>[exp]</i> C [<i>line</i>]	Continue execution from current value of program counter. If <i>line</i> is specified, execution continues up to that line. Breakpoint's count is set to <i>exp</i> .
<i>[exp]</i> CB [<i>line</i>]	Continue execution in background from current value of program counter. If <i>line</i> is specified, execution continues up to that line. Breakpoint's count is set to <i>exp</i> .
D	Delete all breakpoints.
Dy	Delete all breakpoints without prompt for confirmation.
<i>[number]</i> d	Delete breakpoint number.
<i>cpu</i> eC	Start execution on the current CPU and switch to <i>cpu</i> .
<i>[cpu]</i> ec	Select CPU or show current CPU number.
g <i>line</i>	Go to the specified line in the current procedure.
<i>address</i> gi	Go to the specified address.
if (<i>exp</i>) { <i>cmds</i> } [<i>cmds</i>]	Conditionally execute commands.
prst	Reset program counter.
Q	Report breakpoint quietly.
q [<i>y</i>]	Quit debugger (do not save desktop settings).
q s	Save current desktop settings and quit debugger.
R	Reset program counter and start execution.
rst	Reset target system to initial conditions.
<i>[exp]</i> S	Single step for <i>exp</i> lines, step over function calls.
<i>[exp]</i> s	Single step for <i>exp</i> lines, step into function calls.
<i>[exp]</i> Si	Single machine step for <i>exp</i> machine instructions, step over subroutine calls.
<i>[exp]</i> si	Single machine step for <i>exp</i> machine instructions, step into subroutine calls.

- st** Stop the execution of the target immediately.
- [interval] u [d|k|r|s|a|mem|t]** Toggle updating of the appropriate window when the target runs in the background. You can specify the update interval, in seconds. If *interval* is zero, never update automatically.
- ubgw [s|a|k|r|d|mem|t|all]** Refresh the appropriate window, or all open windows, when the target runs in the background. This command is not available for all execution environments.
- use [path]...** Clear source directory search path or use the specified path to search for source files.
- wt** Wait for the completion of the target.
- [exp] x** Force an exit from assertion mode. If *exp* is non-zero, finish executing command list of the current assertion.

13.2.5 RECORD & PLAYBACK

- <file** Play back commands from *file*.
- <<file** Play back commands with single step from *file*.
- >file** Record CrossView Pro commands in *file*.
- >{t|f|c}** Set recording file status, true (**t**), false (**f**) or closed (**c**).
- >** Report status of command recording mechanism.
- >#file** Record emulator commands in *file*.
- >#{t|f|c}** Set emulator recording file status, true (**t**), false (**f**) or closed (**c**).
- >@file** Record CrossView Pro and emulator commands in *file*.
- >@{t|f|c}** Set CrossView Pro/emulator recording file status, true (**t**), false (**f**) or closed (**c**).
- >>file** Log commands and screen output in *file*.
- >>{t|f|c}** Set logging file status, true (**t**), false (**f**) or closed (**c**).

- >>** Report status of command and screen output logging mechanism.
- >&file** Log host-to-target communication in *file*.
- >&{t|f|c}** Turn target communication logging on (**t**), off (**f**) or close (**c**) log file.
- >&** Report status of target communication logging mechanism.
- >*file** Log GDI accesses in *file*.
- >{*t|f|c}** Set GDI accesses log file status, true (**t**), false (**f**) or closed (**c**)

13.2.6 MACROS

- echo string** Display macro expansion of *string*.
- save file** Save current macros to *file*.
- set** Display all macros.
- set macro "cmds"**
Define macro *macro* as command list *cmds*.
- unset** Delete all macros.
- unset macro!**
Delete definition of macro *macro*.
- macro!** Prevent expansion of *macro*.

13.2.7 INPUT/OUTPUT SIMULATION

- ios_open** ["file"[,[mode][,[r],,\$xvw_variable]]]
Open a CrossView Pro File I/O stream.
- ios_wopen** ["terminal_window"[,\$xvw_variable]]
Open a CrossView Pro File I/O stream and map the stream to a terminal window.
- ios_close** {stream | "file"}
Close a CrossView Pro File I/O stream.

- ios_read** *{stream | "file"},address,number_of_maus[,x]*
Read binary data from a File I/O stream. Optionally, interpret the read data as hexadecimal values.
- ios_readf** *{stream | "file"},"format",expression*
Formatted read from a File I/O stream (scanf).
- ios_write** *{stream | "file"},address,number_of_maus[,x]*
Write binary data to a File I/O stream. Optionally, interpret the data as hexadecimal values.
- ios_writf** *{stream | "file"},"format",expression*
Formatted write to a File I/O stream (printf).
- ios_rewind** *{stream | "file"}*
Move File I/O file pointer to the beginning of the file.

13.2.8 FILE SYSTEM SIMULATION

- FSS** { < | > } *&stream | "file"*
Redirect to or from a stream or file.
- FSS_stdio_open** *filename,rwdirection,streamnumber*
Redirect the output of a stream to a file.
- FSS_stdio_close** *streamnumber*
Close the specified stream.

13.2.9 TARGET SYSTEM CONTROL

- dcmp** *[file],[number_of_bits],[d]*
Compare an application *file* with the memory contents and display differences.
- dn**
Download the image part of the current absolute file, specified when CrossView Pro was invoked or loaded with the **N** command.
- dn file**
Download the image part of the absolute file *file*.
- load** *[file]*
Load symbol table of *file* in CrossView Pro and download the image part to the target. This is a combination of **N** and **dn**.

- N** [*file*] Load symbol table of *file* in CrossView Pro.
- n** [*addr*] Set code address bias (for overlays) to *addr*. If no address is given, then display the current bias.
- o** [*cmd*] Enter transparency mode (exit with *ctrl-D*). If *cmd* is present, pass *cmd* to the execution environment.
- !** [*command-line*] Execute shell command *command-line* or invoke new shell.

13.2.10 SAVE AND RESTORE TARGET STATE

This feature is only available when it is supported by the debug instrument.

di_state open *state_name*

Open the state with the specified *state_name*.

di_state save *state_name, number*

Save the state of the debug instrument with the specified *state_name* and *number*.

di_state restore *state_name, number*

Restore the state of the debug instrument with the specified *state_name* and *number*.

di_state close *state_name, delete*

Close the state with the specified *state_name*. *delete* can be 1 to delete the state or use 0 to keep the state.

13.2.11 HELP COMMANDS

- I** Print information about debugger state.

13.2.12 SEARCH COMMANDS

- Z** Toggle case sensitivity in searches.
- /**[*string*] Search forwards in source file for *string*. If *string* is not present, perform previous search again.

- `?[string]` Search backwards in source file for *string*. If *string* is not present, perform previous search again.
- `"string"` Print *string*.

13.3 COMMANDS: DETAILED DESCRIPTIONS

The rest of this chapter provides the detailed descriptions of the CrossView Pro commands.

expression

Function

Print the value or address of an expression.



From the **Data** menu, select **Evaluate Expression...** Enter an expression and optionally select a display format. You may set up a monitor, which instructs the debugger to evaluate a particular expression each time the program stops, from the Source Window by selecting text there and by clicking on the **Watch Expression** button.



Enter the expression in the Command Window. You may specify in which format you want CrossView Pro to display the answer.

Description

In the Command Window, the syntax for this command is:

exp [/ *format* | @ *format*]

Print the value or address of *exp* with format *format*. A / (slash) is used to print the value of *exp* and a @ (commercial at) is used to print the address of *exp*. If *format* is not supplied, the natural (/n) format of the expression is used.

Formats have the syntax:

[*count*] *style* [*size*]

count is the number of times to apply the format *style* and defaults to 1. *style* may be one of:

a c D O U X d o u x E F G e f g i I n P p s t



See Chapter 6, *Accessing Code and Data*, and section 3.5 *Formatting Expressions* in Chapter 3, *Command Language*, for details on each of the format styles.

size indicates the number of bytes to be formatted. Rather than a number for the integer type styles, *size* can also be: **c** for char, **s** for short, **i** for int, and **l** for long.

The default action, if no modifier is specified, is to print the value of *exp* using the /n (normal) format.

Be careful with one letter variable names, as they may be taken as an CrossView Pro command rather than as a variable. If an expression begins with a variable that might be mistaken for a command, then eliminate any white space between the variable and the first operator. For example: use `h=9` instead of `h = 9`.

To display the value of a variable that has the same name as an CrossView Pro command you must use the natural format modifier. For example: to print the value of the variable `C`, use `C/n`.

Variables may be altered as a side effect of evaluation of *exp*. See the example below.

Example

To set variable `aux` to `t` times 8, type:

```
aux = t++*8
```

As a side effect the variable `t` is post-incremented. If you type:

```
$s_aux = func(t,s)
```

CrossView Pro will set special variable `$s_aux` to the result of the function call to `func` with the variables `t` and `s` passed as parameters. If you type:

```
$s_aux/x4
```

Print the value of the special variable `$s_aux` as four hex bytes; you could also use: `$s_aux/x1`.



line

Function

Display the C source line numbered *line* in the current source file.



From the **Edit** menu, select **Find Line...** Enter the line number and click on the **Find** button. Alternately, you may click on the desired source line in the Source Window.



Enter the line number in the Command Window. The syntax is:

line

Description

The current viewing position becomes *line*.

Example

To display the twelfth line in the current source file, type:

12



e, p, P

string

Function

Echo a string to the terminal.



Enter the string to the Command Window.

Description

A string may contain standard C escapes, such as `\n` for a newline. The syntax for a string in the Command Window is:

"string"

Example

This function can be useful for labelling breakpoints. For example, to insert a breakpoint at line 12 and have a message printed when that line is reached, enter:

```
12 b {"At the twelfth line\n"; C}
```

When CrossView Pro reached line 12, the message "At the twelfth line" will be printed and the program will continue. If you only type:

"Debug"

CrossView Pro will simply echo the word "Debug."



Q, *expression*



Function

Instruct CrossView Pro to interpret a command literally, ignoring any macro definitions of the same name. Also, enter a shell command.



The syntax for this command is:

`[string] !`

or:

`! [string]`

Description

This command is useful whenever *string* should be treated literally and not as a potential macro invocation. It can be used, for example, in executing an CrossView Pro command whose name has been defined as a macro.

Example

To enter the host environment under a new shell, type:

`!`

To execute the host `date` command, type:

`!date`

To execute the CrossView Pro command **b** instead of the macro named `b`, type:

`b!`



`set, unset, echo, save`



Function

Search down (forward) for a string.



To search for a string in the Source Window, select **Search String...** from the **Edit** menu and select the **up** radio button. To repeat your search click on the **Find Next Text String** button.



The command line syntax is:

`/ [string]`

Description

The search begins with the line after the current line. If the *string* is found the viewing position is changed to the line containing the string. The execution position is not affected. If you do not specify a string to search for, CrossView Pro will look for the most recent specified string.

Searches wrap around to the beginning of the file. Regular expressions are not recognized.

Example

To look for the next occurrence of `Random` in the current file, beginning with the line after the current line, type:

`/Random`



`?, Z`



Function

Search up (backward) for a string.



To search for a string in the Source Window, select **Search String...** from the **Edit** menu and select the **down** radio button. To repeat your search click on the **Find Next Text String** button.



The command line syntax is:

? [*string*]

Description

The search begins with the line before the current line. If *string* is found, the current line is changed to point to the line containing the string. The execution position is not affected. If you do not specify *string*, CrossView Pro searches for the previously-specified string again.

Searches wrap around to the end of the file. Regular expressions are not recognized.

Example

To look for the previous occurrence of Random in the current file, beginning with the line above the current line, type:

?Random



/, Z



Function

Continuous command playback. Read commands continuously from a file.



To setup command playback, select **Playback | CrossView...** from the **Tools** menu. Enable the **Continuous playback** check box and click on the **Execute** button.



The command line syntax is:

< file

Description

All the commands in *file* will be read and executed. If a playback file contains either a **<** or **<<** command, playback switches to the newly specified file and does not return to the original file.

Record and playback options can also be specified via command line parameters.

If the execution of commands from the playback file is interrupted with the **Halt** button, CrossView Pro will begin reading the remainder of commands in file using single step playback (see the **<<** command.)

Example

To read and execute the commands found in the file `command.cmd`, type:

`<command.cmd`



<<, >, I



Function

Single-step command playback.



To setup command playback, select **Playback | CrossView...** from the **Tools** menu. Disable the **Continuous playback** check box and click on the **Execute** button.



The command line syntax is:

<<file

Description

Commands will be played back one at a time. Each command will be loaded sequentially into the entry field of the Command Window. The command can then be edited and executed.

The carriage return will execute the current command and stop at the next one.

If a playback file contains either a **<** or **<<** command, playback switches to the newly specified file and does not return to the original file. Record and playback options can also be specified via command line parameters.

Example

To read and execute the commands found in the file `command.cmd`, type:

<< command.cmd



<, >, I



Function

Record CrossView Pro commands to a file.



To start recording or toggle the state of the command recording mechanism, select **Record | CrossView...** from the **Tools** menu. Type or select a file to record commands in and click on the **Start** button to start recording. To suspend recording click on the **Suspend** button. To resume recording click on the **Resume** button. To stop recording click on the **Stop** button.



The command line syntax is (note that the greater than sign must be typed as shown):

```
> [!] [file | t | f | c]
```

Description

CrossView Pro will start recording commands in a file if *file* is specified, otherwise, turn recording on (**t**), off (**f**), or close (**c**) the recording file. Specifying a different file while recording is on will cause the old output file to be closed and all successive commands will be sent to the new file. If no arguments are given, the state of the recording mechanism will be displayed.

The optional **!** forces flushing of the output after every write.

The commands recorded can be played back by using the **<** or **<<** command. It is possible to have a command recording file and a screen output recording file to be open concurrently. The file is also closed as a side effect of the **q** command.

Commands issued to the emulator under transparency mode are not recorded.

Files may not be named: **t**, **f** or **c**.

Example

To set (or change) the command recording file to `command.cmd` and turn command recording on, type:

```
>command.cmd
```

To suspend recording commands, type:

>f

To resume recording the commands to the recording file, type:

>t

To stop recording commands and close the file, type:

>c

To display the state of the recording mechanism, type:

>



>>, >&, <, <<, I, q



Function

Record CrossView Pro and emulator commands to a file.



To start recording or toggle the state of the command recording mechanism, select **Record | CrossView...** from the **Tools** menu. Type or select a file to record commands in, select **Include emulator commands** and click on the **Start** button to start recording. To suspend recording click on the **Suspend** button. To resume recording click on the **Resume** button. To stop recording click on the **Stop** button.



The command line syntax is (note that the greater than sign must be typed as shown):

```
>@ [!] [file | t | f | c]
```

Description

CrossView Pro will start recording commands in a file if *file* is specified, otherwise, turn recording on (**t**), off (**f**), or close (**c**) the recording file. Specifying a different file while recording is on will cause the old output file to be closed and all successive commands will be sent to the new file. If no arguments are given, the state of the recording mechanism will be displayed.

The optional **!** forces flushing of the output after every write.

The commands recorded can be played back by using the **<** or **<<** command. It is possible to have a command recording file and a screen output recording file to be open concurrently. The file is also closed as a side effect of the **q** command.

Commands issued to the emulator under transparency mode are also recorded, but each command is preceded by the **o** command.

Files may not be named: **t**, **f** or **c**.

Example

To set (or change) the command recording file to `command.cmd` and turn command recording on, type:

```
>@command.cmd
```

To suspend recording commands, type:

>@f

To resume recording the commands to the recording file, type:

>@t

To stop recording commands and close the file, type:

>@c



>, >#, >>, >&, <, <<, I, q



Function

Record emulator commands to a file.



To start recording or toggle the state of the command recording mechanism, select **Record | Emulator...** from the **Tools** menu. Type or select a file to record commands in and click on the **Start** button to start recording. To suspend recording click on the **Suspend** button. To resume recording click on the **Resume** button. To stop recording click on the **Stop** button.



The command line syntax is (note that the greater than sign must be typed as shown):

```
># [!] [file | t | f | c]
```

Description

CrossView Pro will start recording emulator commands in a file if *file* is specified, otherwise, turn recording on (**t**), off (**f**), or close (**c**) the recording file. Specifying a different file while recording is on will cause the old output file to be closed and all successive commands will be sent to the new file. If no arguments are given, the state of the recording mechanism will be displayed.

The optional **!** forces flushing of the output after every write.

The emulator commands recorded can only be played back by selecting **Playback | Emulator...** from the **Tools** menu. It is possible to have a command recording file and a screen output recording file to be open concurrently. The file is also closed as a side effect of the **q** command.

Files may not be named: **t**, **f** or **c**.

Example

To set (or change) the emulator command recording file to `emu.cmd` and turn command recording on, type:

```
>#emu.cmd
```

To suspend recording emulator commands, type:

>#f

To resume recording the emulator commands to the recording file, type:

>#t

To stop recording emulator commands and close the file, type:

>#c



>, >>, >&, <, <<, I, q



Function

Log Command Window screen output. All Command Window input and output will be saved to a file.



To create a log of Command Window screen output, select **Log | Command Input/Output...** from the **Tools** menu. Type or select a file to log to and click on the **Start** button to start logging. To suspend logging click on the **Suspend** button. To resume logging click on the **Resume** button. To turn off logging click on the **Stop** button.



The command line syntax is:

```
>> [!] [file | t | f | c]
```

Description

Start logging the commands typed and their output in a file if *file* is specified, otherwise, turn logging on (**t**), off (**f**), or close (**c**) the log file. Specifying a different file while logging is on will cause the old output file to be closed and all successive Command window output will be sent to the new file. If no arguments are given, the state of the recording and logging mechanism is displayed.

The optional **!** forces flushing of the output after every write.

Because output is logged as well as commands, files logged using **>>** cannot be played back like those recorded with the **>** command.

It is possible to have both a command recording file and a screen output logging file open concurrently. The log file is also closed as a side effect of the **q** command. Log files may not be named: **t**, **f** or **c**.

Example

To set (or change) screen output recording file to the file `screen.log` and turn screen output recording on, type:

```
>>screen.log
```

To suspend recording the screen output, type:

```
>>f
```


To resume recording the screen output in the recording file, type:

```
>>t
```

To stop recording the screen output and close the file, type:

```
>>c
```

To display the state of the recording mechanism, type:

```
>>
```



```
>, >&, l, q
```



Function

Log communications between debugger and emulator.



To save debugger/emulator communications, select **Log | CrossView-Emulator I/O...** from the **Tools** menu. Type or select a file to log to and click on the **Start** button to start logging. To suspend logging click on the **Suspend** button. To resume logging click on the **Resume** button. To turn off logging click on the **Stop** button.



The command line syntax is:

```
>& [!] [file | t | f | c]
```

Description

Start host-to-execution environment communication logging in a file if *file* is specified; otherwise, turn logging on (**t**), off (**f**), or close (**c**) the log file. This feature is most often used to diagnose problems with CrossView Pro itself.

The optional '**!**' forces flushing of the output after every write.

The commands captured cannot be played back the way commands recorded by the **>** command can. The log file is also closed as a side effect of the **q** command.

Example

To open the file `out.log` and put the following host-to-emulator communications in this file, type:

```
>&out.log
```

To suspend logging communications in the log file, type:

```
>&f
```

To resume logging communications in the log file, type:

```
>&t
```

To stop logging communications and close the file, type:

```
>&c
```



```
>, >>, q
```



Function

Log GDI accesses.



To save GDI accesses, select **Log | CrossView-GDI Accesses...** from the **Tools** menu. Type or select a file to log to and click on the **Start** button to start logging. To suspend logging click on the **Suspend** button. To resume logging click on the **Resume** button. To turn off logging click on the **Stop** button.



The command line syntax is:

```
>*[!][file | t | f | c]
```

Description

Start GDI accesses logging in a file if *file* is specified; otherwise, turn logging on (**t**), off (**f**), or close (**c**) the log file. This feature is most often used to diagnose problems with the Debug Instrument.

The optional **!** forces flushing of the output after every write.

The commands captured cannot be played back the way commands recorded by the **>** command can. The log file is also closed as a side effect of the **q** command.

Example

To open the file `gdi.log` and start logging GDI accesses in this file, type:

```
>*gdi.log
```

To stop logging GDI accesses and close the file, type:

```
>*c
```



```
>, >>, q
```



Function

Display contents of preceding memory location based on the size of the last data item displayed.



The command line syntax is:

^ [*format*]

Description

Use previous format or *format*, if supplied. Formats have the syntax:

[*count*] *style* [*size*]

count is the number of times to apply the format *style* and defaults to 1. *style* may be one of:

a c D O U X d o u x E F G e f g i I n P p s t



See Chapter 6, *Accessing Code and Data*, and section 3.5 *Formatting Expressions* in Chapter 3, *Command Language*, for details on each of the format styles.

size indicates the number of bytes to be formatted. Rather than a number for the integer type styles, *size* can also be: **c** for char, **s** for short, **i** for int, and **l** for long.

This command is most often used in combination with *exp/format* to look at the value of some variable or memory location.

Example

To display the variable *aux* as two octal values of length two, type:

^ aux/2o2

To show the eight bytes before *aux* in hexadecimal format, next type:

^2x4



expression

A

Function

Toggle the state of the assertion mode.



To activate or suspend assertion mode, select **Assertions...** from the Breakpoints menu, and enable or disable the **Assertion Mode Active** check box.



The command line syntax is:

A [**a** | **s**]

Description

Activate (**A a**) or suspend (**A s**) overall state of the assertion mechanism. If no operand is given, toggle the state.

Example

To activate the assertion mechanism, type:

A a

To suspend the assertion mechanism, type:

A s

To toggle the state of the assertion mechanism, simply type:

A



a

a

Function

Define or modify an assertion.



From the **Breakpoints** menu, select **Assertions...** to open the Assertions dialog box. Click the **New...** button to define an assertion. Select an assertion and click the **Edit...** button to modify an assertion.



The command line syntax is:

```
exp a { a | d | s }
a cmds
```

Description

The **a** command is used to invoke two different commands. The syntax for each command is distinct. The first version allows modification of the state of the assertion specified by the expression *exp*. (The assertion can be activated (**a a**), deleted (**a d**) or suspended (**a s**.) The second version creates a new assertion with the given command list *cmds*. Using the mouse, you can create a new assertion or toggle the state of an existing one from the Assertions dialogue box.

Suspended assertions continue to exist, but are not active. Deleted assertions must be explicitly redefined in order to be made active again.

The commands for every active assertion are executed after every source statement is executed. The **x** command in an assertion command list forces an exit from assertion mode.

This command is not allowed when the target runs in the background.

Example

To suspend assertion 3, type:


```
3 a s
```

To delete assertion 1, type:

```
1 a d
```

To set an assertion to stop the program when global variable `myvar` exceeds 3, type:

```
a if (myvar > 3) {x}
```

 **A, l, x**

B

Function

List all of the currently defined breakpoints.



From the **Breakpoints** menu, select **Breakpoints...** to open the Breakpoints dialog box.



The command line syntax is:

B

Description

Breakpoints are listed with numbers associated with them. These numbers can be used to delete individual breakpoints.



break, b, bb, bB, bi, bI, bu, bU, R, C, D, l

b

Function

Set a code breakpoint.



From the **Breakpoints** menu, select **Breakpoints...** to open the Breakpoints dialog box. Click the **Add >** button and select **Code Breakpoint...** to open the Add Code Breakpoint dialog. Enter the name of the source module or click the **Break At...** button to select a source module and enter a line number.

Alternatively, you can set a code breakpoint directly in the source by clicking on a green breakpoint toggle next to the source line.



The command line syntax is:

[line] **b** *[commands]*

Description

You can attach a list of CrossView Pro commands with the breakpoint. If no line is given, set the breakpoint at the current viewing position.

When the breakpoint is hit execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next, any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **b** command.

Example

To set a breakpoint at the current line, type:

b

To set a breakpoint at line 10 that will list all global variables and halt execution, type:

10 b {l g}



break, bd, bD, bdis, bena, bb, bB, bi, bI, bt, bti, btiI, bu, bU, Q

bB

Function

Set a temporary breakpoint at the beginning of a function.



In the Stack Window, click on the desired function and select **Stack Breakpoint | At Function Entry** from the **Breakpoints** menu.



The command line syntax is:

```
[ stack ] bB [ cmds ]
```

Description

The function is designated by the stack level *stack*. If no function is specified, CrossView Pro uses the current function (stack level 0), and associates the list of CrossView Pro commands *cmds* with the breakpoint.

Breakpoints set in the Stack Window are always temporary, meaning they will be deleted after the first time you reach them. A breakpoint set in this manner will not be visible in the Source Window.

When the breakpoint is hit, execution is halted; the breakpoint is then removed. By default the current execution position, function, line number, and source statement are displayed. Next, any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bB** command.

This command is not allowed when the target runs in the background.

Example

To set a temporary breakpoint at the beginning of the current function which prints a stack trace, type:

```
bB {T}
```

To set a temporary breakpoint at the beginning of the function whose stack number is 2, type:

```
2 bB
```



break, b, bb, bd, bD, bi, bI, bt, bti, bti, bu, bU, Q

bb

Function

Set a permanent breakpoint at the beginning of a function.



In the Stack Window, click on the desired function and select **Stack Breakpoint | At Function Entry** from the **Breakpoints** menu. To make the stack breakpoint permanent, select **Breakpoints...** from the **Breakpoints** menu, select the desired breakpoint and click on the **Edit...** button. The Edit Code Breakpoint dialog appears. Click on the **Advanced>>** button and disable the **Remove when hit** check box.



The command line syntax is:

```
[ stack ] bb [ cmds ]
```

Description

Set a breakpoint at the beginning of the function designated by the stack level *stack*. Otherwise, use the current function (stack level 0), and associate the list of CrossView Pro commands *cmds* with the breakpoint.

When the breakpoint is hit, execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next, any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bb** command.

This command is not allowed when the target runs in the background.

Example

To set a breakpoint at the beginning of the current function, which prints a stack trace, type:

```
bb {T}
```

To set a breakpoint at the beginning of a function whose stack number is 2, type:

```
2 bb
```



break, b, bB, bd, bD, bi, bI, bt, bti, btiI, bu, bU, Q

bc

Function

Set a breakpoint's count and reset count.



From the **Breakpoints** menu, select **Breakpoints...**, select the breakpoint for which you want to set the count and reset count and click on the **Edit...** button. The Edit Code Breakpoint dialog appears. Click on the **Advanced** button and enter a breakpoint count.



The command line syntax is:

```
[ number ] bc [ count ] [ reset_count ]
```

Description

Set the *count* and *reset_count* for the breakpoint with breakpoint number *number*. When no arguments are given, the breakpoint at the current viewing position is set to a count of 1 and a reset count of 1. If no breakpoint is present at the current viewing position, the message "No such breakpoint" appears.

Each time a breakpoint is hit, CrossView Pro decrements the *count*. When the *count* reaches 0, execution is halted and the *count* is reset to the *reset_count*.

This command is not allowed when the target runs in the background.

Example

To set a breakpoint's count and reset count to 1 for the breakpoint at the current viewing position, type:

```
bc
```

To set the count to 3 and the reset count to 4 for the breakpoint whose breakpoint number is 2, type:

```
2 bc 3 4
```



break, C

bCYC

Function

Set a temporary cycle count breakpoint.



From the **Breakpoints** menu, select **Breakpoints...** to open the Breakpoints dialog box. Click the **Add >** button and select **Cycle Breakpoint...** to open the Add Cycle Breakpoint dialog. Click the **Advanced** button and enable the **Remove when hit** check box.



The command line syntax is:

```
count bCYC [cmds]
```

Description

Set a temporary breakpoint after the specified cycle *count*. *count* can be any expression evaluating to a number. The list of CrossView Pro commands *cmds* are executed when the breakpoint is hit.

When the breakpoint is hit, execution is halted; the breakpoint is then removed. By default the current execution position, function, line number, and source statement are displayed. Next any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bCYC** command.

Example

To set a temporary breakpoint after 4 clock cycles and list all global variables, type:

```
4 bCYC {1 g}
```



break, b, bcyc, bINST, binst, bTIM, btim, D

bcyc

Function

Set a permanent cycle count breakpoint.



From the **Breakpoints** menu, select **Breakpoints...** to open the Breakpoints dialog box. Click the **Add >** button and select **Cycle Breakpoint...** to open the Add Cycle Breakpoint dialog. Enter a cycle count and click the **OK** button.



The command line syntax is:

```
count bcyc [cmds]
```

Description

Set a permanent breakpoint after the specified cycle *count*. *count* can be any expression evaluating to a number. The list of CrossView Pro commands *cmds* are executed when the breakpoint is hit.

When the breakpoint is hit, execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bcyc** command.

Example

To set a cycle count breakpoint after 4 clock cycles and list all global variables, type:

```
4 bcyc {1 g}
```



break, **b**, **bCYC**, **bINST**, **binst**, **bTIM**, **btim**, **D**

bD

Function

Set a read and/or write data breakpoint over a range of addresses.



From the **Breakpoints** menu, select **Breakpoints...** to open the Breakpoints dialog box. Click the **Add >** button and select **Data Breakpoint...** to open the Add Data Breakpoint dialog. Enter an address or click the **Address...** button to select a symbol to use as the address. Click the **Advanced** button. Enter an address in the **End address** field or click the **Browse...** button to select a symbol to use as the end address. Click the **OK** button to add the data breakpoint.



The command line syntax is:

```
exp1 bD { r | w | b } exp2 [cmds]
```

Description

Set a read, write, or both (read and write) data breakpoint in the address range *exp1* to *exp2* and associate the list of CrossView Pro commands *cmds* with the breakpoint.

When the breakpoint is hit, execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next, any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bD** command.

If *exp1* is the address of a local (stack) variable, the function in which it was declared must be currently active on the stack. If the local variable corresponding to a data breakpoint goes out of scope due to a return from the function in which it is currently active, the data breakpoint will be removed and a message will be printed telling the user that the variable is no longer active.

Example

To set a data breakpoint that includes the entire structure `rec1`, type:

```
&rec1 bD r (int)&rec1+sizeof(rec1)-1
```

This breakpoint will be hit only if any address in the range of addresses is read from.

To set a data breakpoint for the address range 10 to 10f hex (256 bytes) that will list all global variables, type:

```
0x10 bD b 0x10f {l g;}
```

This breakpoint will be hit if any memory locations within the range 10–10f hex are either read from or written to.



break, b, bb, bB, bd, bi, bI, bt, bti, bti, bu, bU, Q

bd

Function

Set a read and/or write data breakpoint at an address.



From the **Breakpoints** menu, select **Breakpoints...** to open the Breakpoints dialog box. Click the **Add >** button and select **Data Breakpoint...** to open the Add Data Breakpoint dialog. Enter an address or click the **Address...** button to select a symbol to use as the address. Click the **OK** button to add the data breakpoint.



The command line syntax is:

```
exp bd { r | w | b } [cmds]
```

Description

Set a read, write or both (read and write) data breakpoint at the address specified by *exp* and associate the list of CrossView Pro commands *cmds* with the breakpoint.

When the breakpoint is hit, execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bd** command.

If *exp* corresponds to a local (stack) variable, the function in which it was declared must be currently active on the stack. If the local variable corresponding to a data breakpoint goes out of scope due to a return from the function in which it is currently active, the data breakpoint will be removed and a message will be printed telling you that the variable is no longer active.

Example

To set a breakpoint at the variable `count` which will all be hit only if the variable is read from memory, type:

```
&count bd r
```

To set a breakpoint at address 10 hex that will list all global variables, type:

```
0x10 bd b {l g}
```

This breakpoint will be hit if address 10 hex is either read from or written to.



break, b, bb, bB, bD, bi, bI, bt, bti, bti, bu, bU, Q

bdis

Function

Disable code breakpoint.



From the **Breakpoints** menu, select **Breakpoints...** On Windows toggle the check box in front of the breakpoint to enable or disable the breakpoint. On UNIX select the breakpoint and click the **Enable** or **Disable** button.



The command line syntax is:

number **bdis**

Description

Disable the code breakpoint associated with the given *number*.

This does not delete the code breakpoint. It disables the code breakpoint until you enable it again with the **bena** command.

This command does not work on data breakpoints, only on code breakpoints

Example

To disable code breakpoint number 3, type:

3 bdis



break, b, bena, D

bena

Function

Enable code breakpoint.



From the **Breakpoints** menu, select **Breakpoints...** On Windows toggle the check box in front of the breakpoint to enable or disable the breakpoint. On UNIX select the breakpoint and click the **Enable** or **Disable** button.



The command line syntax is:

number **bena**

Description

Enable the code breakpoint associated with the given *number*, which was previously disabled by the **bdis** command.

This command does not work on data breakpoints, only on code breakpoints

Example

To enable code breakpoint number 3, type:

3 bena



break, b, bdis, D

bl

Function

Set a temporary low-level breakpoint at a machine instruction.



From the **Breakpoints** menu, select **Breakpoints...** to open the Breakpoints dialog box. Click the **Add >** button and select **Code Breakpoint...** to open the Add Code Breakpoint dialog. Edit the **Break At...** field. In the Advanced dialog enable the **Remove when hit** check box.



The command line syntax is:

```
[addr] bl [cmds]
```

Description

Set a temporary breakpoint at the machine instruction at address *addr*, or the current viewing position's address if *addr* is not specified; the list of CrossView Pro commands *cmds* are executed when the breakpoint is hit.

Make sure that *addr* is the start address of a machine instruction, otherwise the results are unpredictable. When the breakpoint is hit execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bl** command.

Example

To set a temporary breakpoint at the current viewing position's address, type:

```
bl
```

To set a temporary breakpoint at address 100 that will print the addresses of the next five source statements, type:

```
100 bl {P 5}
```



break, b, bb, bB, bd, bD, bi, bt, bti, btl, bu, bU, Q

bi

Function

Set a permanent low-level breakpoint at a machine instruction.



From the **Breakpoints** menu, select **Breakpoints...** to open the Breakpoints dialog box. Click the **Add >** button and select **Code Breakpoint...** to open the Add Code Breakpoint dialog. Edit the **Break At...** field. In the Advanced dialog disable the **Remove when hit** check box.

Alternatively, you can place a breakpoint in the intermixed window or assembly window by double clicking on the desired instruction.



The command line syntax is:

```
[addr] bi [cmds]
```

Description

Set a permanent breakpoint at the machine instruction at address *addr*, or the current viewing position's address if *addr* is not specified; the list of CrossView Pro commands *cmds* are executed when the breakpoint is hit.

Make sure that *addr* is the start address of a machine instruction, otherwise the results are unpredictable. When the breakpoint is hit execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bi** command.

Example

To set a breakpoint at the current viewing position's address, type:

```
bi
```

To set a breakpoint at address 100 that will print the addresses of the next five source statements, type:

```
100 bi {P 5}
```



break, b, bb, bB, bd, bD, bI, bt, bti, bti, bu, bU, Q

bINST

Function

Set a temporary instruction count breakpoint.



From the **Breakpoints** menu, select **Breakpoints...** to open the Breakpoints dialog box. Click the **Add >** button and select **Instruction Breakpoint...** to open the Add Instruction Breakpoint dialog. Type a value in the **Instruction count** field and enable the **Remove when hit** check box in the Advanced dialog.



The command line syntax is:

```
count bINST [cmds]
```

Description

Set a temporary breakpoint after the specified *count* number of machine instructions have been executed. *count* can be any expression evaluating to a number. The list of CrossView Pro commands *cmds* are executed when the breakpoint is hit.

When the breakpoint is hit, execution is halted; the breakpoint is then removed. By default the current execution position, function, line number, and source statement are displayed. Next any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bINST** command.

Example

To set a temporary breakpoint after execution of 5 instructions and list all global variables, type:

```
5 bINST {1 g}
```



break, **b**, **bCYC**, **bcyc**, **binst**, **bTIM**, **btim**, **D**

binst

Function

Set a permanent instruction count breakpoint.



From the **Breakpoints** menu, select **Breakpoints...** to open the Breakpoints dialog box. Click the **Add >** button and select **Instruction Breakpoint...** to open the Add Instruction Breakpoint dialog. Type a value in the **Instruction count** field and disable the **Remove when hit** check box in the Advanced dialog.



The command line syntax is:

```
count binst [cmds]
```

Description

Set a permanent breakpoint after the specified *count* number of machine instructions have been executed. *count* can be any expression evaluating to a number. The list of CrossView Pro commands *cmds* are executed when the breakpoint is hit.

When the breakpoint is hit, execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **binst** command.

Example

To set a permanent breakpoint after execution of 5 instructions and list all global variables, type:

```
5 binst {1 g}
```



break, **b**, **bCYC**, **bcyc**, **binST**, **bTIM**, **btim**, **D**

break

Function

Universal breakpoint command.



From the **Breakpoints** menu, select **Breakpoints...** to add/remove/enable/disable breakpoints.



The general command line syntax is:

break [*type*] *where* [, *option*]...

Description

This is a universal breakpoint command.

type can be one of: **code** | **data** | **instructions** | **cycles** | **time** | **sequence** | **set** | **delete** | **enable** | **disable**. The type can be abbreviated. So, **t** | **ti** | **tim** | **time** are the same. When the type field is not specified the type defaults to **code**.

Depending on the type field the *where* field will evaluate to an address, count, name, breakpoint number or a sequence.

The available options are listed below.

Code breakpoints

Syntax:

break code *address* [, *option*]...

address can be any expression evaluating to an address.

Data breakpoints

Syntax:

break data *address* [, *option*]...

address can be any expression evaluating to an address.

Instruction count breakpoints

Syntax:

break instructions *count* [, *option*]...

count can be any expression evaluating to the number of instructions.

Cycle count breakpoints

Syntax:

break cycles *count* [, *option*]...

count can be any expression evaluating to the number of cycles.

Timer breakpoints

Syntax:

break timer *time* [, *option*]...

time can be any expression evaluating to a time value. Depending on the setting of the **timer_unit** option this value is in seconds or timer ticks (default is in seconds).

Sequence breakpoints

Syntax:

break sequence *sequence* [, *option*]...

sequence is a combination of breakpoints.

Set/change breakpoint attributes

Syntax:

break set *bp_number* | *bp_name* [, *option*]...

bp_number is the breakpoint number. If the breakpoint has a name (*bp_name*) you can use this name instead of a number.

Delete breakpoint attributes

Syntax:

break delete *bp_number* | *bp_name* | **all** [, *option*]...

bp_number is the breakpoint number. If the breakpoint has a name (*bp_name*) you can use this name instead of a number.

Enable/disable breakpoints

Syntax:

break enable *bp_number* | *bp_name*

break disable *bp_number* | *bp_name*

bp_number is the breakpoint number. If the breakpoint has a name (*bp_name*) you can use this name instead of a number.

Options

name=*str*

Change/set the name of a breakpoint. Note that when a name of a breakpoint which name is used in a sequence is changed the name in the sequence is not automatically changed.

temporary[=*bool*]

Single shot breakpoint, temporary breakpoints are deleted after they are hit.

enabled[=*bool*]

Enable or disable a breakpoint.

curr_count=*expr*

Set current count.

reset_count=*expr*

Set reset count.

count=*expr*

Set current and reset count of a breakpoint.

access_type=r | w | rw

Set the access type of a data breakpoint: read (**r**), write (**w**) or read/write (**rw**).

addr=expr

Set the (start)address for a code or data breakpoint.

value=expr

set the value for a data breakpoint.

method=hardware | software | none

Set the breakpoint method.

probe_point[=bool]

Treat the breakpoint as a probe point. When a probe point breakpoint is hit the associated commands are executed and program execution is continued. Probe points do not update CrossView Pro windows.

size=expr

Length of a data or code breakpoint (end_addr = begin_addr+size-1).

end_addr=expr

The end address of a range is inclusive.

end_value=expr

The end value is inclusive.

value_is_absolute[=bool]

For instructions and cycles breakpoints only, the specified value is an absolute count, breakpoint will hit when count has value, otherwise repeat every number of instructions.

commands={ commands }

Set breakpoint commands.

timer_unit=seconds | ticks

The specified timer value is in seconds or ticks.

bool

1 | 0 | true | false

True/false, case insensitive.

expr

Appropriate CrossView expression.

Example

To set a code breakpoint at an address range, type:

```
break code code:0x10, end_addr=code:0x1f
```

To set a code breakpoint at an address range by specifying a size, type:

```
break code:0x10, size=0x10
```

To set a code breakpoint with a name, type:

```
break code:0x10, name=brk_1
```

To disable the breakpoint with name `brk_1`, type:

```
break dis brk_1
```

To set a cycle count breakpoint and treat the value as an absolute count, type:

```
break cycles 1000, value_is_absolute
```



Chapter 7, *Breakpoints*.

bt

Function

Set a task aware code breakpoint.



From the **Breakpoints** menu, select **Breakpoints...** to open the Breakpoints dialog box. Click the **Add >** button and select **Code Breakpoint...** to open the Code Breakpoint dialog. Fill in the **Task ID** field.



The command line syntax is:

```
[line] bt "TaskId" [cmds]
```

Description

Set a task aware code breakpoint at the specified source *line* and associate the list of CrossView Pro commands *cmds* with the breakpoint. If no line is given, set the breakpoint at the current viewing position. The *TaskId* is the identification of the task as displayed in the Tasks Window or specified by the **lk** command.

When the breakpoint is hit, execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next, any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bt** command.

Example

To set a breakpoint for task 4 at the current viewing position, type:

```
bt "4"
```

To set a breakpoint for task 4 at line 10, which lists all global variables, type:

```
10 bt "4" {l g}
```



break, **b**, **bb**, **bB**, **bd**, **bD**, **bi**, **bI**, **bti**, **bti**, **bu**, **bU**, **l**, **Q**

btl

Function

Set a temporary low-level task aware breakpoint at a machine instruction.



From the **Breakpoints** menu, select **Breakpoints...** to open the Breakpoints dialog box. Click the **Add >** button and select **Code Breakpoint...** to open the Code Breakpoint dialog. Edit the **Break At...** field and fill in the **Task ID** field. In the Advanced dialog enable the **Remove when hit** check box.



The command line syntax is:

```
[addr] btl "TaskId" [cmds]
```

Description

Set a temporary task aware breakpoint at the machine instruction at address *addr*, or the current viewing position's address if *addr* is not specified; the list of CrossView Pro commands *cmds* are executed when the breakpoint is hit. The *TaskId* is the identification of the task as displayed in the Tasks Window or specified by the **lk** command.

Make sure that *addr* is the start address of a machine instruction, otherwise the results are unpredictable. When the breakpoint is hit execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **btl** command.

Example

To set a temporary breakpoint for task 4 at the current viewing position's address, type:

```
btl "4"
```

To set a temporary breakpoint for task 4 at address 0xF00 and print the message, type:

```
0xF00 btl "4" {"breakpoint triggered:
               address 0xF00, task 4"}
```



break, b, bb, bB, bd, bD, bi, bI, bt, bti, bu, bU, l, Q

bti

Function

Set a permanent low-level task aware breakpoint at a machine instruction.



From the **Breakpoints** menu, select **Breakpoints...** to open the Breakpoints dialog box. Click the **Add >** button and select **Code Breakpoint...** to open the Code Breakpoint dialog. Edit the **Break At...** field and fill in the **Task ID** field. In the Advanced dialog disable the **Remove when hit** check box.



The command line syntax is:

```
[addr] bti "TaskId" [cmds]
```

Description

Set a permanent task aware breakpoint at the machine instruction at address *addr*, or the current viewing position's address if *addr* is not specified; the list of CrossView Pro commands *cmds* are executed when the breakpoint is hit. The *TaskId* is the identification of the task as displayed in the Tasks Window or specified by the **Ik** command.

Make sure that *addr* is the start address of a machine instruction, otherwise the results are unpredictable. When the breakpoint is hit execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bti** command.

Example

To set a breakpoint for task 4 at the current viewing position's address, type:

```
bti "4"
```

To set a breakpoint for task 4 at address 0xF00 and print the message, type:

```
0xF00 bti "4" {"breakpoint triggered:
               address 0xF00, task 4"}
```



break, b, bb, bB, bd, bD, bi, bI, bt, btl, bu, bU, l, Q

bTIM

Function

Set a temporary time breakpoint.



From the **Breakpoints** menu, select **Breakpoints...** to open the Breakpoints dialog box. Click the **Add >** button and select **Timer Breakpoint...** to open the Add Timer Breakpoint dialog. Enter a value in the **Time** field and enable the **Remove when hit** check box in the Advanced dialog.



The command line syntax is:

```
time bTIM [cmds]
```

Description

Set a temporary breakpoint after the specified *time* (in seconds). *time* can be any expression evaluating to a number. The list of CrossView Pro commands *cmds* are executed when the breakpoint is hit.

When the breakpoint is hit, execution is halted; the breakpoint is then removed. By default the current execution position, function, line number, and source statement are displayed. Next any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bTIM** command.

Example

To set a temporary breakpoint after 0.5 seconds and list all global variables, type:

```
0.5 bTIM {l g}
```



break, **b**, **bCYC**, **bcyc**, **bINST**, **binst**, **btim**, **D**

btim

Function

Set a permanent time breakpoint.



From the **Breakpoints** menu, select **Breakpoints...** to open the Breakpoints dialog box. Click the **Add >** button and select **Timer Breakpoint...** to open the Add Timer Breakpoint dialog. Enter a value in the **Time** field and disable the **Remove when hit** check box in the Advanced dialog.



The command line syntax is:

```
time btim [cmds]
```

Description

Set a permanent breakpoint after the specified *time* (in seconds). *time* can be any expression evaluating to a number. The list of CrossView Pro commands *cmds* are executed when the breakpoint is hit.

When the breakpoint is hit, execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **btim** command.

Example

To set a permanent breakpoint after 0.5 seconds and list all global variables, type:

```
0.5 btim {l g}
```



break, **b**, **bCYC**, **bcyc**, **bINST**, **binst**, **btim**, **D**

bU

Function

Set a temporary up-level breakpoint (to finish the function at a specific stack level).



In the Stack Window, double-click on the desired function. Alternately, you can click on the desired function in the Stack Window and select **Stack Breakpoint | After Call to Function** from the **Breakpoints** menu.



The command line syntax is:

```
[ stack ] bU [ commands ]
```

Description

This command sets a temporary up-level breakpoint immediately after the call to the function designated by the stack number *stack*, otherwise the currently viewed function is used. Associate the list of CrossView Pro commands *commands* with the breakpoint.

When the breakpoint is hit execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bU** command.

Breakpoints set in the Stack Window are always temporary, meaning they will be deleted after the first time you reach them. A breakpoint set in this manner will not be visible in the Source Window.

This command is not allowed when the target runs in the background.

Example

To set a temporary up-level breakpoint immediately after the call to the currently viewed function, type:

```
bU
```

To set a temporary up-level breakpoint immediately after the call to the function at stack level 2, type:

```
2 bU {1}
```

After stopping, this command will cause CrossView Pro to print out the function's local variables and arguments.



break, b, bb, bB, bd, bD, bi, bI, bt, bti, bti, bu, Q

bu

Function

Set a permanent up-level breakpoint (to finish the function at a specific stack level).



Click on the desired function in the Stack Window and select **Stack Breakpoint | After Call to Function** from the **Breakpoints** menu. To make the stack breakpoint permanent, select **Breakpoints...** from the **Breakpoints** menu, select the desired stack breakpoint and click on the **Edit...** button. The Edit Code Breakpoint dialog appears. Click on the **Advanced>>** button and disable the **Remove when hit** check box.



The command line syntax is:

```
[ stack ] bu [ commands ]
```

Description

Set a permanent up-level breakpoint immediately after the call to the function designated by the stack number *stack*, otherwise the currently viewed function is used. Associate the list of CrossView Pro commands *commands* with the breakpoint.

When the breakpoint is hit execution is halted. By default the current execution position, function, line number, and source statement are displayed. Next any commands associated with the breakpoint are executed. The **Q** command can be used to suppress the output from the **bu** command.

This command is not allowed when the target runs in the background.

Example

To set a temporary up-level breakpoint immediately after the call to the currently viewed function, type:

```
bu
```

To set an up-level breakpoint immediately after the call to the function at stack level 2 and, after stopping, print out the local variables and arguments of that function, type:

```
2 bu {1}
```



break, b, bb, bB, bd, bD, bi, bI, bt, bti, bti, bU, Q

bufa

Function

Append the contents of one buffer to another buffer.



The command line syntax is:

```
bufa target_buffer_name,added_buffer_name[,size_limit]
```

Description

Add the contents of buffer *added_buffer_name* to buffer *target_buffer_name*. If *size_limit* is specified, buffer *target_buffer_name* will be trimmed down to the specified size (keeping *size_limit* elements of the tail of the buffer).

Example

To append the contents of `$buffer` to buffer `$all_data`, and keep the last 1024 elements, type:

```
bufa $all_data,$buffer,1024
```



bufd, **graph**, **memget**.

Section 11.5, *Data Analysis*, in Chapter *Special Features*.

bufd

Function

Free a used buffer.



The command line syntax is:

```
bufd buffer_name
```

Description

Discard the specified buffer (if the target data is not to be used anymore).

Example

To discard buffer `$buffer`, type:

```
bufd $buffer
```



bufa, **graph**, **memget**.

Section 11.5, *Data Analysis*, in Chapter *Special Features*.

C

Function

Continue using the current value of the program counter.



In the Source Window, click on the **Run/Continue** button. You can also select **Run** from the **Run** menu.



The command line syntax is:

```
[ exp ] C [ line ]
```

Description

If *exp* is specified and you are stopped at a breakpoint, then the breakpoint count is set to this value. If *line* is specified, a temporary breakpoint is set at that line number. Note that this temporary breakpoint will overwrite any existing breakpoint at that line.

The **C** command can be used in the command lists of breakpoints to resume execution automatically.

This command is not allowed when the target runs in the background.

Example

To continue execution from the current target program counter, type:

```
C
```

To set the breakpoint's count to 4 and continue, type:

```
4 C
```

To set a temporary breakpoint at line 52 and continue, type:

```
C 52
```



break, bc, g, R, CB

CB

Function

Continue execution in background using the current value of the target program counter.



The command line syntax is:

[*exp*] **CB** [*line*]

Description

If *exp* is specified and you are stopped at a breakpoint, then the breakpoint count is set to this value. If *line* is specified, a temporary breakpoint is set at that line number. Note that this temporary breakpoint will overwrite any existing breakpoint at that line.

The **CB** command can be used in the command lists of breakpoints to resume execution automatically.

This command is not allowed when the target runs in the background.

Example

To continue execution from the current target program counter, type:

CB

To set the breakpoint's count to 4 and continue, type:

4 CB

To set a temporary breakpoint at line 52 and continue, type:

CB 52



g, R, C, st, wt

cd

Function

Disable, turn off, gathering of coverage data.



From the **Tools** menu, select **Coverage** if this item was set.



The command line syntax is:

cd

Description

If coverage is supported by your version of CrossView Pro, this command disables the coverage system. Normally, you should disable coverage if you are not interested in the coverage results, as this will often improve the performance of the execution environment.

Example

To disable coverage, type:

cd



ce, nC, nU, pC, pU

ce

Function

Enable, turn on, gathering of coverage data.



From the **Tools** menu, select **Coverage** if this item was not set.



The command line syntax is:

```
ce
```

Description

If coverage is supported by your version of CrossView Pro, this command enables the coverage system. Normally, you should disable coverage if you are not interested in the coverage results, as this will often improve the performance of the execution environment.

Example

To enable coverage, type:

```
ce
```



cd, nC, nU, pC, pU

clear

Function

Clear the Command Output Window.



The command line syntax is:

clear

Description

Use this command if you want to clear the output window part of the Command Window.

Example

To clear the Command Output Window, type:

clear

covinfo

Function

List coverage information.



From the **Tools** menu, select **Code Coverage...**, make your changes and select the **Update** button.



The command line syntax is:

```
covinfo [[all | module_or_function_name][,filename]]
```

Description

If coverage is supported by your version of CrossView Pro and coverage is enabled, this command lists the coverage information. Without arguments (same as **all**) this command lists the coverage information of all modules and functions.

Instead of listing the results you can also save the results in a file with extension `.cov`.

Normally, you should disable profiling if you are not interested in the profiling results, as this will often improve the performance of the execution environment.

Example

To list the coverage information of all modules and functions to the output window, type:

```
ce  
covinfo
```

To list coverage information of function `main` to the output window, type:

```
covinfo main
```

To list coverage information of all modules and functions in file `hello.cov`, type:

```
covinfo all,hello.cov
```



cd, ce, proinfo

cproinfo

Function

List cumulative profiling results or add or remove functions from the list of profiled functions.



From the **Tools** menu, select **Cumulative Profiling Setup...**, make your changes and click the **OK** button. Select **Cumulative Profiling Report...** to see the cumulative profiling report.



The command line syntax is:

```
cproinfo [all[filename] | {add | remove} function]
```

Description

If profiling is supported by your version of CrossView Pro and profiling is enabled, this command lists the cumulative profiling results. Without arguments (same as **all**) this command lists the cumulative profiling information of all functions.

Instead of listing the results you can also save the results in a file with extension `.cpr`.

Normally, you should disable profiling if you are not interested in the profiling results, as this will often improve the performance of the execution environment.

Example

To list the cumulative profiling results of all functions to the output window, type:

```
pe  
cproinfo
```

To dump cumulative profile information of all functions in file `hello.cpr`, type:

```
cproinfo all,hello.cpr
```

To add function `main` to the list of profiled functions, type:

```
cproinfo add main
```


To remove function `main` from the list of profiled functions, type:

```
cproinfo remove main
```



```
proinfo, pd, pe
```

ct

Function

Display a C-execution trace.



From the **View** menu, select **Trace | Source Level**. The Trace Window displays the most recently executed lines of code every time program execution is stopped. CrossView Pro automatically updates the Trace Window each time execution is halted, as long as the window is open.



The command line syntax is:

number **ct**

Description

Display a C-execution trace in the Command window, corresponding to the last *number* of machine instructions executed. Since the **ct** command relies on the emulator's trace buffer, the **ct** command will not be implemented on some emulators.

For each executed line of code, the Trace Window displays:

- The name of the source file
- The name of the function
- The line number and corresponding source code

The window shows all the code executed since the the last time the program halted.

This command is not allowed when the target runs in the background.

Example

To display, in the Command window, the last C statements (corresponding to the last ten machine instructions) executed, type:

10 **ct**



ct i, ct r

ct i

Function

Display a disassembled trace.



From the **View** menu, select **Trace | Instruction Level**. The Trace Window displays the most recently executed lines of code every time program execution is stopped. CrossView Pro automatically updates the Trace Window each time execution is halted, as long as the window is open.



The command line syntax is:

number **ct i**

Description

Display a disassembled trace in the Command window, corresponding to the last *number* of machine instructions executed.

Since the **ct i** command relies on the emulator's trace buffer, the **ct i** command will not be implemented on some emulators (or implemented differently).

This command is not allowed when the target runs in the background.

Example

To display in the Command window the last 20 disassembled instructions executed, type:

20 ct i



ct, ct r

ct r

Function

Display a raw trace.



From the **View** menu, select **Trace | Raw**. The Trace Window displays the most recently executed lines of code every time program execution is stopped. CrossView Pro automatically updates the Trace Window each time execution is halted, as long as the window is open.



The command line syntax is:

number **ct r**

Description

Display a raw trace in the Command window, corresponding to the last *number* of trace frames. This command merely shows the contents of the emulator's trace buffer.

Since the **ct r** command relies on the emulator's trace buffer, the **ct r** command will not be implemented on some emulators.

This command is not allowed when the target runs in the background.

Example

To display in the Command window the last 20 trace frames, type:

20 ct r



ct, ct i

D

Function

Delete all currently defined breakpoints.



From the **Breakpoints** menu, select **Breakpoints...** to open the Breakpoints dialog box. Click on the **Remove All** button.



The command line syntax is:

D[y]

Description

D deletes all currently defined breakpoints. **Dy** does not ask for confirmation.



break, B, d

d

Function

Delete a specific breakpoint.



To delete a code breakpoint directly from the C source, click on the red breakpoint toggle next to the corresponding, source line in the Source Window.

Otherwise, select **Breakpoints...** from the **Breakpoints** menu to open the Breakpoints dialog box. Select the breakpoint you want to remove and click on the **Remove** button.



The command line syntax is:

[*number*] **d**

Description

Delete the breakpoint associated with the given *number*. If no number is given, delete the breakpoint at the current line. If there is no breakpoint at the current line, a **B** command will be executed to display all breakpoints.

Whenever a breakpoint is deleted the remaining breakpoints are renumbered starting at 0.

Example

To delete a breakpoint at the current line, type:

d

To delete breakpoint number 3, type:

3 d



break, b, bb, bB, bd, bD, bi, bI, bt, bti, bti, bu, bU, B, D

dcmp

Function

Compare a file with the downloaded application.



From the **File** menu, select **Compare Application...** Specify an application file and click on the **Compare** button.



The command line syntax is:

```
dcmp [file],[number_of_bits],[d]
```

Description

Compare an application file with the memory contents and display differing memory addresses or addresses and values. If you have already loaded an application you can invoke this command without specifying a file name. You can limit the number of differences by specifying a *number_of_bits*. The value **0** means there is no limit on the number of differences.

This command is not allowed when the target runs in the background.

Example

To compare the currently loaded application, there is no limit on the number of differences and the contents of differing memory addresses are not displayed, type:

```
dcmp
```

To compare the currently loaded application and stop when the number of differences equals 10, type:

```
dcmp ,10
```

To compare the currently loaded application there is no limit on the number of differences and display the contents of differing memory addresses, type:

```
dcmp ,,d
```

To compare file `test.abs`, stop if the number of differences equals 5 and display the contents of differing memory addresses, type:

```
dcmp "test.abs",5,d
```



dn

di_state

Function

Open, save/restore, close a debug instrument state.



From the **Target** menu, select **Save/Restore Target State...**



The command line syntax is:

di_state open *state_name*

di_state save *state_name, number*

di_state restore *state_name, number*

di_state close *state_name, delete*

Description

Before a state can be saved, restored or closed it must be opened first. To open a state use the **di_state open** *state_name* command. When opened successfully the name is added to the available state names list.

With the **di_state save** command you can now save the state of the debug instrument with the specified *state_name* and *number*. With **di_state restore** you can restore a previously saved state of the debug instrument with the specified *state_name* and *number*.

Use **di_state close** to close a state. The *delete* flag can be 1 to delete the state or use 0 to keep the state.

Example

To open and save a state, type:

```
di_state open S1
di_state save S1, 0
```

To restore a state, type:

```
di_state restore S1, 0
```

dis

Function

Disassemble a range of memory.



From the **View** menu, select **Source | Disassembly** or **Source | Source and Disassembly** to open the Disassembly or Source and Disassembly window respectively.



The command line syntax is:

```
dis address [, {address | #count} [,i]]
```

Description

Disassemble a range of memory. The output is interleaved with source lines when **i** is specified. You can enter valid expressions as well for *address* and *count*.

Example

To disassemble 4 instructions starting at 3 bytes behind the start address of the function `main`., type:

```
dis main+3,#4
```

To disassemble memory for (`initval+1`) instructions, starting at the address of the function `main`., type:

```
dis main+3,#initval+1
```

To disassemble from `0x2000` up to and including the instruction at `0x2100` and also interleave C source lines of any function resident in that memory range, type:

```
dis 0x2000,0x2100,i
```



dump, *expression*

dn

Function

Download a file.



From the **File** menu, select **Download Application...** to download the image part of the file to the execution environment.



The command line syntax is:

```
dn [file]
```

Description

Download the image part of the specified *file* to the execution environment. If no *file* is specified, use the file specified when CrossView Pro was invoked, and from which the symbolic information was read during startup, or the file specified in either the **N** command or the Load Symbolic Debug Info dialog.

Downloading a file only copies an image part into target memory. It will not cause CrossView Pro to re-read symbolic information.

This command is not allowed when the target runs in the background.

Example

To download the current file, type:

```
dn
```

To download the IEEE file `demo.abs`, type:

```
dn demo.abs
```

To download the hex file `test.hex`, type:

```
dn test.hex
```



I, N

dump

Function

Dump a range of memory.



From the **View** menu, select **Memory | New** to open a Memory Window.



The command line syntax is:

```
dump address [, [address | #count] [, [style [width ] ] [, filename [a]]]
```

Description

The **dump** command can dump memory as hexadecimal data or as C variables. You can enter valid C expressions as well for *address* and *count*. You can also dump Motorola S records or Intel hex records. Also, you can specify a *filename* in which the dump is to be written or appended.

style can be one of:

```
a c D O U X d o u x E F G e f g n P p R r s t I M
```

Style **I** dumps Intel hex and style **M** specifies Motorola S records output. See Chapter 6, *Accessing Code and Data*, and section 3.5, *Formatting Expressions*, in Chapter 3, *Command Language*, for details on each of the other format styles. The **R** and **r** style are only available for targets that support the fractional type.



Mind the following:

- the commas are required
- the addresses can also be C expressions
- default width is MAU (usually byte) sized words
- additional style **M**: Motorola S records
- additional style **I**: Intel hex
- a semicolon is a command terminator
- the **dump** is end address Inclusive

Example

To dump the first byte of the function `main.`, type:

```
dump main
```

To dump the first 10 bytes of the function `main` as Motorola S records in the file `main.sre`, type:

```
dump main,main+10,M,main.sre
```

To dump the first 5 bytes of the function `main`. as 1 string, type:

```
dump main,main+10,M,main.sre,a
```

To append the first 5 bytes of the function `main`. as 1 string, type:

```
dump main,,c5
```

To dump the resulting value bytes of 'the address of `main` binary anded with 3', type:

```
dump main+1,#main&3
```



dis, *expression*

e

Function

Establish viewing position



From the **File** menu, select **Open Source...** to view a file. In the Source Window, click on the **Find Symbol** button to find a function, or select **Find Symbol...** from the **Edit** menu.

In the Stack Window click once on the function to be examined.



The command line syntax is:

```
e [file | function ]  
stack e
```

Description

The **e** option invokes two distinct commands. The first version establishes the viewing position to be the first line of *file*, the first executable line of the function *function* or the current viewing position if no argument is given.

The second version establishes the viewing position to be the line at stack level *stack* in the stack trace. (See the **t** command.)

The *stack* **e** command is not allowed when the target runs in the background.

The **L** command is equivalent to **0 e**.

Example

To view the function main, type:

```
e main
```

To view the test file test.c, type:

```
e test.c
```

To view the call site of the current function, type:

```
0 e
```

To view the line at stack level 3, type:

3 e



?, /, ei, L, p, P, t

eC

Function

Start execution on current CPU and switch to another CPU.



The command line syntax is:

cpu_number **eC**

Description

Start execution on the current CPU and switch to CPU *cpu_number*.

This command can only be issued when the currently selected CPU is in debug mode.

Example

To start execution on the current CPU and select the CPU indicated by number 1, type:

1 eC



ec

ec

Function

Select a CPU or show current CPU number.



The command line syntax is:

```
[cpu_number] ec
```

Description

The **ec** command allows you to select a CPU in your current Execution Environment if your target has multi-CPU support.

This command can only be issued when the currently selected CPU is in debug mode.

Example

To view the current CPU selection, type:

```
ec
```

To select the CPU indicated by number 1, type:

```
1 ec
```



```
ec
```

echo

Function

Display the definition of a macro name without executing the macro.



From the **Tools** menu, select **Macro Definitions...** to view the definition of a macro.



The command line syntax is:

echo *text*

Description

Perform macro expansion on *text* without executing. This allows you to see how a macro is expanded. It is particularly informative when macros call other macros.

Example

If you type:

echo macro(3)

CrossView Pro will display the expansion of `macro(3)`.



set, unset, save, !

ei

Function

Establish viewing position at a specified address.



From the **Edit** menu, select **Find Address...**



The command line syntax is:

```
[addr] ei
```

Description

The **ei** command establishes the viewing position to be at the instruction specified.

This command is useful for viewing some code in the assembly window, without changing the program counter, since the execution position is not changed.

Example

To view the current viewing position, type:

```
ei
```

To view the instruction at address 0x100, type:

```
0x100 ei
```



?, /, e, L, p, P, t

et

Function

Select the specified task's context.



In the Tasks Window click once on the task to be examined.



The command line syntax is:

```
et "TaskId"
```

Description

Select the specified task's context. The *TaskId* is the identification of the task as displayed in the Tasks Window or specified by the **lk** command.

The current execution position, function, line number, and source statement are displayed. All other windows, except for the Kernel Windows, are updated accordingly.

Subsequent CrossView Pro commands use the context of the selected task. For example, the **t** command shows a stack trace of the selected task.

Example

To select task 4, type:

```
et "4"
```



1

f

Function

Set default address printing format



The command line syntax is:

```
f [ " printf-style-format " ]
```

Description

Set the default address printing format, using a `printf` format specification.

If there is no argument, the format defaults to `%x`, which prints an address in hexadecimal.

This command is intended to allow users to see memory addresses in decimal, octal or a format of their choosing.

Example

To display addresses in octal, type:

```
f "%o"
```

To display addresses in hex, type:

```
f
```



expression

FSS

Function

File System Simulation redirection.



The command line syntax is:

```
FSS { < | > } {&stream | "file"}
```

Description

Redirect a File System Simulation stream to a file or another stream. Redirection to a file can be needed when a stream is only mapped to a window and you want it to be mapped to a file also.

Example

To redirect the output of stream 2 to stream 1, type:

```
FSS 2>&1
```

To retrieve input for stream 1 from stream 4, type:

```
FSS 1<&4
```

To retrieve input for stream 4 from file "data.txt", type:

```
FSS 4<"data.txt"
```

To redirect the output of stream 3 to file "data.txt", type:

```
FSS 3>"data.txt"
```



Section 10.3, *File System Simulation* in Chapter *I/O Simulation*.

FSS_stdio_close

Function

Close a stream previously opened by **FSS_stdio_open**.



The command line syntax is:

```
FSS_stdio_close streamnumber
```

Description

Close the stream indicated by *streamnumber*.

Example

To close stream 1, type:

```
FSS_stdio_close 1
```



FSS_stdio_open.

Section 10.3, *File System Simulation* in Chapter *I/O Simulation*.

FSS_stdio_open

Function

Redirect the output of a stream to a file.



The command line syntax is:

```
FSS_stdio_open filename,rwdirection,streamnumber
```

Description

Redirect the stream indicated by *streamnumber* to the file *filename*. *rwdirection* can be an **r** for read-only, **w** for writable, or **rw** for read/write.

Example

To redirect stream 1 (output, so **w** for writable) to the file `myfile.out`, type:

```
FSS_stdio_open myfile.out,w,1
```

The following command is used to close the stream.

```
FSS_stdio_close 1
```



FSS_stdio_close.

Section 10.3, *File System Simulation* in Chapter *I/O Simulation*.

g

Function

Change the program counter to a new execution position.



Click on a source line and select **Jump to Cursor** from the **Run** menu.



The command line syntax is:

g *line*

Description

This command changes the program counter so that *line* becomes the current execution position. *Line* must be a line in the current function.

This command changes only the program counter. It does not cause the target to begin execution.

Exercise caution when changing the execution position. Oftentimes, each line of C source code is compiled into several machine language instructions. Moving the program counter to a new address in the middle of a series of related assembly instructions is sometimes risky. Moreover, even though you change the program counter, registers and variables may not have the expected values if parts of the code are bypassed.

This command is not allowed when the target runs in the background.

Example

To change the program counter so that the next instruction to be executed corresponds to line 127, type:

g 127



C, gi, R

gi

Function

Change the program counter to a new execution position.



Click on a source line and select **Jump to Cursor** from the **Run** menu.



The command line syntax is:

address **gi**

Description

This command changes the program counter so that *address* becomes the current execution position.

This command changes only the program counter. It does not cause the target to begin execution.

Exercise caution when changing the execution position. The **Jump to Cursor** menu item is not available in the source lines window mode to prevent problems by skipping pieces of C code which are required to be executed. Moving the program counter to a new address in the middle of a series of related assembly instructions is sometimes risky. Moreover, even though you change the program counter, registers and variables may not have the expected values if parts of the code are bypassed.

This command is not allowed when the target runs in the background.

Example

To change the program counter so that the next instruction to be executed corresponds to address 0x0800, type:

0x0800 gi



C, g, R

graph

Function

Create Data Analysis window and execute CXL script.



The command line syntax is:

graph "window","script"[arg]...

Description

Create Data Analysis window *window* and execute CXL script *script*. The display list produced by the script is shown in the specified window. Arguments *arg* are passed as global variables to the script. Each argument is treated as an expression. Arguments starting with a "\$" refer to an acquisition buffer. In all other cases *arg* is evaluated as an expression and will be casted to type double.



If for example register \$R1 should be passed as argument to the script you must write "0+\$R1" to avoid that \$R1 is recognized as an acquisition buffer.

Example

To transform the contents of buffer \$buffer to displayable data in window demo using CXL script x_t.cxl, type:

graph "demo","x_t.cxl",\$buffer,0,1



bufa, **graphm**, **graphp**, **memget**.

Section 11.5, *Data Analysis*, in Chapter *Special Features*.

graph_add_update

Function

Add a command to the sequence of update commands.



For the supplied scripts only. From the **Settings** menu, select **Data Analysis Window Setup...** Enter a new window name and click **New**. Click **Configure...** to open the Data Analysis Window Setup dialog.



The command line syntax is:

```
graph_add_update "window",command
```

Description

Set the sequence of update commands for Data Analysis window *window* manually. These update commands are executed when the Update button on the Data Analysis window is pressed or when the **update** command is issued.



Prior to adding update commands, you have to remove all update commands with the **graph_clear_updates** command.

Example

To retrieve data and show it in window demo, type:

```
graph_clear_updates "demo"
graph_add_update "demo",memget data[$i],100,$buffer
graph_add_update "demo",graphm "demo","show_x_t.cxl"
graph_add_update "demo",graph "demo","x_t.cxl",$buffer,0,1
update "demo"
```



graph_clear_updates, **update**.

Section 11.5, *Data Analysis*, in Chapter *Special Features*.

graph_clear_updates

Function

Clear the sequence of update commands.



The command line syntax is:

```
graph_clear_updates "window"
```

Description

Clear the sequence of update commands for Data Analysis window *window*. This is needed prior to adding new update commands with the **graph_add_update** command.

Example

To retrieve data and show it in window demo, type:

```
graph_clear_updates "demo"  
graph_add_update "demo",memget data[$i],100,$buffer  
graph_add_update "demo",graphm "demo","show_x_t.cxl"  
graph_add_update "demo",graph "demo","x_t.cxl",$buffer,0,1  
update "demo"
```



graph_add_update, **update**.

Section 11.5, *Data Analysis*, in Chapter *Special Features*.

graph_close

Function

Close a Data Analysis window.



The command line syntax is:

```
graph_close "window"
```

Description

With the **graph_close** command you can close the named *window*.

Example

To close window demo, type:

```
graph_close "demo"
```



graph, **graphm**.

Section 11.5, *Data Analysis*, in Chapter *Special Features*.

graph_debug

Function

Debug Data Analysis graph window.



The command line syntax is:

graph_debug *expression*

Description

If *expression* evaluates to a non-zero value, this value is an ORed value of two flags:

- 1 (bit 0) the "graphical data window debugging mode" will be enabled, showing all communication between the scripts and the windows in the command window. This can be useful when developing scripts.
- 2 (bit 1) When errors occur during script processing, these errors are logged to the command window. The total error count (per script) is now shown in a popup window rather than logged in the command window. The errors themselves remain logged in the command window.

Other bits (when value & 3 equals zero, for example 4) are ignored and treated like zero. No parameters result in value 1. A value of zero turns off all debugging.



graph, **graphm**, **graphmn**.

Section 11.5, *Data Analysis*, in Chapter *Special Features*.

graphm

Function

Set Data Analysis window display mode.



The command line syntax is:

```
graphm "window","script"[,arg]...
```

Description

The **graphm** command sets the representation *script* for the specified *window*. Depending on the script, the arguments may vary.

Several scripts are supplied with the product that you can use with the **graphm** command. See section *Supplied Data Analysis Window Scripts* in Chapter *Special Features* for more information.

Example

To set the display mode for window demo using CXL script `show_x_t.cxl` and show "demo" in the title bar of the window, type:

```
graphm "demo","show_x_t.cxl"
```



bufa, **graph**, **graphp**, **memget**.

Section 11.5, *Data Analysis*, in Chapter *Special Features*.

graphmn

Function

Set Data Analysis window display mode.



The command line syntax is:

```
graphmn "window","script"[,arg]...
```

Description

The **graphmn** command works similar to the **graphm** command, but it does not update the graph window. This can be useful where a **graph** and a **graphm** command are followed by each other, preventing the redrawing of the same graphics twice.

Example

To set the display mode for window demo using CXL script `show_x_t.cxl` and show "demo" in the title bar of the window, type:

```
graphmn "demo","show_x_t.cxl"
```



bufa, graph, graphp, memget.

Section 11.5, *Data Analysis*, in Chapter *Special Features*.

graphp

Function

Position Data Analysis window on the screen.



The command line syntax is:

```
graphp "window",left_top_x,left_top_y,width,height
```

Description

With the **graphp** command you can position the named *window* at the specified screen coordinates.

Example

To put window demo at position (0,0) on the screen with a size of 100x100, type:

```
graphp "demo",0,0,100,100
```



graph, **graphm**.

Section 11.5, *Data Analysis*, in Chapter *Special Features*.

gus

Function

Suppress or reactivate window updating.



The command line syntax is:

```
gus {on | off}
```

Description

With **gus on** the GUI updating suppress feature is enabled. This means that the graphical windows are no longer updated. To reactivate the window updating use the **gus off** command.

Example

To suppress the updating of CrossView windows, type:

```
gus on
```



Function

Print out information about the state of CrossView Pro.



The command line syntax is:

I

Description

Print out information about the state of CrossView Pro, including: the CrossView Pro version number, the execution environment version information, the name of the program being debugged (and the number of its files and functions), the state of the assertion mechanism, the state of output recording, the state of command recording, the state of target communication recording and the state of search case sensitivity.

The state of the assertion mechanism tells how many assertions have been defined and whether the overall assertion mechanism is active or suspended; it does not tell whether any individual assertions are active or suspended.



I, a, A, >, >>, >&, Z

if

Function

Conditional command execution.



The command line syntax is:

```
if ( expression ) { cmds } [ { cmds } ]
```

Description

If *expression* evaluates to a non-zero value, execute the first group of commands. Otherwise, the second group of commands, if present, will be executed. This command is nestable.

Leave a space between **if** and *exp*. `if(a==b)` parses as a function call. The **if** statement is used primarily within breakpoint command lists.

Example

If you type:

```
if (a=b) {5t} {C}
```

CrossView Pro will trace back five levels on the stack if *a* is equal to *b*. Otherwise, CrossView Pro will continue.

The command line:

```
if (wait>1000) {wait;l r}
```

will print the value of *wait* and list all registers if the value of *wait* exceeds 1000.

ios_close

Function

Close a File I/O stream.



From the **Settings** menu, select **I/O Simulation Setup...** Select a stream in the **Connections** tab and click on the **Delete** button.



The command line syntax is:

```
ios_close {stream | "file"}
```

Description

You can specify either a filename or a stream number.

Example

To close stream number 1, type:

```
ios_close 1
```

To close file `data.txt` and close 1 stream that is mapped to this file, type:

```
ios_close "data.txt"
```

Only 1 stream is closed, even if multiple streams are attached to this file.
The command displays which stream number has been closed.



ios_open, ios_wopen

ios_open

Function

Open a File I/O stream.



From the **Settings** menu, select **I/O Simulation Setup...** Open the **File I/O** tab and click on the **Configure...** button. Attach a stream (with a file) to a probe point.



The command line syntax is:

```
ios_open ["file"[,mode][,r],$xvw_variable]]]
```

Description

This command is useful to connect a file to a stream at the command line of CrossView. CrossView returns a stream number which is opened with this command in the *\$xvw_variable* and displays it too.

The filename is optional. When the filename is omitted and such a newly opened stream receives data and is not shown in any opened terminal window a new window will be opened that interacts with this stream.

Furthermore the *mode* can be specified when a I/O stream is opened: read, write or append:

- r** Open file for reading. The file pointer is positioned at the beginning of the file.
- r+** Open file for reading and writing. The file pointer is positioned at the beginning of the file.
- w** Truncate file to zero length or create file for writing. The file pointer is positioned at the beginning of the file.
- w+** Open file for reading and writing. The file is created if it does not exist, otherwise it is truncated. The file pointer is positioned at the beginning of the file.
- a** Open file for writing. The file is created if it does not exist. The file pointer is positioned at the end of the file.
- a+** Open file for reading and writing. The file is created if it does not exist. The file pointer is positioned at the end of the file.

All modes can have a **'b'** appended, indicating binary access. The **'b'** can be positioned before or after the **'+'**. This mode affects the **ios_read** and **ios_write** commands. The **ios_read** command writes host data to target memory. In binary mode MAUs (minimum addressable units) are filled with a number of bytes that fits in 1 MAU. For example, a MAU with a size of 24 bits will be filled with $24/8 = 3$ bytes. Otherwise the least significant 8 bits of a MAU will be filled with 1 byte and the highest 16 bits will be filled with zeros. The **ios_write** command writes target memory to the host. In binary mode for each MAU the number of bytes to be written equals the number of bytes that fits in 1 MAU. For a MAU size of 24 bits CrossView Pro will write 3 bytes to the host. If the mode is not binary CrossView Pro will write the least significant 8 bits (1 byte) of each MAU to the host.

CrossView Pro opens all files by default in **w+** mode, overwriting the opened file if it already exists.

The optional **'r'** specifies to rewind to the beginning of the file when the end of file is reached.

\$xvw_variable is a user special variable in CrossView Pro which holds the value of the newly opened stream number. This variable can also be used in the read and write commands to read from or write to the *file*.

Example

To open a new File I/O stream, type:

```
ios_open
```

To open file `data.txt` and assign the new stream number to `$ios_nr`, type:

```
ios_open "data.txt",,, $ios_nr
```

To open file `data.txt` in read-only mode and wrap around when end of file is reached, type:

```
ios_open "data.txt",r,r, $ios_nr
```



ios_wopen, ios_close, ios_read, ios_write

ios_read

Function

Read binary data from an I/O *stream*.



The command line syntax is:

```
ios_read {stream | "file"},address,number_of_maus[,x]
```

Description

You can specify a File I/O stream number or a filename. *address* is the memory location where the read data will be stored. *number_of_maus* is the length of the data to be read in MAUs (minimum addressable units).

The optional '**x**' specifies that the read data should be interpreted as hexadecimal values. The hexadecimal format is a whitespace separated (no TAB) hexadecimal string without the **0x** prefix.

If the stream was opened in binary mode (see **ios_open**), MAUs are filled with a number of bytes that fits in 1 MAU. For example, a MAU with a size of 24 bits will be filled with $24/8=3$ bytes. Otherwise the least significant 8 bits of a MAU will be filled with 1 byte and the highest 16 bits will be filled with zeros.

Example

To read 16 minimum addressable units from stream 4, type:

```
ios_read 4,0x100,16
```

To read from stream \$istrm 1 MAU hex value, type:

```
ios_read $istrm,0x100,1,x
```



ios_readf, ios_write, ios_open

ios_readf

Function

Formatted read from an I/O *stream* (scanf). Store the data at the location defined by the expression.



The command line syntax is:

```
ios_readf {stream | "file"}, "format", expression
```

Description

You can specify a File I/O stream number or a filename. *format* is a format specifier as used in the scanf C library function. *expression* can be any CrossView Pro expression.

Valid format specifiers are:

%d	Decimal.
%x	Hexadecimal (without 0x prefix).
%c	Char.
%s	String.
%f	Float.

Example

To read a hex value from stream 4 and store it the value of program variable ch1, type:

```
ios_readf 4, "%x", &ch1
```

To read a hex value from stream 4 and store it in register R2, type:

```
ios_readf 4, "%x", $R2
```

To read two hex values from stream \$i1strm and assign them to program variable ch1 and target register R2, type:

```
ios_readf $i1strm, "%x %x", &ch1, $R2
```



ios_read, ios_write, ios_open

ios_rewind

Function

Move File I/O file pointer to the beginning of the file.



From the **Settings** menu, select **I/O Simulation Setup...** Open the **File I/O** tab and click on the **Configure...** button. Attach a stream to a probe point. In the New Stream dialog enable the **Wrap around** check box.



The command line syntax is:

```
ios_rewind {stream | "file"}
```

Description

With **ios_rewind** the file pointer is moved to the beginning of the file.

Example

To move the file pointer of the file connected to stream 4 to the beginning of the file, type:

```
ios_rewind 4
```

To move the file pointer of the file connected to stream \$istrm to the beginning of the file, type:

```
ios_rewind $istrm
```

To move the file pointer to the beginning of file my.txt, which is connected to a stream, type:

```
ios_rewind "my.txt"
```



ios_read, ios_write, ios_open

ios_wopen

Function

Open a File I/O stream and map the stream to a terminal window.



From the **Settings** menu, select **I/O Simulation Setup...** Open the **File I/O** tab and click on the **Configure...** button. Attach a stream (which is only connected to a terminal window) to a probe point.



The command line syntax is:

```
ios_wopen ["terminal_window"],$xvw_variable]
```

Description

When the name matches the name of an existing terminal window the newly opened stream is mapped to this terminal window.

\$xvw_variable is a user special variable in CrossView Pro which holds the value of the newly opened stream number. This variable can also be used in the read and write commands to read from or write to the *terminal_window*.

You can close the opened stream with **ios_close**.

Example

To create a new terminal window and map the newly created stream to it. The name of the new terminal window will be like #x., type:

```
ios_wopen , $ios_nr
```

To open a new stream and if there is a terminal window with the name "My terminal" map stream to it, otherwise create a new terminal and name it "My terminal"., type:

```
ios_wopen "My terminal", $ios_nr
```



ios_open, ios_close

ios_write

Function

Write binary data to an I/O *stream*.



The command line syntax is:

```
ios_write {stream | "file"},address,number_of_maus[,x]
```

Description

You can specify a File I/O stream number or a filename. *address* is the memory location where the data will be read from. *number_of_maus* is the length of the data to be written in MAUs (minimum addressable units).

The optional '**x**' specifies that the data should be interpreted as hexadecimal values. The hexadecimal format is a whitespace separated (no TAB) hexadecimal string without the **0x** prefix.

If the stream was opened in binary mode (see **ios_open**), for each MAU the number of bytes to be written equals the number of bytes that fits in 1 MAU. For a MAU size of 24 bits CrossView Pro will write 3 bytes to the host. If the mode is not binary CrossView Pro will write the least significant 8 bits (1 byte) of each MAU to the host.

Example

To write 16 minimum addressable units to stream 4, type:

```
ios_write 4,0x100,16
```

To write 1 MAU hex value to stream \$ostrm, type:

```
ios_write $ostrm,0x100,1,x
```



ios_read, ios_writef, ios_open

ios_writef

Function

Formatted write to an I/O *stream* (printf).. The data is obtained from the C expression, for example a variable.



The command line syntax is:

```
ios_writef {stream | "file"}, "format", expression
```

Description

You can specify a File I/O stream number or a filename. *format* is a format specifier as used in the printf C library function. *expression* can be any CrossView Pro expression.

Valid format specifiers are:

%d	Decimal.
%x	Hexadecimal (without 0x prefix).
%c	Char.
%s	String.
%f	Float.

Example

To write the hex value of program variable ch1 to stream 4, type:

```
ios_writef 4, "%x", ch1
```

To write the hex value of register R2 to stream \$ostrm, type:

```
ios_writef $ostrm, "%x", $R2
```

To write the hex values of program variable ch1 and target register R2 to stream 4, type:

```
ios_writef 4, "%x %x", &ch1, $R2
```



ios_read, **ios_write**, **ios_open**

L

Function

Synchronize the viewing and execution positions.



To synchronize the positions manually, click on the **Find PC** button in the Source Window or select **Find PC** from the **Edit** menu.



The command line syntax is:

L

Description

This command synchronizes the viewing and execution positions. It also lists the current file, function and line number of the current program counter. The viewing position is always moved to match the execution position.

The **L** command is synonymous with a **0 e** command and does not affect the execution position.



This command is not allowed when the target runs in the background.

Example

To synchronize the viewing and execution positions, then list current file, function, and line number, type:

L



e, 1



Function

List.



In general, the dialog box in which you define a feature also contains a list.



The command line syntax is:

```
l{ a | b | d | f | g | k | l | L | m | p | r | s | S} [string]  
l [func]  
l stack
```

Description

In the first case above, list one of the following: **a**ssertions, **b**reakpoints, **d**irectories, **f**iles, **g**lobals, **k**ernel state data, **l**abels (on module scope), all **L**abels, **m**emory map (of application code sections), **p**rocedures, **r**egisters, **s**pecial variables, **S**ymbol tables. If *string* is present, then list only those items that start with *string*.

In the second case, list the values of all parameters and locals of the function *func*. Without a function, this command lists all parameters and locals of the current function in view.

In the third case, list all parameters and locals of the function at depth *stack*.

The **lf** and **lm** commands also show the address of the modules' first procedure. The **lm** command is identical to **lf**, list files, but the list of files is sorted on ascending segment addresses. *func* must be a function on the stack or the current function.

For configurations that support real-time kernels, the **lk** command can have one of the following arguments (**lk** is the same as specifying **lk t**):

- t** – Display tasks.
- m** – Display mailboxes.
- q** – Display queues.
- p** – Display pipes.
- s** – Display semaphores.
- e** – Display events.
- h** – Display HISRs (High-level Interrupt Service Routines)
- si** – Display signals.
- ti** – Display timers.
- pm** – Display partition memory.
- dm** – Display dynamic memory.
- r** – Display resources.
- misc** – Display miscellaneous information.

Example

To list defined assertions and the state of the assertion mechanism, type:

```
l a
```

To list all locals and parameters of the current function, type:

```
l p
```

Data is displayed using the normal (**/n**) format. To list all the parameters and locals of the function **fcn**, type:

```
l fcn
```

To list queue information for the current tasks (only if your configuration supports it), type:

```
l k q
```



L, et

load

Function

Load a program's symbol file and download the image part.



From the **File** menu, select **Load Symbolic Debug Info...** This dialog allows you to specify the file.



The command syntax is:

```
load [filename]
```

Description

This command performs the **N** and **dn** commands successively.

Downloading a file only copies the image part into target memory (**dn**). It will not cause CrossView Pro to re-read symbolic information (**N**). The **load** command does both.



This command is not allowed when the target runs in the background.

Example

To load the symbol table of file `demo.abs` in CrossView Pro and download the image part, type:

```
load demo.abs
```



dn, N

M

Function

List the data currently being monitored.



Refer to the Data Window. Each time the program stops, the debugger evaluates all monitored expressions and displays the results in the Data Window.



The command line syntax is:

M

Description

List all C expressions being monitored by CrossView Pro. The listing associates a unique number with each expression. This number is used to specify the deletion of monitored data.



m

m

Function

Monitor (watch) an expression. (Also delete a monitor.)



From the Source Window, double-click on an expression. A new monitor is created in the Data Window or the Expression Evaluation dialog is opened if the **Bypass Expression Evaluation Dialog** check box in the Data Display Setup dialog is not set. If the latter is the case, click on the **Add Watch** button to create a new monitor in the Data Window. To remove an existing monitor, select the monitor in the Data Window and click on the **Delete Selected Data Item** button.



The command syntax is:

```
m exp  
number m d
```

Description

The **m** command has two distinct functions. The first monitors the given expression. The second deletes the monitoring of the expression specified by *number*.

Data monitoring takes place whenever the program stops execution, that is, for a breakpoint, assertion, single step, or user interrupt (*ctrl-C*). In window mode, the values of all currently monitored data are displayed in the Data window. Each piece of monitored data has a unique identifying number that is used when deleting it.

Example

To monitor the value of the variable `myvar`, type:

```
m myvar
```

To monitor the address of variable `myvar`, type:

```
m &myvar
```

To monitor the element `alpha+1` of `array`, type:

```
m array[alpha+1]
```

To delete expression number 2 of the monitored data, type:

2 m d



M, b, a, s, R, C

mcp

Function

Memory copy.



From the Memory Window, click on the **Copy Memory** button to open the Copy Memory dialog. Enter the start address and the end address (inclusive) of the memory region you want to copy. Enter the destination address and click on the **OK** button.



The command syntax is:

addr_start **mcp** *addr_end*, *addr_dest*

Description

The **mcp** command copies a block of target memory starting at address *addr_start* to destination address *addr_dest*. The size of the memory block is defined as: '*addr_end* - *addr_start* + 1'. The data item located at address *addr_end* is included in the copy.

If your target supports multiple memory spaces then it is legal to copy data between different memory spaces. Of course *addr_start* and *addr_end* must be located in the same memory space. This command does not have any effect on code breakpoints.

Example

To copy the contents of variable `buf` to address `0x200`, type:

```
&buf mcp &buf+sizeof(buf), 0x200
```



mF, mf

memget

Function

Retrieve data from the target into a buffer.



The command line syntax is:

memget *expr,count,buffer_name*

Description

The **memget** command is used to retrieve data from the target system and to store the data in the acquisition buffer *buffer_name*. Data in the acquisition buffer is of type `double`. CrossView Pro will automatically handle data conversion based upon the type of expression *expr*.

Expression *expr* contains the iterator "*i*" which initially starts at 0 and increments to *count*-1.

Notation convention:

"*expr*<*i*{*n*}>" means "*expr*" in which all instances of "*i*" are substituted by "*n*".

To correctly retrieve the data from the target CrossView Pro needs to know the start address, the size of the data elements, and the number of items to fetch. The number of items to fetch from the target is specified by *count*. The following algorithm is used to fill the acquisition buffer:

```

addr0      = (char *) &expr<i{0}>
addr1      = (char *) &expr<i{1}>
delta      = addr1 - addr0
elem_size  = sizeof(expr<i{0}>)
type       = C-type(expr<i{0}>)
for (i = 0; i < count; i++)
{
    value = read elem_size MAUs from address addr0 + (i * delta)
    buffer[i] = convert_to_double(type, value);
}

```

Example

1. C structure access.

```
struct
{
    double re,
           im;
    int f;
} data[100];
```

To store the `data[x].re` values into acquisition buffer `$a`:

```
memget data[$i].re,100,$a
```

To store the `data[x].im` values into acquisition buffer `$b`:

```
memget data[$i].im,100,$b
```

2. Memory access.

To retrieve 18 integer values from memory starting at address `0x100` and store these in acquisition buffer `$buffer`:

```
memget ((int[]) 0x100)[$i],3*6,$buffer
```



bufa, bufd, graph, rawmemget.

Section 11.5, *Data Analysis*, in chapter *Special Features*.

mF

Function

Memory single fill.



From the Memory Window, click on the **Fill Single Memory Address** button to open the Single Fill Memory dialog. Enter the start address the memory region you want to fill. Enter one or more expressions separated by commas and click on the **OK** button.



The command syntax is:

addr **mF** *expr* [,*expr*]...

Description

The **mF** command fills target memory with data. The value defined by *exp* is written to address *addr* in target memory. Multiple *exprs* separated by commas may be entered. Each *exp* is written to a subsequent MAU.

If your target supports multiple memory spaces then *addr* may refer to any memory space.

If the size of a given *exp* occupies more than one MAU, only the least significant MAU will be written to memory. This command does not have any effect on code breakpoints.

Example

To store value 0x12 at memory location 0x400 and value 0xAB at location 0x401, type:

0x400 mF 0x12, 0xAB



mcp, mf

mf

Function

Memory fill, repeating the specified pattern until the specified region is filled.



From the Memory Window, click on the **Fill Memory** button to open the Memory Fill dialog. Enter the start address and end address (inclusive) of the memory region you want to fill. Enter one or more expressions separated by commas and click on the **OK** button.



The command syntax is:

```
addr_start mf addr_end, expr [expr]...
```

Description

The **mf** command fills a block of target memory with a pattern. The memory region starting at address *addr_start* and ending at address *addr_end* is filled with the pattern defined by *exp* [*exp*]. Multiple *exp*s separated by commas may be entered. Each *exp* is written to a subsequent MAU.

The specified pattern is repeated until the end address of memory region is reached.

If your target supports multiple memory spaces then *addr* may refer to any memory space.

If the sizeof a given *exp* occupies more than one MAU, only the least significant MAU will be written to memory. This command does not have any effect on code breakpoints.

Example

To store values 0x01 and 0x02 at succeeding memory locations in the range 0x400 to 0x404, type:

```
0x400 mf 0x404, 0x01, 0x02
```

The result of this command is:

```
address: 0x400 0x401 0x402 0x403 0x404
value:    1      2      1      2      1
```

 **mcp, mf**

ms

Function

Memory search.



From the Memory Window, click on the **Find Memory** button to open the Search Memory dialog. Enter the start address and end address (inclusive) of the memory region you want to search. Enter one or more search patterns separated by commas and click on the **OK** button.



The command syntax is:

```
addr_start ms addr_end, expr [,expr]...
```

Description

The **ms** command searches for a pattern within a block of target memory. The memory region starting at address *addr_start* and ending at address *addr_end* (inclusive) is searched for the pattern defined by *exp* [*,exp*]. Multiple *exps* separated by commas may be entered. Each *exp* corresponds to a subsequent MAU.

If your target supports multiple memory spaces then *addr* may refer to any memory space.

This command does not have any effect on code breakpoints.

Example

Suppose the memory range 0x400 to 0x4ff was filled using the following commands:

```
0x400 mf 0x4ff, 0
0x400 mf 0x404, 1, 2
```

To search for the values 0x01 and 0x02 at memory locations in the range 0x400 to 0x4ff, type:

```
0x400 ms 0x4ff, 0x01, 0x02
```

The result of this command is:

```
FOUND pattern at 0x400
FOUND pattern at 0x402
```



mcp, mF, mf

N

Function

Load a program's symbol file.



From the **File** menu, select **Load Symbolic Debug Info...** This menu item allows you to specify the file.



The command syntax is:

N *[[path]filename[.abs]]*

Description

Load the symbol table of the specified file in CrossView Pro. If no filename is given, the file being debugged is reloaded. In this case only the breakpoints set by the user are removed. Monitors, I/O simulation streams, assertions and CrossView Pro local variables remain active.

If a new file (different filename) is loaded, all breakpoints, monitors, I/O simulation streams, assertions and CrossView Pro local variables are removed.

If a path is supplied, CrossView Pro changes its current directory according to the specified path. In case a relative search path to source files was provided at startup time, CrossView Pro will search relative to the new working directory.

This command is automatically executed during CrossView Pro startup when a filename was given on the command line. Use the **dn** command to send the associated executable code to the target.

Example

To load the symbol table of file `demo.abs` in CrossView Pro, type:

N demo.abs



dn

n

Function

Set address bias



From the **File** menu, select **Load Symbolic Debug Info...** In the Load Symbolic Debug Info dialog you can edit the **Code address bias** field.



The command syntax is:

n [*addr*]

Description

Set address bias of overlay files to *addr*. If no address is given, then display current bias.

If a program is to be loaded at a different address than that indicated in the linked and located (absolute object) file, then the address information in the debugger's symbol file will be incomplete, since it does not know where the program is actually going to be loaded. This command will normalize the addresses by adding the bias to every address.

Example

To add a bias of 1000 to every address in the code, type:

n 1000

To display the current bias, type:

n

nC

Function

Set the viewing position to the next covered block of statements.



Use the scroll bar and click on the desired line.



The command line syntax is:

nC

Description

If code coverage is supported by your version of CrossView Pro, this command enables you to skip to the next block of statements that have been executed while the program was running on the target.

Example

To move the cursor to the next executed block, type:

nC



nU, pC, pU

nU

Function

Set the viewing position to the next not covered block of statements.



Use the scroll bar and click on the desired line.



The command line syntax is:

nU

Description

If code coverage is supported by your version of CrossView Pro, this command enables you to skip to the next block of statements that have not been executed while the program was running on the target.

Example

To move the cursor to the next not executed block, type:

nU



nC, pC, pU

O

Function

Enter emulator mode.



From the View menu, select **Command | Emulator**. If you know the emulator-level command language, you can communicate directly with the emulator from this window.



The command line syntax is:

o *string*

Description

Pass *string* to emulator and show the emulator response.

The **o** command lets you communicate with the emulator directly via emulator commands.

Do not issue one-shot transparency emulator commands that result in large output (or otherwise require intervention other than a carriage return to terminate output). Instead, enter transparency mode first, then issue the command.

Example

To send the string `map` to the emulator, type:

o `map`

opt

Function

Set or display specific options.



Option values can be changed in various dialogs and menus.



The command line syntax is:

```
opt [ option_name [= option_value]]
```

Description

If no arguments are passed, all options with their current value are listed. By specifying an option's name, the current value of that option is displayed. By specifying an option name followed by a valid value, the option is set to that new value.

The options are a sub-set of CrossView's so-called "special variables". See Chapter 3, *Command Language*, for a list of all special variables.

Example

To display all options, type:

```
opt
```

To disable mixing of disassembly code and source lines in the assembly window, type:

```
opt mixedasm=off
```



1

P

Function

Print source lines, including machine addresses.



In the Source Window, the machine address of the line at the current viewing position is displayed in the address field in the upper left corner.



The command line syntax is:

[*line*] **P** [*exp*]

Description

Print *exp* lines of source starting at line *line*, including machine addresses. If *exp* is omitted, print one line. If *line* is omitted, start from the current viewing position.

Example

To print source lines 4, 5, 6, 7 and 8 (displaying machine addresses) of the current source file, type:

4 P 5



P

p

Function

Print source lines.



C source is displayed in the Source Window.



The command line syntax is:

[*line*] **p** [*exp*]

Description

Print *exp* lines of source starting at line *line*. If *exp* is omitted, print one line. If *line* is omitted, start from the current viewing position.

Example

To print source lines 4, 5, 6, 7 and 8 of the current source file, type:

4 **p** 5



P

pC

Function

Set the viewing position to the previous covered block of statements.



Use the scroll bar and click on the desired line.



The command line syntax is:

pC

Description

If code coverage is supported by your version of CrossView Pro, this command enables you to skip to the previous block of statements that have been executed while the program was running on the target.

Example

To move the cursor to the previous executed block, type:

pC



nC, nU, pU

pd

Function

Disable, turn off, profiling.



From the **Tools** menu, select **Profiling** if this item was set.



The command line syntax is:

pd

Description

If profiling is supported by your version of CrossView Pro, this command disables the profiling system. Normally, you should disable profiling if you are not interested in the profiling results, as this will often improve the performance of the execution environment.

Example

To disable profiling, type:

pd



pe

pe

Function

Enable, turn on, profiling.



From the **Tools** menu, select **Profiling** if this item was not set.



The command line syntax is:

pe

Description

If profiling is supported by your version of CrossView Pro, this command enables the profiling system. Normally, you should disable profiling if you are not interested in the profiling results, as this will often improve the performance of the execution environment.

Example

To enable profiling, type:

pe



pd

proinfo

Function

List profiling results.



From the **Tools** menu, select **Profiling Report...**
Make your changes and select the **Update** button.



The command line syntax is:

```
proinfo [[all | module_or_function_name][filename]]
```

Description

If profiling is supported by your version of CrossView Pro and profiling is enabled, this command lists the profiling results. Without arguments (same as **all**) this command lists the profiling information of all modules and function.

Instead of listing the results you can also save the results in a file with extension `.pro`.

Normally, you should disable profiling if you are not interested in the profiling results, as this will often improve the performance of the execution environment.

Example

To list the profiling results of all modules and functions to the output window, type:

```
pe  
proinfo
```

To list profile information of function `main` to the output window, type:

```
proinfo main
```

To list profile information of all modules and functions in file `hello.pro`, type:

```
proinfo all,hello.pro
```



```
cproinfo, pd, pe
```

prst

Function

Reset the application being debugged to initial conditions. That is, set the program counter to the start address of the application.



From the **Run** menu, select **Reset Application**.



The command line syntax is:

prst

Description

The program counter is set to the start address of the application being debugged. This command does NOT perform a hardware reset of the target system. That is, no registers are modified except for the program counter.



This command is not allowed when the target runs in the background.



R, rst

pU

Function

Set the viewing position to the previous not covered block of statements.



Use the scroll bar and click on the desired line.



The command line syntax is:

pU

Description

If code coverage is supported by your version of CrossView Pro, this command enables you to skip to the previous block of statements that have not been executed while the program was running on the target.

Example

To move the cursor to the previous not executed block, type:

pU



nC, nU, pC

Q

Function

Quiet breakpoint reporting.



The command line syntax is:

Q

Description

If this appears as the first command in a breakpoint's command list, the debugger does not make the usual announcement of:

function: line number: source file

when the breakpoint is hit.

The purpose of this command is to allow quiet breakpoint reporting. For example, to check the value of a variable without cluttering the screen with text.

Example

If you type the following:

```
21 b {Q; var1}
```

CrossView Pro will set a breakpoint at line 21. When that breakpoint is hit, CrossView Pro will print the value of `var1`, but will not print the current function, line number, and source file.



b

q

Function

Quit a debugging session.



From the **File** menu, select **Exit**.



The command line syntax is:

q [**s** | **y**]

Description

CrossView Pro will prompt you if you really want to quit if you do not specify anything. Note that the current desktop settings are NOT saved then!

Typing **q s** saves the current desktop settings and quits the debugger without confirmation.

Typing **q y** does not save the current desktop settings and quits the debugger without confirmation.

Inside a command line procedure call it will just quit from this.

When the target runs in the background CrossView Pro will first stop the target.

R

Function

Reset program and begin execution from initial conditions.



From the **Run** menu, select **Reset Application** and then **Run**.



The command line syntax is:

R

Description

Reset the application being debugged and begin execution from initial conditions. The program counter is set to the start address of the application being debugged. This command does NOT perform a hardware reset of the target system. That is, no registers are modified except for the program counter.



This command is not allowed when the target runs in the background.



C, g, prst

rawmemget

Function

Retrieve data from the target into a buffer.



The command line syntax is:

```
rawmemget address,type,count,buffername [interleave]
```

Description

The **rawmemget** command is used to retrieve data from the target system and to store the data in the acquisition buffer *buffername*. Data in the acquisition buffer is of type `double`. CrossView Pro will automatically handle data conversion based upon the *type* of the data. It reads *count* elements of type *type* from the target starting at address *address* into the buffer.

interleave indicates the distance between successive elements. The default value is `sizeof(type)`.

Example

To retrieve 18 integer values from memory starting at address 0x100 and store these in acquisition buffer `$buffer`:

```
rawmemget 0x100,int,3*6,$buffer
```



bufa, bufd, graph, memget.

Section 11.5, *Data Analysis*, in chapter *Special Features*.

rst

Function

Reset target system to initial conditions.



From the **Run** menu, select **Reset Target System**.



The command line syntax is:

rst

Description

The target is initialized according to the power-up sequence for the processor. Almost all registers, including the system stack pointer and program counter are initialized.



A target system reset may have undesired side effects. To be sure that the application code is correct, a download must be performed after a target system reset.



This command is not allowed when the target runs in the background.



R, prst

S

Function

Single step C statements, stepping over function calls.



To step *over* a function, click on the **Step Over** button in the Source Window. You can also select **Step Over** from the **Run** menu. Check the **Step Mode** menu item in the **Run** menu: **Source line step** must be selected.



The command line syntax is:

[*exp*] **S**

Description

If you try to step over a call to a function which contains a breakpoint (or which calls another function with a breakpoint) then the breakpoint will be hit.

Stepping over a function means that CrossView Pro treats function calls as a single statement and advances to the next line in the source. This is a useful operation if a function has already been debugged or if you do not want to take the time to step through a function line by line.

When multiple statements are present on one line, they are all executed by this single step.



This command is not allowed when the target runs in the background.

Example

To step one C statement, type:

s

To step five C statements, type:

5 s



C, s, si, Si

S

Function

Single step C statements, stepping into function calls



To step *into* a function (single step), click on the **Step Into** button in the Source Window. You can also select **Step Into** from the **Run** menu. Check the **Step Mode** menu item in the **Run** menu: **Source line step** must be selected.



The command line syntax is:

[*exp*] **s**

Description

Single step *exp* (default is 1), C statements, stepping into function calls.

Stepping into a function means that CrossView Pro enters the function and executes its prologue machine instructions halting at the first C statement. When the end of the function is reached, CrossView Pro brings you back to the line after the function call. The debugger changes the source code file displayed in the Source Window, if necessary.



This command is not allowed when the target runs in the background.

Example

To step one source instruction, type:

s

To step five source instructions, type:

5 s



C, S, si, Si

save

Function

Save macros.



From the **Tools** menu, select **Macro Definitions...** to open the Macro Definitions dialog box. From this dialog box, you can save macros with the **Save** button. To save macro definitions in a file other than the current one, click on the **Save as...** button.



The command line syntax is:

save *file*

Description

Save all currently defined macros in the specified file. This file is in the format of a sequence of **set** commands, and thus can be loaded by reading it as a playback file. See the **<** and **<<** commands.

An existing save file with the same name will be overwritten.

Example

To save the definitions of the currently defined macros in the file `mac.sav`, type:

save mac.sav



set, unset, echo, !, <, <<

set

Function

Definition and display of macros.



To create a macro, select **Macro Definitions...** from the **Tools** menu. Click on the **New...** button and add a new macro.



The command line syntax is:

```
set [ name [ "cmds" ] ]
```

Description

The set command allows for definition and display of macros. If *name* and *cmds* are supplied, a macro entry is made associating the *name* with the commands. If only *name* is supplied, the body of the specified macro is displayed.

If no arguments are supplied the names of all currently defined macros are displayed. Macro definitions must contain the body of the macro in double quotation marks.

Macros may take arguments. In the body of a macro formal arguments are referred to as \$*n*, where *n* is the argument number starting from 1.

It is important to understand that macro expansion takes place for all names. Therefore, if you wish to pass the name of an existing macro to a command, such as **set**, you must escape it with '!', to keep CrossView Pro from expanding the name.

Example

To display the names of all currently defined macros, type:

```
set
```

To display the body of the macro named macro, type:

```
set macro!
```

To define macro to be a macro which lists the registers then enters the function given by its first argument, type:

```
set macro "l r; e $1"
```

To invoke this macro, you might type, for example:

```
macro(main)
```



```
unset, echo, save, !
```

Si

Function

Single step machine instructions, stepping over subroutine calls



From the **Run** menu, select **Step Mode | Instruction step**. Then click on the **Step Over** button in the Source Window, or select **Step Over** from the **Run** menu.



The command line syntax is:

[*exp*] **Si**

Description

Single step *exp* (default is 1) machine instructions, stepping over subroutine calls.

If you try to step over a call to a subroutine which contains a breakpoint (or which calls another subroutine with a breakpoint) then the breakpoint will be hit.

The next instruction to be executed is shown as a disassembled instruction, not as a C statement.



This command is not allowed when the target runs in the background.

Example

To step one machine instruction, type:

si

To step five machine instructions, type:

5 si



C, s, S, si, R

si

Function

Single step machine instructions, stepping into subroutine calls



From the **Run** menu, select **Step Mode | Instruction step**. Then click on the **Step Into** button in the Source Window, or select **Step Into** from the **Run** menu.



The command line syntax is:

```
[ exp ] si
```

Description

Single step *exp* (default is 1), machine instructions, stepping into subroutine calls.

The next instruction is shown as a disassembled instruction, not as a C statement.



This command is not allowed when the target runs in the background.

Example

To step one machine instruction, type:

```
si
```

To step five machine instructions, type:

```
5 si
```



C, s, S, Si, R

st

Function

Stop the execution of the target immediately.



The command line syntax is:

st

Description

This command stops the running process immediately.



CB, wt

T

Function

Stack trace with local variables



The command line syntax is:

[*exp*] **T**

Description

Produce a trace of functions on the stack and show local variables. Only the first *exp* levels of the stack trace will be displayed. If *exp* is omitted, all of the levels of the stack trace (up to 20) will be printed.

This command works independently of the Stack Window.



This command is not allowed when the target runs in the background.

Example

To print out a stack trace of 20 levels with corresponding local variables, type:

T

To print out the top five levels of the stack trace with corresponding local variables, type:

5 T



e, l, t

t

Function

Stack trace.



From the **View** menu, select **Stack**. The Stack Window shows the current situation in the stack after the program has been stopped. It displays the following information for each stack frame:

- The name of the function that was called
- The value of all input parameters to the function
- The line number in the source code from which the function was called



The command line syntax is:

[*exp*] **t**

Description

Produce a trace of functions on the stack.

exp specifies the number of levels of the stack trace to be displayed. If omitted, up to 20 levels of the stack trace will be printed.

Each stack level shown in the Stack Window is displayed with its level number first. The levels are numbered sequentially from zero. That is, the lowest/last level in the function call chain is always assigned zero.



This command is not allowed when the target runs in the background.

Example

To print out a stack trace of 20 levels, type:

t

To print out the top five levels of the stack trace, type:

5 t



e, l, T

td

Function

Disable, turn off, trace.



From the **Tools** menu, select **Trace** if this item was set.



The command line syntax is:

```
td
```

Description

If trace is supported by your version of CrossView Pro, this command disables tracing (both instruction level, high level and raw). Trace is automatically disabled when you close the Trace Window.

Example

To disable tracing, type:

```
td
```



```
te
```

te

Function

Enable, turn on, trace.



From the **Tools** menu, select **Trace** if this item was not set.



The command line syntax is:

```
te
```

Description

If trace is supported by your version of CrossView Pro, this command enables tracing (both instruction level, high level and raw). Trace is automatically enabled when you open a Trace Window.

Example

To enable tracing, type:

```
te
```



```
td
```

u

Function

Toggle the updating of the appropriate window when the target runs in the background.



The command line syntax is:

```
[interval] u [d|k|r|s|a|mem|t]
```

Description

The following windows can be updated:

d (Data), **k** (Stack), **r** (Register),
s (Source), **a** (Assembly), **mem** (Memory), **t** (Trace)

With *interval* you can specify the update interval (in seconds). If *interval* is zero, no window is automatically updated.

The updating of the Data Window is ON at startup, the others are OFF

If all windows are being updated and/or many monitor commands are active it will increase the load on the communication between CrossView Pro and the target.



This command is not available if the background mode is not supported (check the addendum).

Example

To toggle the updating of the Register Window, type:

```
u r
```

To toggle the updating of the Source Window, type:

```
u s
```

To disable period updating, type:

```
0 u
```



CB, ubgw

ubgw

Function

Update the appropriate window when the target runs in the background.



From the **View** menu, select **Background Mode** and select one of the refresh options.



The command line syntax is:

```
ubgw [ s | a | k | r | d | mem | t | all ]
```

Description

The following windows can be updated:

s (Source), **a** (Assembly), **k** (Stack), **r** (Register), **d** (Data), **mem** (Memory), **t** (Trace), **all** (all open windows)

Without an argument, the **ubgw** command refreshes all windows selected by the background mode (**u** command).

The **ubgw all** command refreshes all open windows.

This command is not available if the background mode is not supported (check the addendum).

Example

To update the Source Window, type:

```
ubgw s
```

To update the Memory Window, type:

```
ubgw mem
```



u

unset

Function

Delete a macro definition.



From the **Tools** menu, select **Macro Definitions...** to open the Macro Definitions dialog box. Highlight the name of the macro and click on the **Delete** button.



The command line syntax is:

```
unset [ name !]
```

Description

The **unset** command deletes a macro. If *name* is supplied, the specified macro is deleted. If no arguments are supplied, all currently defined macros are deleted after CrossView Pro confirms your intent.

It is important to understand that macro expansion takes place for all names. Therefore if you wish to pass the name of a macro to a command, for example **unset**, you must escape it with '!', to keep from expanding the name.

Example

To delete all macros, type:

```
unset
```

CrossView Pro will first ask for confirmation. To delete all the macro definitions at the same time, click on the **Delete all** button in the Macro Definitions dialog box.

To delete the macro named macro, type:

```
unset macro!
```



set, echo, save, !

update

Function

Update a Data Analysis window.



Click on the **Update Data Analysis Window** button in a Data Analysis window.



The command line syntax is:

```
update "window"
```

Description

Update Data Analysis window *window* by issuing a sequence of update commands. These update commands were added with the **graph_add_update** command.



When you use the **update** command in a complex breakpoint, you should append a '!' character to prevent early macro expansion.

Example

To retrieve data and show it in window demo, type:

```
graph_clear_updates "demo"  
graph_add_update "demo",memget data[$i],100,$buffer  
graph_add_update "demo",graphm "demo","show_x_t.cxl"  
graph_add_update "demo",graph "demo","x_t.cxl",$buffer,0,1  
update "demo"
```

To update window demo as part of a complex breakpoint, type:

```
0x100 bi {update! "demo"}
```



graph_add_update, **graph_clear_updates**.

Section 11.5, *Data Analysis*, in chapter *Special Features*.

use

Function

Change source directories run-time.



From the **Target** menu, select **Settings...** to open the Target Settings dialog box. Click on the **Configure...** button and specify the names of the directories containing your source files. Relative paths are allowed.



The command line syntax is:

```
use [ path ]...
```

Description

The **use** command changes the source directories. Without a *path* this command empties the search path, except for the path **.** (current directory). If one or more *paths* are supplied, this command adds the, semicolon separated, paths to the list of searched directories. Relative paths are allowed.

Example

To clear the source directory path, type:

```
use
```

To search for source files in the directory `/project/src` and in the `src` directory relative to your current directory, type:

```
use /project/src;../src
```



1 d

wt

Function

Wait for the completion of the target.



The command line syntax is:

wt

Description

This command can only be used if the target runs in the background mode.

This command waits for the running process to stop.

Waiting can be interrupted by typing *ctrl*-C. The target continues to run without interruption. It could be that some informational messages from the target are displayed in the command window. They can be ignored.



CB, st



Function

Force an exit from assertion mode.



The command line syntax is:

[*exp*] **x**

Description

Normally this command stops execution immediately, but if *exp* is present and its value is non-zero, then CrossView Pro finishes executing the entire command list of the current assertion.

Example

To define an assertion to stop the program when the value of global variable `myvar` exceeds 10, type:

```
a if (myvar > 10) {x}
```

To define an assertion to suspend the assertion mechanism and continue program execution when global variable `myvar` exceeds 10, type:

```
a if (myvar > 10) { A s; 1 x; C}
```



a, A, 1

Z

Function

Toggle case sensitivity in searches



From the **Edit** menu, select **Search String...** to open the Search String dialog box. This dialog contains the **Case Sensitive** check box.



The command line syntax is:

Z

Description

Toggle case sensitivity in searches. The initial state of this toggle depends on information in the currently loaded absolute file. Use the **I** command to find out the state of the case sensitivity.

This command affects everything: file names, function names, variables and string searches.



/, ?

REFERENCE

CHAPTER

14

ERROR MESSAGES



TASKING



14

CHAPTER

14.1 WHAT THIS CHAPTER COVERS

The following is a list of common user error messages, and some suggested ways to solve the problem.

CrossView Pro is a complex program running on several hosts. From time to time, slight differences between the documentation and the program's operations do occur. The list of errors presented below and the suggested remedies may not be, therefore, entirely comprehensive.

If you get a message that begins with "XVW Error" or "XVW Fatal Error" please contact TASKING technical support for help.

14.2 ERROR MESSAGES

(in alphabetical order):

"member-name" is not defined for "enum enum"

You cannot assign or compare an enum type with a name that is not in the enumeration's members. Try casting the enum to a different type.

'save' must have a filename; type 'help save' for more information

The **save** command requires a file to be supplied. Note: if the supplied file name already exists, it will be overwritten.

***** Fatal XVW error**

CrossView Pro has detected a error which it can not handle. If information is displayed, you may be able to detect the source of the error and correct it. Otherwise, if the message persists, please contact TASKING Technical Support.

0xvalue is an invalid value. The register *register* is unchanged.

The *value* supplied is incorrect for the specified register. Verify that both the value and the register are correct and retry.

Adding 2 pointers not allowed

You cannot add two pointers together in an expression. If you intended to add to a pointer, make sure that the argument is a value, not another pointer.

Address not allowed for '! or ~ or % operator'

The "Not", "One's complement", and "Modulus" operators cannot be used with an address. If you intended to perform the operation on the *contents* of the address, please be sure to dereference the pointer.

Addresses not allowed in '* or / operator'

The multiply and divide operators cannot be used with address data. If you intended to perform the operation on the *contents* of the address, please be sure to dereference the pointer.

Addresses not allowed in 'bitwise logical or logical or shift operators'

Bitwise logical (&, ^, or |), logical (&& or ||), and shift (<< >>) operators only work on data, not addresses. If you intended to perform the operation on the *contents* of the address, please be sure to dereference the pointer.

Attempt to set breakpoint at invalid address

The memory location is not available. If the memory location is not out of the target chip's range, you may need to map the target system's memory to allow access to this location.

Bad argument to the *command* command

The argument you have given to the **sio** or **f** command is not allowed. Refer to the *Command Reference* chapter, for allowable arguments and their meanings.

Bad assertion number: *number*

The number *number* is not a valid assertion number. List assertions with the **la** (list assertions) command to determine which assertion numbers are valid.

both expressions must be addresses for 'relational operator'

If one of the expressions is an address type, both expressions for relational operators (<, <=, >, >=, ==, and !=) must be address types. Retry with both expressions as either addresses or arithmetic types.

Breakpoint is (or at the address of) an CrossView internal breakpoint. It can not be deleted.

You may not install a breakpoint *over* an CrossView Pro internal breakpoint. See *Breakpoints and Assertions* chapter for more information.

com* return code=*code

The MS-DOS version of CrossView Pro received a status condition from the monitor communication channel which it can not handle. If the condition persists, please contact your system administrator, or call the TASKING Technical Support staff for assistance.

***command* takes no arguments.**

The command *command* needs no arguments. Refer to the *Command Reference* chapter, for the command syntax.

Can not open file (*file*)

CrossView Pro could not open the file *file*. Check the spelling of *file* and check that the file is in the correct directory. You should also check the permission of *file*. With MS-DOS, check the CONFIG.SYS file for the maximum number of open files allowed. Increase the number and reboot if necessary.

Can not output to input stream

An attempt was made to output to an input stream. The most common case is incorrectly setting up your simulated i/o streams. Correct and retry.

Can not scroll that window

The window you have tried to scroll is not scrollable. Examine your choice of window and/or your choice of windowing commands.

Can't define macro: out of space

There is not enough host memory to add your macro. Eliminate one or more unused macros before adding a new one.

Can't expand macro: out of space

There is not enough host memory to expand your macro. Eliminate one or more unused macros before adding a new one.

Can't monitor data: out of space

CrossView Pro cannot add any more variables or expressions to monitor. You must delete one or more variables or expressions before adding any more.

Can't open *logfile-name* as log file

CrossView Pro could not open the specified host-to-target system communications logfile. Check the spelling of *logfile-name* and that *logfile-name* is in the correct directory. Check permissions of *logfile-name*. With MS-DOS, check the CONFIG.SYS file for the maximum number of open files allowed. Increase the number and reboot if necessary. Make sure the filename is valid for the host Operating System.

Can't open *output-file-name* as output file

CrossView Pro could not open the specified output file. Check the spelling of *output-file-name* and that *output-file-name* is in the correct directory. Check permissions of *output-file-name*. With MS-DOS, check the CONFIG.SYS file for the maximum number of open files allowed. Increase the number and reboot if necessary. Make sure the filename is valid for the host operating system.

Can't open *playback-file-name* as playback file

CrossView Pro could not open the specified playback file. Check the spelling of *playback-file-name* and that *playback-file-name* is in the correct directory. Check permissions of *playback-file-name*. With MS-DOS, check the CONFIG.SYS file for the maximum number of open files allowed. Increase the number and reboot if necessary. Make sure the filename is valid for the host operating system.

Can't open *record-file-name* as record file

CrossView Pro could not open the specified recording file. Check the spelling of *record-file-name* and that *record-file-name* is in the correct directory. Check permissions of *record-file-name*. With MS-DOS, check the CONFIG.SYS file for the maximum number of open files allowed. Increase the number and reboot if necessary. Make sure the filename is valid for the host operating system.

Can't open file '*file*'

CrossView Pro could not open the specified file. Check the spelling of *file* and that *file* is in the correct directory. Check permissions of *file*. With MS-DOS, check the CONFIG.SYS file for the maximum number of open files allowed. Increase the number and reboot if necessary. Make sure the filename is valid for the host operating system.

Can't perform trace, out of memory

There is not enough host memory to support tracing. Reduce memory demands and retry again. If the problem persists, please contact the TASKING Technical Support staff for assistance.

Can't set breakpoint; either the current file has no symbols, or line *line#* is not inside any procedure in the current file.

CrossView Pro was unable to set the breakpoint that you specified. First check the location of line *line#* and verify that it is in the current procedure being debugged. If it is within the current procedure, then you may need to compile/assemble/link/locate for debugging. Refer to chapter *Overview* for details.

Can't start a new process. Feature not implemented.

Your host system does not support shell commands. Any attempt to issue shell commands will cause this message to be displayed.

Can't write to a read-only SFR.

The SFR register is a read-only register. It can not be set or altered.

Cannot allocate memory for symbol table

Allocating memory for storing the symbol table failed. Remove some tasks from memory or add more memory to your computer system.

Cannot allocate symbol table memory buffers

The symbol table is too large for CrossView Pro. You may need to selectively compile with the **-g** switch only those files and procedures that most interest you.

Cannot allow that combination of operand(s) and operator

The operand(s) is/are incompatible for this type of operation. For example, you may not add two structures. Please verify the operation and data types you are using.

Character constant is missing ending '

Character constants must be delimited with single quotes. Example: 'a'.

Command '*command*' not allowed while emulator running in background

The target is running, this command is not allowed unless the target is stopped. See the **st** command.

couldn't *error-message*

VMS is reporting a condition that CrossView Pro can not handle. If the condition persists, please contact your system administrator, or call the TASKING Technical Support staff for assistance.

Data already being monitored "*task-id*":'*symbol*'

The variable or expression *symbol* is already being monitored by CrossView Pro. You do not need to enter it again.

Display format required

The display command expected an output format option that was not supplied. See chapter *Command Language* for valid format options and their meanings.

Double not allow in '*% or ~ operator*'

You may not use the one's complement or modulus operators on double floating point types.

Double not allow in '*bitwise operator*'

You may not use *bitwise* operators (&, ^ and |) on double floating point types.

ERROR: you must enter ?,i,r,d

CrossView Pro's line editor only supports the following commands: **?-help**, **i-insert**, **r-replace**, **d-delete**, and **<cr>** to execute command.

Establish a file context first.

The command executed requires an active file. Verify the file you specified to CrossView Pro on start up.

Establish a procedure context first

The command executed requires an active procedure. Either execute the command from within a procedure, or give a procedure name as an argument to the command.

Exiting procedure call state

An unknown system signal caused the end of a command line function call.

Expecting stream number

The following forms of the **sio** command expect a stream number:

stream sio {i | o} {file | screen}

stream sio d

stream sio p prompt

Expression garbaged

The symbol table contains a type that is unknown to CrossView Pro. Please verify that you are using the compiler and utilities supplied to you. If the condition persists, please contact the TASKING Technical Support staff for assistance.

file has already been edited, going to NEW file

The command executed requires that the file be edited only once. A new file has been created.

failed to allocate the SIO tables

Entries for recording simulated input/output information could not be allocated due to lack of host memory. Please contact your system administrator, or call the TASKING Technical Support staff for assistance.

Float not allowed in '% or ~ operator'

You may not use the modulus or one's complement operators on floating point types. Change the data type to an appropriate type, for example, integer.

Float not allowed in 'bitwise or shift operator'

You may not use the *bitwise* (&, ^, or |) or *shift* (>>, or <<) operators on floating point types. Change the data type to an appropriate type, for example, integer.

Framing Error on COM port number

The host computer detected a data frame communication error on COM port *number*. Check the host and target communication set up as well as line connections. If the problem persists, please contact your system administrator, or call the TASKING Technical Support staff for assistance.

I can't put something that big in the child process

The size of the expression exceeds the buffer size needed to spawn a child process. Be sure you have linked `end.c` in your application. This module implies space for CrossView Pro in your execution environment. Refer to section *Building Your Executable* in chapter *Overview*. If this condition persists, please contact the TASKING Technical Support staff for assistance.

I don't have symbols for this procedure

You will need to re-compile, assemble, link and locate with the proper debugging options before using this command. See section *Building Your Executable* in chapter *Overview* for details.

I have no source file for this address

The program counter holds an address which is outside all the address ranges that CrossView Pro knows about. This may happen if program execution has reached a file that was not compiled with the `-g` generate debug symbols switch.

I need a linenumber

The `go g` command requires a line number. Enter a line number and the command will be executed.

Illegal address for Emulator Hardware Breakpoint

The address specified is out of emulator hardware breakpoint memory range. Verify the address and retry.

Illegal argument ("0") to 'p' command

You must specify a number greater than 0 for the `p` command, which prints the specified number of lines.

Illegal argument to '*command*' command: '*argument*'

You have passed an illegal argument to the specified command. Refer to chapter *Command Reference* for legal arguments.

Illegal argument to `ct`: '*argument*'

You have passed an illegal argument to the C-trace command. Refer to chapter *Command Reference* for legal arguments.

Illegal data monitor command

You have passed an illegal argument to the **m** data monitor command.

Legal commands are:

- m exp** to set up monitoring
- id* **m d** to delete monitoring of a specific expressions
- m d** to delete monitoring of all expressions

Illegal third arg to set: 'argument'

The **set** command may have only two arguments: the name by which the macro is known and the command string to be executed when the macro is invoked. Enclose the command string in quotes, separating the individual commands with semicolons. Refer to chapter *Command Reference* for more information.

Improper floating point format length

You have specified a format length that is inconsistent with floating numbers. Legal lengths are 4 and 8 bytes.

Improper integer format length

You have specified a format length that is inconsistent with integer numbers. Legal length are 1, 2, and 4 bytes. You may also choose **b**, **s**, or **l** for 1, 2, and 4 byte integers.

Improper string format length

You have specified a format length that is inconsistent with character strings. Choose a positive number.

Input buffer overflow

CrossView Pro is over-running the input buffer. Contact your system administrator to either increase the input buffer or lower the communication line baudrate.

Input communications buffer overflow on COM port

CrossView Pro is over-running the input buffer. Contact your system administrator to either increase the input buffer or lower the communication line baudrate.

Input from stdin longer than *max-input-size* characters: *input-string* Command truncated

The input data is longer than the input buffer, therefore the data was truncated at *max-input-size*. Try to reduce the input data and/or commands.

Internal error while setting an instruction level breakpoint

If this error condition persists, please contact the TASKING Technical Support staff for assistance.

Invalid assertion maintenance command

You have entered an illegal assertion command. Valid commands are:

- a a** to activate assertions
- a d** to delete assertions
- a s** to suspend assertions

Invalid value for uplevel break.

You have entered an illegal value for an uplevel break. The form of the command is *exp bU* or *exp bU*, where *exp* determines how many returns from functions should occur before the break. Execute the **t** command to find out how many levels down in the stack you are, then choose an appropriate value for the uplevel break. See chapter *Command Reference* for more information.

Invoking procedure calls not allowed while emulator is running in the background

The target is running, this command is not allowed unless the target is stopped. See the **st** command.

Macro Expansion error: expansion looping

CrossView Pro looped 50 times while trying to expand this macro without completing the expansion. Check the logic of the macro arguments. It may need to be corrected or simplified.

Macro Expansion error: expansion too large

The macro expansion exceeds 200 commands. The macro must be simplified.

Macro Expansion error: missing '('

See the command reference page or use the help command to review macro command syntax.

Macro Expansion error: missing ')'

See the command reference page or use the help command to review macro command syntax.

Macro Expansion error: missing ','

See the command reference page or use the help command to review macro command syntax.

Macro Expansion error: not enough args

See the command reference page or use the help command to review macro command syntax.

Macro Expansion error: out of space

There is not enough memory to expand the macro. Eliminate one or more unused macros before adding a new one.

Maximum trace size is: *max-trace-size*

CrossView Pro can perform C tracing only up to *max-trace-size* source lines. Choose a number less than *max-trace-size* with the **ct** command.

Missing { after if command

The required format for the **if** command is: **if** *exp* {*commands*}

Missing file name or 'screen'

The **sio** command was missing a required parameter for setting up a simulated i/o stream. See chapter *Command Reference* for command definition and format.

Missing format character

You did not specify a display format type with your command. Either remove **'/'** from the command, or add a format character.

Missing prompt string

You did not specify a prompt string with the **sio** command. Either remove **p** from the **sio** command, or add a prompt string.

Must supply 'b' or 'f'

The color command requires a value of **f** for foreground or **b** for background to modify the screen color.

Must supply 'r', 'w' or 'b'

Both the data range (**bd**) and data (**bd**) breakpoint commands require the type of data modification to generate a break condition. Use **r** for read, **w** for write, and **b** for both read/write. Please see chapter *Command Reference* for more information.

Must supply data to be monitored

You did not specify a variable or expression to the **m** monitor command. Please provide a variable or expression to be monitored, for example, `m myvar`.

Must supply second address with bD command.

The **bD** command requires two addresses. Either specify an upper limit if you want to break anywhere in memory range, or use the **bd** command if you want to break on an individual address.

**Negative /baudrate value ignored. (VAX)
or****Negative baud rate (-S) value ignored.**

The baudrate specified was a negative value. Please specify a legal value or use the default.

**Negative /TIMEOUT value ignored. (VAX)
or****Negative timeout interval (-I) value ignored.**

The time out value specified was negative. Please specify a legal timeout value or use the default.

No child process

The CrossView Pro internal data structure containing user information about child processes is not as expected. Please contact the TASKING Technical Support staff for assistance.

No current file

Undefined special variable, `$file`; probably due to debugging where no symbols are present.

No current line number

Undefined special variable, `$line`; probably due to debugging where no symbols are present.

No current procedure

Undefined special variable, `$proc`; probably due to debugging where no symbols are present.

No host memory

There is not enough space in memory to execute this command. Check whether you have unnecessary processes running in the background or resident in memory.

No host memory for command

There is not enough space in memory to execute this command. Check whether you have unnecessary processes running in the background or resident in memory.

No macros to save; file not created

CrossView Pro found no macros to save, therefore the **save** command did not create a file.

No Match – *pattern*

CrossView Pro did not find the specified *pattern* in its search of this file. Check your spelling or case-sensitivity. Use the **Z** command to toggle case-sensitivity if necessary.

No memory space

There is not enough host memory to execute this command. Check whether you have unnecessary processes running in the background or resident in memory.

No more hardware breakpoints available

The target system uses hardware breakpoints to support the data breakpoint function. To continue, you must explicitly delete a data breakpoint before placing a new one.

No more room for directories (> *max-dir-size*)

You can reference no more than *max-dir-size* directories for source files.

No more SIO windows, I/O to command window.

Only four SIO streams can be displayed simultaneously in the SIO window. Subsequent SIO streams' output will be displayed in the command window.

No name of symbol file specified

CrossView Pro cannot deduce the name of a symbol file. No filename was given to the **N** command and no symbol file was currently loaded.

No playback name specified

Give the name of the playback file to be used for this session.

No process

CrossView Pro only allows one process to be debugged at the same time.

No such breakpoint

The breakpoint number was incorrect. List breakpoints with the **lb** command to find the correct breakpoint.

No such field name "*name*" for "<structure | union> *name*"

The field *name* is not recognized for the specified structure or union. Check the spelling of field name. The **/t** format will show you the names and types of a particular structure's or union's fields.

No Such Line

CrossView Pro can not find the specified line number in any of its known files. Please check the source window or a source listing for legal line numbers.

No such procedure, "*name*".

CrossView Pro does not recognize *name* as a procedure name. Check the spelling and whether the file was compiled/assembled/linked/located for debugging. Check that the file is in the appropriate directory.

No such procedure or file name: *procedure*

CrossView Pro does not recognize *procedure* as a procedure or file name. Check the spelling and whether the file was compiled/assembled/linked/located for debugging. Check that the file is in the appropriate directory.

No such PSW register state

Check register name and selected target.

No such register

The target processor does not have a register with that name.

No such sr reg state

Check register name and selected target.

No such stream

The stream you tried to delete does not exist. Check the stream number, correct, and retry.

No symbols – unable to determine end-of-procedure

CrossView Pro has no symbol information for this procedure. To facilitate debugging this procedure, you must compile, assemble, link and locate with the appropriate switches. Refer to the *Overview* chapter for details.

No symbols available in active procedures.

To get symbol information you must compile, assemble, link, and locate with the appropriate switches. Refer to the *Overview* chapter for details.

No symbols for that procedure

To get symbol information you must compile, assemble, link, and locate with the appropriate switches. Refer to the *Overview* chapter for details.

No User or System special variable matches prefix *name*

The *string* argument of the **ls** command did not match any user or system special defined variables. Check spelling and case-sensitivity and retry.
You may also enter **ls** to print out all the user and system special defined variables.

Not enough memory available to start up windows. Either use the **-nw (no window) option or remove resident programs from memory.**

CrossView Pro has detected that there is not enough host memory to execute the windowing software. You may need to use the **-nw** option to start up CrossView Pro in line mode. Check whether you have unnecessary processes running in the background or resident in memory.

Not enough memory to execute shell command.

The attempt to create a child process for the shell command failed due to the lack of host memory. Check whether you have unnecessary processes running in the background or resident in memory.

Not enough memory to start window mode

CrossView Pro has detected that there is not enough host memory to execute the windowing software. You may need to use the **-nw** option to start up CrossView Pro in line mode. Check whether you have unnecessary processes

Not enough space

CrossView Pro has detected a general error due to lack of host memory. Check whether you have unnecessary processes running in the background or resident in memory.

Not in known territory. Could not set breakpoint.

CrossView Pro's current location is not in a file or procedure that it knows about. The breakpoint request can not be performed.

Not in window mode

The command issued requires CrossView Pro windows to be active. Use the **WW** command and repeat the previous command.

Not that many levels active on the stack.

A stack level was specified that does not exist. Execute the **t** command to determine levels on the stack. See chapter *Command Reference* for more information.

Oops called with sig = *signal-number*

CrossView Pro has received a signal that it can not handle. Continuing from this point may result in a fatal process condition. If this condition persists, please contact your system administrator, or call the TASKING Technical Support staff for assistance.

Placement of the breakpoint handler must be in one of the restart vectors. Choose a value from 0 to 7.**Try again. (Hit <cr> to exit)?**

The specified placement for the breakpoint handler was not valid for this target. CrossView Pro is requesting a valid location.

Procedure "*name*" is not active on the stack.

The procedure *name* was not found on the current stack. Execute the **t** command to list functions which are active on the stack.

Procedure '*name*' is not at that stack depth

The procedure *name* was not found on the specified stack. Execute the **t** command to list functions which are active on the stack.

Procedure "*procedure*" is not active

The procedure *procedure* was not found on the current stack. Execute the **t** command to list functions which are active on the stack or **lp** for list of procedures known to CrossView Pro.

Program not completely loaded

An error occurred during loading a symbol file. Check what cause the problem (illegal filename or file format). You may retry to load a symbol file.

Prompt too long (> *max-number*)

Choose a prompt of no more than *max-number* characters.

Ran out of memory reading symbol file into memory

Reduce the size of the symbol file by re-compiling only the "interesting" files with the **-g** debug switch.

Read I/O request could not be queued

VMS detected an error for a read I/O queue which CrossView Pro can not handle. If the condition persists, please contact your system administrator, or call the TASKING Technical Support staff for assistance.

Readprompt I/O request could not be queued

VMS detected an error for a read I/O queue which CrossView Pro can not handle. If the condition persists, please contact your system administrator, or call the TASKING Technical Support staff for assistance.

Redo: line too large

Limit line length to fewer than 256 characters.

Result type too large for command line call.

A command line function call must pass the result back in a register. The specified function does not. You cannot call functions whose return value is greater than an integer, for instance floating point types and structures.

Result type undefined

Type casting from the expression or variable to the result type was not possible.

Second address smaller than first

When specifying a range of addresses for a data breakpoint, the second address must be higher than the first.

Sim I/O request too long (>*max-number* bytes)

The I/O request exceeds the maximum length.

Simulated I/O stream out of range

Choose a stream value between 0 and 7.

Sorry, the "v" command is not supported on this host

No visual editor is available on this host.

Stream already active

Either choose another stream, or deactivate this one before re-assigning it.

String constant is missing ending "

String constants must be delimited with double quotes: "

Subtracting 2 pointers not allowed

You cannot subtract two pointers in an expression. If you intended to subtract from a pointer, make sure that the argument is a value, not another pointer.

Symbol file is either unreadable or too short

The symbol file is not an absolute IEEE-695 file, or the file format is not correct, or the file is not an IEEE-695 file at all.

Symbol file is not formatted correctly

The symbol file is not intended for the type of microprocessor you are using.

Symbol not in current procedure

There is no symbol by this name in the current procedure. Check the spelling of the symbol name.

The '*command*' command accepts no args

The command *command* does not accept any arguments. See chapter *Command Reference* for more information on *command*.

The window would be too large; Total lines must not be greater than *max-size*

The window size options specified would create a window that would have exceeded the screen size. Retry with corrected window size options.

There is insufficient information to do a structure dump

CrossView Pro cannot uniquely identify the structure or part of the structure to be dumped.

There is no associated source.

The program counter holds an address which is outside all the address ranges that CrossView Pro knows about. This may happen if program execution has reached a file that was not compiled with the **-g** debug switch.

There is no available source line for the current address.***\$pc= address***

CrossView Pro is reporting that the current position has no associated source line. This may happen if you are trying to debug a routine that was not compiled with **-g** debug switch or are trying to debug a runtime library routine.

This does not appear to be a struct or a union

The data entered is not recognizable as a structure or union. Check the specified variable.

Timed read I/O request could not be queued

VMS reported a condition on a timed read i/o request that CrossView Pro could not handle. If the condition persists, please contact your system administrator, or call the TASKING Technical Support staff for assistance.

Too many args to unset: '*argument*'

You may specify only one macro at a time, for example, **unset** *name*, or you may remove all macros at once with **unset**.

Too many assertions (>*max-number*)

The maximum number of assertions allowed is *max-number* as shown in the error message. Remove a previous assertion before trying to add one, or reinvoke CrossView Pro with the **-a** option to increase the maximum number of assertions.

Too many breakpoints (> *max_number*)

The maximum number of breakpoints allowed is *max-number* as shown in the error message. You must explicitly delete a breakpoint before adding any new ones. Alternatively, you could re-invoke CrossView Pro with the **-b** option to increase the maximum number of breakpoints.

Too many locals (> *max-number*)

Eliminate some existing locals or reinvoke CrossView Pro with the **-s** switch to increase the number of locals allowed.

Too many modules

The symbol file describes an application that was constructed from more than 1818 modules.

Too many processes (> *max-number*)

CrossView Pro allows only one process to be debugged.

Too many streams (> *max-number*)

The maximum number of I/O streams, *max-number*, has been reached. You must eliminate an I/O stream before adding a new one.

Trace size is required

The required format of the command is *exp* **ct**, where *exp* is the number of statements to trace. Re-enter the command with a value for *exp*.

Type '*r*', to run program from power-on conditions or '*c*' to continue with current program pointer

This is to inform you that command *r* is not implemented and that you should use *r* or *c*.

Type of *command-line-expression* is too complex

The command line function returns a data type that CrossView Pro cannot handle. An example would be a function returning a structure.

Unexpected breakpoint type '*type*'

CrossView Pro has encountered a breakpoint with an unknown type attribute. Verify the previous break commands and re-try. If the condition persists, please contact the TASKING Technical Support staff for assistance.

Unknown command '*command*' (<*number*>)

CrossView Pro does not recognize *command*, and has echoed the command number for Technical Support purposes. Please check the spelling and retry. If the condition persists, please contact the TASKING Technical Support staff for assistance.

Unknown data monitor id '*number*'

The monitor number *number* that you tried to delete does not exist. Use the **M** command to list currently monitored variables.

Unknown data size

Valid data sizes are 1, 2, 4, or 8 bytes.

Unknown display mode

See chapter *Accessing Code and Data*, for a list of display mode options.

Unknown name '*name*'

Variable *name* is not in scope or is undefined.

Unknown procedure "*name*".

The function *name* does not exist in any file that CrossView Pro knows about. The file containing *name* may not have been compiled with the **-g** debug switch.

Unknown macro '*name*'

CrossView Pro does not recognize the macro name as given. Please check the spelling. You may list all current macros by using the **set** command with no arguments, or display the Macro window for currently defined macros.

Unknown window

CrossView Pro does not recognize the window name as given. See chapter *Command Reference* for valid window arguments.

Unsupported format type (*parameter*)

Supported types are **c** (character), **x** (hex), and **o** (octal).

Value *number* is not defined for this enum.

The member specified was not part of the enumerated set. Please check the spelling and verify that the correct enum was used.

Value exceeds depth of stack.

A stack level was specified that does not currently exist. Please check the value and retry. Check the stack window for valid stack levels, or execute a **t** command (trace stack) to determine the depth of the stack.

VMS error : cannot establish handler for signals

CrossView Pro on VMS could not establish proper error handlers. If the condition persists, please contact the TASKING Technical Support staff for assistance.

VMS error : cannot establish pasteboard

CrossView Pro on VMS can not establish the running environment. If the condition persists, please contact your system administrator, or call the TASKING Technical Support staff for assistance.

VMS error : cannot establish virtual keyboard

CrossView Pro on VMS can not establish the running environment. If the condition persists, please contact your system administrator, or call the TASKING Technical Support staff for assistance.

VMS error code = *number* \ Attempt to get message text fail.

CrossView Pro on VMS received an error while attempting to provide an error diagnostic message from the host error message library. If the condition persists, please contact your system administrator, or call the TASKING Technical Support staff for assistance.

Warning: NULL pointer dereference

The expression contained a null pointer dereference. Check the expression for possible errors, or verify that the pointer should in fact be null.

Warning: pointer dereference with invalid segment selector.

The pointer is addressing invalid memory and the dereference may report unexpected data results. Check the initialization of the pointer or verify that it has been set correctly.

Warning: too few parameters.

The command given was not invoked with the proper number of arguments. CrossView Pro will supply the command with defaults which may or may not produce the result you expected.

Warning: Using *file-b* instead of *file-a*

CrossView Pro could not find *file-a*, or *file-a*'s status was such that CrossView Pro could not use it. If *file-b* is not correct, check *file-a* spelling and its directory.

Warning: X=Y: X is *x-size* bytes and Y is *y-size* bytes

The assignment of two different size variables may cause unexpected results. Please correct the condition if possible. This condition is common when assigning string variables where string *y* is shorter than string *x*.

Warning: X=Y: X is *x-size* words and Y is *y-size* words

The assignment of two different size variables may cause unexpected results. Please correct the condition if possible. This condition is common when assigning string variables where string *y* is shorter than string *x*.

**Warning: CrossView comment terminated by end of command line
*source-command-line***

The playback file has a comment that was not terminated. It is by default terminated, but if the next line was the continuation of the comment, then unexpected results may occur. Please terminate comment strings on each line to avoid this warning.

Windows not enabled; use WW to enable

The command issued can only be used when windows are enabled.

Write I/O request could not be queued

CrossView Pro received a condition that it could not handle. If the condition persists, please contact your system administrator, or call the TASKING Technical Support staff for assistance.

Write-only register. Value may not be valid.

CrossView Pro set a write-only register but has no way of verifying the correctness of the register contents.

Wrong storage class for data breakpoint

You may not set a data breakpoint at the address of a register variable or special variables.

CrossView could not disassemble the emulator's trace buffer because the address information in the buffer is incorrect.

The trace buffer may be corrupted. Re-check the commands leading to this condition, and retry. If the condition persists, please contact the TASKING Technical Support staff for assistance.

XVW error – *message*

or

XVW Fatal error – *message*

These messages are generated by internal conditions that should not normally occur. The *message* is usually encrypted and should be brought to the attention of the TASKING staff. Please contact the TASKING Technical Support staff for assistance.

XVW:main – Cannot continue, incomplete initialization.

CrossView Pro's initialization was interrupted and could not be completed. Please re-start CrossView Pro, and if the condition persists, contact the TASKING Technical Support staff for assistance.

You can't goto a line outside of the current procedure

The specified line number is outside the current procedure. Change the line number to one within the procedure or enter the correct procedure before executing this command.

You may not assign from a host system string/array

The expression given performs an assignment that CrossView Pro can not perform at this time.

You may not assign from a void function

The expression attempts to assign a variable from a void function. Please check the return value of the function and verify that you are referencing the correct function.

You may not assign more than *max-size* bytes to a special variable

An attempt was made to assign greater than the maximum number of bytes to a special variable. Check the expression for errors, and check the variable's spelling.

You may not assign to a constant

The value of a constant cannot be changed. Check the name that you have specified.

You may not mix assignment of a scalar and an aggregate

An attempt was made to assign incompatible types of data. Please check the data types and retry.

You need to supply a program name.

CrossView Pro requires a program name to debug on the invocation line.



ERRORS

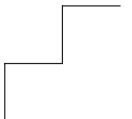
CHAPTER

15

GLOSSARY



TASKING



15

CHAPTER

15.1 WHAT THIS CHAPTER COVERS

This chapter defines terms common to CrossView Pro and source-level embedded systems debugging. Italicized items in definitions are also glossary entries.

15.2 GLOSSARY TERMS

A

absolute file. The IEEE-695 file (*.abs*) that contains symbolic debug information and the final executable code of the target system.

active window. The window last selected by the user in CrossView Pro that commands operate on as a default. An active window's title appears in a different color (on color monitors) or inverse video (on monochrome).

analysis. *See* **trace analysis**.

analysis window. The window where you can select to Modify, Begin, Halt, Display or Reset a trace analysis. *See also* **trace analysis**.

assertion. A command or set of commands to be executed before every line of source code, assessing the application state on validity. Assertions are especially useful in tracking down hard to find bugs when other methods fail. Individual assertions may either be active or suspended. *See also* **assertion mode**.

assertion mode. A mode of CrossView Pro operation under which assertions will be executed. Before CrossView Pro executes a source line of code, it assesses all assertions active. Since CrossView Pro is single stepping, breakpoints will not be effective. As long as there is at least one assertion active, CrossView Pro operates in assertion mode. A program running in assertion mode will be stopped when an asserted command executes the **x** (exit assertion mode) command.

B

background mode. A target dependent feature in CrossView Pro that lets the execution environment run and at the same time allows you to enter a reduced set of CrossView Pro commands, for example to monitor memory contents.

bias. A value added to program code addresses to tell CrossView Pro where the application has actually been loaded into memory. The bias can be set in the Load Application dialog or with the **-n** startup option.

breakpoint. A mechanism for stopping target program execution, for example at a particular line of code (*see* **code breakpoint**), when a memory address is accessed (*see* **data breakpoint**), or at a return from a function (*see* **up-level breakpoint**). There are two general kinds of breakpoints. Hardware, which the emulator or on-chip debug support sets in its circuitry, and software, which are special instructions placed in user code. Since the number of simultaneous hardware breakpoints is limited in number, CrossView Pro uses both kinds by default. Other types of breakpoints are for example **instruction count breakpoint**, **cycle count breakpoint**, **timer breakpoint** and **sequence breakpoint**. *See also* **probe point**.

breakpoint window. A CrossView Pro dialog displaying all breakpoints, and any attached commands.

C

C-trace window. A CrossView Pro window keeping a record of the most recently executed C or machine statements.

cache. Some microprocessors keep a copy of the most recently executed instructions in on-chip memory to speed-up execution.

code breakpoint. A breakpoint that halts program execution when a particular line of code is reached. A code breakpoint can have a command list. A breakpoint can be set on a line of source code or at the address of a machine instruction. *See also* **count**.

code coverage. *See* **coverage**.

command window. A CrossView Pro window that gives access to CrossView Pro via a command line interface with history.

command list. A series of CrossView Pro commands and/or C (assignment) statements attached to a code or data breakpoint, executed when the breakpoint is hit.

count. The number of times a breakpoint must be hit to finally stop execution. Breakpoints are created with a count of 1. The **C** command may be used to change the count of a breakpoint.

coverage. With **code coverage** the source line is marked for each source code line that is executed. Through code coverage you can find executed and non-executed areas of the application program. **Data coverage** allows you to verify which memory locations, i.e. which variables, are accessed during program execution. Additionally, you can see stack and heap usage. The availability of this feature depends on the execution environment.

cycle count breakpoint. A breakpoint that halts program execution after a specified number of CPU cycles. A cycle count breakpoint can have a command list.

current function. The function that is currently being executed. The current function is always at level 0 on the stack. Also stored in the CrossView Pro special variable `$PROCEDURE`.

D

data breakpoint. A breakpoint that halts program execution when a particular memory address (or an address within a particular range) is written to, read from, or both. A data breakpoint may have a command list and a **count**.

data coverage. *See coverage.*

data monitoring. CrossView Pro allows you to monitor expressions and variables in the Data window. CrossView Pro updates their values whenever execution stops.

data window. A CrossView Pro window displaying the values of monitored expressions.

diagnostic output. Program output designed for debugging purposes. With CrossView Pro, probe points and data monitoring can be used for diagnostic output, eliminating the need for intrusive and annoying printf calls compiled into code.

disassembly window. A CrossView Pro window showing a part of the disassembled program space. It also displays other information such as the current execution position, viewing position and installed breakpoints.

dot operand. The period character "." used in an expression to represent the last value CrossView Pro calculated. The dot operand is useful as shorthand.

E

embedded system. Computer(s) executing an application program built to run in (semi) real-time. An embedded system usually is part of a larger, non-computer system, hence the term "embedded." The TASKING product line is designed for embedded systems programming.

emulator. A device used to monitor and control various aspects of a microprocessor's operation. An emulator usually is built around two chips, the target microprocessor and a controlling chip. The controller chip can start and stop the target chip's program execution, and can examine and change registers and memory. An emulator can be connected via a probe to a hardware prototype to fully emulate the behavior of the target chip. *See* **ROM monitor**.

__end_. A run-time library routine used to implement command line function calls. It must be linked into the object code.

execution position. The source line to be executed next. *See* **viewing position**.

F

File System Simulation (FSS). A facility to redirect all C library file I/O operations on the target, to the host system via CrossView Pro. File system simulation is often used to provide input to an application for which no hardware I/O is available yet and to log test results.

format. The manner in which CrossView Pro displays addresses and data; for instance, hexadecimal, character and octal are different formats. You may include special format codes when specifying variables.

H

hardware breakpoint. *See* **breakpoint**.

help window. A window explaining the use of CrossView Pro windows and dialogs and summarizing the syntax and function of CrossView Pro commands.

history mechanism. A facility for modifying and executing previous CrossView Pro commands.

host system. The computer system on which CrossView Pro is run. The host system is connected to the target system, usually with an RS-232 cable.

I

image part. This is the downloadable part of the absolute file that contains the executable code of the target program. *See also* **absolute file**.

instruction count breakpoint. A breakpoint that halts program execution when a number of instructions have been executed. An instruction count breakpoint can have a command list.

interrupt key. The key that interrupts ongoing processes. On many systems this is *ctrl-C*.

I/O Simulation. A technique to intercept input and output for debugging purposes. I/O Simulation is often used for testing a program before the actual input and output hardware devices are present. *See also* **stream**.

L

local variable. A variable that can only be referenced from within its defining function.

low-level breakpoint. A code breakpoint placed on an individual machine instruction. Low-level breakpoints can be set with the **break code address** command.

M

macro. A user-created shorthand for a CrossView Pro command sequence. Macros can accept parameters and can be saved to a file.

main(). The function where a C program's execution begins. *See also system startup code.*

MAU. *See* **minimum addressable unit.**

memory map. The configuration of an emulator's memory that specifies which addresses are read-only, and which addresses are read/write. With many emulators, you must first set up a memory map before using CrossView Pro, for example via transparency commands.

minimum addressable unit. For a given processor, the amount of memory located between an address and the next address. It is not necessarily equivalent to a word or a byte. Abbreviated **MAU**.

monitoring. *See* **data monitoring.**

O

object language. A representation for target machine instructions, with the ability to represent either relocatable or absolute address locations.

on-line help. A complete summary of all CrossView Pro commands and individual descriptions available while CrossView Pro is running.

on-line tutorial. A playback file supplied with CrossView Pro that demonstrates CrossView Pro's capabilities.

output buffer. The location in memory where CrossView Pro directs I/O simulation output. *See also* **I/O Simulation.**

P

patch. A technique to alter program flow (without recompiling the source code) with CrossView Pro commands and/or C expressions. With CrossView Pro, it is possible to use breakpoints to alter program flow by patching in new code or moving the execution position around existing code.

pop-up window. A window that appears in certain situations that overlaps the current display. Pop-up windows usually contain information (like a command definition) that need not be continuously displayed.

probe. A part of an emulator that can be inserted in place of the target chip in the actual embedded systems hardware.

probe point. A special kind of breakpoint. When a probe point breakpoint is hit, the associated commands are executed and program execution is continued.

profiling. For each source code line that is executed, the timing information is given.

Q

quiet command. A **Q** instruction at the start of the command list of a breakpoint suppressing the default display of *function: line number: source file*.

R

record & playback. The ability to save CrossView Pro commands (and, if desired, Command window output) to a file. CrossView Pro can play back simple text files consisting solely of CrossView Pro commands.

register window. A CrossView Pro window showing the contents of the target microprocessor's registers.

reserved special variables. Special variables (\$LINE, \$PROCEDURE, \$FILE) whose values CrossView Pro maintains to reflect the current status of the debugging session. *See also* **special variables**.

ROM monitor. A program which supervises or controls, at an elementary level, the overall operation of an embedded system. Because of the limited hardware features of most boards containing ROM monitors, some CrossView Pro features may not be supported. *See also* **emulator**.

RS-232 cable. A cable that exchanges asynchronous data between the host and target systems.

S

scope. The extent to which a variable can be referred to. Global variables are always in scope; local variables are only in scope when their defining function is the current function.

select. To make a window active.

sequence breakpoint. A breakpoint that halts program execution when breakpoints are hit in a specified sequence. A sequence breakpoint can have a command list.

single stepping. Executing a source statement or a machine instruction then halting. Single stepping lets you observe a program executing in stop-motion, to observe registers, variables and program flow.

skidding. When a microprocessor executes a few instructions after a data breakpoint halts execution. On some microprocessors, execution may not stop until all instructions in its pipeline have been executed. It is important to realize therefore that a target program may not halt at the precise instruction where the data breakpoint occurred.

software breakpoint. *See* **breakpoint.**

source level debugger. A debugger capable of correlating source code and variable names with object code. CrossView Pro is a source level debugger.

source window. A CrossView Pro window displaying the high-level language program code. It also displays such information as the current execution position, viewing position and installed breakpoints.

special variable. A variable independent of the target program that CrossView Pro maintains for the user's benefit. Special variables start with a \$ and are defined when first mentioned. CrossView Pro also maintains **reserved special variables** that contain information about the state of the debugging session.

stack depth. The level that a particular return address from a function resides on the stack. The current function is always at stack depth zero.

stack traceback. An operation in which CrossView Pro reads the return addresses and passed parameters off the stack to reconstruct program flow.

stack window. A CrossView Pro window showing the function calls on the stack, with the values of the parameters passed to them.

startup options. Special command line switches passed to CrossView Pro when the debugger is first loaded. These options control items such as the number of assertions allowed, or can perform various actions such as to start recording screen output to a file.

stream. A particular input or output data path for I/O simulation. Per method, File System Simulation, File I/O or Debug Instrument I/O, a unique stream numbering scheme is used.

switches. *See* **startup options**.

symbolic debugger. A type of debugger generally limited to dealing with global, non-dynamic variables. Symbolic debuggers know nothing of the data types; they translate global names and global subroutines into addresses. *See also* **source level debugger**.

symbol information. The necessary information for CrossView Pro to correlate object code with source code. The symbol information is part of the absolute file. *See also* **absolute file**.

system startup code. A run-time library routine written in assembly language source that initializes the target environment before calling `main()`. *See also* **main()**.

T

target communication. The low-level communication between the host and the target system. For the most part, CrossView Pro handles target communications, allowing the programmer to concentrate on the high-level information.

target microprocessor. The chip on which the target program runs.

target system. The targeted microprocessor where the embedded application runs.

terminal window. A CrossView Pro window containing all the input and output streams directed to the screen. CrossView Pro can display several windows at a time.

timer breakpoint. A breakpoint that halts program execution after a specified number of seconds or timer ticks. A timer breakpoint can have a command list.

trace buffer. A target-resident buffer that contains the most recent instructions executed by the target microprocessor. CrossView Pro uses this buffer to deduce a C-trace.

trace analysis. An emulation bus analyzer captures bus cycle information from the address, data, and status buses of an emulation processor in sync with the processor clock. The states captured show a history of activity on the emulation processor bus.

transparency mode. The mode in which CrossView Pro passes user input directly to the emulator. Transparency mode is often used when setting up memory maps.

U

up-level breakpoint. A code breakpoint set at the return from a function at a specified stack depth.

V

viewing position. The line of source code currently being viewed. This line contains the dashed line cursor. Some commands operate by default on the viewing position. The viewing position and the execution position are initially the same, but you may adjust each individually.

APPENDIX

A

INTERPROCESS COMMUNICATION



A

APPENDIX

1 COM INTERFACE

1.1 INTRODUCTION

CrossView Pro provides a COM object interface on MS-Windows platforms. The purpose of the COM object interface is to make the command-line interface of the command window available to the outside world. Simultaneously, a callback mechanism is provided which allows the outside world to tap into events that occur within CrossView Pro (for example a breakpoint hit message). This is achieved by a COM connection point interface to which multiple programming languages can connect.

The CrossView Pro COM object can be used in programming languages like Python, Visual C++ or Visual Basic. Applications that are COM clients can also make full use of the CrossView Pro COM object interface. COM is a binary reusable object technology, linked tightly to MS-Windows. COM is closely related to ActiveX and Automation. ActiveX consists of a set of predefined interfaces to be implemented in a COM object and used to create pluggable GUI components. Automation is a similar set of predefined COM interfaces, but with a wider range of applications than ActiveX.

1.2 USING THE COM OBJECT INTERFACE

1.2.1 RUN-TIME ENVIRONMENT

The CrossView Pro COM object executes as an out-of-process server. Only one client per instantiated CrossView Pro COM object can connect. Each CrossView Pro executable has a unique identification (so-called UUID or GUID), independent of the version number. This is especially important for Visual Basic which stores the TypeLib UUID. This requires recompilation if the UUID changes across different versions of the same CrossView Pro executable.

1.2.2 COMMAND LINE OPTIONS

To prevent initialization dialogs at CrossView Pro startup (for example a dialog to specify which CPU type you use), you can use several command-line options which you can specify via the `Init()` method.

Use the following options instead of startup dialogs:

- tcfg** *file* Specifies a target configuration file which contains, among other things, the GDI module to be loaded among other things. This overrides the filename specified in `xvw.ini`.
- C** *cpu* Specifies the CPU type.
- D** *device_type,opt1[,opt2]* Specifies communications parameters such as communication port and baud rate.
- G** *path* Specifies the startup directory for CrossView Pro
- ini** Specifies the `xvw.ini` file.



Section 4.4, *Startup Options* in Chapter *Using CrossView Pro*

1.2.3 STARTUP DIRECTORY

The startup directory of CrossView Pro determines where the `xvw.ini` file is written. When CrossView Pro is invoked via its COM interface on MS-Windows, the startup directory is usually `C:\WINNT\system32`. You can change the location of the `xvw.ini` file with the **-G** command line option. This feature is useful when you are using two different CrossView Pro instances simultaneously.

1.3 COM INTERFACES

The following interfaces are provided with CrossView Pro:

ICommandLine

Default interface; provides CrossView Pro command interpreter access.

ICommandLineEvents

Connection point; provides the events output stream of CrossView Pro.
Works as a callback.

1.3.1 ACTIVATING THE COM OBJECT

Command line options are passed to CrossView Pro via the `Init()` method. It is necessary to call the `Init()` method before you can use the CrossView Pro COM object. CrossView Pro does not start as COM object, until after you have actually called the `Init()` method. If you do not need to pass any options, invoke `Init()` with an empty string.

Registering the server

Before you can use the COM object, you must register it in the MS-Windows Registry. Run CrossView Pro from the command line as follows:

```
xfw68 -RegServer
```

Similarly, you can remove the COM object from the Registry:

```
xfw68 -UnRegServer
```

To avoid the popup message when registering, two more command line options are available that are useful when you use batch files:

-RegServers	Same as -RegServer , but without message
-UnregServers	Same as -UnregServer , but without message box

1.3.2 METHODS

This section lists the methods that are supported by the CrossView Pro COM object's default interface 'ICommandLine'. The data types and return values are expressed as COM base types. For example, BSTR is a wide-character UNICODE string type, which is the same type as Visual Basic strings.

Init()

```
void Init(BSTR CommandlineOptions)
```

Passes command line options to the CrossView Pro COM. It is necessary to call the `Init()` method before you can use the CrossView Pro COM object. If you do not need to pass any options, invoke `Init()` with an empty string.

CommandlineOptions

The string with the command line options. The options are written as on a regular command line.

The list of supported command-line options can be found in the CrossView Pro User Manual.



See Section 4.4, *Startup Options* in Chapter *Using CrossView Pro* for a complete overview of all available command line options.

Execute

```
BOOLEAN Execute(BSTR Command, long SequenceNumber,  
                BSTR *Result)
```

Passes a command to CrossView Pro, executes it and returns TRUE or FALSE after the command has been executed.

Command The command to be executed by CrossView Pro.

SequenceNumber

A number that is unique for each command. You can use this number to distinguish the output in the events stream. If you do not use this, specify a value of 0.

Result The textual output of the command window, encapsulated in an annotated format. See `CmdAnnotatedOutput` in section 1.4 *Events* for the format description. Specify `NULL` if you do not want any output.

Returns: `TRUE` on success, `FALSE` on error.

ExecuteNoWait

```
BOOLEAN ExecuteNoWait(BSTR Command,  
                      long SequenceNumber)
```

Queues a command for execution and returns `TRUE` or `FALSE` after the command has been passed but before it is executed.

Command The command to be executed by CrossView Pro.

SequenceNumber

A number that is unique for each command. You can use this number to distinguish the output in the events stream. If you do not use this, specify a value of 0.

Returns: `TRUE` if the command is successfully passed, `FALSE` on error.

Halt

```
void Halt(void)
```

Halts the execution of the current command.

1.3.3 IMPLEMENTATION DETAILS

A multi-threading (MTA) type of apartment is used with a free-threading model, for example, `ThreadModel=Free`. However, each `CLSID` can have its own distinct `ThreadingModel`. Only one client can connect to a COM object instance of CrossView Pro. Each next `CoCreateInstance()` will result in a new CrossView Pro COM object instance being created.

Be aware that DLLs are not supposed to call `CoInitialize` themselves. Once the concurrency model for a thread is set, it cannot be changed. A call to `CoInitialize` on an apartment that was previously initialized as multithreaded will fail and return `RPC_E_CHANGED_MODE`.

Typically, the COM library is initialized on a thread only once. Subsequent calls to `CoInitialize` or `CoInitializeEx` on the same thread will succeed, as long as they do not attempt to change the concurrency model, but will return `S_FALSE`. To close the COM library gracefully, each successful call to `CoInitialize` or `CoInitializeEx`, including those that return `S_FALSE`, must be balanced by a corresponding call to `CoUninitialize`. However, the first thread in the application that calls `CoInitialize(0)` or `CoInitializeEx(COINIT_APARTMENTTHREADED)` must be the last thread to call `CoUninitialize()`. If the call sequence is not in this order, then subsequent calls to `CoInitialize` on the STA will fail and the application will not work.

Because there is no way to control the order in which in-process servers are loaded or unloaded, it is not safe to call `CoInitialize`, `CoInitializeEx`, or `CoUninitialize` from the `DllMain` function.

So, take care when establishing more CLSIDs in a GDI module.

1.4 EVENTS

CrossView Pro provides an events source, into which a client can tap via a COM connection point. Examples of events are "reporting which breakpoint has been hit" and "symbols have been loaded". Currently the following events are defined. Each event is terminated by a newline character. Prepare your client for new events, basically by ignoring unrecognized ones.

CommandInterpreterBusy

The debugger's command interpreter is executing a command line, or a GUI operation is in progress. A command line can comprise multiple target execution commands, so arbitrary Running and Stopped events may occur before the command line has been finished. An example for using this event is the disabling of menu entries in your tool.

You can send multiple `CommandInterpreterBusy` events without the `CommandInterpreterReady` counterpart. New commands can be sent to the debugger after this event has been issued, but they will be queued until the debugger is ready for new command input.

CommandInterpreterReady

The entire command line or GUI operation has either been executed completely or aborted. You can send multiple `CommandInterpreterReady` events without the `CommandInterpreterBusy` counterpart.

CommandCanceledByUser

The entire command line or GUI operation has can be canceled by the user, usually via the Halt button.

In case of DDE, the CrossView Pro command queue will be emptied. The command queue of all other IPCs, for example COM, will be preserved. This has been designed for the multi-core debugger which relies on commands –submitted by the multi-core debug system– always being executed, even if the user hits the Halt button.

Note that every command can be canceled this way, even when asking a variable's value. Often no value will be returned at all, because Halt aborted the evaluation.

HaltButtonPressed

Tells that the user has pressed the Halt button. This is necessary because in CrossView Pro Halt means stop executing the current command line. If an external client needs to know this too, the Halt button must be reported explicitly. If not, only when the Halt button actually is hit during a command line execution, cancellation is the case, and reported via an event. If the command line just finished, nothing is being done, so needs to be canceled, hence no cancellation is reported either.

An example would be a client interpreting breakpoint hits and issuing continue commands to resume execution. If the halt button should also stop the client from doing this, the `HaltButtonPressed` event must be used.

Running

Started executing the target.

RunningInBackground

Started target execution in background mode. This is usually a mode in which a restricted set of operations can be performed, for example read from a memory location.

Stopped cause

Stopped target execution. The cause is reported. Possible causes:

STEP A single step of any kind was finished. Be aware that when using single-step, the debugger does not report any breakpoints the program counter arrives at.

BREAKPOINT "*name*"

Breakpoint name was hit. This includes cycle breakpoints, time elapsed or number of instructions types. Breakpoints that the user has set are reported as well.

Nameless breakpoints are reported using as name *#number#*, where number is the CrossView Pro administration number. If no name or number is known, NAMELESS BREAKPOINT will be used.

ASSERTION *number*

Assertion *number* was hit.

UNKNOWN The process has stopped. The cause is unknown or cannot be described with one of the previous reasons.

One of the causes may be that the user pressed the Halt button.

More causes may be added in the future.

Reset

Hardware reset command has been executed by the debugger.

ResetProgram

Software reset of the program command has been executed by the debugger.

ViewedLineNrChanged *number*

The line being displayed changed to the specified one. If the source window is closed, or the cursor is not in a file but somewhere in assembly, this event will not be sent.

SourceFileChanged "*filename*"

The debugger displays an other source file. An empty file name "" will be sent if no source is being displayed at all.

DidLoadSymbols "*filename*"

The symbols of an application have been loaded.

DidAddSymbols "filename"

An application's symbols have been added to the ones already present.

DidDownloadImage "filename"

The code and data image of an application has been downloaded into target memory.

DestroyedAllSymbols "filename"

The symbol table of the application *filename* has been destroyed.

BreakpointsChanged

The list of breakpoints changed (for example when a breakpoint was added).

AssertionsChanged

Either the list of assertions or assertion mode changed (for example when an assertion was added). Note that the assertion numbering can be entirely altered when an assertion is removed.

MenuEntrySelected "id-string"

The menu entry *id-string* was selected by the user. Only menu entries created with the `AddDDEMenuEntry` or `AddCOMMenuEntry` command are reported.

CmdAnnotatedOutput<n>
annotated-output

Provides the command window output in an annotated form.

The first line indicates the error status and says OK, ERROR or NOT EXECUTED. The second line has the form `SEQ:sequence_number`, where the sequence number is either 0 or the number specified with the command. Although the sequence number is optional (it may be omitted in some commands) this line is always present. The next lines are either output or error messages. A label indicates the type (OUTPUT or ERROR) and the number of lines that follow.

Example

```

ERROR
SEQ: 9284
OUTPUT:1
Hello World
ERROR:1
No such name: xy

```

The reason behind this event is the inevitable merging of all data streams into one when TCP/IP server is provided next to for example the DDE server.

Quit

The debugger is about to terminate. This is not necessarily the last event, nor is it guaranteed that a `CommandInterpreterReady` event was send before. The **quit** event may not be send at all, due to technical restrictions.

1.5 COM EXAMPLES**1.5.1 PYTHON EXAMPLES**

To use COM objects for Python, you must first install the Python interpreter and the Win32COM extensions. You can use the Python interpreter distributed with the TASKING EDE. Or you can download the Python interpreter from <http://www.python.org> (May 2001) or use `win32all.exe` from <http://aspn.activestate.com/ASPN/Downloads/ActivePython/Extensions/Win32all> (May 2001).

Synchronous Calls

Replace all occurrences of `Xfw<targ>` in the example below by the name of your CrossView Pro executable to make the text applicable.

```

#
# Example without events callback
#

import win32com.client
# Python 1.4 requires "import ni" first.

```

```

class Xfw<targ>:
    "Xfw<targ> via COM wrapper class"
    def __init__(self, cmdline_options = ""):
        try:
            self.COMobject = win32com.client.Dispatch(
                "Xfw<targ>.CommandLine")
            self.COMobject.Init(cmdline_options)
        except Exception,e:
            print '(Is the Xfw<targ> COM object installed,
                using "xfw<targ>.exe -RegServer"?)'
            raise e

    def Execute(self, text, sequence_number = 0):
        result = self.COMobject.Execute(text, sequence_number)

        # convert Unicode to Python string
        retval = (result[0], str(result[1]))

        return retval

def test_xfw<targ>_com_object():
    xvw = Xfw<targ>(r"-sd c:\\testdir")

    (success, result) = xvw.Execute("echo Hello from Python")
    print "received", result

    (success, result) = xvw.Execute("l d")
    print "success=", success
    print result
    (success, result) = xvw.Execute("++$hoi")
    print result
    (success, result) = xvw.Execute("++$hoi")
    print result
    (success, result) = xvw.Execute("++$hoi")
    print result

    del xvw

if __name__ == "__main__":
    test_xfw<targ>_com_object()

```

Events Callback

```

#
# Example with Events callback
#

import win32com.client
# Python 1.4 requires "import ni" first.
import win32ui

import re

seen_ready_event = 0

class xvw_events:
    def OnCrossViewEvent(self, strUnicode):
        global seen_ready_event
        print "CrossViewEvent: " + str(strUnicode)
        if (re.match("CommandInterpreterReady.*", str(strUnicode))):
            seen_ready_event = 1

class Xfw<targ>:
    def __init__(self, cmdline_options = ""):
        self.COMobject = win32com.client.DispatchWithEvents(
            "Xfw<targ>.CommandLine", xvw_events)
        self.COMobject.Init(cmdline_options)

    def Execute(self, text, sequence_number = 0):
        result = self.COMobject.Execute(text, sequence_number)

        # convert Unicode to Python string
        retval = (result[0], str(result[1]))

        return retval

if __name__ == "__main__":
    xvw = Xfw<targ>("-sd testdir1")

    print xvw.Execute('"hello Python";$hoi++')

    while seen_ready_event == 0:
        win32ui.PumpWaitingMessages(0, -1)

    print "terminating"

    del xvw

```

Python Makepy Utility

In the examples above Python will load the type info dynamically from the COM object. This is called 'dynamic' binding or 'late' binding in PythonCOM jargon. However, PythonCOM also provides a mechanism to generate a Python module which contains this type info and thus speeds up the loading process. This is called early binding in the PythonCOM package.

Python uses the makepy utility to support early-bound automation. Makepy is a Python script that translates the COM type library to a Python module which wraps the COM object's interfaces. Once you use the makepy utility, early binding for the objects is automatically supported. There's no need to do anything special to take advantage of the early binding.

Advantages:

- Method invocation is faster.
- Constants defined in the type library are available via the COM interface module.
- It allows much better support for advanced parameter types. Especially parameters declared by COM as BYREF can only be used with makepy wrapped objects.

Disadvantages:

- The makepy wrapper script depends on the COM object to be wrapped by makepy. Generation can be automated.
- The module that is generated by makepy, can be large. The file generated for Microsoft Excel for example, is about 800 Kb.

To speed up starting a Python script that loads the CrossView Pro COM object, you can generate a Python module with `makepy.py`:

```
cd ...\\python20\\win32com\\client
python makepy.py ...\\xfw<targ>.exe
```

This script will place a module in the `win32com\\gen_py` subdirectory.

For more information on COM programming with Python refer to *Python Programming on Win32 – Help for Windows Programmers* (Mark Hammond & Andy Robinson; 1st Edition January 2000; 1-56592-621-8).

1.5.2 VISUAL BASIC EXAMPLES

Replace all occurrences of Xfw<targ> in the example below by the name of your CrossView Pro executable to make the text applicable.

Synchronous Calls

This example demonstrates plain commands being executed in CrossView Pro, without receiving any events from CrossView Pro.

```
Dim Xvw As Object
Dim Result As String

' here we invoke the PowerPC \xvw{}
' replace xfw<targ> by your executable name
Set Xvw = CreateObject("Xfw<targ>.CommandLine")
Call Xvw.init("")

Call Xvw.Execute("I", Result, 0)
MsgBox Result

End
```

Events Callback

Visual Basic provides a special feature, **WithEvents**, to connect to the connection point of a COM interface. It is also available in VBA 5.0. You must use **WithEvents** in a variable declaration. There is a catch, however: you can only use it in a class module (including form modules) and it must appear in the declaration section. You cannot declare a variable using **WithEvents** in the body of a procedure. For this example, first select Xfw<targ> type library in the Project References dialog:

1. In Microsoft Word or Microsoft Excel, start the Visual Basic editor and go to Tools|References or:

In Visual Basic, go to Project|References.



Note that VBA differs from VB. See the Word example for VBA.

2. Search and check the CrossView COM Interface Type Library entry.

```
Option Explicit

Public WithEvents Xvw As Xfw<targ>

Private Sub Form_Load()
    Dim Result As String

    Set Xvw = CreateObject("Xfw<targ>.CommandLine")
    Call Xvw.Init(" ")

    Call Xvw.Execute("echo Hello", Result, 0)
End Sub

Private Sub Xvw_CrossViewEvent(ByVal EventText As String)
    MsgBox "Called back with: " & EventText
End Sub
```

1.5.3 WORD EXAMPLES

Here is an example of connecting to CrossView Pro PowerPC. It starts `xfw<targ>` and shows all messages that CrossView Pro sends to Word. Visual Basic for Applications provides a special feature, **WithEvents**, to connect to the connection point of a COM interface. You must use **WithEvents** in a variable declaration. There is a catch, however: You can only use it in a class module (including form modules) and it must appear in the declaration section. You cannot declare a variable using **WithEvents** in the body of a procedure.

Replace all occurrences of `Xfw<targ>` in the example below by the name of your CrossView Pro executable to make the text applicable. To add the example to Word:

1. Start the Visual Basic editor and go to **Tools | References**
2. Search and check the CrossView COM Interface Type Library entry
3. Insert a class module, via the menu bar: **Insert | Class Module**
4. Change its name to `clsXfw<targ>` in the properties pane
5. Paste the following text:

```

'
' Class module clsXfw<targ>
'
'Option Explicit

' members
Public WithEvents oXfw<targ> As Xfw<targ>

Private Sub Class_Initialize()
End Sub

Private Sub oXfw<targ>_CrossViewEvent(ByVal strEvent
As String)
    MsgBox strEvent
End Sub

```

6. Insert a module, via the menu bar: Insert | Module

7. Paste the following text:

```

'
' Module testXfw<targ>
'
Option Explicit

Dim oXfw<targ>1 As New clsXfw<targ>

' run automatically when your Addin loads
' and your Addin will automatically load when Word
loads.
Public Sub AutoExec()
    Set oXfw<targ>1.oXfw<targ> = New Xfwppc
    call oXfw<targ>1.oXfw<targ>.Init("")
End Sub

```

1.5.4 EXCERPT OF THE MIDL DEFINITION

The 'ICommandLine' interface is dual, the 'ICommandLineEvents' connection point interface is not. Replace all occurrences of Xfw<targ> in the example below by the name of your CrossView Pro executable to make the text applicable.

```
interface ICommandLine
{
    HRESULT Init([in] BSTR CommandLine);

    HRESULT Execute([in] BSTR Command,
        [in] long SequenceNumber,
        [out] BSTR *Result,
        [out, retval] VARIANT_BOOL *Ok);

    HRESULT Halt(void);
    HRESULT ExecuteNoWait([in] BSTR Command,
        [in] long SequenceNumber,
        [out, retval] VARIANT_BOOL *Ok);
};

library CrossViewLibXfw<targ>
{
    dispinterface _ICommandLineEvents
    {
        methods: void CrossViewEvent([in] BSTR);
    };

    coclass Xfw<targ>
    {
        [default] interface ICommandLine;
        [default, source] dispinterface _ICommandLineEvents;
    };
};
```


2 DDE SERVER INTERFACE

2.1 INTRODUCTION

CrossView Pro offers an Interprocess Communications (IPC) option using the Microsoft Windows Dynamic Data Exchange (DDE) interface for external control of CrossView Pro. The DDE interface offers direct access to the CrossView Pro command interpreter. Via the DDE interface you can execute every CrossView Pro command that you can access via the regular CrossView Pro command window, and retrieve the output produced by the executed command.

2.2 DDE ITEMS AND TOPICS

DDE function calls always return, whether they succeed or fail. They do not report application command errors. Retrieve and interpret the **cmdoutput** item or **cmdannotatedoutput** item to check for errors.

Help

Topic

System

Item

Help

Operations

Request, Advise

Description

Returns a brief overview of the topics and items in ASCII text format.

cmdoutput

Topic

Command

Item

cmdoutput

Operations

Request, Advise

Description

Retrieves all command window output of the last executed command via the Command topic. This item empties itself after it has been requested.

cmdannotatedoutput

Topic

Command

Item

cmdannotated output

Operations

Request, Advise

Description

The first line indicates the error status and says OK, ERROR or NOT_EXECUTED. The second line has the form SEQ:*sequence_number*, where the sequence number is either 0 or the number specified with the **execext** command. Although the sequence number is optional (it may be omitted in some commands) this line is always present. The next lines are either output or error messages. A label indicates the type (OUTPUT or ERROR) and the number of lines that follow.

Example

```
ERROR
SEQ: 9284
OUTPUT:1
Hello World
ERROR:1
No such name: xy
```

execext

Topic

Command

Item

execext:options:string

Operations

Execute

Description

Passes the specified string without interpreting it to CrossView Pro's command interpreter (see also `Command\cmdannotatedoutput`). The **execext** prefix is part of the entire command string; it makes a distinction between the various commands. For example **exec**, **execext** or **halt**, received via the Command topic.

Options

wait=yesno *yesno* is 1 or 0. If you specify *wait=1* is, the **execext** command blocks the DDE transaction until CrossView Pro has finished executing the command. Issue the **Halt** command in this case via a second conversation.

Be aware of the time limitation imposed by the DDE interface. It can wait for a period of 25 days. Use **exec** combined with either waiting for an Advise on the **cmdoutput** item, or with interpreting the event item to handle very long lasting commands.

When you do not specify a value, 1 is assumed by default.

seq=number A unique number to identify a command's specific result in the stream of events output via the event item. See the **event** item and **cmdannotatedoutput** item for more details.

silent=yesno *yesno* is 1 or 0. If 1, the command window output will be suppressed. See section 2.5.5 *Using CrossView Pro as Pure Server* for the **gus** command.

When you do not specify a value, 1 is assumed by default.

Example

```
execext:seq=424564,wait:echo test
```

exec

Topic

Command

Item

exec

Operations

Execute

Description

Passes the specified string without interpreting it to CrossView Pro's command interpreter (see also Command\cmdoutput).

A major difference with regular MS-Windows applications is the immediate acknowledge of a command, before it has been completed. This is because the sender does not have to wait for the answer and can perform other tasks meanwhile. For example, you are able to issue a **halt** command to stop the debugger.

To simulate wait-till-completion command execution, wait until the cmdoutput item is assigned to the command's output via an Advisory link event, or interpret the event item.

The **exec:** prefix is part of the entire command string; it makes a distinction between the various commands. For example, **exec**, **execext** or **halt**, received via the Command topic.

halt

Topic

Command

Item

halt

Operations

Execute

Description

Forces CrossView Pro to stop target execution. You can issue the command via a second conversation.

event

Topic

Command

Item

event

Operations

Advise

Description

Reports event occurrences to the client, asynchronously. An event is reported by a string. To ensure capturing all events, use an Advise link. CrossView Pro only keeps the last event.

Request is not meant to be used; it can only be used after establishing an Advise link.

result

Topic

Command

Item

result:*name*

Operations

Execute

Description

The *name* that you specify provides a serve as DDE requestable item to obtain a message which describes the reason why a DDE command failed to execute. It does not return the CrossView Pro error message. It is always deleted after it has been requested.

The **result:** prefix is part of the entire command string; it makes a distinction between the various commands. For example, **exec**, **execext** or **halt**, received via the Command topic.

2.3 DDE EVENTS

2.3.1 PACKET FORMAT

Each event is wrapped in a record and one DDE message contains one or more of these records. This means that multiple events can be received simultaneously in one DDE transaction. This is done because DDE can lose ("combines") events when XTYPF ACKREQ mode is selected, and because this channel will be redirected to TCP/IP in the future for portable IPC support in CrossView Pro.

To handle events with more than one line, a header (not a newline) is used to distinguish between the individual events. The header format is:

EVENT: *number-of-characters*<newline>

So you must always split events that arrive in one DDE message. An example of such a multi-event DDE message is:


```

EVENT: 27
SourceFileChanged "demo.c"
EVENT: 23
ViewedLineNrChanged 93
EVENT: 27
Stopped BREAKPOINT "input"
EVENT: 24
CommandInterpreterReady
EVENT: 79
CmdAnnotatedOutput
OK\r
OUTPUT:1\r
Error breakpoint name 'input' is not unique!\r

```



For an overview of all available events, see section 1.4 *Events*

2.4 CROSSVIEW PRO DDE SPECIFIC OPTIONS AND COMMANDS

2.4.1 COMMAND LINE OPTIONS

--ddeservername=name

This command line option specifies a unique DDE server name. This way it is easier to distinguish between multiple instances of the same debugger.

If you do not use this option, the server name is the name of the CrossView executable. To distinguish between multiple DDE servers with the same name, you must connect to all DDE servers using **DdeConnectLists()** and obtain distinguishing information.

2.4.2 COMMANDS

With regard to DDE support, the following commands are available enhance integration support.

AddDDEMenuEntry

Syntax:

```
AddDDEMenuEntry "label", "id-string" [,AlwaysEnabled]
```

Creates a menu entry with given *label* and *id-string*. The label also specifies the path from the main menu bar, for example:

```
AddDDEMenuEntry "Options|CaseTool|Configure..." ,
                 "config-menu-entry"
```

An entry cannot be removed or replaced once it has been created. Neither is there support for enabling or disabling entries via this interface, this is somewhat problematic, since we are communicating across an asynchronous interface, so the disable may not be executed immediately.

AlwaysEnabled is either 1 (true) or 0 (false, default). CrossView Pro by default disables the menu entry when the command window disallows entering a command, for example when running an application.

To define the shortcut character of a menu entry, place a '&' before the character. The shortcut character will be underlined. To add a separator line in the menu, start the next menu entry with a '+'. The separator line will precede this menu entry. For example:

```
"&Options|&CaseTool|+&Reset"
```

2.5 EXAMPLES

2.5.1 EVALUATING AN EXPRESSION

To get the value of an expression, pass it to the command interpreter. The syntax of the returned value is:

identifier = value

The *value* can even be a complete structure or union. For example, execute via the Command topic:

```
execext:main
```

The returned string could look like:

```
main = 0x0
```

2.5.2 READING TARGET MEMORY

You can retrieve target memory either via requesting a variable's value, or with the **dump** command. The **dump** command can dump both byte (MAU) sized hex values or C type values, for example long or double. The resulting output must be interpreted to get the values.

The basic syntax of the returned values for plain MAU size hex dumps is:

address: value value ASCII-dump

The basic syntax of the returned values for formatted dumps is:

address = value value

For example, execute via the Command topic a hex dump command:

```
execext:dump main,#16
```

The returned text could be:

```
0x2000: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

For example, execute via the Command topic a formatted dump command, requesting 16bit integers:

```
execext:dump data1,#16,d2
```

The returned text could be:

```
0x2000 = 0 0 0 0 0 0 0 0
0x2010 = 0 0 0 0 0 0 0 0
```

The number of values per line differs. This depends on both the size and type of the values, as well as the architecture of the processor that is connected to the debugger.

2.5.3 WRITING INTO TARGET MEMORY

To write to target memory, use one of the following three methods.

1. Assign a value to a variable.
2. Use one of the the mF or mf commands.

For example, the following stores the byte (MAU) sized values 1 2 3 4 5 in memory starting at memory location 0x2000.

```
0x2000 mF 1, 2, 3, 4, 5
```

3. Write into memory using a type cast.

For example:

```
*(long*)0x2000 = 0x12345
```

2.5.4 REQUESTING CURRENT FILE AND LINE NUMBER

To determine the location of the source window cursor position, request the following special variables:

\$FILE The file in which the source window cursor position is. If the position is outside any file, the error message 'No current file' is returned.

\$PROCEDURE The name of the function in which the source window cursor position is. If the position is outside any function, the error message 'No current function' is returned.

\$LINE The line number of the cursor in the source window. If the position is outside any file, the error message 'No current line' is returned.

You can also use the command "**!s**" to get all special variables, including the ones above. If a variable is not set, it is not included in the list, or set with the error message as described above.

To make sure the cursor is at the current execution position, precede the **L** command before requesting the variable. For example, issue:

```
L; $FILE; $PROCEDURE; $LINE
```

Error messages appear when a variable fails.

To obtain the current execution positions, you can also interpret the result of the **L** command directly.

2.5.5 USING CROSSVIEW PRO AS PURE SERVER

To have CrossView Pro act as server only, updating windows can be turned off with the command **gus on**. This inhibits all windows from being updated, except for the command window. Note that also the GUI does not refresh anymore.

Also the **execext:silent=1:...** command via DDE inhibits the command window output.

APPENDIX

B

REGISTER MANAGER



B

APPENDIX

1 INTRODUCTION

CrossView Pro uses a so-called “register definition file” that specifies the register name to register number mapping for CrossView Pro. In case a register is a memory mapped register, the register number specifies the register’s memory address.

A number of register definition files are included in the CrossView Pro release. Register definition files use the following naming convention: `regcpu.dat`.

If you use another derivative you can use the register manager to create new register definition files.

2 INVOCATION

CrossView Pro has a user definable set of special function registers (SFRs). CrossView Pro reads the registers from a binary file (`regcpu.dat`) specified by the `cpu_type` field in `xvw.ini` or a target configuration file (`*.cfg`):

```
cpu_type: 68000
```

You can overrule the CPU type (and thus the `regcpu.dat` file) to be used by CrossView Pro with the **-C** option. The tool for generating this binary file from a text file is the Register Manager **rm68**.

The invocation syntax is:

```
rm68    [ register-file [,register-file]... ] [-o outfile]
rm68    -?
rm68    -V
```



When you use a **UNIX** shell (Bourne shell, C-shell), arguments containing special characters (such as `'()`' and `?`) must be enclosed with `" "` or escaped. The **-?** option in the C-shell becomes: `"-?"` or `-\\?`.

The optional *register-file* (or several files) contains the user’s definition of SFRs (Special Function Registers). The syntax of this text file is described in the next section. The **rm68** tool always generates a fixed set of registers in the binary output file. If no *register-file* is supplied, only this set is generated. A list of the fixed set appears in the section *Fixed Register Set*.

The **-o** *outfile* option lets you specify the name of the generated binary register file. If you omit this option, the default *outfile* name is `reg*.dat`.

The **-?** option causes **rm68** to print a tiny manual, and the **-V** option prints the version header only.

3 SYNTAX OF A REGISTER FILE

The syntax of the register file is quite simple. Comment starts with a semi-colon (;) and ends at the end of the line.

Register file syntax:

name size { [base+]address | REGISTER:id} attribute [; comment]

Field Values:

name A unique register name. If a name is multiply defined an error is issued. If a fixed name is redefined, a warning is issued and the redefinition is ignored.

size The SFR size in bits (8, 16 and 32).

base+ You can use this optional prefix for SFR addresses that are relative to a special base register address. Supported base addresses are:

IPSBAR_BASE	Base address specified by Internal Peripheral System Base Address Register (MCF5280, MCF5282)
MBAR_BASE	Base address specified by Module Base Address Register (ColdFire, MC68330, MC68340, MC68360)
MBAR2_BASE	Base address specified by Module Base Address Register (MCF5249)
SIMCR_BASE	Control register block address indicated by MM bit of SIM Configuration Register (MC68331, MC68332, MC68F375, MC68376)
BAR_BASE	Base address specified by Base Address Register (MC68302)

Setting the IPSBAR_BASE and SIMCR_BASE is explained in the following section. See your CPU Manual for more information on how to initialize the other base registers.

- address*

An absolute address or an offset relative to a base register. The last character identifies the format: **h**(ex), **b**(inary), **o**(ctal), default is decimal. Alternatively you can use the \$ prefix for hexadecimal numbers.
Hex format examples: 400h, \$27
- id*

A unique number 23 or higher. You can use the REGISTER:*id* construction to add an internal register. Register numbers 0-22 are in use by the fixed register set.
- attribute*

The following attributes are available:

R

read only

W

write only

RW

read and write

-

not accessible

Examples:

CCR	8	REGISTER:23	RW	; Condition Code Register
IMR	16	MBAR_BASE+\$0036	RW	; Interrupt Mask Register
MCR	16	\$ffffa00	RW	; Module Control Register

4 SFR BASE ADDRESS REGISTER SPECIAL VARIABLES

Some 68K/ColdFire targets use base registers to determine where SFR addresses are mapped in memory. In case of the base registers IPSBAR (MCF5280 and MCF5282) and SIMCR (MC68331, MC68332, MC68336, MC68F375 and MC68376), CrossView Pro cannot determine the contents of the base register because its base address depends on the contents of the base register itself. To know the contents of these base registers, you can pass these to CrossView Pro through special variables: \$IPSBAR_BASE and \$SIMCR_BASE.

For example, if the MM bit in the SIMCR base address register is set, then the SFR registers are located at memory locations 0xFFFF000–0xFFFFFFF. If the MM bit is not set (zero), the SFR registers are located at 0x7FF000–0x7FFFFFFF. The corresponding value for \$SIMCR_BASE will be:

If SIMCR[MM] = 0, \$SIMCR_BASE = 0x7FF000
If SIMCR[MM] = 1, \$SIMCR_BASE = 0xFFFF000

At startup, CrossView Pro use the reset values of the corresponding base address registers to initialize the special variables:

\$IPSBAR_BASE = 0x40000000
\$SIMCR_BASE = 0xFFFF000

You can specify an alternative value, via the command line:

xfw68 --IPSBAR_BASE=value
xfw68 --SIMCR_BASE=value



It is your responsibility to keep the contents of these special variables up-to-date. This means that when the IPSBAR register or SIMCR is changed, you have to change the value of the corresponding special variable.

For example, to change the IPSBAR register from the CrossView Pro command window, use the following commands (in that order):

\$IPSBAR = 0x80000001
\$IPSBAR_BASE = 0x80000000

5 FIXED REGISTER SET

rm68 defines the following registers, and you cannot overwrite them.

D0...D7, A0...A7, PC, SR, USP, FP, SP, CCNT, CBRK

Table B-1: Fixed Register Set

D0...D7, A0...A7, PC, SR, USP

The standard 68K registers. Target-specific core registers (i.e. ISP, DTT0) are specified in supplementary `reg*.def` files provided with the debugger package. See the section *Derivatives* for details.

FP, SP

Pseudo registers defining the current frame pointer and stack pointer, respectively.

CCNT, CBRK

Pseudo registers used by the simulator for profiling.

6 DERIVATIVES

The CrossView Pro package for the 68K/ColdFire contains a number of register register files (`reg*.def`) and the corresponding binary files (`reg*.dat`) in the `etc` directory. You can create your own version of an existing file or build a new one for a derivative which is not (yet) supported.

For instance, to create a binary version for the 68020, type:

```
rm68 reg68020.def -o reg68020.dat
```

The following rules are used to find the `reg*.dat` files:

1. Look if the files are present in the current directory
2. Use the `etc` subdirectory of your product tree.

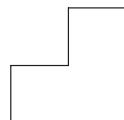
REGISTER MANAGER

APPENDIX C

SOUND SUPPORT (MS–Windows)



TASKING



C

APPENDIX

You can have sound effects being played when a predefined event in CrossView Pro occurs. You can configure the sound in the Sound settings of the Control Panel of MS-Windows. Similar to assigning a sound to a system event, you can assign a sound to a CrossView Pro event.

Currently the following events are supported:

- Breakpoint hit
- File has been downloaded
- CrossView Pro has started execution
- CrossView Pro is exiting
- Run command/button
- Step command/button
- StepOver command/button
- Halt command/button
- Symbols Loaded
- Fatal (system) error occurred
- Non-fatal error

How to add sound support

1. Firstly all events must be specified to MS-Windows. You can do this by adding the following lines to the Registry under:

```
My Computer\HKEY_CURRENT_USER\AppData\Local\Microsoft\Windows\CurrentVersion\EventLabels\
```

Use **regedit** to start the registry editor.

snd_xvw_bphit	"XVW Breakpoint Hit"
snd_xvw_download	"XVW Program Download"
snd_xvw_start	"XVW Start"
snd_xvw_exit	"XVW Exit"
snd_xvw_run	"XVW Run"
snd_xvw_step	"XVW Step Into"
snd_xvw_stepover	"XVW Step Over"
snd_xvw_stop	"XVW Stop"
snd_xvw_syms_load	"XVW Load Symbols"
snd_xvw_syserror	"XVW syserror"
snd_xvw_uerror	"XVW uerror"

2. You must also add the same list of keys (without values) to

```
My Computer\HKEY_CURRENT_USER\AppData\Local\Microsoft\Windows\CurrentVersion\EventLabels\
```

3. Now go and start the Sound settings in your Control Panel. Here you can assign a sound to each event. You can also assign None to an event, which prevents CrossView Pro from playing a sound if that specific event occurs.

4. For the sound effects to become operational, you also have to edit the `xvw.ini` file. You can do this using any editor, e.g. the Windows **notepad** command. Add the following line at an arbitrary line to your `xvw.ini` file:

```
sound_effects: TRUE
```

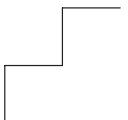
It is also possible to disable the sound effects by changing this line into:

```
sound_effects: FALSE
```

Now all sound effects are disabled.

ADDENDUM

SIMULATOR



ADDENDUM

1 INTRODUCTION

This addendum contains information specific to the simulator version of CrossView Pro for the 68K/ColdFire microprocessor family.

In general, the simulator for the 68K/ColdFire family attempts to duplicate the behavior of the common architecture of the microprocessor family. The simulator is a generic 68K/ColdFire family instruction set simulator, and this package does not support processor-specific features. However, there are several functional areas which deserve a brief discussion. The following sections describe simulator-specific features.

2 SUPPORTED FEATURES

Except for the restrictions mentioned in section 3 in this addendum, the simulator version of the debugger cleanly supports all the standard features of CrossView Pro, including single stepping, breakpoints, trace support, C expression evaluation and record/playback capability. With respect to setting breakpoints the simulator version of the debugger is capable of supporting all breakpoint types, including separate data-read and data-write breakpoints. The simulator also supports code and data coverage and profiling.

Because this is a simulator version, you do not have to setup communication at startup, as with an emulator.

The transparency mode is available to enter simulator commands.

2.1 MAPPING MEMORY

Simulator memory is defined in terms of address blocks. Each address block has a type (RAM, ROM, or IO_PORT), an associated address range, and a slot number. In the default simulator configuration, slot number 0 is type RAM and has an address range of 0 to 0x1FFFF (128K).

If your program won't fit in the first 128K of memory (covered by the default address block in slot 0), then you need to define more simulation memory before downloading your program. If you attempt to download into an address for which there is no simulation memory, an error message appears.

2.2 SIMULATING I/O VIA I/O PORT ADDRESS BLOCKS AND DEVICES

Simulation of a device is performed using the generic 'IO_PORT' address block and device.

IO_PORT is a generic set of “registers” representing a device whose behaviour is user-defined (a UART for example). A single device also called the I/O PORT, is associated with this address block type. The device contains a window which displays the current contents of the I/O Port registers, and which also allows the behavior of the simulated device to be controlled, either interactively or through the use of an I/O Port control script.

When an IO_PORT address block is created via the **S_ABA** simulator command, an I/O Port Device Window for the associated I/O Port device will appear. This window displays the current value of a portion of the register set, as well as two buttons labeled **I/O Control** and **Setup**.

For example, enter the following command in the Emulator Command Window:

```
s_aba 14 io_port ffa000
```

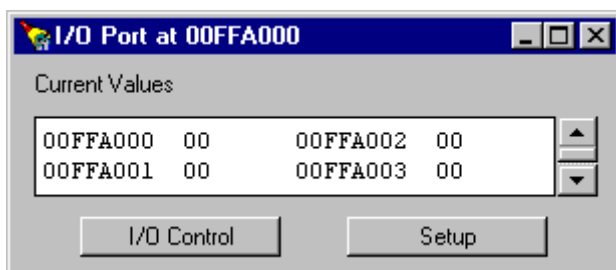


Figure Sim-1: I/O Port device window

The **Setup** button brings up the I/O Port Setup dialog box, allowing you to specify such attributes as the radix and size of displayed values, enabling and disabling of event logging, the name of the log file, and the control mode for this address block/device (Interactive or Script). See the *I/O Port Logging* section below for more information.

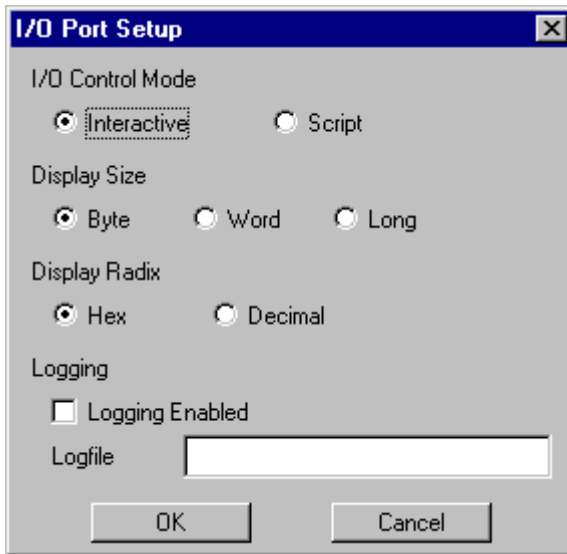


Figure Sim-2: I/O Port Setup dialog

The behavior of the device may be controlled either interactively or through the use of a control script. The **I/O Control** button in the device display window brings up a separate dialog through which the behavior of the device is controlled. The dialog that is displayed after you click this button depends on the **I/O Control Mode** field in the I/O Port Setup dialog. The choices are **Interactive** and **Script**.

If the control mode for the device is **Script** mode, selecting the **I/O Control** button will bring up the I/O Port Script Control dialog. This dialog allows you to load and execute a script file which controls the device's behavior. (See *I/O Port Script Control* for more details).

If the control mode for the device is **Interactive** mode, selecting the **I/O Control** button will bring up the I/O Port Interactive Control dialog. This dialog allows you to input a value into a device register and generate interrupts at any time, even during instruction simulation. (See *I/O Port Interactive Control* for more details).

I/O Port Script Control

If the **I/O Control Mode** field in the I/O Port Setup dialog is set to be **Script**, the I/O Port Script Control dialog will be displayed.

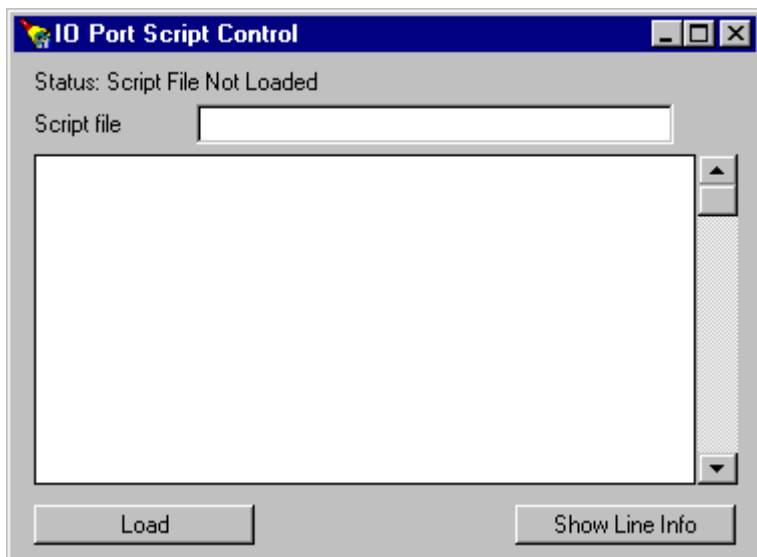


Figure Sim-3: I/O Port Script Control dialog

The script-based control window allows you to load and execute a script file which controls the device's behavior. Using the I/O Port Control Script language, the simulated device can be programmed to automatically respond to CPU accesses to certain registers, change the values of registers, delay for a specified amount of time, and generate interrupts. See *I/O Port Control Script Language* for additional information on how to generate a script. The script can be loaded, suspended, resumed, stopped, and restarted at any time, even while instruction simulation is in progress.

To load a script, click the **Load** button. Once a script is loaded, the **Start** button will start the execution of the script (simulation must be in progress for the script to run). Once a script is started, the **Suspend** button will suspend script execution until you click the **Resume** button. The **Stop** button will stop a script that has been started. You can restart the script with the **Restart** button. To unload a script, click the **Unload** button.

The script control window also provides a 'show line information' mode under which the current values of script variables and outstanding time delays are displayed. To enable the 'show line information' mode, click the **Show Line Info** button. To disable this mode, click the **No Line Info** button.

I/O Port Control Script Language

The following is a list of statements supported in the I/O Port Control Script language. See *Scripting Examples* for some scripting examples.



RVAL means an existing variable or a specified static value. V1 | V2 | ... means that one of the specified values MAY optionally appear. <V1 | V2 | ...> means that one of the specified values MUST appear.

IOVAR <Var name> <BYTE | WORD | LONG> <RVAL offset>

Declare an I/O Variable, which gives the name, size, and offset of one of the registers. Offset must be less than the size of the address block (e.g. use 0 for the first address in the block).

INTR <RVAL Pri>

Send an interrupt using the autovector for the specified priority.

INTR <RVAL Pri> <RVAL Vec>

Send an interrupt using the specified priority and vector.

DELAY <Rval count> USEC | MSEC | SEC

Delay for the specified number of microseconds, milliseconds, or seconds. If no unit is specified, microseconds are used. US, MICROSECONDS, and MICROSECS are aliases for USEC; MS, MILLISECONDS, and MILLISECS are aliases for MSEC; SECONDS is an alias for SEC.

WAIT FOR <READ | WRITE | ACCESS> I/O Var Name

Wait for a read, write, or any access to a register by the (simulated) CPU during instruction simulation. If no register is specified, a read/write/access to any register in the address block will satisfy the wait.

<Variable> = <RVAL>

Set the value of the script local variable or I/O variable to the specified RVAL. If the variable name is not recognized, a local variable with this name is created; this is the mechanism by which local variables are created.

<Variable> = <RVAL> <BINARY_OP> <RVAL>

Assign the result of the arithmetic operation to the script local variable or I/O variable, creating a new local variable if necessary.

BINARY_OPs supported are +, -, *, /, << (left shift), >> (right shift), % (modulo), & (binary AND), | (binary OR), and ^ (binary XOR).

Accepted aliases are MOD for %, AND for &, OR for |, and XOR for ^.

IF <RVAL> <COND_OP> <RVAL> THEN

....

ELSE IF <RVAL> <COND_OP> <RVAL> THEN

....

ELSE

....

END IF

Execute the first block of statements whose IF or ELSE IF condition is satisfied. If no condition is satisfied, execute the ELSE block (if present). Condition operators supported are =, !=, <, <=, >, and >=. Accepted aliases are == for =, and <> for !=.

LOOP DO END LOOP

Loop indefinitely, executing the block of statements. DONE is an alias for END.

LOOP <RVAL> TIMES DO END LOOP

Loop the specified number of times, executing the block of statements.

WHILE <RVAL> <COND_OP> <RVAL> DO END WHILE

Loop executing the block of statements while the specified condition is true.

CONTINUE

Go to the start of the nearest enclosing loop/while

BREAK

Go to the first statement after the end of the nearest enclosing loop/while.

EXIT

Terminate execution of the script.

RAND

Special 'variable' which yields a (pseudo-)random number whenever used as an RVAL. Cannot be assigned.

// or ; (semi-colon)

Comment strings. These and the following characters on a line are ignored.

General script notes: Identifiers/tokens must be separated by white space (spaces or tabs). Zero or one statement per line. In order to allow the language to resemble C a little more, the following are true: Parentheses are allowed but are simply matched up and ignored. Open bracket is an alias for the ignored DO and THEN reserved words. Close bracket is an alias for END. Close bracket followed by ELSE is an alias for ELSE. Reserved words (e.g. INTR, DELAY) are case insensitive. Variable names are case sensitive, and only the first 32 characters are significant. All script local variables, including RAND, are 32-bit unsigned integers. Atomic execution of control scripts is guaranteed. This means that while the simulator is executing script statements, no instructions are simulated. Instruction simulation resumes only when all active scripts are executing WAIT or DELAY statements. When an instruction is executed which satisfies an outstanding WAIT or DELAY, the simulator executes script statements without simulation of instructions, until the script again WAITs or DELAYs.

Scripting Examples

Some script examples follow:

```
; Example 1
; Loop 10 times, inputting random values and
; generating interrupts.
; Register format:
; Offset Size Register Name Description
; 0 LONG InputReg Input Value
IOVAR InputReg LONG 0 ; 32-bit register at offset 0
LOOP 10
    InputReg = RAND ;Write a random value to the register
    INTR 6 ;Generate an interrupt using priority
            ;6 autovector
    WAIT FOR READ ;Wait for the CPU to fetch the
                ;input value

END LOOP
```

```
// Example 2
// Generate interrupts until the CPU writes a STOP
// command to the command register.
// Register format:
// Offset Size Register Name Description
// 0 BYTE CmdReg Command Register
// Register declarations
IOVAR CmdReg BYTE 0;
// 'Constant Variables'
STOP_CMD = 1;
DELAY_NUSEC = 800;
INTR_PRI = 6;
INTR_VEC = 100;
// Loop until the STOP command is received, sending
// interrupts and waiting.
while (CmdReg != STOP_CMD)
    INTR INTR_PRI INTR_VEC;
    DELAY DELAY_NUSEC;
```

```

// Example 3
// Accept input X from the CPU, generate some f(X), and
// interrupt the CPU to signal completion of the
// function.
// Register format:
//   Offset  Size  Register Name  Description
//   0       LONG  InputValueReg  Input value X
// provided by CPU
//   4       LONG  CmdReg          CPU->device command
// -- if non-zero,
//
//           8     LONG  OutputValueReg  Output value f(X)
// returned to CPU
// Register declarations
IOVAR InputValueReg LONG 0
IOVAR CmdReg LONG 4
IOVAR OutputValueReg LONG 8
loop
    // Wait for the command register to become non-zero.
    while (CmdReg == 0)
        WAIT FOR WRITE CmdReg;

    // Retrieve input X, find f(X), send output and
    // generate interrupt.
    X = InputValueReg;
    fX = X + 100;                // Trivial f(X)
    OutputValueReg = fX;
    INTR 4

```

I/O Port Interactive Control

If the **I/O Control Mode** field in the I/O Port Setup dialog is set to be **Interactive**, the I/O Port Interactive Control dialog will be displayed.

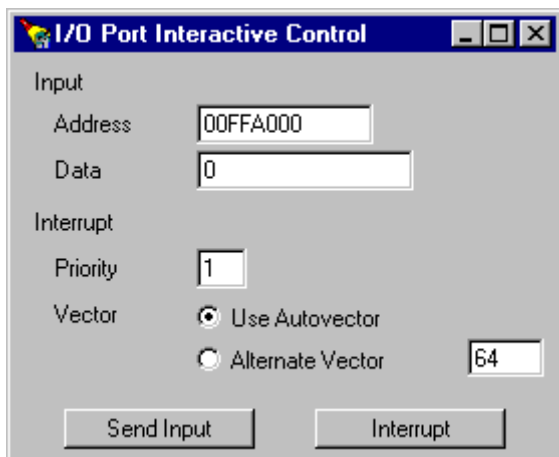


Figure Sim-4: I/O Port Interactive Control dialog

The interactive control window allows you to input a value into a device register and generate interrupts at any time, even during instruction simulation. To input a value into a device register, enter the desired register address into the **Address** field and the desired data value to be sent to this register into the **Data** field. Once this has been done, click the **Send Input** button to send the data to the specified register.

Interrupts can also be generated from this window. You can select the priority of the interrupt and specify if the interrupt uses the autovector for that priority interrupt or uses an alternate vector. If you want to use an alternate vector, you can define the vector in the space provided. Once you have defined the interrupt parameters, click the **Interrupt** button when you wish to send the interrupt.

I/O Port Logging

The "I/O Port Setup" window has fields that allow you to enable logging and specify a log file where this information will be stored. I/O Port Logging allows you to see the information sent from or to an I/O device. An example listing of the information stored in a typical I/O device log file is shown below:

```
DEV 02 < 00FFA000
DEV 02 < 00FFA000
DEV 000003E8 < 00FFA004
CPU 00 > 00FFA000
CPU 06 > 00FFA001
CPU 01 > 00FFA002
CPU 00 > 00FFA003
CPU 000003E8 > 00FFA004
CPU 02 > 00FFA000
DEV 01 < 00FFA002
DEV 06 < 00FFA001
DEV 02 < 00FFA000
DEV 02 < 00FFA000
DEV 000003E8 < 00FFA004
```

What does this information show you? Notice that each line either starts with "DEV" or "CPU". If a line starts with "DEV", it means that the I/O script for the device manipulated the data in the manner shown. If a line starts with "CPU", it means that your code (and thus, the CPU) manipulated the data in the manner shown. An example of each type of access is shown as follows:

```
DEV 02 < 00FFA000           ;Script read byte 02 from device
                             ;reg @ addr 00FFA000
DEV 06 > 00FFA001           ;Script wrote byte 06 to device
                             ;reg @ addr 00FFA001
DEV 000003E8 < 00FFA004     ;Script read long word 000003E8
                             ;from device reg @ addr 00FFA004
CPU 00 > 00FFA000           ;Code wrote byte 00 to device
                             ;reg @ addr 00FFA000
CPU 06 < 00FFA001           ;Code read byte 06 from device
                             ;reg @ addr 00FFA001
CPU 000003E8 > 00FFA004     ;Code wrote long word 000003E8 to
                             ;device reg @ addr 00FFA004
```



I/O logging can also be controlled via the **S_ABDEVSET** simulator command. See *Setting I/O Device Attributes* for more information.



2.3 SETTING I/O DEVICE ATTRIBUTES

Each I/O device has an associated list of attributes, which may be viewed and set. Use the simulator commands in the following table to view or set I/O device attributes:

Command	Purpose
S_ABDEVL	Lists all devices currently configured
S_ABDEVLV	Verbose listing of device attributes
S_ABDEVSET	Set device attributes

The following is an example of a command with sample output from the simulator.

To list the names of all devices currently configured:

```
s_abdevl

Slot   Dev   Device Name
14      0    I/O Port at 00FFA000
```

To list the attributes for a particular device:

```
s_abdevl 14 0

Device_Name='I/O Port at 00FFA000'
Display_Window_Enabled=TRUE
Display_Window_Position='(83,30,188,330)'
Display_Width=BYTE
Display_Radix=HEX
Control_Mode=INTERACTIVE
Control_Window_Enabled=FALSE
Interactive_Control_Window_Position=
    '(291,50,491,320)'
Script_Control_Window_Position='(239,42,489,407)'
Script_File=''
Logging_Enabled=FALSE
Logfile=''
```

To list the verbose attributes for a particular device:

```
s_abdevlv 14 0
```

```

Device_Name{STRING}='I/O Port at 00FFA000'
Display_Window_Enabled{BOOLEAN}=TRUE
Display_Window_Position{WINDOW_POSITION}=
    '(83,30,188,330)'
Display_Width{ENUM BYTE,WORD,LONG}=BYTE
Display_Radix{ENUM HEX,DECIMAL}=HEX
Control_Mode{ENUM INTERACTIVE,SCRIPT}=INTERACTIVE
Control_Window_Enabled{BOOLEAN}=FALSE
Interactive_Control_Window_Position
    {WINDOW_POSITION}='(291,50,491,320)'
Script_Control_Window_Position{WINDOW_POSITION}=
    '(239,42,489,407)'
Script_File{FILENAME}=''
Logging_Enabled{BOOLEAN}=FALSE
Logfile{FILENAME}=''

```

To set selected attributes for a device:

```

s_abdevset 14 0 device_name='Timer'logfile=
mdgs.log

```



Logging occurs only if the device has logging enabled and logfile is specified; the format of the logging information is address block type-specific. Logging can also be enabled from the "I/O Port Setup" window. See *I/O Port Logging* for additional information.

3 RESTRICTIONS

Facilities for background mode are absent in the simulator version of CrossView Pro. As a consequence, the CrossView Pro commands **CB**, **st**, **u**, **ubgw** and **wt** for background mode, are not available.

4 SIMULATOR COMMANDS

S_ABA (Add an Address Block)

This command adds an address block at the specified address. If no end address is specified, the default size for the specified type is used. The format of this command is:

S_ABA *addr_block_slot type start_address [end_address]*

where:

addr_block_slot is the slot where the block is to be located

type is the address block type (RAM, ROM, or IO_PORT)

start_address is the beginning address of the block

end_address is the ending address of the block

S_ABD (Delete an Address Block)

This command deletes an address block in specified slot. The format of this command is:

S_ABD *addr_block_slot*

where *addr_block_slot* is the slot to be deleted.

S_ABDEVL (List All Devices Currently Configured)

This command, when entered with no options, lists all devices that are currently configured. If a certain device is specified, all attributes for the device are shown. If you wish to list all devices that are currently configured, the format of this command is:

S_ABDEVL

If you wish to view the attributes for a certain device, the format of this command is:

S_ABDEVL *addr_block_slot device_number*

where:

addr_block_slot is the slot where the device is located

device_number is the number of the device

S_ABDEVLV (Verbose Listing Of Device Attributes)

This command provides a verbose listing of the attributes for the specified device. This verbose listing gives a description of the type along with valid values for each attribute. The format of this command is:

S_ABDEVLV *addr_block_slot device_number*

where:

addr_block_slot is the slot where the device is located

device_number is the number of the device

S_ABDEVSET (Set Device Attributes)

This command allows you to set the value of one or more attributes for the specified device. The format of this command is:

S_ABDEVSET *addr_block_slot device_number attribute_name=value*

where:

addr_block_slot is the slot where the device is located

device_number is the number of the device

attribute_name is the name of the attribute to be changed

value is the desired value for the attribute

S_ABL (Display Address Block Attributes)

This command displays the type, size, start, and end addresses for the specified address block, or for all active slots if none is specified. To display the above information for all active slots, enter the following command:

S_ABL

To display the above information for a specific slot, enter the following command:

S_ABL *addr_block_slot*

where:

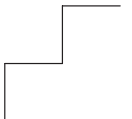
addr_block_slot is the slot where the device is located.



SIMULATOR

ADDENDUM

SmartMON ROM MONITOR



ADDENDUM

1 INTRODUCTION

This chapter introduces SmartMON and some of its features. It includes the following major sections:

- Overview
- SmartMON's Debugging Features
- SmartMON Distribution Contents

1.1 OVERVIEW

Welcome to SmartMON. SmartMON is a software-only, real-time debugger that resides on your 68xxx target system. Once activated, you have access to a powerful set of commands that let you control and monitor your application software directly through CrossView Pro source-level debugger. Figure Rom-1 shows the typical SmartMON debugging configuration.

You can use a PROM programmer to burn SmartMON into ROM or use a Flash programmer to program a FLASH device.

- SmartMON is a ROM monitor, and ROM monitors have existed since the first 4-bit microprocessor. However, SmartMON gives you several important advantages over older monitor technology:
- SmartMON has a far more extensive command language. In addition to standard monitor features like software breakpoints, read/write memory, read/write registers, and start/break execution, SmartMON also allows tracing (instruction and data), data breakpoints, conditional breakpoints, breakpoints on ROM code, and block memory operations.
- SmartMON incorporates features that make it a valuable tool for field testing and manufacturing QA. Its extensive custom diagnostics, in addition to the SmartMON system call facility, allow SmartMON to be used throughout the life cycle of your embedded application. For example, SmartMON can be used by field-test engineers to trouble-shoot systems via a terminal interface.
- Unlike most monitors, SmartMON has a built-in interface to the CrossView Pro C source-level debugger. With CrossView Pro and SmartMON, you get a powerful, real-time C source level debugging solution that delivers increased efficiency for C programmers.

- Finally, SmartMON has been ported to many off-the-shelf single board computers (SBC). The driver packages for some of these SBC's are all included in this release. Batch files have been supplied to build SmartMON for many popular VME boards or 683xx or ColdFire evaluation boards (sold by Motorola).

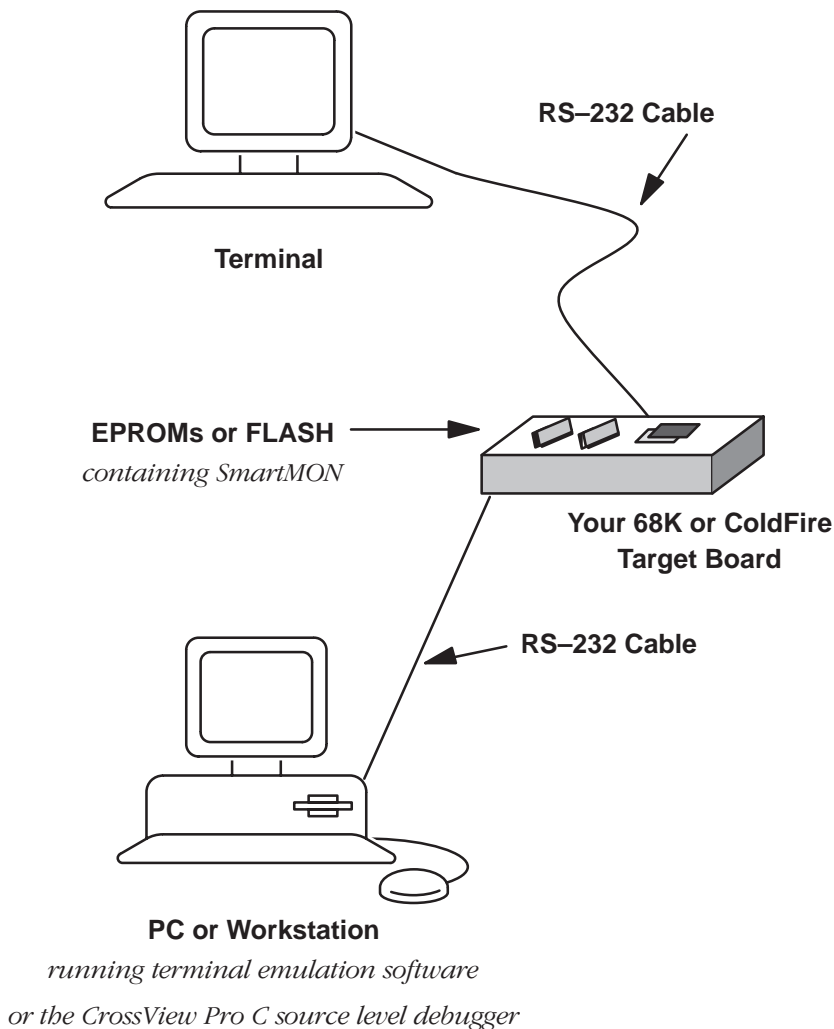


Figure Rom-1: SmartMon debugging configuration

1.2 SMARTMON'S DEBUGGING FEATURES

The table below shows a list of SmartMON's commands that can be used to debug your 68xxx application. A description of the major types of commands follows.

Command	Function	Command	Function
CF	configure	GO	go
BD	disable break point	HE	online help
BE	enable break point	IN	initialize sequence
BF	block fill	RB	remove break point
BM	block move	SB	set break point
DB	display break point	SI	single step
DC	display configuration	SM	set memory
DF	diagnostic functions	SO	step out of range
DI	disassemble	SR	set register
DL	download S record	SS	search for string
DM	display memory	TE	enable trace
DR	display registers	TD	disable trace
DT	display trace	UD	user diagnostics

Table Rom-1: ROM monitor commands

1.2.1 INITIALIZE AND DOWNLOAD

SmartMON includes a special initialize command that re-boots the software system and executes the RMAIN initialization code. RMAIN is part of the Target Interface Package (TIP) which must be modified for your hardware environment.

Through RMAIN, you have the choice of entering the debugger upon initialization and power-up, or booting your application software directly. If you boot your software upon initialization, SmartMON can remain dormant in your product and invoked at anytime with a control-C character through the serial port.

1.2.2 STEPPING, EXECUTING, AND HALTING

SmartMON provide a full set of commands that let you single step, multi-step, or run your application in real-time. You can optionally request to see disassembly or register display with every step command.

The debugger supports both interrupt-driven and polled I/O serial communications. If you use an interrupt-driven I/O driver, you can halt your software application by typing a control C at the host or terminal. If you are using a polled I/O driver, you may want to include a hardware means to break execution, for example, an abort button.

1.2.3 SETTING BREAKPOINTS

SmartMON offers numerous breakpoint commands that result in a powerful and flexible means to control your application program. The debugger implements real-time breakpoints by substituting trap instructions at specified RAM addresses. You can append conditions to these real-time breakpoints to form complex breakpoints. SmartMON also cleanly handles breakpoints in interrupt service routines.

Through a unique assertion mode, you can instruct SmartMON to break on ROM based code. The debugger builds a table containing ROM based breakpoints, and then after a **GO** command is issued, it steps the processor and compares the next instruction's address with the addresses in the breakpoint table.

Also through this assertion mode, SmartMON implements data breakpoints. It builds a table of memory or register values that you wish to halt the processor. While the assertion mode does not execute in real-time, it can offer an invaluable tool in finding complex system bugs.

1.2.4 FULL DISASSEMBLER

SmartMON includes a full-featured disassembler that gives you an intuitive display of target memory, as well as the trace buffer. (Extended version only.)

1.2.5 DISPLAYING AND SETTING MEMORY AND REGISTERS

SmartMON gives you full control over your target's memory and registers. With the display memory command, the debugger shows you the hex values and the ASCII equivalents.

1.2.6 TRACING

SmartMON includes a unique trace capability allowing a history of program execution to be stored in the target's local memory (trace buffer). This gives the user the ability to playback program execution. The program playback includes not only the instruction executed, but also the data movements associated with the actual operation. (Data movement is available only in the extended version of SmartMON.)

For example, let's say the contents of one register, the source, is to be transferred or moved to another register, the destination. The playback will show the values contained in the source register, the destination register prior to the move, and the destination register after the move. When tracing is enabled, user code does not execute in real time, but debugging with tracing typically provides a higher confidence level when you begin debugging your code in real-time. Tracing may be selectively enabled or disabled.

1.2.7 DIAGNOSTIC CAPABILITIES

SmartMON provides memory tests and scope loops for testing specific areas of the target. Also embedded in SmartMON is a diagnostic executive which can control the execution of user-written diagnostics. These diagnostics, may be burned into ROM with SmartMON or down-loaded into RAM and executed. SmartMON creates a menu of available tests, which may be executed in batch mode or one at a time. (Extended version only.)

1.2.8 SYSTEM CALLS

SmartMON also supplies a mechanism that lets you access SmartMON's features and services from your application code. By embedding system calls in your diagnostic routines, you can create a suite of interactive tests that can be run during field testing, manufacturing QA, or when isolating failures in the repair department.

SmartMON handles all character transmission and reception, including flow control and line buffering. These services allow the diagnostic engineer to make his tests interactive with the user. The system calls may also be used by the system software designer as a means of reporting errors to a local terminal or modem.

1.3 SMARTMON DISTRIBUTION CONTENTS

SmartMON supports a variety of Motorola 68K/ColdFire target microprocessors.

SmartMON is normally delivered with the CrossView Pro Source Level Debugger. The SmartMON distribution contains the following subdirectories:

- **lnfiles** – Two object files are supplied: `smon68kb.ln` and `smon68ke.ln`. These are the core of the basic and extended versions of SmartMON, respectively.
- **boards** – There are several subdirectories under `boards`, one for each supported target board. Everything is included to build the ROM monitor for the target board. Alternatively, you may use one of the supported targets as the basis for customizing the ROM monitor to another board.
- **drivers** – Sample drivers for controlling UART chips reside in this subdirectory.

2 USING SMARTMON

This chapter describes the operation of SmartMON on your embedded microprocessor board. It should be read and understood before you proceed to the next chapter. Understanding SmartMON's basic operation will allow you to more easily follow the TIP (Target Interface Package) examples and make the necessary modifications for your board. This chapter includes the following major sections:

- Overview
- SmartMON's Resource Requirements
- SmartMON's Use of Interrupts and Traps
- The Three Operational Modes of SmartMON
- How SmartMON Sets Breakpoints
- SmartMON's Tracing Features
- Single Stepping and Step-out-of-range
- The Six Different Submodes of Execution Mode
- How SmartMON Processes I/O
- How SmartMON is Initialized
- Run-time Notes

2.1 OVERVIEW

SmartMON is a command line driven, software debugger that resides on your 68xxx target board. It controls and monitors your embedded software application in response to user commands that are sent to it over the serial communications line.

SmartMON allows you to run your application in real-time, without any performance penalty placed on your target system. To achieve this real-time performance, you should execute your code (issue a **GO** command) with only code breakpoints set, and no ROM based code breakpoints set.

SmartMON also gives you advanced features for tracking-down complex bugs. Some of these features, such as data breakpoints, breakpoints on ROM code, and tracing, require that SmartMON be activated during the running of your application code, which means that your code does not run in real-time. However, you will still find that these features afford you quite a bit of flexibility and power in isolating target problems.

2.2 SMARTMON'S RESOURCE REQUIREMENTS

SmartMON requires the following resources from your target system:

Code Space

54 Kbytes for the extended version, 20 Kbytes for the basic version. SmartMON is typically burned into ROM, but it can also be downloaded and booted from RAM.

Data Space

6–21 Kbytes of RAM (This includes SmartMON's internal data structures and a 1 Kbyte trace buffer. The trace buffer can be expanded in 1K increments up to 16K).

Traps

The 68xxx trace vector and 2 user-defined traps, one used for breakpoints, and the other used for entering SmartMON. For interrupt-driven I/O, a serial port interrupt is also required.

2.3 SMARTMON'S USE OF INTERRUPTS AND TRAPS

Most microprocessors support interrupts. An interrupt changes the normal flow of program execution and passes control to an interrupt service routine (ISR). When the ISR has completed, control returns to the next instruction in the normal program flow.

The 68000 has 3 interrupt lines, allowing 7 levels of interrupts to occur. For instance, if an interrupt is generated by a serial device (indicating a character has been received or transmitted), the processor passes control to an ISR routine to handle that interrupt. This is achieved by vectoring to an address specified in the Exception Vector Table (EVT). The address of the ISR is loaded into the vector table during SmartMON initialization. The addresses of the breakpoint and system traps are taken from information provided by the configuration table, found in the user equates file, `USREQU.68k`. The I/O ISR address is taken from `portinit` in `io_drv.68k`.

A TRAP works in much the same way as an interrupt. Unlike interrupts, however, traps are actual machine instruction that cause exceptions. High level or compiled code usually does not contain traps. SmartMON makes use of this by temporarily placing them into user code for breakpoints.

When the trap is encountered, execution is passed to a SmartMON exception-handling routine. Traps, therefore, provide a means to halt program flow deliberately and at specific points in user code. Once the program is halted, SmartMON can ascertain information about the target.

For example, when power is first turned on, control jumps to the routine pointed to by the restart vector. When a breakpoint trap occurs in user code, control is passed to SmartMON through the breakpoint trap vector. Similarly, when a serial port interrupt occurs, the address in the vector table points to the serial port interrupt service routine.

Finally, the SmartMON system uses a trap that is referred to as the “pointer to SmartMON.” This is the trap that is used to enter the debugger for most circumstances (other ways to enter or activate the debugger are via the breakpoint trap or a call to the initialization routine, *RMINIT*).

Restart Vectors to Initialization Code

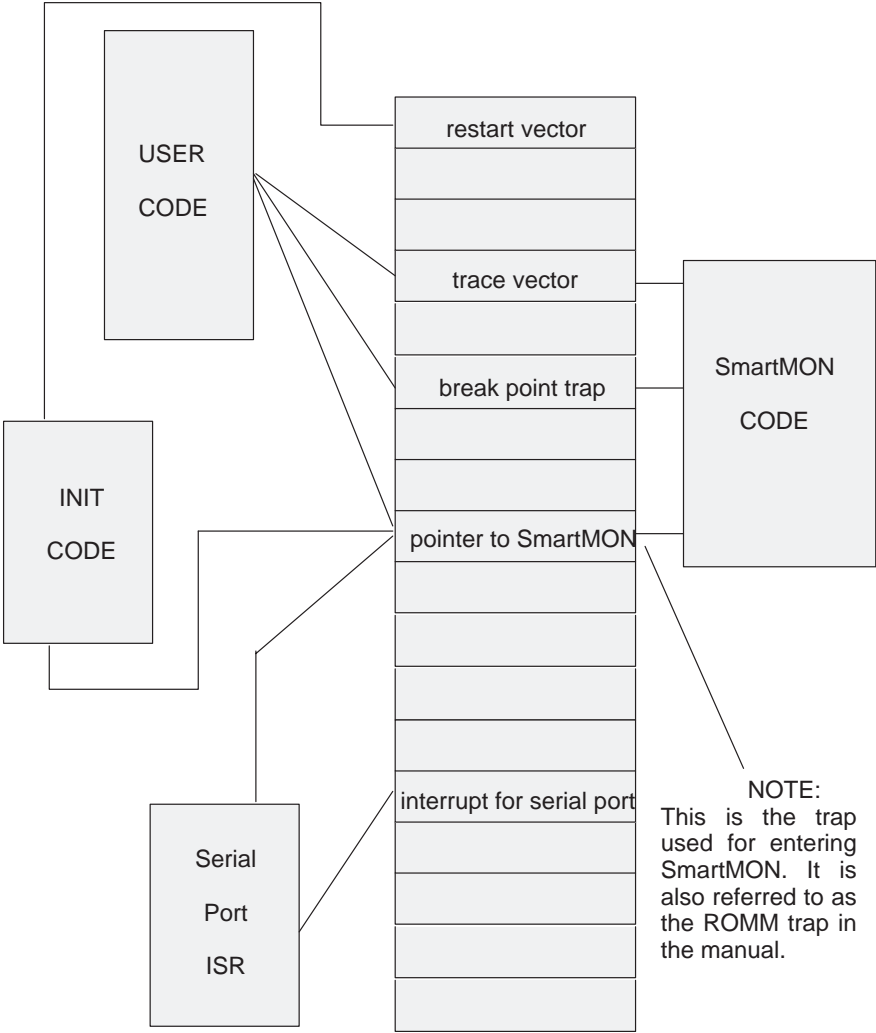


Figure Rom-2: Traps and Interrupts Used by SmartMon

Typically before this ROMM trap is used, a code is loaded into D0 that tells SmartMON the reason why it has been entered. For example, D0=5 is a ROMM_GO code, which tells SmartMON that it is being invoked after system initialization or directly from the application code.

The general term for using this SmartMON enter trap is “system calls.” It is very important to note that system calls have two distinct functions:

Required System Calls

There are two instances when you must use system calls. One is in the `RMAIN.68K` routine after the system has been initialized and you are ready to enter the debugger (although one option is not to boot the debugger at all, but rather directly boot your application code and have SmartMON lie dormant). This is the `ROMM_GO` system call, `D0=5`. The other instance of required system calls is in the serial port driver routines to re-enter SmartMON after the ISR has executed (or after your polled I/O routine, if you are not using interrupt driven I/O). Here, the system call is not an option, but rather the predetermined scheme for SmartMON to process characters. The next chapter, Creating the TIP, explains more about using these required system calls in the initialization module (`RMAIN.68K`) and the I/O driver (`IO_DRV.68K`).

Optional System Calls

SmartMON gives you the ability to use system calls directly from your application code. The best use of these optional system calls is for character I/O routines, where you want your code to output a test message by sharing SmartMON’s resources and serial driver routine.



Both the required and optional system calls are described further in the *System Calls* chapter.

2.4 THE THREE OPERATIONAL MODES OF SMARTMON

At any one instance, SmartMON is in one of these three modes:

Command Mode

When the monitor is continuously active and running on the target microprocessor, it is in command mode. In this mode, SmartMON may accept commands from a host or terminal to perform such tasks as examining the stack, registers, and memory, or to resume target program execution.

Execution Mode

Whenever the user application code is running, the monitor is in execution mode. Since it is not active, it cannot accept commands from the terminal or from CrossView Pro. There are five ways to force SmartMON to become active, thereby leaving execution mode and entering command mode.

- Encounter a previously set breakpoint.
- Type a control-C from CrossView Pro or a control-X from a dumb terminal (or from CrossView Pro's transparency mode).
- Use the ROMM_GO system call (D0=5) from the application code.
- Turn tracing on. Through the trace vector, SmartMON will become active after each instruction is executed.
- Encounter an interrupt or trap that was not assigned by the user.

Download Mode

Whenever you issue a download command from command mode, SmartMON goes into a special download mode in which all subsequent data is loaded into the target system's RAM. The download information is in Motorola standard S-record format. This download mode is terminated when an EOF (end-of-file) S-record is received. Download can also be terminated by typing a control-C from CrossView Pro or a control-X from a dumb terminal (or from CrossView Pro's transparency mode).

2.5 HOW SMARTMON SETS BREAKPOINTS

Breakpoints are the primary mechanism used to control the execution of your program. While SmartMON is a software-only debugger that does not have hardware to monitor the bus (which means that hardware breakpoints cannot be accomplished), it does offer several flexible breakpoint schemes. These are:

- Real-time simple instruction breakpoints on RAM code.
- Real-time instruction breakpoints on RAM code, with comparisons and actions to be automatically executed after the breakpoint is hit (complex breakpoints).
- Non-real-time instruction breakpoints on ROM code, either simple or complex.
- Non-real-time data breakpoints.

- Any of the above breakpoints with tracing enabled; the first two breakpoint classes become non-real-time when the tracing feature is enabled

The sections below describe how SmartMON sets the breakpoints listed above.

2.5.1 SETTING BREAKPOINTS ON RAM CODE WITHOUT TRACE MODE ACTIVE

If the user wishes to set a breakpoint in his code, he must use the SB (set breakpoint command), which will instruct SmartMON to install a breakpoint at a specific location in memory. When this command is received, SmartMON goes to that specific location in memory and removes the instruction stored at that location and installs a TRAP instruction in its place. This means that when the user's program encounters that location, a TRAP will occur which will activate SmartMON. Once activated, SmartMON stores away the register contents and removes the TRAP instruction replacing it with the original code. This allows real-time execution of user code during debugging.

2.5.2 INSTRUCTION BREAKPOINTS ON ROM CODE

As previously mentioned, breakpoints are achieved by installing trap instructions in the user code. However, this can only be done when the user code is in Read/Write memory. In order to achieve breakpoints in read only memory, a breakpoint table is built, and the user program is run with trace enabled. After the execution of each instruction, SmartMON examines the table to look for a match. If the current address equals one of the addresses in the table, a break will occur.

2.5.3 DATA BREAKPOINTS

Using the breakpoint table also enables SmartMON to break on data. The user can define a conditional break when data at a particular location equals a specified value. This is achieved in trace mode. After each instruction execution, the address specified is checked for the required value. If the value is equal to the conditions set forth then a break occurs.

2.5.4 COMPLEX BREAKPOINTS

Complex breakpoints allow many options in debugging. It allows you to, after reaching an address, check either a register or memory location for specified values before taking the breakpoint. This is very useful for debugging modular code which has global variable access. You may want to use this feature for single stepping a certain routine after a specific variable changes. This would be accomplished by setting a complex breakpoint in the function, after the specific variable reaches a certain value. Then, you may begin single stepping.

2.6 SMARTMON'S TRACING FEATURES

One of the unique features of SmartMON is its Tracing capabilities. SmartMON has the ability to replay code execution, which is achieved by running your code with trace enabled. This tracing may be set for Program Counter (PC) only or full trace with data movements. Once enabled, information is stored in a trace buffer (an area in target RAM reserved for SmartMON) after every instruction is executed. When execution is halted and command mode is entered, the trace buffer may be displayed, either forward or backward, showing the disassembled code that has executed. This feature is further enhanced when full trace is enabled. Full trace includes the data movements associated with each executed opcode in the trace information.

Tracing would typically be used when you are trying to find a problem in a subroutine. Let's say you are single stepping through code with full trace enabled, when you encounter the subroutine in question. You may issue the GON (go next instruction) command, which allows the subroutine to execute without having to single step through each instruction. When the routine returns, SmartMON indicates the call is complete. At this point, simply play back the trace buffer to not only see if the bug is in the subroutine, but also to find the source of the problem. This technique allows you to quickly find the offending module. Alternatively, simply setting a breakpoint and running with trace enabled will allow you to play back information prior to that breakpoint.

2.6.1 TRACE POINTS

One issue that causes problems is using tracing to debug time-critical code. SmartMON supplies a trace point feature which uses conditional breakpoints to allow trace to be enabled or disabled. For example, if an ISR is time critical and you do not wish to include a trace history on this event, setting a Trace disable at the entry to the ISR and a trace enable at the exit allows the routine to run in real-time.

```
>SB ea0000 > td;when ea0000 is encountered stop trace
>SB ea000f > te;when ea000f is encountered start trace
```

This feature may be used in another way. Lets say that you have a common time out loop and you would like to know what code executed prior to entering this loop. You could accomplish this by setting a conditional breakpoint and disabling trace at the beginning of this loop. You may stop the code by setting a breakpoint after the time out loop routine or by typing ^C if you know the code is stuck looping. The trace buffer will now contain the code that was executed prior to this loop.

2.6.2 TRACE BUFFER OPERATION

The trace buffer is a FIFO architecture and stores the previously executed instruction. The trace buffer is configurable from 1k to 16k bytes in size. This allows users with limited memory space to tailor the buffer size to their environment. When tracing with data movements is enabled, approximately 50 instructions are stored per kilobyte. When tracing PC only is active, 128 instructions per kilobyte is possible. This gives you 2048 of the previous instructions executed when using a 16k trace buffer using PC only trace and 800 instruction with full data movements.

2.7 SINGLE STEPPING AND STEP-OUT-OF-RANGE

These two SmartMON features are accomplished by the debugger's general tracing capability. Single stepping allows the user to step through his code for debugging. This is done by an **SI** command, which instructs SmartMON to execute a single instruction. To accomplish this, SmartMON sets the trace bit in the processor status register. which causes an exception after each instruction is executed. This trace exception causes SmartMON to be re-activated, and only one instruction in user code is executed.

The Step-out-of-range function allows the user to set a range of addresses within which to execute. If the code steps outside this range, then the program will halt and SmartMON is entered. This function also operates by setting the processor's trace bit and forcing a trace exception after each instruction is executed.

2.8 THE SIX DIFFERENT SUBMODES OF EXECUTION MODE

Within execution mode, SmartMON has six different way of executing your code. These submodes depend on what types of breakpoint are set, and whether or not tracing has been set. The execution submodes are:

Running Real-time

User code is running real-time and SmartMON has no impact on system performance.

Running Real-time with Breakpoints

User code is running real-time and breakpoints are installed; when a breakpoint is encountered, user program execution will stop and SmartMON becomes active.

Tracing (PC only)

User code is running with tracing enabled; after executing each line of user code, SmartMON is activated and stores the PC value in the trace buffer then continues to allow the next line of user code to execute. User code now executes approximately 100 times slower than normal.

Tracing (PC only) with Assertions

User code is running with tracing enabled; in addition to storing the PC value, other conditions are being tested such as data breakpoints, step out of range, etc.

Tracing (PC and Data Movements)

User code is running and tracing is enabled; in addition to storing the PC value, the instruction is disassembled and the data movements associated with the operation are stored in the trace buffer. This will cause the user code to execute at approximately 800 times slower than normal.

Tracing (PC and Data Movements) with Assertions

User code is running with full tracing and other conditions are being tested.

2.9 HOW SMARTMON PROCESSES I/O

SmartMON allows two different types of I/O schemes, interrupt driven I/O and polled I/O. Interrupt driven I/O is recommended because it provides the best performance and because it allows you to break the execution of the application code by typing a control-C at the host or terminal . However, sometimes the polled I/O scheme must be used because of the device type, or a lack of available interrupts.

With either interrupt driven or polled I/O, SmartMON requires a driver routine called `IO_DRV.68K`. This routine, along with five other routines, make up the TIP, Target Interface Package. The next chapter tells you how to modify or supply a TIP for your board. Because I/O drivers, both interrupt-driven and polled, are supplied for the most widely used USARTS, chances are that you will only have to choose the proper driver from the supplied library and make minor modifications.

The next several pages give you an overview of how SmartMON processes serial I/O.

2.9.1 INTERRUPT DRIVEN I/O

Whenever either the target (SmartMON) or the host wants to send a character, it places it in the serial device, called the USART (universal synchronous/asynchronous receiver/transmitter). The serial I/O device generates an interrupt, which invokes an interrupt service routine. For most serial devices, there are separate ISRs for transmit and receive; however, sometimes only one ISR handles both cases.

A flowchart of how SmartMON and the serial driver process characters for interrupt driven I/O is shown in Figure Rom-3. For example, suppose the Host sends a character to SmartMON. This character is received by the serial port device, which generates an interrupt to the processor indicating that a character is available in its input buffer. This interrupt signal causes the processor to pass control to an ISR (the address of which is stored in the appropriate place in the EVT). The receive ISR then takes the character from the serial port buffer and places the character in the D1 register. Then, an `INT_RX` code is loaded into D0 and the ISR traps to SmartMON. SmartMON recognizes the `INT_RX` code in D0, so it knows to process the character that was passed to it in D1.

When SmartMON wishes to send a character to the Host, it follows a slightly more complicated procedure. SmartMON must first indicate to the USART that a message is about to be sent. To start this process, SmartMON calls a routine called `TX_CHAR`, with the first character of the string to be transmitted in register D1. `TX_CHAR` takes the character from D1 and places it in the serial port output buffer. The remaining characters are transmitted by the transmit ISR. This is possible because after the first character has been transmitted the interrupt generated indicates that the serial device is ready for the next character. The transmit ISR, when first entered, places an `INT_TX` code in D0 and traps to SmartMON. SmartMON returns to the ISR with the next character for transmission in D1. This process continues until SmartMON has no more characters to send, in which case it places a `NO_CHAR` in D0.

SmartMON's steps to transmit characters is summarized below:

1. SmartMON has a message for the host.
2. In order to prepare the USART to transmit characters, SmartMON calls `TX_CHAR` with the first character in the message.
3. `TX_CHAR` writes the first character of the message to the serial port, enables serial port interrupts and returns to SmartMON.

4. When the USART is ready for the next character, an interrupt occurs which invokes the transmit ISR.
5. The transmit ISR places an `INT_TX` code in `D0` and traps to SmartMON.
6. SmartMON recognizes the `INT_TX` code in `D0`, places the next character in `D1` and returns to the ISR.
7. The transmit ISR writes the character in `D1` to the serial port.
8. Upon completion the ISR traps to SmartMON with `INT_COMP`. SmartMON does not return to the ISR after this system call.
9. Steps 4-8 are repeated for each character of the message. When SmartMON has no more characters to send, it places a `NO_CHAR` value in `D0`.

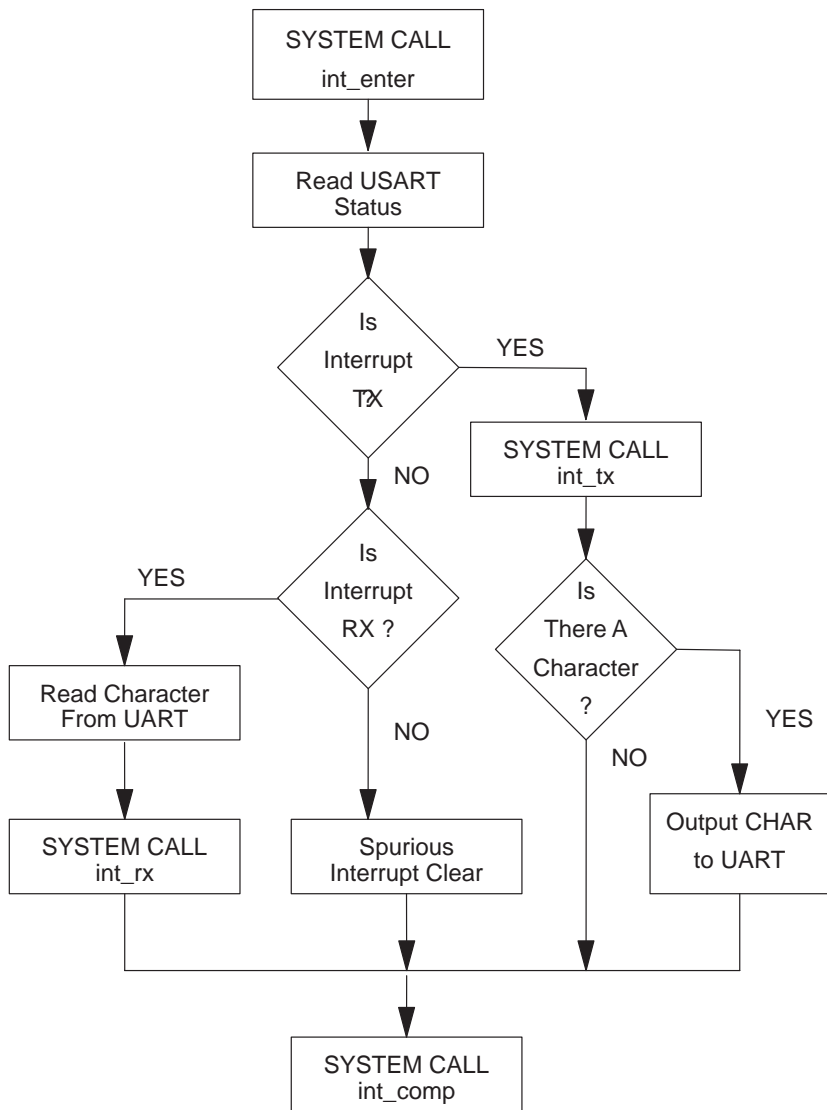


Figure Rom-3: Character processing for interrupt driven I/O

Figure Rom-3 shows the flow of the serial port ISR. The ISR first traps to SmartMON with an `INT_ENTER` (interrupt enter) code to signal that an interrupt has occurred. Control is then returned back to the ISR, which checks for a character having been transmitted. If so, it asks SmartMON for a character by making an `INT_TX` system call. If SmartMON returns a character, it is written to the USART. The ISR then exits after calling SmartMON with an `INT_COMP` (interrupt complete) code in `D0`.

If a character has been received, the ISR reads the character from the USART, and passes it to SmartMON with a `INT_RX` system call. The ISR then exits after making an `INT_COMP` system call.

2.9.2 POLLED I/O

In the polled environment, characters are passed from SmartMON to the Host by placing a character in the serial port transmit buffer. Unlike interrupt-driven I/O, which will create an interrupt when the USART transmit buffer is empty, the `TX_CHAR` driver must monitor the USART's status by waiting for the transmit buffer empty status to indicate that the character has been transmitted. At that point, the `TX_CHAR` routine will make a system call to request another character. If a character is available, it is placed in the serial device's transmit buffer. This process continues until the system call indicates that there are no more characters available. At this point, the `TX_CHAR` routine returns to SmartMON. See Figure Rom-4 for the flowchart of a polled transmit routine.

When the host transmits a character to SmartMON, the program is looping and continually calling `RX_CHAR` looking for another character. When a character is received, it is placed in SmartMON's line buffer. This process continues until a complete message is received. At which point, SmartMON will process the command. If additional characters are sent during command processing, an overrun condition may occur. This should not be a problem under normal operation. See Figure Rom-5 for the flowchart of a polled receive routine.

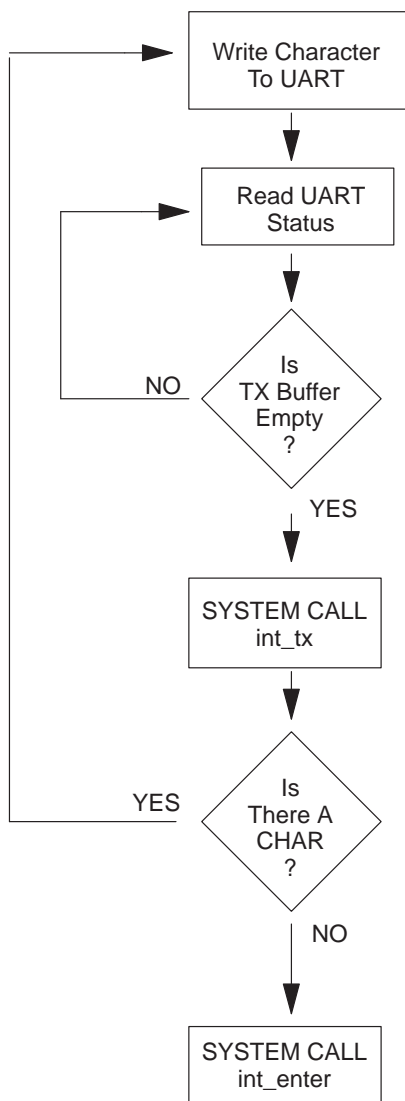


Figure Rom-4: Transmitting polled I/O characters

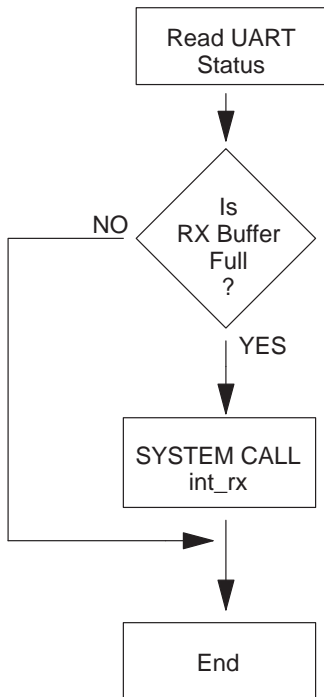


Figure Rom-5: Receiving polled I/O characters

2.9.3 CHARACTER BUFFERING

SmartMON communicates with the host by means of a line buffer. This means a command is not processed until a line has been entered. A line is defined as a series of characters terminated with a Carriage Return. This allows the editing of a line before it is processed. Once a complete command has been entered, processing begins. SmartMON allows type ahead for one line which means that characters may be entered at the same time the previous command is being processed. This type ahead feature also generates an XOFF to the host when a complete command is in the type ahead buffer.



2.9.4 I/O SYSTEM CALLS

As discussed earlier in the *SmartMON's Use of Interrupts and Traps* section earlier in this chapter, SmartMON must be re-entered after the I/O driver routine, `IO_DRV.68K`, has received or transmitted or character (or determined that there are no more characters to process.) As with all system calls, the way to do this is via the ROMM trap (the SmartMON enter trap) and an enter code in `D0`. The code in `D0` tells the debugger why it has been entered. In the case of the I/O routines, the I/O data, if any, will be located in `D1`.

The function codes and parameters passed via the ROMM trap are defined as follows:

Codes	Function	Response
code in <code>D0</code> = 01	ISR enter code	none
code in <code>D0</code> = 02	ISR exit code	none
code in <code>D0</code> = 03	transmit character	<code>tx_char</code> (<code>D1</code>), no char flag (<code>D0</code>)
code in <code>D0</code> = 04	receive character	<code>rx_char</code> (<code>D1</code>)
code in <code>D0</code> = 05	SmartMON enter code	none

Table Rom-2: Function codes

2.10 HOW SMARTMON IS INITIALIZED

SmartMON is initialized through one of two functions. The first function, `(RM_INIT)`, is called by the board initialization code, `(RMAIN.68k)`. SmartMON configures itself according to the data contained in the configuration table found within the `USREQU.68k` file. This allows the user to define SmartMON's resources, such as the beginning of SmartMON's RAM space, which traps are to be used, etc. The `(RM_INIT)` function builds the vector table with a generalized exception and installs all the traps to be handled by SmartMON. It also sets up all data structures and variables required by SmartMON.

The second function is (NV_RM_INIT). This is a non-vectorized ROM monitor initialization routine. This is supplied for those users who wish to build their own vector table. This normally applies to applications where the vector table is in Read Only Memory (ROM). This initialization routine sets up SmartMON without building the vector table. The user must install SmartMON's vector addresses into the vector table in order for SmartMON to operate properly.



The next chapter, *Target Interface Package*, explains these two initialization options in greater detail.

2.11 RUN-TIME NOTES

This section provides some helpful notes about running your application code in conjunction with SmartMON. Figure Rom-6 shows graphically how SmartMon interacts with your application code.

2.11.1 STACKS

SmartMON will always execute in supervisory mode. The User will allocate at least 6K of RAM for SmartMON's variables, stack space, and trace buffer. This will suffice for proper SmartMON operation. However, in some cases, such as system calls, where SmartMON has not gained total control over the processor and the user's code, SmartMON will rely on the user's supervisory stack. SmartMON needs a minimum of 24 bytes in order to operate. We do recommend having more stack space available.

2.11.2 INTERRUPT SERVICE ROUTINES

If you have interrupt service routines (ISR) that will continue to execute while SmartMON is active, there are a few precautions that should be followed. Upon entry, your ISR must save all of the microprocessor's registers before they are used. Any register that is used for pointing to global data space must be reinitialized. Upon exit, all registers must be restored. This will ensure proper SmartMON operation. Please note that the system calls INT_ENTER, INT_COMP, INT_RX, and INT_TX are only required for SmartMON's I/O communications.

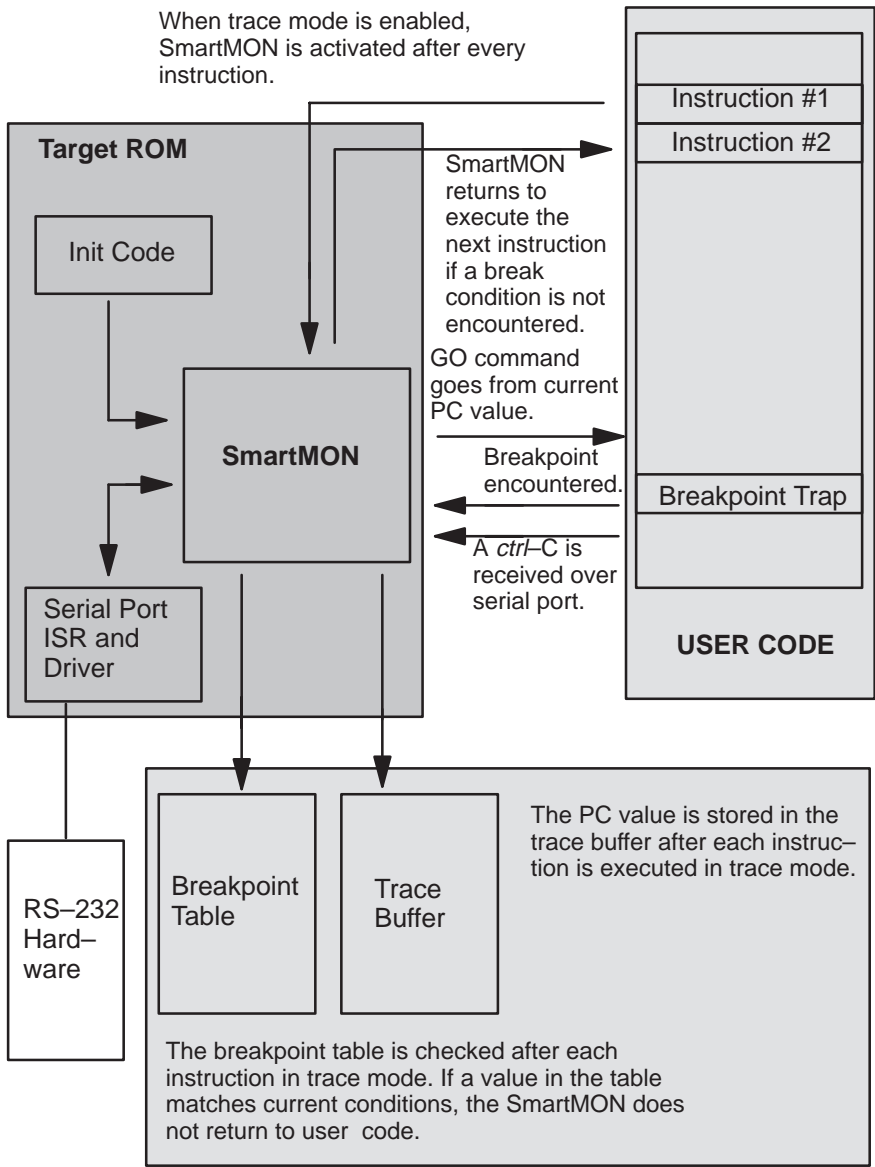


Figure Rom-6: Application code interaction

2.11.3 DOWNLOADING AN ISR FOR DEBUGGING

If the user is going to download interrupt service routines for debugging, then the addresses of these exception routines may be pre-defined. Once these routines are debugged, then its only a matter of changing their addresses to reside in ROM. This task can be accomplished with some additions to the `RMAIN.68K` module and the use of your Locating Tools.

Example

SmartMON will be burned into PROM, whose address starts at 000000. There will be 64K of RAM space residing at hex address E90000, which will be used for downloading and debugging user code. The Vector Base Register will reside at hex address E80000. We would like to debug a timer interrupt service routine named “timer”. The vector address of the timer ISR is at hex offset 200 off the VBR. We have defined other user defined vector routines in `RMAIN.68K` using the declare long address (`dc.l` routine name). The following is the modifications to the `RMAIN.68K` code:

Define the timer routine to be externally referenced:

```
XREF TIMER
```

Declare the timer routine to be a long address:

```
DC.L TIMER
```

Install the timer routine’s address at hex offset 200 of the vector base register:

```
MOVE.l # TIMER,$E80200
```

During the locate part of the build process in the locate command file add:

```
Declare the _timer routine’s address to be hex E90000.
```

After SmartMON has been built and burned into PROMS, then at offset 200 off the VBR, there will be the timer ISR’s address 00E90000. The only modification necessary to the user’s build process is to add a locate statement directing the timer routine to reside at hex E90000. For example:

```
LOCATE ( timer : #E90000);
```


After downloading the user code, the timer ISR will reside at address \$E90000. This will coincide with the vector offset pointer at \$200. The timer ISR may now be debugged.

2.11.4 SYSTEM CONTROL

The routines `SYSSTP.68K` and `SYS_GO.68K` have been provided so that target hardware functions may be disabled and re-enabled while SmartMON is active. This is particularly useful for controlling timer hardware. For example, if you need to suspend your timer when not operating in real-time, then these functions would be used. These routines are under your control and may be customized for your particular needs. SmartMON simply calls these routines when the debugger is activated and deactivated, no matter what these routines are customized to do.

3 TARGET INTERFACE PACKAGE

This chapter describes in detail how to provide an interface from SmartMON to your own custom hardware. The SmartMON object module is designed to be completely target and environment independent. Information is included here on how to provide SmartMON with routines and information, called the Target Interface Package (TIP), that are specific to your hardware. This chapter includes the following major sections:

- What is the TIP?
- TIP Module #1: usreq.68k
- TIP Module #2: rmain.68k
- TIP Module #3: io_drv.68k
- TIP Modules #4 and #5: sysstp.68k and sys_go.68k
- TIP Module #6: diag_tbl.68k

3.1 WHAT IS THE TIP?

The Target Interface Package (TIP) is a set of six assembly language modules that you modify for your target environment. All six modules must be included for SmartMON to operate. The TIP provides SmartMON with the following functions:

- a configuration table with labels for important target-specific data
- hardware and software initialization code
- a UART driver
- system start functions
- system stop functions
- a custom diagnostic table



The six file names and their functions are shown below:

TIP Component	File Name
Environment Configuration Data	usrequ.68k
Board Initialization	rmain.68k
Communication Software (UART driver)	io_drv.68k
Application-specific System Startup Code	sys_go.68k
Application-specific System Stop Code	sysstp.68k
User Diagnostics	diag_tbl.68k

Table Rom-3: TIP file names

3.2 TIP MODULE #1: USREQ.68K

The usrequ.68k file is one of the six TIP modules that must be included with the SmartMON object module in order for the debugger to operate on your board.

usreq.68k contains a table of important hardware information about the environment. The information is read from this file during ROM monitor initialization.

3.2.1 VALUES REQUIRED BY SMARTMON

SmartMON requires the information shown in the table below:

Label	Contents
Config_TBL	A data configuration table which consists of the following:
tx_char	A pointer to a serial port transmit routine.
rx_char	A pointer to a serial port receive routine.
sys_stop	A pointer to a routine called when SmartMON is activated.
sys_go	A pointer to a routine called when SmartMON is deactivated.
DIAG_TABLE	A pointer for custom diagnostics.
RM_RAM	Start of the SmartMON's data space (6K is the minimum RAM required – additional RAM may be allocated to the trace buffer).
RM_VBR	Location of the vector table. Set this value to 0 for processors that do not have a vector base register.
RM_INT	The Interrupt level of the UART.
RM_BRK	The trap assigned to breakpoints.
RM_TRP	Trap assigned to SmartMON system calls.
Micro	The microprocessor type.
RM_TSZ	Size of the trace buffer in 1K blocks of RAM.

Table Rom-4: Labels

3.2.2 MORE INFORMATION ON THE USREQU.68K LABELS

All of the labels shown on the previous page must be included in the usrequ.68k files for your board. The following is detailed information about each of these labels:

tx_char	(size=long) This is a pointer to the user-supplied transmit routine which the ROM monitor uses to transmit data over the I/O port. See the section on io_drv.68k for details on the actual routine.
rx_char	(size=long) This is a pointer to the user-supplied routine rx_char used by the ROM monitor when the I/O port is set up to operate in polled mode. This routine must exist in all environments during the time the ROM monitor is active and waiting for character input. If polled mode is not being used the routine need only contain an RTS instruction. Another use of this routine is to keep a watchdog timer alive. If sys_stop of sys_go do not satisfy your requirements for disabling a watchdog timer, this routine may be used to keep the watchdog alive. See the section on io_drv.68k for details on the actual routine.
sys_stop	(size=long) This is a pointer to the sys_stop routine. The ROM monitor calls this routine when it is activated. Activation occurs as the result of a breakpoint, a ctrl-X, a ctrl-C, or a ROM go system call. See sysstop.68k for details on the actual routine.
sys_go	(size=long) This is a pointer to the sys_go routine which is called by the ROM monitor when a GO command is issued. This allows you to reactivate critical hardware prior to executing user code. See sys_go.68k for details on the actual routine.
DIAG_TABLE	(size=long) The extended version of the ROM monitor uses this pointer to install diagnostics into the user diagnostic menu. In the Base version this must be replaced with a: dc.l \$00000000. See diag_tbl.68k and the <i>SmartMON Command Language</i> chapter on how to write user diagnostics.

RM_RAM	(size=long) This is the RAM start address that has been reserved for the ROM monitor. During initialization, the ROM monitor will set up its internal data structures, variables and buffers starting at this location. This data space is a minimum of 6K in size. A typical location of this RAM would be after the vector table (VBR value+\$400 hex).
RM_VBR	(size=long) This is the location of the VBR to be used by the target. If the target does not have VBR (such as the 68000 and 68302) then this value should be set to zero.
RM_INT	(size=word) This mask value, used when the ROM monitor is active, ensures that ROM monitor will not respond to interrupts of a lower value while it is active. If you are using interrupt driven I/O make sure the level is no higher than your serial port. If you wish to mask higher level interrupts than the serial port, then polled I/O must be implemented. For those interrupts that are going to remain active when the ROM monitor is running, the associated ISR must save all the registers they are using. In addition, registers such as A5 have to be reinitialized in the ISR.
RM_BRK	(size=byte) This is the trap number assigned to breakpoints. When the ROM monitor installs a break point, this is the trap that will be used.
RM_TRP	(size=byte) This is the trap number assigned to system calls.

Micro (size=word) This is used to tell the ROM monitor which type of microprocessor is being used. Enter the value for Micro that corresponds to the CPU in your target board:

target CPU	Micro	target CPU	Micro	target CPU	Micro
68000	\$0000	68060	\$0060	MCF5204	\$5204
68EC000	\$EC00	68070	\$0070	MCF5206	\$5206
68010	\$0010	68302	\$0302	MCF5206E	\$5207
68EC010	\$EC10	68306	\$0306	MCF5249	\$5249
68020	\$0020	68307	\$0307	MCF5272	\$5272
68EC020	\$EC20	68332	\$0332	MCF5280	\$5280
68030	\$0030	CPU32	\$0C32	MCF5282	\$5282
68EC030	\$EC30	68340	\$0340	MCF5307	\$5307
68040	\$0040	MCF5102	\$5102	MCF5407	\$5407
68EC040	\$EC40	MCF5202	\$5202		

Table Rom-5: Microprocessor selection

RM_TSZ (size=byte) This is the size (in 1K blocks) to be assigned to the trace buffer. Value may range from \$1 to \$F.

RM_INIT is the start location of the ROM monitor image. During initialization the ROM monitor subtracts \$30 from this address and reads the usr_equ structure, takes the value of RM_RAM and copies the user equates into this space. Therefore the section RMUSER_EQU must be located exactly \$30 before the start of the ROM monitor code. See the section on linking and locating the ROM monitor for more information.

After installing the ROM monitor software onto your host, you will find a ready-made copy of usrequ.68k in the vme105, vme133, 68302ads, and other board directories. This file describes the parameters for the named target board. Simply take the file and modify the values to support your environment. You may find samples that more closely match your hardware configuration in subdirectories of the boards directory. In order to modify the files, you must understand the memory map for your target and the interrupt structure.

3.3 TIP MODULE #2: RMAIN.68K

The next TIP module which you must modify and supply with SmartMON is `rmain.68k`. `rmain.68k` is the file called by SmartMON to initialize the hardware environment.

Initialization includes all preliminary activities that place the system into a known state before application execution. The initialization code is executed when the system is powered up or reset. This module should be located at the beginning of the PROM or wherever the Reset vector resides.

The User's hardware initialization code simply makes a call to the starting location of SmartMON code and the debugger initializes itself based on the data stored in the configuration table. SmartMON then returns to the user code and is ready for operation. The debugger is then activated by either typing a ^C or by making a ROMM_GO system call. Initialization code consists of the following:

- Define initial stack pointer and restart vectors.
- Set the status register to SUPERVISORY MODE and turn off interrupts.
- Optionally, set up devices that do **NOT** require an interrupt vector, but must be initialized before anything else can operate; such as an MMU (memory management units).
- Install reset vector and stack addresses into the exception vector table {EVT}.
- Call RM_INIT or NV_RM_INIT to initialize SmartMON. This initialization code resides at SmartMON's starting address (see RM_INIT call for details).
- Optionally, set up devices that **DO** require an interrupt vector. This type of device initialization must take place after the RM_INIT call, because the SmartMON initializes all vectors in the EVT during this call.
- Call the serial port initialization routine, PORTINIT, which sets the serial port device up to the correct configuration and installs the correct vector into the EVT for the ISR.
- Set the stack frame pointer A6 to zero.
- Clear all registers.
- Enable interrupts.

- Make a system call by using a ROMM trap (SmartMON enter trap) with `D0=ROMM_GO=0` (see `ROMM_GO` for details) or jump straight to user's code.

3.3.1 STACKS

The initial stack should be set up to have at least 24 bytes of scratch area. This stack is used for the initial call to `RM_INIT` and for any system calls, such as serial communications, before any application code has been downloaded and new stacks setup. During user code debug, you still need at least 24 bytes. However, we recommend allocating 1K bytes for each stack. The initial stack(s) may be reused or reallocated for this purpose. For example, your target has 128K of RAM available starting at address 0. Figure Rom-8 shows a typical memory map layout for a 68000 example.

3.4 RM_INIT CALL

During the `RM_INIT` function, SmartMON fills all of the exceptions in the exception vector table with a generalized exception routine. This assists in the debugging of user code by trapping all unassigned vectors. SmartMON then installs the specific exception routines required for its operation into the vector table. Upon return from this function, the user may then install his own vectors. Be careful not to overwrite SmartMON's vectors for this may cause unpredictable results. Interrupts must be disabled before the `RM_INIT` call is made and enabled after `RM_INIT` is complete.

If the vector table is hard coded, then a separate initialization routine (`NV_RM_INIT`) should be used. This function initializes SmartMON but does not build the vector table. It is then the responsibility of the user to hard code the vectors required by SmartMON. The table below Figure Rom-8 contains a list of addresses which the user should install into the vector table for SmartMON.

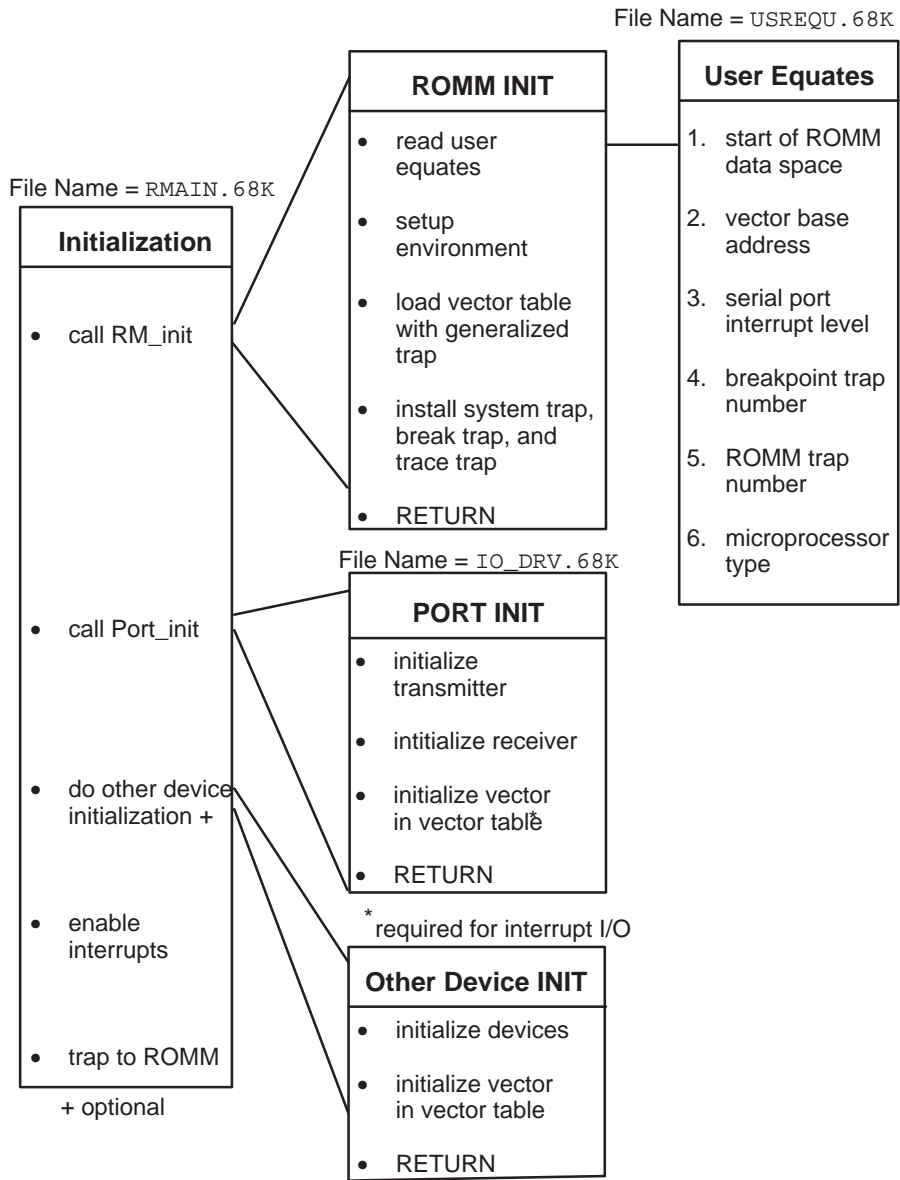


Figure Rom-7: Initialization

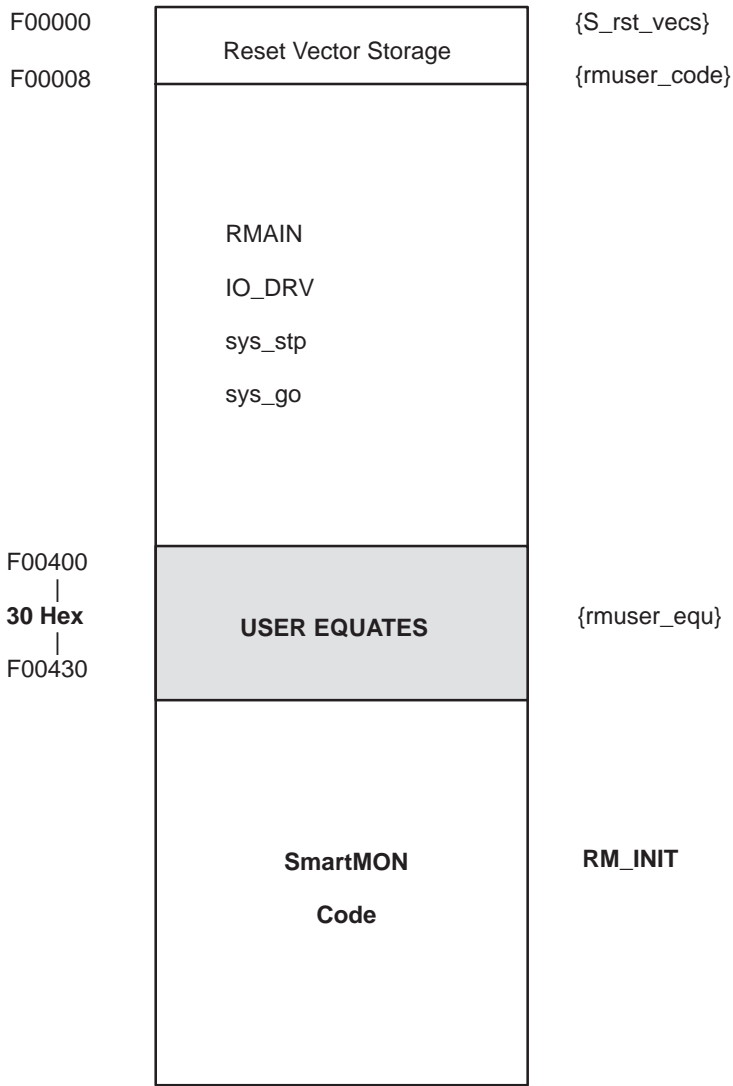


Figure Rom-8: Memory map

SmartMON Addresses

BASE ADDRESS = Start Address of SmartMON image		
Offset	Address (Symbolic Name)	Function
0	RM_INIT	SmartMON's vectored initialization address.
4	NV_RM_INIT	SmartMON's non-vectored initialization address.
8	BUS ERROR (be_trp)	Bus error {Vector #2} (optional).
C	ADDRESS ERROR (ae_trp)	Address error {Vector #3} (optional).
10	ILLEGAL INSTRUCTION (illtrp)	Illegal instruction error {Vector #4} (optional).
14	DIVIDE BY ZERO ERROR (divtrp)	Zero divide error {Vector #5} (optional).
18	PRIVILEGE VIOLATION (prvtrp)	Privilege violation {Vector #8} (optional).
1C	GENERAL EXCEPTION (gentrp)	Generalized exception may be used on unused vectors (optional).
20	TRACE VECTOR (trcistr)	Trace exception used by SmartMON for single stepping.
24	BREAKTRAP (break_trp)	Trap vector used by SmartMON for breakpoints.
28	SYSTEM CALL TRAP (ROMM)	Trap vector used by SmartMON for communications and system calls.

Table Rom-6: SmartMON addresses

Lets assume the following conditions for an example:

- The exception vector table is going to be hardcoded.
- The target EPROM is located at address 0000.
- SmartMON is going to be located at address 1000.
- TRAP #15 is to be used for system calls.

When SmartMON is called to be initialized, the call will be made to address 1004 (NV_RM_INIT – The NON-Vectorized Initialization Routine). The user must ensure that the exception vector for Trap #15 (Vector #47) is loaded with address 1028. In addition, the user must also install the addresses of all his exception routines at their corresponding vector locations.

3.5 ROMM_GO SYSTEM CALL

When all of the above routines have been completed, you may decide whether to enter the ROM monitor. Normally during the development stage, the ROM monitor is entered on power-up. If you wish to leave the ROM monitor in the prototype for product support, then, on power-up, the ROM monitor would not be entered and the user code would boot and run. The ROM monitor would only become active if a terminal was connected and a ctrl-C or ctrl-X was entered.

If you wish to power up directly into the ROM monitor, then at the end of the initialization code, place a ROMM trap with a ROMM_GO code (the value 5) in register D0. The ROMM trap is the standard system call to invoke the ROM monitor.

If the application program is to run directly upon power-up, omit the ROMM trap from the initialization code.

3.6 CREATING YOUR OWN RMAIN.68K

The pseudo code on the following page outlines the operation of RMAIN. Working examples of rmain.68k, as well as all of the other TIP modules, are included for many single board computers in the boards directory.

The pseudo-code for the operation of rmain.68k is shown below:

```
* PSEUDO CODE
*   section S rst vecs
*       define initial stack pointer
*       define restart vector (start:)
*
*   section rmuser code
*
* label= start:
*   set status register to supervisory mode
*   with INTS disabled
*
*   set up VBR to value defined in USREQU.68k
*   copy stack pointer to vector 0 in vector table
*   copy restart vector to vector 1 in vector table
*
*   call RM_INIT to initialize the ROM Monitor
*
*   initialize other target hardware
*   add other ISR address to vector table
*
*   call portinit to initialize serial port
*
*   set stack frame pointer A6 to zero
*   clear all registers
*
*   set status register to supervisory mode
*   with INTS enabled
*   system call ROMM_GO
*   END
*
```

3.7 TIP MODULE #3: IO_DRV.68K

SmartMON communicates with the dumb terminal or host computer through a serial communication device. `io_drv.68` is the module which provides the serial driver for SmartMON.

Data is passed to and from the debugger through a trap handler. A character is passed to the monitor by trapping to it with the character to be processed in register D1. A character is passed from the monitor by trapping to the ROM monitor and returning to the calling routine with the character in register D1. You must therefore supply a routine which passes characters to and from the ROM monitor and transmits or receives characters through the serial port device.

`io_drv.68k` contains four major functions:

- `portinit:` A routine which is called by `rmain.68k` to initialize the serial port.
- The serial port ISR: The interface between SmartMON and the serial port device.
- `TX_CHAR:` The transmit character routine.
- `RX_CHAR:` The receive character routine.

Each of these functions are described in this section.

3.8 PORTINIT CALL

`portinit` is a routine in `io_drv.68k` which is called by the initialization code `rmain.68k`. This routine configures the serial port to the correct configuration and loads the serial port interrupt vector into the vector table.

3.9 SERIAL PORT INTERRUPT SERVICE ROUTINE

The serial port ISR is the interface between the serial port device and the ROM monitor. When the serial port device generates an interrupt, a software routine to service the interrupt is invoked. This routine is known as an Interrupt Service Routine (ISR) or an interrupt handler.

An interrupt is generated when the device is ready to accept another character for transmission or when it has received a character available to input. For some types of UARTs, the same interrupt signals that a character has been received or one may be transmitted. In this case you can write only on ISR to handle both input and output.

A receiver ISR gets a character from the UART and passes it to the ROM monitor by means of a ROMM trap with a INT_RX code in D0 and the received character in D1.

A transmitter ISR traps to the ROM monitor with a INT_TX code in D0. The ROM monitor returns to the ISR with a character for transmission in D1. If the call to the ISR was the result of the last character being transmitted the ROM monitor will not return a character but, will set NO_CHAR in D0. This status indicates to the ISR that there are no more characters for transmission.

Some UARTs use the same interrupt to signal that the transmitter is empty or the receiver is full. In this case, the ISR must determine the device's status before proceeding.

3.10 TX_CHAR

In addition to the ISRs a routine called TX_CHAR must exist which the ROM monitor calls with the first character of a new message. As previously stated, the transmit ISR is only activated when the UART is ready to accept another character for transmission. Therefore, in order to initiate this process, the first character in a message must be loaded into the UART through TX_CHAR routine. All the remaining characters in the message will be transmitted by the transmit ISR. The transmit ISR requests the next character in the message from the ROM monitor until all characters have been transmitted.

3.11 RX_CHAR

RX_CHAR is a receive character driver used in polled I/O. This label must exist even in the interrupt-driven version of the ROM monitor, but need only contain a single RTS instruction.

3.12 HOW TO CREATE YOUR OWN IO_DRV.68K

The source for several I/O drivers is included in the drivers directory of the SmartMON release. There are also complete IO_DRV.68K modules included with the complete TIPs in the boards directory.

The pseudo-code for the operation of io_drv.68k is shown below:

```
*      PSEUDO CODE
*
*      section rmuser code
*
*      label = portinit:
*          initialize the uart
*
*          load uart ISR address in vector table
*          return
*
*      section rmuser code
*
*      label = uart int:
*          INT ENTER system call to ROM Monitor
*          read uart status register
*          IF RX buffer full
*              THEN GOTO RECEIVE INT
*          ENDIF
*          IF TX buffer empty
*              THEN GOTO TRANSMIT INT
*          ENDIF
*          GOTO ERROR
*
*      RECEIVE INT:
*          read uart receive buffer
*
*          INT_RX system call to ROM Monitor
*          INT COMP system call      ;no rtn from this call
*      END
*
*      TRANSMIT INT:
*          INT_TX system call
*          IF character to transmit
*              THEN write character to uart output buffer
*          ELSE clear interrupt
*      ENDIF
```

```

*          INT COMP system call      ;no rtn from this call
*      END
*
*      ERROR:
*          clear the error
*          INT COMP system call      ;no rtn from this call
*      END
*
*      label = tx_char:
*          write character to uart output buffer
*          rts
*
*      label = rx_char:
*          rts
*

```

3.12.1 SERIAL PORT POLLED I/O

In the polled I/O mode, characters are passed from SmartMON to the Host using the serial device's transmit buffer. The transmit routine, TX_CHAR, must monitor the device's status register by waiting for the transmit buffer empty status to indicate that the character has been transmitted. At this point, TX_CHAR must make a system call, INT_TX, to request another character. If a character is available, then the character is placed in the serial device's transmit buffer. This process continues until there are no more characters available to transmit. This is indicated by the return code of the INT_TX system call. Once all the characters are transmitted, then the INT COMP system call is made, from which there is no return. Execution either returns to User code or invokes the SmartMON debugger.

For character reception, SmartMON continually calls the receive routine, RX_CHAR, looking for another character. When a character is received, RX_CHAR must make a system call, INT_RX, to inform SmartMON that a character has been received. This process continues until a complete message is received. At this point, SmartMON will process the command. Additional characters received during the processing of this command may result in an overrun condition.

3.12.2 TX_CHAR USING POLLED I/O

The transmit character routine, TX_CHAR, is used by SmartMON for sending out all the characters of a message.

3.12.3 RX_CHAR USING POLLED I/O

The `RX_CHAR` routine must poll the device to check for a character received. Once a character is received, then it is passed to SmartMON by means of a system call trap with a `INT_RX` code in D0 and the received character in D1.

3.12.4 CREATING A POLLED I/O IO_DRV.68K

Several popular UART drivers have been provided for installation ease. Check to see if any are applicable to your specific application. If none of the drivers are applicable, one may be used as a template for writing your own. See the *Using SmartMON* chapter to better understand the polled I/O mode of operation.

3.13 TIP MODULES #4 AND #5: SYSSTP.68K AND SYS_GO.68K

In an operating system environment when SmartMON is in command mode, all normal operation should be suspended. In order to achieve this, the ROM monitor will disable interrupts while in the debugger state. However, you must supply any other target hardware specific operations that need to be suspended.

These operations must be placed in the `sysstp.68k` and `sys_go.68k` modules. The `sys_stop` routine is used for suspending operation and the `sys_go` for reactivating the operations. These routines are always called by the ROM monitor when the debugger is activated and deactivated.

3.13.1 SYS_GO

This routine, used for activating a user-defined operation, must be included even if there are no special requirements. In this case, the module need contain only an RTS instruction.

3.13.2 SYS_STOP

This routine for suspending a user-defined operation, must be included even if there are no special requirements. In this case, the module need contain only an RTS instruction.

3.14 TIP MODULE #6: DIAG_TBL.68K

The `diag_tbl.68k` contains a data structure that tells SmartMON about the user-supplied diagnostics that will be included with the debugger.

The routine must be provided with the TIP, even if you do not plan to include and custom diagnostics. In this case, the only information required is the first line of the example below, the number of test, where $n=0$. Refer to the *Diagnostics* chapter for more information.

```
*****
*                               module name diag_tbl.68k
*****

XDEF DIAG_TABLE
DIAG_TABLE:

    DC.W  000n                ; number of tests = n

    DC.L  TST1_MSG            ; Menu message for test #1
    DC.L  TST1_MAIN           ; User Diagnostic Test #1
    DC.L  TST1_ERR            ; User Error Routine for #1
    ..
    ..
    DC.L  TSTn_MSG            ; Menu message for test #n
    DC.L  TSTn_MAIN           ; User Diagnostic Test #n
    DC.L  TSTn_ERR            ; User Error Routine for #n
```

4 BUILDING SMARTMON

This chapter describes how to use the 68K/ColdFire C Toolkit to create an image of SmartMON that can be downloaded to a PROM burner, or downloaded directly into RAM on your target board. It includes the following major sections:

- Overview of the Build Process
- Notes on Building Applications to Work with SmartMON
- Starting-up SmartMON with CrossView Pro
- Troubleshooting
- Starting SmartMON with a Terminal or Terminal Emulator

4.1 OVERVIEW OF THE BUILD PROCESS

The steps required to build SmartMON for your board are listed below. Note that Steps 1 through 4 are covered in the previous chapter, *Target Interface Package*.

1. Fill in the information in `usrequ.68k` to describe your board.
2. Modify the `rmain.68k` system initialization module, which will set up the restart vector for the processor, call `RM_INIT` to initialize SmartMON, initialize the serial port (call `portinit`), initialize other devices, and optionally `TRAP` to the debugger.
3. Supply the ISRs and initialization code for the USART (`io_drv.68k`).
4. Develop code for `sys_stop` and `sys_go`, if necessary (`sysstp.68k` and `sys_go.68k`). Fill in the diagnostic structure contained in the `diag_tbl.68k` file.
5. Assemble all the modules.
6. Link and locate code into appropriate place in user memory space.
7. Format for programming EPROMS.
8. Download code to PROM programmer.
9. Burn EPROMS.



The examples in this chapter are for a VME105 board with a 68010 microprocessor. The SmartMON release contains batch files for building the monitor and a demo program for each 68000 family board that is directly supported. The batch files are under the specific board directory. If you are building SmartMON for your own custom board, you may wish to examine the batch files for a board that contains your microprocessor. However, the batch files and the locator command files may need to be modified for your board. Where you locate your ROM Monitor and applications depends on the memory map of your board.

4.1.1 PREPARING THE BUILD ENVIRONMENT

For your convenience, create a WORK directory and then copy the six modified TIP source modules and the SmartMON object module into this directory. Make sure that your path and environment variables are properly set to ensure that you have access to all of the tools from this WORK directory.

4.1.2 ASSEMBLING THE TIP

The six source files of the TIP need to be assembled. Using the TASKING 68010 assembler on PC/DOS, the commands and switches are:

```
asm68010 rmain.68k -l rmain.lst -s -d -g -P
asm68010 io_drv.68k -l io_drv.lst -s -d -g -P
asm68010 sys_go.68k -l sys_go.lst -s -d -g -P
asm68010 sysstp.68k -l sysstp.lst -s -d -g -P
asm68010 usrequ.68k -l usrequ.lst -s -d -g -P
asm68010 diag_tbl.68k -l diag_tbl.lst -s -d -g -P
```



If you are using the base version of SmartMON with CrossView Pro, the `diag_tbl.68k` module is not required.

The assembly options used above are not required, but their use simplifies debugging when developing new TIP modules for custom boards.

Here is a summary of the assembler switches used:

- l***file* Generate listing in file *file*.
- s** Generate source listing.
- d** Create debug information.

-g Generate global symbol information.

-P Show generated structure syntax.

Please refer to the *68K/ColdFire C Compiler/Assembler User's Manual* for more information.

4.1.3 LINKING AND LOCATING THE OBJECT MODULES

The linking locator utility, LLINK, allows you to specify a link file and a locate command file on the command line. The link file, which in this example is named `rm.ols`, contains object file names (in IMST's `.ol` or `.ln` format) that will be linked together. The locate command file, named `loc68k.cmd` in this example, gives you a mnemonic means to specify the memory map and place the SmartMON and TIP labels at the key addresses.

The `rm.ols` link file contains the following:

```

rmain.ol
io_drv.ol
sysstp.ol
sys_go.ol
diag_tbl.ol
usrequ.ol
smon68ke.ln

```

Note that the `diag_tbl.ol` would not be listed in the `rm.ols` file when building the base version of SmartMON (`smon68kb.ln`). To build the base version you would also have to remove the reference to `DIAG_TABLE` in `usrequ.68k`. See the source file for the appropriate line.

For the VME105 example, the `loc68k.cmd` locator file would contain the following locator directives:

```

locate ( S_RST_VECS : #F00000 );
locate ( RMUSER_CODE : after #F00000 );
locate ( RMUSER_EQU : #F00400 );
locate ( est_bug : after #F00430 );

```



The ROM monitor will not function unless the user equates (section `RMUSER_EQU`) are exactly 30 (hex) locations before the start of the ROM monitor code (`est_bug`).

The LLINK utility command line that references these link and locate files is as follows:

```
llink -i rm.ols -o rm.ab -c loc68k.cmd -v
```

Here is a summary of the relevant linking locator switches:

-i rm.ols	Take name of input modules from file <code>rm.ols</code> . See file contents below.
-o rm.ab	Write output to file <code>rm.ab</code> .
-c loc68k.cmd	Read locator commands for <code>loc68k.cmd</code> .
-v	Report linker actions as they are performed.

4.2 FORMATTING

The formatter takes an `.ab` (linked and absolute located) object module and converts it to an object module in one of several industry standard download formats. The default format, which we will use in this case, is Motorola S records.

The formatter also allows us to apply a bias to the composite `.ab` file and to split the object module into odd and even PROMs. We will take advantage of these features in this example. Note that if we were planning to download our composite SmartMON S records file into RAM, we would not need biasing or split object modules.

The following are the formatter commands used on our composite `rm.ab` file that was generated by the LLINK utility above:

```
form rm.ab -b 0 2 -f xm -a F00000 -o rmev.hex  
form rm.ab -b 1 2 -f xm -a F00000 -o rmod.hex
```

If the previous sequence of commands runs without any error you will be left with the two hex files, `rmev.hex` and `rmod.hex`. These files are ready to be downloaded and burned into PROMs. Here is a summary of the formatter switches used:

-b 0 2	Control PROM byte slicing, output every second byte, even address.
-b 1 2	Output every second byte, odd address.

- f xm** Format the hex files in extended Motorola hex format.
- a F00000** Apply bias of F00000 to each record. The bias allows you to program PROMs (which need to have code loaded at address 0 of their memory space) while locating your code and data beginning at address F00000 of the target memory space.
- o filename** Write output hex file to *filename*. We specified *rmev.hex* and *rmod.hex*, respectively.

4.2.1 PROGRAMMING THE PROMS

After the two hex files (even and odd) have been built, they must be burned into PROM or EPROM chips. Before programming the chips, you should check the PROM programmer's documentation to make sure it is able to program the chips. Also compare the microprocessor and board requirements against the PROM/EPROM chip specifications to see if the chips are the correct speed and if they are compatible with the microprocessor and board. Different burners operate differently. Please consult the PROM programmer's manual to learn how to create the PROMs.

4.3 NOTES ON BUILDING APPLICATIONS FOR SMARTMON

This section describes building a demo application program to run with SmartMON. All of the demonstration files can be found in the demo subdirectory. The *pmain* startup code and the *locate* command file must be customized to work with a specific single board computer and microprocessor. Customized versions of *pmain.68k* and *apploc.cmd* can be found in each board's subdirectory. Some examples are:

68010	VME 105
68020	VME 133
68030	VME 147
CPU32	68332 EVS
68302	68302 ADS

The steps for building the demo program are described in the following sections.



The following steps are only an example. You must know the memory map and other characteristics of your board in order to make any modifications.

4.3.1 STEP 1: MODIFY PMAIN.68K

`pmain.68k` contains the startup code required for executing user applications. The source for `pmain.68k` is found in the run-time library on the 68K/ColdFire product distribution. To execute user code with the ROM monitor you will first need to modify `pmain.68k`, assemble it, and link in the resulting object module, `pmain.ol`, instead of the default run-time library version. If you plan eventually to run user code without the ROM monitor, be sure to save the original `pmain.68k`.

You will need to modify the code in `pmain.68k` to relocate the System Stack Pointer (SSP/MSP) and the User Stack Pointer (USP). This means changing the address of the stack pointers in two lines of code. The stack pointer must be changed so that the stacks do not reside in the same area of RAM memory that the ROM monitor uses. The following examples illustrate the changes for `pmain.68k` when the target system is a Motorola VME105 single board computer. This is the same target system used in other examples in the manual. The following lines must be changed:

Change the commented line:

```
movea.l    #$00007ffc,A7          ;Set SSP (A7) to
                                   ;absolute address
                                   ;00007ffc
```

To:

```
movea.l    #$00ea7ffc,A7          ;Set SSP (A7) to
                                   ;absolute address
                                   ;00ea7ffc
```

Change the commented line:

```
movea.l    #$00007f00,A7          ;Set USP (A7) to
                                   ;absolute address
                                   ;00007f00
```

To:

```
movea.l    #$00ea7f00,A7    ;Set USP (A7) to
                             ;absolute address
                             ;00ea7f00
```

Because SmartMON will use the reset vectors for the stack pointer and program counter, `pmain.68k` should not allocate data to these memory locations. The segment, therefore, that defines the reset vectors must be removed. The following lines of `pmain.68k` must be commented out for the VME105 board:

```
org 0
dc.l      $00007ffc
dc.l      __main
```

To:

```
*   org      ea0000
*   dc.l     $00ea7ffc
*   dc.l     main
```

The modifications to `pmain.68k` listed above must be done for all user programs that will run on hardware with SmartMON present.



Refer to the *Run-time Library* appendix of the *68K/ColdFire C Compiler/Assembler Reference Manual* for information on incorporating modified routines into the run-time library.



Many options can interfere with normal SmartMON operations or cause unsuspected results.

If Watch Dog Timer (WTD) is being used, be sure that `sys_go.68k` and `sysstp.68k` are also modified to disable and re-enable the WTD time to prevent it from resetting the target board during monitor control.

Upon return from the `main` procedure, you may go into an infinite loop. If the trap to monitor is used, the monitor will reset itself and lose all breakpoint information and any other operating functions (e.g., trace enable).

4.3.2 STEP 2: BUILD THE DEMO OBJECT MODULES

Compile and assemble the demo files while generating symbolic information for the CrossView Pro C source-level debugger. The command lines are:

```
c68000 demo.c -d -do
asm68000 pmain. -d
asm68000 addone.68k -d
```

Link and locate the code, saving symbolic information while linking in the debugging routines. The user code must be located in an area of RAM memory not used by the ROM monitor. The address to start locating the user code is determined by subtracting the original value of the SSP in `pmain.68k` from the value that it was changed to. Using the above examples they would be:

```
00ea7ffc - 00007ffc = 00ea0000
```

You should create a locator command file, `locdemo.cmd`, for the VME105 board which would contain the following line:

```
LOCATE({code}{}{data}{usep}{constant}:AFTER #ea0000);
```

Assuming the modified `pmain.ol` will be linked in directly (instead of using the librarian to insert it into the run-time library), the command to run the linking locator would be:

```
llink demo.ol pmain.ol addone.ol end.1n -o
-L libc68kdm.nf -v -c demo.loc
```



Do not use the `llink -x` switch. The `-x` switch, which is used to build programs to be debugged with emulator-based versions of the debugger, will link in the run-time library object modules `end.1n` and `break.1n`. Although `end.1n` should be linked with the application, `breakpt.1n` will interfere with the way the ROM monitor handles code breakpoints. Instead, `end.1n` should be linked in explicitly on the `llink` command line. `end.1n` contains code which allows you to take advantage of CrossView Pro's ability to evaluate function calls on the debugger command line.

The file `end.1n` can be found in the appropriate compiler run-time library. If the library containing `pmain.68k` has been modified on a permanent basis it is not necessary to explicitly link in `pmain.ol`.

To format the code, saving symbolic information, type:

```
form demo.ab -x
```

The above four steps will produce a CrossView Pro symbol table file, `demo.abs`, and a hex file, `demo.hex`, that can be downloaded and executed.

4.4 STARTING-UP SMARTMON WITH CROSSVIEW PRO

Upon startup, CrossView Pro tries to establish contact with SmartMON, using the settings specified on the CrossView Pro command line by environment variables.

CrossView Pro has the ability to determine where the user code is and what procedures are currently on the stack. When CrossView Pro first communicates with the ROM monitor, the following communication takes place between CrossView Pro and the ROM monitor:

1. CrossView Pro issues an **IN** command to determine the processor type.
2. CrossView Pro requests the first instruction of user code, looking for **LINK** instructions, so it can correctly synchronize the source code with machine instructions.
3. CrossView Pro requests the following registers: PC, A5, A6, and A7. If the PC is within the bounds of the code to be debugged the appropriate source code is displayed.
4. CrossView Pro then looks at the value of A6, the frame pointer. If this value is not zero, CrossView Pro will follow the frame pointer to determine what procedures have been called. CrossView Pro will follow the current active stacks, depending on the value of the status register. This allows CrossView Pro to build a stack window showing the current status of the stack. It is very important that A6 is set to zero in **RMAIN**. This ensures that CrossView Pro does not have to chase down an invalid frame pointer.

When a reset and run is issued (CrossView Pro **R** command), the following communication takes place between CrossView Pro and SmartMON:

1. CrossView Pro issues a **DC** command to find the vector base register.

2. CrossView Pro issues a display memory at vector zero of the VBR. This is the initial stack pointer for the system and is setup by RMAIN. CrossView Pro sets either the ISP or SSP to the value contained at that location.
3. CrossView Pro sends a command to reset all the registers to zero.
4. CrossView Pro sets the status register to 2700.
5. CrossView Pro goes to the symbol table and finds MAIN and loads that address into the PC.
6. CrossView Pro then sets any breakpoints.
7. CrossView Pro then issues a **GO** command.

If the debugger has trouble communicating with the ROM monitor, it prints a message starting with:

```
Sorry, the monitor is not responding.
```

This indicates that no reply was detected after sending data to the target system. The debugger will try to reset the ROM monitor. If the debugger can then get a response, you will be given another chance to communicate in transparency mode. If the debugger still cannot get any response, it will give up and exit. See the *Troubleshooting* section later in this chapter.

If you get no response when invoking CrossView Pro, restart, and enter transparency mode. Also, it is best to enable target/emulator output logging for technical support help. Use the **DC** (Display Configurator) command to see if the monitor is responding with the correct information. If you get no response, there is no communication with the ROM monitor. Refer to the following *Troubleshooting* section.

4.5 TROUBLESHOOTING

Most problems in starting up CrossView Pro for a debugging session stem from improperly setting up the target system or from an improper connection between the host computer and the target.

Here are some common problems:

- Specifying the wrong device name when invoking the debugger.

- If you have installed the monitor with interrupt-driven I/O, problems may exist with interrupt handling. Try installing the ROM monitor with a polled-driven I/O driver to verify this is the problem.
- The ROM trap number defined in `usrequ.68k` is not the same trap number used in `rmain.68k` and `io_drv.68k`.
- Specifying a baud rate different from the one the UART is configured to expect.
- Not supplying power to the target system.
- Using the wrong kind of RS-232 cable.
- Plugging the cable into an incorrect port on the target or host. Some target boards and hosts have several ports.

4.5.1 LOCATING THE TIP

The important thing to note is that the configuration table, found in `USREQU.68K`, **MUST** reside 30 hex locations before the start of SmartMON code. This requirement is absolutely necessary in order for SmartMON to initialize its environment. Locating the rest of the TIP is not as restrictive, since the addresses will be resolved by the configuration table's contents.

For example, if you wanted to place SmartMON at address 1000 Hex, then the configuration table must reside at 0FD0 Hex (1000 - 30). In this example, the user supplied code is defined in three sections:

<code>S_RST_VECS:</code>	This section contains only the addresses of the restart vectors.
<code>RMUSER_CODE:</code>	This section contains the initialization code, the system start and stop functions and the I/O driver.
<code>RMUSER_EQU:</code>	This section contains the configuration table and the user equates.

The reason for defining these sections is not only for clarity but also to ensure that the restart vectors and the user's configuration table may be placed at specified addresses of the EPROMS.

A Sample Locate Command File: Vectored

```

declare( RM_INIT :           #f00430 );
locate ( S_RST_VECS :       #f00000 );
locate ( RMUSER_CODE :      after #f00000 );
locate ( RMUSER_EQU :       #f00400 );
locate ( est_bug :          #f00430 );

```

A Sample Locate Command File: Non-vectored

```

declare( NV_RM_INIT :       #f00434 );
declare( Trace_VEC :        #f00450 );
declare( BreakTrap_VEC :    #f00454 );
declare( SysCallTrap_VEC :  #f00458 );
locate ( S_RST_VECS :       #f00000 );
locate ( RMUSER_CODE :      after #f00000 );
locate ( RMUSER_EQU :       #f00400 );
locate ( est_bug :          #f00430 );

```

4.5.2 PROGRAMMING EPROMS

First, download the TIP image to the beginning of the PROM programmer's buffer. Then download the SmartMON's image to the correct offset of the PROM programmer's buffer. The correct offset being 30 hex locations after the configuration table. Last, burn the PROM and verify its contents.



If you are splitting your image to burn two PROMs, divide the start location of SmartMON by two. Use this number for SmartMON's offset.

When downloading a file to your PROM programmer buffer, the PROM programmer should prompt you with the starting address in the buffer that you wish to load your code. If the programmer does not, check the manual for setting buffer offset.

Using the VME105 example above and assuming the TIP has been formatted with the correct bias and the split images are named TIP.odd and TIP.eve. Starting with the odd file, download TIP.odd to the PROM programmer buffer starting a offset 0000. Then take SmartMON image 68kxe.odd and download it to the PROM programmer starting at hex location 0218 (half of 430 because it is split). Burn and verify the PROM. Repeat for the even side.



68kxe.odd and 68kxe.eve are not supplied but can be created by linking rom68ke.ln with a command locate and then formatted to generate the two files.

4.6 STARTING SMARTMON WITH A TERMINAL OR TERMINAL EMULATOR

Once the PROMs are programmed, simply plug them into the correct sockets on your target hardware. Apply power and the SmartMON banner should appear on your terminal/PC's screen. The SmartMON banner will look as follows:

```
SmartMON target: M68xxx                      Version x.xx
Copyright (c) 1997 Tasking, Inc., As Modified
>
```

5 SMARTMON COMMAND LANGUAGE

This chapter is a reference for the commands that can be issued to SmartMON in direct communication mode. It includes the following major sections:

- Overview
- Control Characters
- Operation Modes
- Command Descriptions

5.1 OVERVIEW

This chapter is a reference for the interactions between the user and SmartMON in direct communication mode. Here, direct communication mode means that you are talking to SmartMON via a dumb terminal or terminal emulation program from a PC or workstation.

For information on how to drive SmartMON via the CrossView Pro source-level interface, see the *CrossView Pro Debugger User's Manual*. CrossView Pro supports a direct target communications mode, called emulator mode, in which you will be able to invoke all of the SmartMON commands described in this section. You may wish to use emulator mode with CrossView Pro in order to access SmartMON's tracing, diagnostics, data break points, and block memory operations features.

SmartMON uses a standard ASCII protocol with XON/XOFF flow control. It uses two 512 character buffers; one for commands and the other for responses. A command will not be interpreted until a termination character is received. In other words, a command will not be executed until a line of characters, including the termination character, has been entered. At most, only two commands can be stored in the character buffer at a given time, the command that is being executed and the one about to be executed. Most commands have responses associated with them. All responses will end with a Carriage Return (CR) and Line Feed (LF), followed by a prompt (>).

Character processing is case insensitive. A command name is at least two characters in length. Most commands can accept optional arguments or parameters. A space character is used as the delimiting character between command names and arguments.

5.2 CONTROL CHARACTERS

The following control codes may be entered for command line editing, interrupting SmartMON, and flow control processing.



The presence of the upward caret, "^", before a character indicates that the Control or CTRL must be held down while striking the character key.

^C (interrupt)	This character will terminate any operation, including an XOFF condition, flush the character buffer, and return a response prompt. If the user program is operating with serial port interrupt disabled the command will not be seen until interrupts are enabled.
^H (backspace)	The cursor is moved back one space and the character at that position is deleted.
^J (CR)	The carriage return character is used as the line termination character. In some cases, it may repeat the last command.
^Q (XON)	This character will restart the flow of characters.
^S (XOFF)	This character will stop the flow of characters.
^X (line delete)	The cursor is backspaced to the beginning of the line. The last line of characters in the buffer is flushed. A response prompt is sent.

5.3 OPERATION MODES

SmartMON has three modes of operation, each of which changes the user's interface slightly. The rest of this section will describe these modes, which are as follows:

- Command Mode
- Download Mode
- Execution Mode

5.3.1 COMMAND MODE

The target does not execute instructions in this mode. Commands typed in at the terminal/PC are interpreted by SmartMON. Command Mode is the user's interface to the target. The user can control resources and place the target into a known state in this mode. Most interactions take place in command mode, which is distinguishable by its prompt. For more details on all the command see the *Command Descriptions* section of this chapter.

5.3.2 DOWNLOAD MODE

This mode is used to send down the user's application code to the target's RAM. Download mode is used mostly during the debug stages of a project. This mode is entered through the Download command (**DL**) and stays in effect until either the end of file is reached or an interrupt control character is received. For more information, see the **DL** command in the *Command Descriptions* section of this chapter. Also, see the format of a Motorola S-Record in the *Object Module Formats* appendix in the *68K/ColdFire C Compiler/Assembler User's Manual*.

5.3.3 EXECUTION MODE

This mode is in effect when the target is executing instructions. This usually happens as a result of issuing a **GO** Command. Execution mode operates in either real-time, non real-time, or some combination of both, depending on the conditions that are setup in command mode. Data breakpoints and code breakpoints in ROM code results in non real-time execution, while either no breakpoints or code breakpoints in RAM code permits real-time execution. Some complex breakpoints conditions will run in real-time up to a point and then go into non real-time execution. There are many combinations possible. See the Set Breakpoint command (**SB**) in the *Command Descriptions* section of this chapter. A breakpoint or an interrupt control character will place SmartMON back into command mode. For more details on entering the execution mode, see the **GO**, Single Step (**SI**), and Step Out Of Range (**SO**) commands in the *Command Descriptions* section of this chapter.

5.4 COMMAND DESCRIPTIONS

This section contains descriptions of each of the commands that can be used with SmartMON. Each section will detail information about a particular command and its options. Most sections are provided with one or more examples for your convenience. These examples are formatted for clarity in this manual. Actual screen displays may vary from machine to machine. Some commands may be repeatable by hitting a carriage return. Some of these will simply repeat the command exactly, like **DB** – Display Breakpoints; while others will execute the command but increment the addresses, like **DM** – Display Memory. Repeatable commands are represented by an asterisks “*” in their section names. All the command responses will end with a carriage return, line feed, and a prompt, as indicated by <CRLF>.

BD

Function

Breakpoint Disable.

Syntax

BD*type* [*addr*]

type = **C** = code
 D = data
 R = range

addr = An optional breakpoint address.

Description

This command causes SmartMON to disable all (if *addr* is omitted), some, or one of the software breakpoints currently enabled.

Example

In this example we will disable a breakpoint at address EA0000. First, display all breakpoints, which also contain current status, and then disable the breakpoint. Finally, verify the status of the breakpoint. Notice the status change of the “cmp_flags”.

```
>DB
Data Break-Points
1. 0EA0000 data mask = 000ff cmp_flags = enabled Word_cmp_BEQ
>BDC EA0000
>DB
1. 0EA0000 data mask = 000ff cmp_flags = disabled Word_cmp_BEQ
>
```



DB – Display Breakpoints
SB – Set Breakpoint

BE

Function

Breakpoint Enable.

Syntax

BE*type* [*addr*]

type = **C** = code
 D = data
 R = range

addr = An optional breakpoint address.

Description

This command causes SmartMON to enable all (if *addr* is omitted), some, or one of the software breakpoints currently disabled.

Example

For this example, we will enable the breakpoint at address EA0000, which has previously been disabled. First, show all breakpoints and their status and then enable the breakpoint. Finally, verify the status of the breakpoint by issuing a **DB** command. Notice the status change of the “cmp_flags”.

```
>DB
1. 0EA0000 data mask = 000ff cmp_flags = disabled Word_cmp_BEQ
>
>BEC EA0000
>DB
1. 0EA0000 data mask = 000ff cmp_flags = enabled Word_cmp_BEQ
>
```



DB – Display Breakpoints
SB – Set Breakpoint

BF

Function

Block Fill.

Syntax

BF[*unit*] *base_addr hex_value count*

unit = **B** = byte
 W = word (default)
 L = long

base_addr = The start address for the fill, in hex.

hex_value = The value to be used to fill memory, in hex.

count = The number of units, in hex.

Description

This command causes SmartMON to fill a block of data starting at *addr* for *count* number of the specified unit size.

Example

In this example, we will fill a block of memory, 32 bytes (20 hex) in length with a hex word_string = 4121. First, display memory contents and then fill the block. Finally, verify the command.

```
>DM 1000 20
001000 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....
001010 0000 0000 0000 0000 0000 0000 0000 0000 0000 .....

>BF 10000 4121 20

>DM 1000 20
001000 4121 4121 4121 4121 4121 4121 4121 4121 4121 A!A!A!A!A!A!A!
001010 4121 4121 4121 4121 4121 4121 4121 4121 4121 A!A!A!A!A!A!A!
>
```


For this example, we will change the command unit size from the last example of a word to a size long.

```
>DM 1000 20
001000 0000 0000 0000 0000 0000 0000 0000 0000 .....
001010 0000 0000 0000 0000 0000 0000 0000 0000 .....

>BFL 10000 4121 20

>DM 1000 20
001000 0000 4121 0000 4121 0000 4121 0000 4121 ..A!..A!..A!..A!
001010 0000 4121 0000 4121 0000 4121 0000 4121 ..A!..A!..A!..A!
>
```



This command is only available in the extended version of SmartMON.



DM – Display Memory

BM

Function

Block Move.

Syntax

BM[*unit*] *src_addr dest_addr count*

unit = **B** = byte
 W = word (default)
 L = long

src_addr = The start address of the data to copy.

dest_addr = The address where the data should be copied to.

count = The number of units, in hex.

Description

This command causes SmartMON to move blocks of data from one address to another.

Example

In this example, we will move a 32 byte (20 hex) block from one area of memory to another. First, display the memory contents of both the source and destination address using the display memory command with a longer block. Then, execute the block move command and display the new memory contents.

```
>DM 1000 20
001000 5448 4953 2049 5320 4120 5445 5354 2121 THIS IS A TEST!!
001010 0000 0000 0000 0000 0000 0000 0000 0000 .....

>BM 1000 1010 0F
>DM 1000 20
001000 5448 4953 2049 5320 4120 5445 5354 2121 THIS IS A TEST!!
001010 5448 4953 2049 5320 4120 5445 5354 2121 THIS IS A TEST!!
>
```



This command is only available in the extended version of SmartMON.



DM – Display Memory

CF

Function

Configure.

Syntax

CF [*flags*]

flags = **TF** = tracing with full data movements
 TP = tracing program counter only (default)

Description

This command allows the user to change the tracing configuration parameters of SmartMON. Tracing full or tracing program counter only (the default) may be selected. When tracing full is selected, SmartMON not only stores the program counter, but also the data movements associated with each instruction.



This command is only available in the extended version of SmartMON.



DT – Display Trace Buffer

DB

Function

Display Breakpoint.

Syntax

DB

Description

This command causes SmartMON to display all software breakpoints and their status.

Example

Code breakpoint #1 causes SmartMON to go into command mode from execution mode. Notice that this breakpoint was set to break after reaching this address for a second time. Code breakpoint #2 is set to break anytime the instruction at this address will be executed. Code breakpoint #3 is a temporary breakpoint set as a result of issuing a GON command. The GON command was issued placing SmartMON into execution mode and code breakpoint #1 was taken placing SmartMON back into command mode. A data breakpoint was set to break if the word at address EA0000 becomes equal to 00FF. A data range breakpoint was also set to break if the value at address EA1234 goes lower than 7 or higher than 9.

```
>DB <cr>
```

```
Code Break-Points
```

```
1. 010000 count = 0002 actual = 0002 enabled ***
2. 010010 count = 0001 actual = 0000 enabled
3. 010408 count = 0001 actual = 0000 enabled (temporary)
```

```
Data Break-Points
```

```
1. EA0000 data mask = 000000ff cmp_flags = enabled Word_cmp_BEQ
```

```
Data Range Break-Points
```

```
1. EA1234 Low = 0007 High = 0009 cmp_flags = enable outside check
>
```



Breakpoint status includes whether the break is enabled or disabled and the conditions on which to break. For viewing convenience, each breakpoint is assigned a number within each category type: code, data, and data range. In addition to this, after a breakpoint has occurred, a flag indicates which breakpoint has interrupted execution mode and placed SmartMON into command mode.

Either type of data breakpoint causes SmartMON to go into non real-time execution mode.



GON – Go to Next Instruction
SB – Set Breakpoint

DC

Function

Display Configuration.

Syntax

DC

Description

The command allows the user to examine the environmental resources that are configured for SmartMON, which include:

- microprocessor type
- address of vector base register
- beginning of SmartMON's RAM space
- ending address of SmartMON's RAM including trace buffer
- interrupt mask level
- trap number used for breakpoints
- trap number used for system calls
- size of the trace buffer
- type of storage used by the trace buffer, PC only or PC with data movements
- mode of operation, if entering the execution mode

All of this information, except for the trace buffer storage and mode of operation, is read directly from the configuration table, which is setup as part of the TIP elements. The trace buffer storage may be changed by using a Configure command (**CF**). The mode of operation is determined by many conditions, including tracing and conditional breakpoints.

Example

In this example, we will display a configuration setup for SmartMON. The microprocessor is a Motorola 68010, with the vector base register setup at address E80000. SmartMON's RAM space, including the trace buffer, starts at E90000 and ends at E91800. Interrupts greater than level 4 will be allowed to run while SmartMON is active. The I/O and breakpoint traps will be assigned to traps 13 & 14, respectively. There is a 1K byte buffer setup for storing trace information, which is configured for the program counter only. If no other breakpoint conditions are set before you issue a **GO** command, the execution mode will run in real-time.

>DC

SmartMON uses the following resources:

```
Micro Type = M68010
VBR = 00E80000
RAM Start = 00E90000
RAM End = 00E91800
INT Mask = 0400
Break Trap = 14
I/O Trap = 13
Trace Buffer = 1K
Trace Buffer Storage = PC only **
Mode of Operation = ..Running Real-Time
```

** = may be modified with CF command

>



CF - Configure
GO - Go

DF

Function

Diagnostic Function.

Syntax

DF[*unit*] *n* [*args*]

unit = **B** = byte
 W = word (default)

n = A number corresponding to a particular diagnostic.

args = Optional parameters containing addresses and hex strings.

Description

This command will run a diagnostic function. See the *Diagnostics* chapter for more details.



This command is only available in the extended version of SmartMON.

DI

Function

Disassemble.

Syntax

DI [*addr*] [*count*]

addr = The address to begin disassembling (the PC is the default).

count = The number of lines (in hex) to disassemble:
 1 – default
 20 – maximum

Description

This command causes SmartMON to disassemble target code beginning at either the program counter or the address specified for a length of count number of lines. This command can also be used in combination with the Single Step (**SI**) command.

Example

In this example, we will disassemble 8 lines of user's code located at address 10000. The display will show the address, the opcodes, and the disassembled instructions.

>**DI 10000 8**

0010000	2200	move.l	D0,D1
0010002	4282	clr.l	D2
0010004	D401	add.b	D1,D2
0010006	E289	lsl.l	#1,D1
0010008	66FA	bne.s	\$10004
001000A	E20A	lsl.b	#1,D2
001000C	55C2	scs	D2
001000E	60FE	bra.s	\$1000E
>			



This command is only available in the extended version of SmartMON.



SI – Single Step

DL

Function

Download.

Syntax

DL [*addr*]

addr = An optional address offset to be added to each S record.

Description

The download command invokes a special mode of operation, called the download mode. Once entered, all information sent over the serial port is assumed to be data (S record format) until an EOF is encountered or the interrupt control character (^C) is received.

During downloading, the characters received are echoed back to the host. Also, as a result of receiving a good S record and checksum, a positive acknowledgement “+” (PACK) is returned. A negative acknowledgement “-” (NACK) will be returned for a bad S record or checksum. A PACK will also be returned upon the initial download request before the first data record is transferred.

SmartMON returns to command mode when an EOF is encountered. If a download is aborted with a ^C, all download records following the abort will be treated as commands and will be handled as invalid. It is therefore the responsibility of the host to stop transmission of records after issuing the ^C.

For relocatable code, you may send an optional offset with the command. This offset will be added to the address specified by the S record. Hence, the code may be downloaded to any available memory space.

Example

In this example, the code was originally located at address ED0000, but we will add an offset of 1000 to this address. The display will show the positive acknowledgements and the echoed characters.

```
>DL 1000
```

```
+S00600004844521B
```

```
+S213ED0000000700ED005600ED006C00ED007800F7
```

```
+S213ED000FED00A400ED00BA00ED00C600ED00EA2E
```

```
+S213ED001E00ED010000ED013000ED017700ED0182
```

```
+S213ED002D8E00ED01C800ED025800ED026E00EDFD
```

```
+S804000000FB
```

```
>
```


DR

Function

Display Registers.

Syntax

DR [*reg_name*]

reg_name = A valid register name for the target.

Description

This command causes SmartMON to display the contents of a particular register or registers. If no arguments are specified then all the registers will be displayed. The active stack will be flagged with an asterisk “*”.

Example

In this example, we will display all the registers of a Motorola 68010 based target system. Notice the SSP or Supervisor Stack Pointer is the current stack pointer used in A7 (Address Register 7).

>DR

```
D0 = 00000000 D1 = 00000000 D2 = 00000000 D3 = 00000000
D4 = 00000000 D5 = 00000000 D6 = 00000000 D7 = 00000000
A0 = 00000000 A1 = 00000000 A2 = 00000000 A3 = 00000000
A4 = 00000000 A5 = 00000000 A6 = 00000000 A7 = 00E84000
USP = 00E85000 SSP*= 00E84000 PC = 00F00086 SR = 2000
VBR = 00E80000 SFC = 0007 DFC = 0007
>
```

In this example, first set some registers with values and then display these registers in different order.

>SRL A0 100000 A3 120000 D1 1234 D3 55

>DR D1 D3 A0 A3

```
D1 = 00001234 D3 = 00000055 A0 = 00100000 A3 = 00120000
>
```



SR – Set Register

DT

Function

Display Trace.

Syntax

DT[*arg1*][*arg2*] [*count*]

arg1 = **B** = backwards (default)
 F = forwards

arg2 = **F** = full trace with data movements

count = Number of locations to be displayed (default is 20 lines of information).

Description

This command allows the user to examine the contents of the trace buffer. The trace buffer contains the program history up to the point of SmartMON becoming active. This is the result of leaving execution mode and entering command mode. The trace buffer will have valid history only if tracing had previously been enabled, either directly with the Trace Enable (**TE**) command or indirectly by setting a data breakpoint, for example. The trace buffer contains PC history and/or data movement information, depending on the configuration parameters selected using the Configure command (**CF**).

Example

In this example, we display the last four instructions that were executed. SmartMON was previously configured for capturing full trace with data movements. Notice the instructions with data movements. The display will show the address and data, where applicable, of the source, destination before the instruction was executed, and destination after the instruction was executed.

```
>DTBF 4

4.-----
00EC00f2 bra.l $00EC0000
3.-----
00EC0000 moveq.l #$ffffffff,d0 FFFFFFFF,00000000->FFFFFFF
2.-----
00EC0004 move.l d0,d1 FFFFFFFF, 00000000 ->FFFFFFF
1.-----
00EC0008 move.b d0,(a0) (00EE0000)(00EE0000) 000000FF, 00 -> FF
>
```

In this example, we will show a trace display, without the data movements, of three instructions that were previously executed.

```
>DTB 3

19. 00ea6070 2080          move.b d1,(a0)
18. 00ea6074 10BC00AA      move.l #$$AA,(a0)
17. 00ea6076 2210          move.l (a0),d1 >
```



- CF** – Configure
- SB** – Set Break
- TE** – Trace Enable

GO

Function

Start Execution.

Syntax

GO[*flag*] [*addr*]

flag = **D** = trace disabled
 T = trace enabled

addr = An optional hex address where execution mode will resume.
 The default is the PC.

Description

This command causes SmartMON to go into execution mode. The execution mode will continue until either a breakpoint occurs or an interrupt is received from the host. If breakpoints are enabled, a TRAP instruction is inserted into the target code at each breakpoint address. This will allow real-time execution of target code and still allow for breakpoints to be taken.

If the location where execution begins contains a breakpoint TRAP, then that breakpoint is temporarily disabled until the program is stepped off of the breakpoint. If an assertion has been set then, the **GO** command will enable the TRACE buffer and check for break conditions after each instruction is executed. After the **GO** command has been issued, SmartMON will display one of the following messages indicating the mode of execution:

- ..RUNNING REAL-TIME
- ..RUNNING REAL-TIME WITH BREAKPOINTS
- ..TRACING (PC only)
- ..TRACING (PC only) WITH ASSERTIONS
- ..TRACING (PC and Data Movements)
- ..TRACING (PC and Data Movements)
 WITH ASSERTIONS

The last function performed, before starting the execution of user's code, is a call `SYS_GO`. This function is used to enable user specific operations that are suspended when SmartMON is active. Refer to the *System Control* section of the *Using SmartMON* chapter.

The *flag* argument is a way to combine the Trace Enable (**TE**) or Trace Disable (**TD**) commands with the **GO** command. This is optional and if not specified there will be no effect on the execution mode, other than conditions previously set. The tracing mode will remain in effect until it is specifically changed by another **GO** command or by the **TE** or **TD** commands.

Example

In this example, we will first examine the instructions located at address 10000 and then set a breakpoint and verify that it is the only condition setup. We will also disable tracing and issue the **GO** command, which will allow execution mode to operate in real-time. Notice the message displayed.

```
>DI 10000 F

0010000      2200      move.l      D0,D1
0010002      4282      clr.l       D2
0010004      D401      add.b       1,D2
0010006      E289      lsr.l       1,D1
0010008      66FA      bne.b      $10004
001000A      E20A      lsr.b      #1,D2
001000C      55C2      scs        D2
001000E      60FE      bra.b      $1000E

>SB 1000E

>DB

Code Break-Points

1. 01000E count = 0001 actual = 0000 enabled

>GO 10000

..RUNNING REAL-TIME WITH BREAKPOINTS

!BREAK! - Breakpoint at 001000E
>
```

In this example, we will assume only one address breakpoint is set at location 1000E. First, we issue a **GO** with trace enabled and then go again from the same place. Finally, disable tracing and go again. Notice that tracing is still enabled for the second **GO** command.

```
>GOT 10000
```

```
..TRACING (PC only) WITH ASSERTIONS
```

```
!BREAK! - Code Breakpoint at 001000E
```

```
>GO 10000
```

```
..TRACING (PC only) WITH ASSERTIONS
```

```
!BREAK! - Code Breakpoint at 001000E
```

```
>GOD 10000
```

```
..RUNNING REAL-TIME WITH BREAKPOINTS !
```

```
BREAK! - Breakpoint at 001000E
```

```
>
```



- DB** - Display Breakpoints
- DI** - Disassemble
- SB** - Set Breakpoint
- TE** - Trace Enable
- TD** - Trace Disable

GON

Function

Go to Next Instruction.

Syntax

GON

Description

This command causes SmartMON to set a temporary breakpoint at the address of the next instruction following the current instruction, then go into execution mode. This command is useful when debugging modular code because it allows a subroutine to execute without having to step through the code. The call `SYS_GO` is the last function performed before starting execution of user code.

Example

In this example, first show the user's code and a subroutine of that code, then set a breakpoint on the call to the subroutine. Execute up to that subroutine, then issue a **GON** command. This will skip stepping through the subroutine, but break upon its return.

```
>DI 6000 4

0006000          7003          moveq.l    #3,D1
0006002          7201          moveq.l    #1,D1
0006004          D4006FF8      bsr.w      $7000
0006008          E289          move.l     D0,D1
000600A          D401          add.b      D1,D2

>DI 7000 2

0007000          D081          add.l      D1,D0
0007002          4E75          rts

>SB 6004

>GO 6000

..RUNNING REAL-TIME WITH BREAKPOINTS

!BREAK! - Breakpoint at 6004
```

>GON

..RUNNING REAL-TIME WITH BREAKPOINTS

!BREAK! - Breakpoint at 6008

>



DB - Display Breakpoints

DI - Disassemble

GO - Start Execution

SB - Set Breakpoint

HE

Function

Help.

Syntax

HE [*cmd*]

cmd = An optional two-letter command name.
If omitted, the help menu will be displayed.

Description

This command causes SmartMON to display the help menu. This menu shows the syntax of the command set. Additional help for each command may be specified by passing the command name as an argument on the **HE** command line.

Example

In this example, we will display help for the Display Register (**DR**) command.

```
>HE DR
```

```
DR {register}{cr} = display registers
```

```
{register} if not specified all registers displayed  
>
```



This command is only available in the extended version of SmartMON.

IN

Function

Initialize.

Syntax

IN

Description

This command causes SmartMON to identify its version number, set default configuration parameters, and put SmartMON into command mode.

Example

For this example, we will show an **IN** (initialize sequence) for a Motorola 68000 based system.

```
>IN
```

```
SmartMON target: M68000 Version: 4.0
```

```
Copyright (c) 1997 Tasing, Inc., As Modified  
>
```

In this example, we will show an **IN** (initialize sequence) for a Motorola 68010 based system.

```
>IN
```

```
SmartMON target: M68010 Version: 4.0
```

```
Copyright (c) 1997 Tasing, Inc., As Modified  
>
```



Initialization of SmartMON does NOT imply initialization of the target system.

The revision numbers used in the examples are sample numbers for this manual only.

MM

Function

Modify Memory.

Syntax

MM[*unit*] *addr values*

unit = **B** = byte
 W = word (default)
 L = long

addr = The start address for the set, in hex.

values = A space-separated list of values to write to memory.

Description

This command causes SmartMON to fill memory with the hex byte values specified and without verification of the write (write only).

Example

In this example, we will display the contents of memory both before and after we set three bytes of memory to the specified values.

```
>DMB 200010 3
200010: FF 0F 00      ...
>MMB 200010 0B 7F 34
>DMB 200010 3
200010: 0B 7F 30      ..0
>
```



DM – Display Memory

RB

Function

Remove Breakpoint.

Syntax

RB[*type*] [*addr*]

type = **C** = code (default)
 D = data
 R = range

addr = An optional address that specifies the breakpoint to be removed.

If *type* and *addr* are omitted, then all breakpoints will be removed.

Description

This command causes SmartMON to remove a software breakpoint at the address specified. If no arguments are specified, then all breakpoints will be removed. If an address is specified, then the default argument is assumed to be a code breakpoint.

Example

In this example, first display some breakpoints that were previously setup and then remove a code breakpoint at address 10000. Finally, verify that the breakpoint has been removed. Notice that the second code breakpoint will become the first breakpoint after the RB command is executed.

```
>DB
```

```
Code Break-Points
```

```
1. 010000 count = 0002 actual = 0000 enabled  
2. 010010 count = 0001 actual = 0000 enabled
```

```
Data Break-Points
```

```
1. EA0000 data mask = 000ff cmp_flags = enabled  
   Word_cmp_BEQ  
>
```



```
>RBC 10000
```

```
>DB
```

```
Code Break-Points
```

```
1. 010010 count = 0001 actual = 0000 enabled
```

```
Data Break-Points
```

```
1. EA0000 data mask = 000ff cmp_flags = enabled  
   Word_cmp_BEQ
```

```
>
```



DB – Display Breakpoints

SB

Function

Set Conditional Breakpoints.

Syntax

For setting a conditional breakpoint and checking a register value:

SB *addr* [*count*] > **R** *reg* [*value*] [*condition*]

For setting a conditional breakpoint and checking a memory location:

SB *addr* [*count*] > **D** [*data_addr*] [*value*] [*condition*]

For setting a conditional breakpoint to enable/disable tracing:

SB *addr* [*count*] > { **TE** | **TD** }

addr = The address to set the breakpoint.

count = The number of times that the breakpoint must be encountered before testing for *condition*.

reg = The name of a target register to check.

value = The test value to compare.

data_addr = The memory location to compare against *value*.

condition = *unit type test*

unit = **B** = byte

W = word

L = long

type = **A** = and

C = compare

test = **E** = equal

N = not equal

The default condition is **LCE** – long compare break if equal.

TE, **TD** = Enable or disable trace respectively.

Description

This command causes SmartMON to set a conditional software breakpoint at the address specified. The conditional tests will be done when the address is encountered *count* times.

The following is a list of those conditions which may be checked:

- Check an ADDRESS or DATA REGISTER for specific value.
- Check a MEMORY location for a particular value.
- Enable Tracing.
- Disable Tracing.

When entering the execution mode, the mode of operation message, displayed by the **GO** command, will only indicate the mode that was started.

Example

In this example, we will set a conditional breakpoint to occur when the program counter reaches address 10000 and register D0 has a value of 000FFFFF. First set the conditional breakpoint and then issue the **GO** command.

```
>SB 10000 > R D0 FFFFFF

>GO FF00

..RUNNING REAL-TIME WITH BREAKPOINTS

!BREAK! - Breakpoint at 10000
>
```

In this example, we will set a conditional breakpoint to trace a specific subroutine's execution. First set a conditional breakpoint that will enable tracing during the subroutine and also set a conditional breakpoint that will disable tracing at the end of that subroutine. Finally, issue the **GO** command. Begin by displaying the code that will execute. Notice that when entering execution mode, the code will be running real-time; only during the subroutine will the code not run in real-time.

```
>DI 6000 5

0006000          7003          moveq.l    #3,D1
0006002          7201          moveq.l    #1,D1
0006004          D4006FF8      bsr.w      $7000
0006008          E289          move.l     D0,D1
000600A          D401          add.b      D1,D2
```

>DI 7000 4

0007000	D081	add.l	D1,D0
0007002	E289	move.l	D0,D1
0007004	D401	add.b	D1,D2
0007006	4E75	rts	

>SB 7000 > TE

>SB 7006 > TD

>SB 600A

>GO 6000

..RUNNING REAL-TIME WITH BREAKPOINTS

!BREAK! - Breakpoint at 600A

>DTB

4.	0007000	D081	add.l	D1,D0
3.	0007002	E289	move.l	D0,D1
2.	0007004	D401	add.b	D1,D2
1.	0007006	4E75	rts	

>



In the basic version of SmartMON, the **SB** command is translated to **SBC** and any option after *count* is treated as a syntax error.



DI - Disassemble
DT - Display Trace Buffer
GO - Start Execution

SBC

Function

Set Address Breakpoint.

Syntax

SBC *addr* [*count*]

addr = The code address for the breakpoint's location.

count = An optional count that specifies the number of times the breakpoint must be encountered before executing the breakpoint.

Description

This command causes SmartMON to set a software breakpoint at the address specified.

Example

In this example, set a breakpoint at address 10000. We do not want this breakpoint to be taken until this code has executed 16 (10 hex) times. Verify that the breakpoint exists and issue a **GO** command.

```
>SB 10000 10
```

```
>DB
```

```
Code Break-Points
```

```
1. 010000 count = 0010 actual = 0000 enabled
```

```
>GO
```

```
..RUNNING REAL-TIME WITH BREAKPOINTS
```

```
!BREAK! - Breakpoint at 10000
```

```
>
```



Since a code breakpoint is the default, SB may be used instead of SBC if desired.



DB – Display Breakpoints
GO – Start Execution

SBD

Function

Set Data Breakpoint.

Syntax

SBD *addr data condition*

addr = The data address to evaluate.

data = The data value to compare to see if the breakpoint should be executed.

condition = *unit type test*

unit = **B** = byte
 W = word
 L = long

type = **A** = and
 C = compare

test = **E** = equal
 N = not equal

The default is **LCE** – long compare break if equal.

Description

This command causes SmartMON to compare data at the address specified while tracing through execution mode. The data is compared against the value residing at the specified address using the conditions set forth by the arguments. The breakpoint will occur if the test condition is satisfied.

Example

In this example, we set a data breakpoint for the word data residing at address E00000. We would like to find the section of code that is overwriting a variable at this address. The breakpoint will be taken if the data changes to any value other than FFFF. First, display the contents of the variable and then set the data breakpoint, issue the **GO**, and verify the variable's contents.

```
>DMW E00000 1
E00000: FFFF          ....

>SBD E00000 FFFF WCN

>GO

..TRACING (PC only) WITH ASSERTIONS

!BREAK! - Data Breakpoint at F00440

>DMW E00000 1
E00000: FFFE          ....
>
```

In this example, we set a data breakpoint for the word data residing at address E00010. We would like to find what section of code is setting a bit flag at this address. Setup the breakpoint to occur if the least significant bit gets set. First, display the contents of the flag word and then set the data breakpoint, issue the **GO**, and verify the flag word's contents.

```
>DMW E00010 1
E00010: FFFE          ....

>SBD E00000 0001 WAE

>GO

..TRACING (PC only) WITH ASSERTIONS

!BREAK! - Data Breakpoint at F03510

>DMW E00000 1
E00010: FFFF          ....
>
```



DM – Display Memory
GO – Start Execution

SBR

Function

Set Data Range Breakpoint.

Syntax

SBR[*unit*] *addr low_data high_data [test]*

unit = **B** = byte
 W = word
 L = long (default)

addr = The data address to evaluate.

low_data = The low value of the data range.

high_data = The high value of the data range.

test = **E** = Equal or break on data outside range
 – no break on boundaries (default).
 N = Not equal or break on data inside range
 – break on boundaries.

Description

This command will set up a data breakpoint condition where the value at a specific location must fall either within or outside a certain range. The breakpoint will occur if the test condition is satisfied.

Example

In this example, we will set a data range breakpoint for the word data residing at address E00000. We would like to determine when a certain variable will exceed a specified limit. The breakpoint will be setup to occur when the value of this variable is outside a certain range. First, display the contents of the variable and then set the data range breakpoint, issue the **GO** command, and verify the variable's contents. Notice the values are long only.

```
>DML E00000 1
E00000: 0000A000          .....

>SBR E00000 9FFF B000 E

>GO

..TRACING (PC only) WITH ASSERTIONS

!BREAK! - Data Range Breakpoint at F01424

>DM E00000
E00000: 0000 B001 0000 0000 0000 0000 0000 0000.....
>
```

In this example, we will set a data range breakpoint for the word data residing at address E00000. We would like to determine when a certain variable will fall within our specified limit. The breakpoint will be taken when the value of this variable is inside a certain range. First, we will display the contents of the variable. Then we will set the data range breakpoint, issue the **GO** and verify the variable's contents.

```
>DML E00000 1
E00000: 0000A000          .....

>SBR E00000 AAAA B000 N<cr>

>GO

..TRACING (PC only) WITH ASSERTIONS

!BREAK! - Data Range Breakpoint at F0186A

>DM E00000
E00000: 0000 AAAB 0000 0000 0000 0000 0000 0000 .....
>
```



DM – Display Memory
GO – Start Execution

SI

Function

Single Step Instruction.

Syntax

SI [**DR**] [**DI**] [*count*]

DR = Display all registers, if specified.

DI = Disassemble the last instruction executed, if specified.

count = An optional number of target instructions to execute. The default is one instruction.

Description

This command causes SmartMON to go into execution mode with tracing enabled.

Example

In this example, we will single step the target for one instruction. We would like to see the contents of the registers and the instruction that was executed.

>SIDIDR

..TRACING (PC only) WITH ASSERTIONS

```
D0 = 00000000 D1 = 00000000 D2 = 00000000 D3 = 00000000
D4 = 00000000 D5 = 00000000 D6 = 00000000 D7 = 00000000
A0 = 00200000 A1 = 00000000 A2 = 00000000 A3 = 00000000
A4 = 00000000 A5 = 00000000 A6 = 00000000 A7 = 00000000
USP = 00000000 SSP = 00000000 PC = 00000000 SR = 0000
VBR = 00000000 SFC = 0000 DFC = 0000
```

```
MOVE.W #$7F00,(A0)
```



This command is only available in the extended version of SmartMON.

SM

Function

Set Memory.

Syntax

SM[*unit*] *addr values*

unit = **B** = byte
 W = word (default)
 L = long

addr = The start address for the set, in hex.

values = A space-separated list of values to write to memory.

Description

This command causes SmartMON to fill memory with the hex byte values specified.

Example

In this example, we will display the contents of memory both before and after we set three bytes of memory to the specified values.

```
>DMB 200010 3

200010: FF 0F 00      ...

>SMB 200010 0B 7F 34

>DMB 200010 3

200010: 0B 7F 34      ..4
>
```



DM – Display Memory

SO

Function

Step Out of Address Range.

Syntax

SO *start end*

start = The starting address for the range.

end = The ending address for the range.

Description

This command causes SmartMON to go into execution mode with trace enabled. The execution mode will continue until any of the following conditions occurs: a breakpoint is encountered, an instruction outside of the specified range is about to be executed, or an interrupt is received from the host.

Example

In this example, we will single step the code until the program falls outside the specified range.

```
>SO 0001000 0001fff
..TRACING (PC only) WITH ASSERTIONS
!BREAK! - Code Step out of Range at 0002154
>
```

SR

Function

Set Registers.

Syntax

SR[*unit*] *reg value* [*reg value ...*]

unit = **B** = byte
 W = word
 L = long (default)

reg = A valid target register (you may not modify the value of A7).
 The keyword **all** may be specified to mean all registers other
 than A7.

value = A hex value to write to the register.

Description

This command causes SmartMON to modify the register or registers specified.

Example

In this example, we will set address register A0 and data register D2 with word values. Then display these registers to verify their contents. Notice that only the lower word of the register will be affected by this command.

```
>DR A0 D2
```

```
A0 = 00000000 D2 = FFFFFFFF
```

```
>SR A0 005B D2 7F34
```

```
>DR A0 D2
```

```
A0 = 0000005B D2 = FFFF7F34
```

```
>
```



DR – Display Registers

SS

Function

Search for String.

Syntax

SS[*unit*] *addr range string*

unit = **A** = ASCII (default)
 H = hex

addr = The starting address of the search.

range = The size of search in bytes.

string = The string to be searched for.

Description

This command allows the user to search for a string in memory.

Example

In this example, we will search for the ASCII string "hello" for 32 (20 hex) memory locations. Find one string and continue the search until all the locations have been checked. There will be only one occurrence of this string in the memory that will be searched.

```
>SSA 10000 20 hello
```

```
String found at 1000f
```

```
><CR>
```

```
String not found
```



This command is only available in the extended version of SmartMON.

TD

Function

Trace Disable.

Syntax

TD

Description

This command causes SmartMON to disable tracing of target execution code. Tracing may not be disabled if assertions are set. Programs will run in real-time when tracing is disabled.

Example

In this example, we will assume no other assertions are set. Disable trace and issue a **GO** command. Execution mode will run in real-time.

```
>TD
>GO
..RUNNING REAL-TIME
```

In this example, we will assume that another assertion is set. Disable trace and issue a **GO** command. Execution mode will NOT run in real-time.

```
>TD
>GO
..TRACING (PC only) WITH ASSERTIONS
```



GO – Start Execution

TE

Function

Trace Enable.

Syntax

TE

Description

This command causes SmartMON to enable tracing of target execution code.



The default configuration parameter for tracing is trace the PC history only. If tracing with full data movements is enabled (see the **CF** command), then the data associated with “move” type commands will be saved in the trace buffer. Once the trace parameter has been selected, all subsequent **TE** commands will be in that mode.



CF – Configure

UD

Function

User Diagnostics.

Syntax

UD[*type*] *number*

type = **C** = run test continuously
 H = run test continuously, halt on error
 I = install RAM diagnostics

The default is to run tests once.

number = *n* – number of tests to be run
 A – all tests

Description

SmartMON allows the user to create his own diagnostics to run under the monitor. See the *Diagnostics* chapter for more information.



This command is only available in the extended version of SmartMON.

6 SYSTEM CALLS

This chapter is a reference for the system calls used to access selected functional routines contained within SmartMON.

6.1 INTRODUCTION

System calls can be used to access selected functional routines contained within SmartMON. The access to SmartMON is through the user defined TRAP# called RM_TRP (USREQU.68K). In order to select the appropriate function, a system call code is loaded into D0 and a trap is then made to SmartMON. SmartMON decodes the value passed in D0 and takes the appropriate action.

System calls are used by the user-supplied I/O driver. The serial port ISR informs SmartMON of a pending character or buffer empty after transmission of a character. System calls may also be used by the user's application code to access communication services.

The following information includes a description of the system calls, the function codes, its return values, and any necessary guidelines.



Not all system calls are available in the basic version of SmartMON.

EVT_COPY

Function

A user may wish to create a separate vector table in memory to contain its exception vectors. If the user wishes to use SmartMON services, he must copy over those vector addresses used by SmartMON.

When this system call is issued, SmartMON will copy the current EVT (Exception Vector Table) to a location starting at an address specified in A0. When SmartMON copies it performs a write read verify. If this check fails an error is returned.

Example

```
LEA      start evt,A0          ;load address pointer
MOVE.W   #EVT_COPY,D0         ;load system call
TRAP     #ROMM                 ;trap
```

Input

```
D0      =  EVT_COPY 000A
A0      =  address of new EVT
```

Output

```
D0      =  return code
```

Returns

```
0000    =  RET OK successful return
0001    =  FAIL unable to write EVT
```



VBR must not be changed until after this system call.

IN_CHAR

Function

When this system call is issued SmartMON will read a ">" character from the input buffer. SmartMON returns the next character from its input buffer.

If there is no character present in the buffer, IN_CHAR returns the error code NO_CHAR.

Example

```
MOVE.W    #IN_CHAR,D0
TRAP      #ROMM           ;trap
```

Input

D0 = IN_CHAR 0010

Output

D0 = return code
D1[7:0] = character

Returns

0000 = RET OK successful return
0001 = NO_CHAR

IN_STR

Function

When this system call is issued SmartMON will read a string from the input buffer. SmartMON places the string in a buffer pointed to by an address in A0. A string consists of a number of characters terminated with a <cr>.

If a complete string is not present in the buffer, IN_STR returns the error code NO_STR.

Example

```
LEA      buff_ptr,A0          ;load address pointer
MOVE.W   #IN_STR,D0
TRAP     #ROMM                ;trap
```

Input

D0 = IN_STR 0011
A0 = address of buffer

Output

D0 = return code

Returns

0000 = RET OK successful return
0001 = NO_STR



This function is only available in the extended version of SmartMON.

INT_COMP

Function

This call exits an ISR (interrupt service routine). When you `INT_ENTER` at the start of the ISR, you must terminate the ISR with this call. Refer to `INT_ENTER` for enter interrupt service handler. There is no return to the interrupt service routine for this system call. Execution resumes wherever code was executing previous to the interrupt service routine's execution.

Input

D0 = INT_COMP 0002

Output

no return is possible



Because register D0 is needed to make the `INT_COMP` call to SmartMON, it is assumed that the original value of D0 saved at the start of the ISR is now on top of the stack. The stack must have the following format when calling `INT_COMP`:

```
                D0
ISP OR SSP-->  SR
                PC
                Format I/D *
```

*MC68010 and 68020 only

INT_ENTER

Function

This call is used to signal SmartMON that an ISR has been entered. INT_COMP must be used to exit the ISR that begins with INT_ENTER. Refer to the INT_COMP command in this section.

Example

```
MOVE.W    D0, -(sp)           ;save D0
MOVE.W    #INT_ENTER, D0      ;load D0
TRAP      #ROMM               ;trap
```

Input

D0 = INT_ENTER 0001

Output

D0 = return code

Returns

0000 = RET OK successful return



Your ISR must save the contents of register D0 of size word onto the stack before this call is made.

INT_RX

Function

An ISR uses this call to transfer each character to SmartMON as it is received from the supported I/O device.

Input

D0 = INT_RX 0004

Output

D0 = return code

Returns

0000 = RET OK successful return

INT_TX

Function

An ISR uses this call to inform SmartMON that it is ready to transmit another character to the supported I/O device. SmartMON returns the next character from its output buffer to the ISR.

If there is no character present in the buffer, INT_TX returns the error code of 0001, which is a NO_CHAR return code.

Input

D0 = INT_TX 0003

Output

D0 = return code
D1[7:0] = character

Returns

0000 = RET OK successful return
0001 = NO_CHAR

OUT_CHAR

Function

When this system call is issued SmartMON will write a character out to the serial port.

Example

```
MOVE.B    CHAR,D1
MOVE.W    #OUT_CHAR,D0
TRAP      #ROMM           ;trap
```

Input

D0	=	OUT_CHAR 0013
D1	=	character to be transmitted

Output

D0	=	return code
----	---	-------------

Returns

0000	=	RET OK successful return
------	---	--------------------------

OUT_DATA

Function

When this system call is issued SmartMON will read data from the buffer pointed to by an address in A0 for a count contained in D1. The data is converted to ASCII hex before transmission.

Example

```
LEA      buff_ptr,A0      ;load address pointer
MOVE.W   COUNT,D1        ;load the number of bytes
MOVE.W   #OUT_DATA,D0
TRAP     #ROMM            ;trap
```

```
buff_ptr: dc.b $24,$48,$fe
```

Input

D0 = OUT_DATA 0015
A0 = address of buffer

Output

D0 = return code

Returns

0000 = RET OK successful return
0001 = NO STR



This function is only available in the extended version of SmartMON.

OUT_STR

Function

When this system call is issued SmartMON will read a string from the a buffer pointed to by an address in A0. A string consists of a number of characters terminated with a null.

If more than 254 characters are present without a null, OUT_STR returns the error code STR_TO_LONG.

Example

```
LEA      buff_ptr,A0          ;load address pointer
MOVE.W   #OUT_STR,D0
TRAP     #ROMM                ;trap
```

```
buff_ptr: dc.b "Hello", $a,$d,$o
```

Input

```
D0      = OUT_STR 0014
A0      = address of buffer
```

Output

```
D0      = return code
```

Returns

```
0000    = RET OK successful return
0001    = STR TO LONG
```



This function is only available in the extended version of SmartMON.

RD_STR

Function

When this system call is issued SmartMON will read a string from the input buffer. SmartMON places the string in a buffer pointed to by an address in A0. A string consists of a number of characters terminated with a <cr>.

This function does not return to the caller until a <cr> is received.

Example

```
LEA      buff ptr,A0          ;load address pointer
MOVE.W   #RD_STR,D0
TRAP     #ROMM                ;trap
```

Input

D0 = RD_STR 0012
A0 = address of buffer

Output

D0 = return code

Returns

0000 = RET OK successful return



This function is only available in the extended version of SmartMON.

ROMM_GO

Function

When this system call is issued SmartMON becomes active and a ">" will appear on the terminal. This call is typically made at the end of RMAIN.68K, but may be placed anywhere in user code to invoke SmartMON.

Example

```
MOVE.W    D0, -(sp)           ;save D0
MOVE.W    #ROMM_GO, D0        ;load D0
TRAP      #ROMM               ;trap
```

Input

D0 = ROMM_GO 0005

Output

D0 = no return possible

Returns

0000 = RET OK successful return

7 DIAGNOSTICS

This chapter describes diagnostic functions which are utility routines and special tests that are useful for exposing memory problems. Diagnostics can be performed both by using SmartMON diagnostic commands or by using user-written custom diagnostic routines. This chapter includes the following major sections:

- SmartMON Diagnostics
- User Diagnostics



Diagnostic functions and their utilities are only available in the extended version of SmartMON.

7.1 SMARTMON DIAGNOSTICS

7.1.1 OVERVIEW

The diagnostic functions are a group of utility routines and special tests. They are useful for exposing memory problems and giving the user the ability to write and execute custom diagnostics, both ROM and RAM based.

SmartMON has provided RAM tests, scope loops, and CRC tests for basic conveniences. For integrating customized diagnostics, see the *User Diagnostics* section later in this chapter. The rest of this section will describe the tests that are already incorporated into SmartMON.

7.1.2 RAM TESTS

These pre written RAM tests check to see if target memory is operating properly. Simple and complete RAM tests are supported in both single pass and continuous pass modes.

DFO

Function

Simple RAM Test, Single Pass.

Syntax

DF[*unit*] **0** *start* *end*

unit = **B** = byte
 W = word (default)
 L = long

start = The starting memory location to test.

end = The ending memory location to test.

Description

Run a simple RAM test for a single pass.

Example

A simple RAM test will be executed on a 128 word memory space. There will be no errors.

```
>DFW 0 00000 000FF
```

```
>complete
```

A simple RAM test will be executed on a 128 word memory space. There will a bad memory bit 12 at address 0000E. A bit will be stuck low.

```
>DFW 0 00000 000FF
```

```
memory failure: $0000E=$4555 not $5555
```

```
complete
```

```
>
```


DF1

Function

Complete RAM Test, Single Pass.

Syntax

DF[*unit*] **1** *start end*

unit = **B** = byte
 W = word (default)
 L = long

start = The starting memory location to test.

end = The ending memory location to test.

Description

Run a complete RAM test for a single pass.

Example

A complete RAM test will be executed on a 128 word memory space.
 There will be no errors.

```
>DFW 1 00000 000FF
```

```
>complete
```

A complete RAM test will be executed on a 128 word memory space.
 There will a bad memory bit 12 at address 0000E. A bit will be stuck low.

```
>DFW 1 00000 000FF
```

```
memory failure: $0000E=$efff not $ffff
```

```
memory failure: $0000E=$0000 not $1000
```

```
complete
```

```
>
```

DF2

Function

Simple RAM Test, Continuous.

Syntax

DF[*unit*] **2** *start end*

unit = **B** = byte
 W = word (default)
 L = long

start = The starting memory location to test.

end = The ending memory location to test.

Description

Run a simple RAM test continuously. This test can be stopped by an interrupt control character (^C) from the host.

Example

A simple RAM test will be executed continuously on a 128 word memory space. There will be no errors.

```
>DFW 2 00000 000FF
TEST IS LOOPING PRESS ^C TO ABORT
NUMBER OF COMPLETE LOOPS = $XXXX
```

A simple RAM test will be executed continuously on a 128 word memory space. There will a bad memory bit 12 at address 0000E. A bit will be stuck low.

```
>DFW 2 00000 000FF
TEST IS LOOPING PRESS ^C TO ABORT
memory failure: $0000E=$4555 not $5555
PASS# = 1
memory failure: $0000E=$4555 not $5555
PASS# = 2
UNTIL ^C
```

DF3

Function

Complete RAM Test, Continuous.

Syntax

DF[*unit*] **3** *start* *end*

unit = **B** = byte
 W = word (default)
 L = long

start = The starting memory location to test.

end = The ending memory location to test.

Description

Run a complete RAM test continuously. This test can be stopped by an interrupt control character (^C) from the host.

A complete RAM test will be executed continuously on a 128 word memory space. There will be no errors.

```
>DFW 3 00000 000FF
```

```
TEST IS LOOPING PRESS ^C TO ABORT
```

```
NUMBER OF COMPLETE LOOPS = $XXXX
```

A complete RAM test will be executed continuously on a 128 word memory space. There will a bad memory bit 12 at address 0000E.

```
>DFB 3 00000 000FF
```

```
TEST IS LOOPING PRESS ^C TO ABORT
```

```
memory failure: $0000F=$ef not $ff memory failure:
$0000F=$00 not $10
```

```
PASS# = 1
```

```
memory failure: $0000F=$ef not $ff memory failure:
$0000F=$00 not $10
```

```
PASS# = 2
```

```
...
```

DF4

Function

CRC Text.

Syntax

DF 4 *start end*

start = The starting memory location to test.

end = The ending memory location to test.

Description

Run a CRC test over a specified range of memory.

Example

A CRC test will be executed on a 128 word memory space.

```
>DF 4 00000 000FF
```

```
CRC = 437B
```

DF5

Function

Scope Loop: Read from Location.

Syntax

DF[*unit*] **5** *addr*

unit = **B** = byte
 W = word (default)
 L = long

addr = The memory location to test.

Description

The scope loop routines are useful when troubleshooting with an oscilloscope. Read/write continuously from/to an address, and write then read data are supported routines.

This command will continuously read from the specified address. This test can be stopped by an interrupt control character (^C).

Example

Scope loop reading a location.

```
>DF 5 E80000
```

```
DF 5 reading a location press ^C to abort
```

DF6

Function

Scope Loop: Write to Location.

Syntax

DF[*unit*] **6** *addr* *value*

unit = **B** = byte
 W = word (default)
 L = long

addr = The memory location to test.

value = The hex value to write to *addr*.

Description

Continuously write a specified pattern to the address specified. This test can be stopped by an interrupt control character (^C).

Example

Scope loop writing data to a location.

```
>DF 6 E80000 5555
```

```
DF 6 writing a location press ^C to abort
```

DF7

Function

Scope Loop: Write and Compliment.

Syntax

DF[*unit*] 7 *addr* *value*

unit = **B** = byte
 W = word (default)
 L = long

addr = The memory location to test.

value = The hex value to write to *addr*.

Description

Consecutively write a specified pattern to the address specified and then write its complement. This test can be stopped by an interrupt control character (^C).

Example

Scope loop writing data then its complement to a location.

```
>DF 7 E80000 5555
```

```
DF 7 writing value then complementing  
press ^C to abort
```

DF8

Function

Scope Loop: Write Rotating Value.

Syntax

DF[*unit*] **8** *addr* *value*

unit = **B** = byte
 W = word (default)
 L = long

addr = The memory location to test.

value = The hex value to write to *addr*.

Description

Write a specified pattern to the address specified and rotate the pattern.
This test can be stopped by an interrupt control character (^C).

Example

Scope loop writing rotating data to a location.

```
>DF 8 E80000 0001
```

```
DF 8 writing value then rotating press ^C to abort
```


DF9

Function

Scope Loop: Write then Read.

Syntax

DF[*unit*] **9** *addr* *value*

unit = **B** = byte
 W = word (default)
 L = long

addr = The memory location to test.

value = The hex value to write to *addr*.

Description

Write a specified pattern to the address specified and then read it back.
 This test can be stopped by an interrupt control character (^C).

Example

Scope loop writing then reading data to/from a location.

```
>DF 9 E80000 5555
```

```
DF 9 writing value then read press ^C to abort
```

7.2 USER DIAGNOSTICS

7.2.1 OVERVIEW

In the area of manufacturing, SmartMON's ability to control and execute user diagnostics makes an ideal interface between the manufacturing technician and the diagnostics. These services create a menu of available tests with the ability to execute the tests individually or as a set. The debugging tools also allow the diagnostic engineer to debug his diagnostics in the same environment they will eventually run in. SmartMON supplies the following services to support these diagnostics:

- A menu listing of all user diagnostics.
- The ability to run a test or all the tests once or continuously with an option to halt on error.
- The ability to report errors to the host.

The user can include up to eight diagnostics as part of SmartMON image which will be linked and burned into the target EPROMS. This allows the user to select and run these tests under SmartMON.

SmartMON also supports eight downloaded diagnostics. After the diagnostics have been downloaded, a UDI (User Diagnostic Install) command with the address of the diagnostic table is issued. SmartMON will then install these diagnostics into the UD menu. Figure Rom-9 contains the layout of the diagnostic table structure.

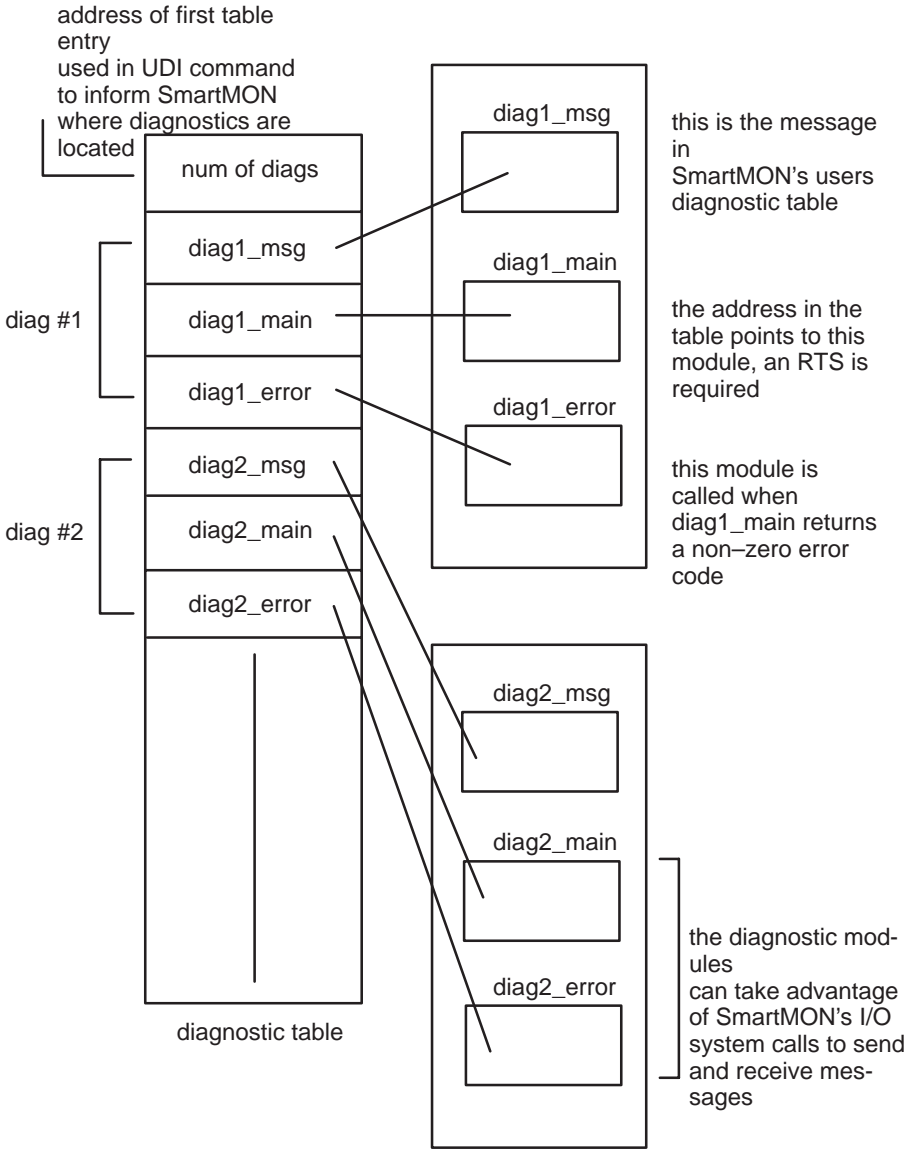


Figure Rom-9: Diagnostic table structure

User Diagnostic Commands

Syntax

UD*type number*

type = **C** = Run test continuously.
 H = Run test continuously halt on error.
 I = Install RAM diagnostics.
 Default is to run test once.

number = # number of test to be run
 A = all tests

If *type* and *number* are omitted, then a list of available tests is displayed.

Example

In this example, we will display the list of available tests on the target.

```
>UD
                                USER DIAGNOSTICS

                                ROM BASED                                RAM BASED

1.  USER DIAGNOSTIC MEMORY TEST    9.  not installed
2.  TEST TWO                        A.  not installed
3.  not available                   B.  not installed
4.  not available                   C.  not installed
5.  not available                   D.  not installed
6.  not available                   E.  not installed
7.  not available                   F.  not installed
8.  not available                   10. not installed

type UD<type> <test number> to execute test

TO install downloaded diagnostics type UDI <address>
      <address> = address of RAM DIAG_TABLE
```

In this example, we will run test #1 once. There are no errors found during this test.

```
>UD 1

RUNNING TEST NUMBER 1 :- USER DIAGNOSTIC
MEMORY TEST

COMPLETE
```

In this example we will run test #1 once. An error will be found during this test. The message “TEST FAILED” will be printed by SmartMON, while the message following it is printed out by the user’s error routine.

```
>UD 1

RUNNING TEST NUMBER 1 :- USER DIAGNOSTIC
MEMORY TEST
TEST FAILED
ERROR MESSAGE:- TEST FAILED DURING READ
>
```

In this example, we will run all tests continuously. When each test is about to be executed, its menu message will be printed by SmartMON. These tests can be stopped the interrupt control character (^C).

```
>UDC A

RUNNING TEST NUMBER 1 :- USER DIAGNOSTIC
MEMORY TEST
```

7.2.2 HOW TO WRITE A USER DIAGNOSTIC

The user creates each test in three sections: the diagnostic message, the diagnostic test and the error reporting routine. The starting addresses of each of these sections are then placed in the diagnostic table. In order for SmartMON to access the diagnostics the starting address of this table is required. For ROM based diagnostics, this table address is specified in the configuration table found in USREQU.68K. After downloading RAM based diagnostics, the table’s address is specified by issuing the UDI command with the table’s address. The first value in each of these tables is the number of diagnostics to follow. Each diagnostic must have three entries in the table (an address for each of the three sections in the diagnostic) as follows:

1. TST MSG – Diagnostic Message

2. TST MAIN – Diagnostic Test Routine
3. TST ERR – Error Reporting Routine

Diagnostic Messages

This is the first section of a diagnostic and simply contains a message. This message is used by SmartMON to describe the test. The message can be up to 30 characters long and must be terminated with a null character.

The following message will be printed out by SmartMON when either the UD menu is selected or when the test is executed. This is an example of a test message defined:

```
TST1 MSG: DC.B "USER DIAGNOSTIC MEMORY TEST",0
```

Diagnostic Test

This is the actual test to be executed. The test routine must be terminated with an RTS and the error code returned in D0.

The following is an example of what the end of a diagnostic test might look like:

```
TST1 MAIN
      :      :
      BEQ.S   TST1 GOOD   ; any errors found ?
      MOVE.B  #01,D0      ; yes, errors found
      BRA.S   TST1 END    ; skip good return
TST1 GOOD MOVE.B  #00,D0   ; no, test passed
TST1 END   RTS
```

Error Reporting

This part of the test allows the user to print additional information about a particular failure. This function is called by SmartMON when the diagnostic test produced an error. The only exception is when the test is to run in continuous mode and not halt on errors. If the user does not wish to produce additional information, this function need only contain an RTS.

This is an example of an error reporting routine. It will check for a valid user-defined error code and will print a corresponding error message. This routine will only be called if its diagnostic test has failed:

```
TST1 RSP:

RSP1 CMPI      01,D0      ;check if error 1 occurred
      BNE      RSP2      ;no keep checking
      MOVE     #OUT_STR,d0 ;setup for string xmit
      LEA      ERROR 1,A0 ;point to error message
      TRAP     #ROMM      ;make system call
      RTS      ;exit

RSP2 CMPI      02,D0      ;check if error 2 occurred
      BNE      RSP3      ;not a valid error code
      MOVE     #OUT_STR,D0 ;setup for string xmit
      LEA      $ERROR 2,A0 ;point to error message
      TRAP     #ROMM      ;make system call
RSP3 RTS      ;exit

ERROR 1:      DC.B  "TEST FAILED DURING READ",$a,$d,0

ERROR 2:      DC.B  "TEST FAILED DURING WRITE"$a,$d,0
```

7.2.3 LINKING DIAGNOSTICS WITH SMARTMON

For SmartMON to know what user diagnostics are available, the following data structure must be created:

```
      XDEF      DIAG_TABLE

DIAG_TABLE: DC.W  0001      ;number of tests = 1
              DC.1  TST1 MSG ;function containing msg
              DC.1  TST1 MAIN ;diagnostic test
              DC.1  TST1 RSP  ;error routine
```

This table must be available even if no diagnostics are to be included, in which case, the table must look as follows:

```
      XDEF      DIAG_TABLE

DIAG_TABLE:      DC.W  0000      ;number of tests = 0
```

7.2.4 DOWNLOADING AND RUNNING USER DIAGNOSTICS

When diagnostics are downloaded, SmartMON has to be informed of their location in memory. This is done by creating the diagnostic table as described above which becomes part of the downloaded image. In this case, the diagnostics are not linked with SmartMON. In order for SmartMON to recognize the diagnostics, the starting address of the diagnostic table (RAM_DIAG) must be known. To define a location where you would like the diagnostic table to reside, add the following line to your locate command file:

```
LOCATE (RAM_DIAG : #<address>)
```

These tests can then be included in SmartMON's diagnostic menu by issuing the UDI command with the address of the DIAG_TABLE. The UDI <address> command takes the table and loads the diagnostic names into SmartMON's UD menu, after which these tests can be selected and run. Unlike the ROM based diagnostics, if the user chooses not to use downloadable diagnostics, a DIAG_TABLE with test number set to 0 does not have to exist.

7.2.5 HOW SMARTMON PROCESSES UD COMMANDS

When SmartMON is initialized, part of the process is to install the ROM based diagnostics. This is done by examining the DIAG_TABLE to determine how many diagnostics are available. For each diagnostic available, SmartMON takes the diagnostic message and places it in the menu, allowing the user to display the available tests with a UD command.

7.2.6 INSTALLING RAM BASED DIAGNOSTICS

When the UDI <address> command is issued, SmartMON goes to the address specified and examines the DIAG_TABLE to determine how many diagnostics are available. For each diagnostic available, SmartMON takes the diagnostic message and places it in the menu, allowing the user to display the available tests with a UD command.

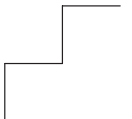
7.2.7 RUNNING A TEST

Entering a UD 1 command causes SmartMON to print out a test running message and the test message located at address TST1 MSG. SmartMON then calls the TST1 MAIN diagnostic and the test will execute. Upon returning to SmartMON, the error code in register D0 is checked and a message is displayed indicating the test passed or failed. If a non-zero value was returned in D0, TST1 ERR the error routine is then called and the error code is also passed in D0. This routine may now print out any additional information using system calls. Upon completion the routine is terminated with a RTS.

If a UDC 1 command had been entered, then TST1 MAIN would be called repeatedly and the PASS COUNT would be updated. If an error is encountered the error message is displayed and the test is halted. Entering a <cr> will continue execution of the test.

ADDENDUM

BACKGROUND DEBUG MODE



ADDENDUM

1 INTRODUCTION

This addendum explains how to use CrossView Pro with the Background Debug Mode (BDM), a special feature available on Motorola CPU32 family processors. It includes the following major sections:

- Background Debug Mode as a CrossView Pro Execution Environment
- BDM Hardware and Software Installation
- BDM Command Interface (Emulator Mode)
- Troubleshooting
- Other Considerations at this Time

2 BACKGROUND DEBUG MODE AS A CROSSVIEW PRO EXECUTION ENVIRONMENT

Background Debug Mode (BDM) is a special feature available on the Motorola CPU32 family processors. The feature is implemented in CPU microcode and incorporates a full set of debug options. The BDM is documented in the *Development Support* section of Motorola's *CPU32 Reference Manual*.



CrossView Pro for BDM is only available on MS-Windows.

Additional System Requirements

The following are the hardware requirements:

- Parallel printer port (LPT1, LPT2, or LPT3)
- Macraigor Wiggler cable (optional with BDM version of CrossView Pro for 68K processors)
- P&E ClodFire BDM cable (optional with BDM version of CrossView Pro for ColdFire processors)
- CPU32 or CPU32+ or ColdFire target (MC68330, MC68331, MC68332, MC68333, MC68340, or MC68360, MCF5102, etc)

2.1 ADDITIONAL SOFTWARE CONTENTS

Chip Select Initialization Files

CPU32 targets have a programmable chip-select sub-module. After a power-on reset, you must initialize the chip-select registers. The initialization values depend on memory map, wait states required, and other properties of the target hardware. The CrossView Pro delivery contains chip-select initialization files for many popular target boards.

TASKING is constantly adding support for new targets. If you do not see an initialization file for your target, contact your TASKING Sales Engineer for updated information. This release includes Chip Select Initialization for the following targets:

Target Board	Filename
EST SBC 360	cse360.cmd
Matrix 360	csmat360.cmd
Motorola EVK332	csevk2.cmd
Motorola EVK340	csm340.cmd
Motorola MPD16/32	csmpd32.cmd
Motorola QUADS	mot_quad.cmd
Vesta SBC332	csv332.cmd
AVNET MCF5282	av_mcf5282.cmd
NetBurner MCF5206E	nb_mcf5206e.cmd

3 BDM INSTALLATION

3.1 HARDWARE INSTALLATION

The hardware installation consists of connecting a Macraigor Wiggler cable or P&E cable from your PC to the target board.



It is strongly recommended that both the PC and the target board be powered off during the installation. It is also strongly recommended by Motorola that installation to the BDM connection follow electrostatic conventions to prevent damage to the target CPU.

The power off sequence should be as follows:

1. Power off the target board first.
2. Power off the PC using the procedure recommended by the PC manufacturer.

The power on sequence should be as follows:

1. Power on the PC first.
2. Then power on the target board.

The Macraigor Wiggler cable (for 68K processors) is a DB25 connector with a ribbon cable that has a 10 pin in-line connector. The DB25 connects to the PC parallel port (usually labelled LPT1, LPT2, or LPT3).

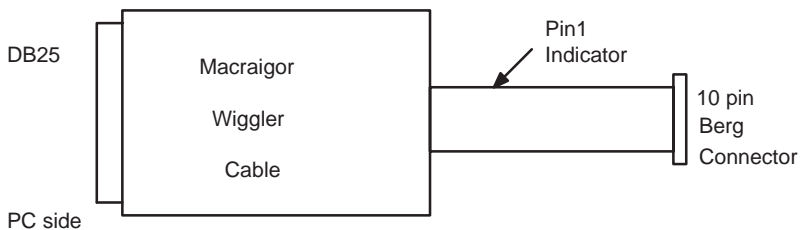


Figure Bdm-1: Macraigor Wiggler Cable



The P&E ColdFire BDM cable has a ribbon cable with a 26 pin connector. Its use is identical to the Macraigor Wiggler cable. For additional information on this cable see <http://www.pemicro.com>.

The 10/26 pin Berg connector must be connected in the correct orientation.



Some 68K targets use a 10 pin connector while others use an 8 pin connector.

On the 10 pin connector, simply plug the cable in with Pin 1 connected to Pin 1.

On targets that use 8 pin Berg connectors, the cable needs to be shifted so that Pin 3 of the cable is connected to Pin 1 of the target.

If the ribbon cable is not connected to the DB25 housing, reconnect it so that the Pin 1 indicator is near the center of the DB25 housing. Note that the Pin 1 indicator in the 10 pin plug should be on top.

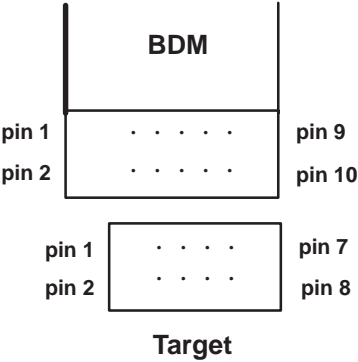


Figure Bdm-2: 68K BDM Cable Connection

If the ICD cable is not long enough to reach between the PC and the target board, use a standard printer extension cable or a direct 25 to 25 pin connector.

3.2 SOFTWARE INSTALLATION

By default the Macraigor or P&E drivers are installed during the TASKING 68K/ColdFire installation.

3.3 CONFIGURATION OPTIONS

Target Configuration File

In the CrossView Pro Target Settings dialog box make sure you have selected the correct Target configuration: **68KBDM Wiggler** or **ColdFire BDM**.

Macraigor Wigglers (68K) / P&E (ColdFire)

The CrossView Pro installation performs all the required installation for the device drivers on MS-Windows. You need to specify the parallel port to which the BDM interface is connected (LPT1, LPT2, or LPT3). This is done in the CrossView Pro Communication Dialog box. By default CrossView Pro uses LPT1.

3.4 TARGET ENVIRONMENT SETUP

BDM provides faster and less intrusive debugging than typical ROM-based debuggers and in-circuit emulators. With BDM, the user's target runs in real-time between breakpoints and does not require any target resources (ROM, RAM, interrupts, etc.).

As with an in-circuit emulator, the memory for CPU32 targets must be mapped after a target board power-up. This is because the hardware reset clears any chip selects to the default values. In order for CrossView Pro to download and debug, the CPU32 chip selects must first be set to reflect the hardware memory configuration.

The initialization of chip selects and other memory map options should be added to the startup code for your application (the `__main` routine in the standard TASKING 68K/ColdFire libraries). However, until you have written the necessary startup code, you can perform initialization through BDM commands in CrossView Pro's emulator mode.

Sample playback files are provided as part of the CrossView Pro distribution. The sample in the table below (`csv332.cmd`) was developed for a Vesta SBC332 with 128K of RAM. For details on the commands see the *Command Descriptions* section later in this chapter. To playback a chip-select initialization file:

1. From the **Tools** menu, select **Playback | Emulator...** to open the Emulator Playback dialog box.

- 2. Type the playback filename or use the **Browse...** button to select the file. The default filename extension is `.cmd`.
- 3. Enable the **Continuous playback** check box.
- 4. Click on the **Execute** button to start the playback.

Command	Explanation
rs	Reset the target and enable Background Debug Mode. This also causes the memory map to be reset.
sr a6 0	Clear the stack frame pointer. This prevents CrossView Pro from trying to decode invalid stack frames.
smw fffffa20 000d	Configure SYPCR to disable Software Watchdog Timer (WDT), enable bus monitor functions, and use 16MHz system clock for timing.
smw fffffa04 7f00	Set SYNCR to default.
smw fffffa00 424f	Set MCR to allow the break signal to freeze timers (including WDT), configure bus monitor to enable external arbitration, and allow system registers to be set in user mode.
smw fffffa4c 0004 smw fffffa50 0004	Set up chip selects for 128KB RAM starting at 0.
smw fffffa48 0604	Set up chip selects for 128KB ROM at 60000.
sml fffffa44 ffffffff	Enable all chip select pins.
smw fffffa4e 5830 smw fffffa52 3830	Chip select for RAM: read/write no wait states.
smw fffffa4a 78f0	Chip select for EPROM: read-only, 3 wait states.

Table Bdm-1: Sample Chip-select Initialization File for Vesta SBC332

4 BDM COMMAND INTERFACE (EMULATOR MODE)

BDM driver communication is performed via the `SendDriverMessage` function within the Windows API. The driver has a 300-character input buffer for input commands. It will send back a complete response, except for the Dump Memory command. The Dump Memory command will restrict the output to 256 memory units (approximately 2560 bytes).

Control Characters

The following control codes may be entered for command line editing, interrupting BDM.



The presence of the upward caret, "^", before a character indicates that the Control or CTRL must be held down while striking the character key.

- | | |
|------------------|---|
| ^C (interrupt) | This character will terminate any operation, flush the character buffer and return a response prompt. This includes returning the target back to Background Mode. |
| ^J (CR) | This returns a prompt. |
| ^X (line delete) | This acts the same as ^C. |

4.1 OPERATION MODES

BDM has three modes of operation, each of which changes the user interface slightly. The rest of this section will describe these modes, which are as follows:

- Command Mode
- Download Mode
- Execution Mode

Command Mode

The target does not execute instructions in this mode. Commands typed in at the CrossView Pro emulator mode are interpreted by the BDM driver. Command Mode is the user's interface to the target. The user can control resources and place the target into a known state. Most interactions take place in command mode, which is distinguishable by its prompt. For more details on all the commands see the *Command Descriptions* section.

Download Mode

This mode is used to send down the user's application code, in Motorola S-Record format, to the target's RAM. Download mode is used mostly during the debug stages of a project. This mode is entered through the Download command (**DL**) and stays in effect until either the S7, S8, or S9 record, or an interrupt control character is received. For more information, see the DL command in the *Command Descriptions* section of this chapter.

Execution Mode

This mode is in effect when the target is executing instructions. This usually happens as a result of issuing a **GO** Command. Execution mode operates in real-time. See the Set Breakpoint command (**SB**) in the *Command Descriptions* section of this chapter. A breakpoint or an interrupt control character will place the BDM driver back into command mode. For more details on entering execution mode, see the **GO**, Single Step (**SI**), and Step Out Of Range (**SO**) commands in the *Command Descriptions* section of this chapter. During execution mode most commands are disabled. An attempt to issue a disabled command will result in the message “!error! Not in Background Mode”.

4.2 COMMAND DESCRIPTIONS

This section contains descriptions of each of the commands that can be used with the BDM driver. Each section will detail information about a particular command and its options. Most sections are provided with one or more examples. These examples are formatted for clarity and actual screen displays may vary from machine to machine. All the command responses will end with a carriage return, line feed, and a prompt.

DB

Function

Display Breakpoint

Syntax

DB

Description

This command causes the BDM driver to display all software breakpoints.

Example

>DB

```
1. 010000
2. 010010
```



GO – Go
SB – Set Breakpoint

DC

Function

Display Configuration

Syntax

DC

Description

The command allows the user to examine the environmental resources that are configured for the BDM driver, which include:

- microprocessor type (CPU 32 family)
- communication port
- delay factor

Example

>DC

Background Debug Mode
Target=M6833X M68340 M68360
Communication Port=Lpt1
Delay Factor=1

>



GO – Go

DDD

Function

Debug DFC Display

Syntax

DDD

Description

This command displays the current debug mode value of the Destination Function Code.

BDM uses a default value of 5 to allow complete access to the target memory areas. Under special conditions this value may be modified in order to access a special register (i.e., mbar on the MC68360). This is not the execution value of the DFC, only the Debug Command mode value.

Example

```
>DDD  
>Debugger DFC = 5  
>
```



DDS – Debug DFC Set
DSD – Debug SFC Display
DSS – Debug SFC Set

DDS

Function

Debug DFC Set

Syntax

DDS <Level (0-7)>

Description

This command modifies the debug Destination Function Code (DFC) used by BDM in Command Mode.

The most common reason to modify the values of DFC is to set the mbar register of the MC68360.



The Debug DFC should be returned to the default value of 5 after completion of the operation requiring it to be changed.

Example

```
>DDS 1  
>
```



DDD – Debug DFC Display

DSD – Debug SFC Display

DSS – Debug SFC Set

DL

Function

Download

Syntax

DL

Description

The download command invokes a special mode of operation, called the download mode. Once entered, all information sent to the BDM driver is assumed to be data (S record format) until a Motorola EOF record (S7, S8, S9) is encountered or the interrupt control character (^C) is received.

A positive acknowledgement “+” (PACK) is returned if no problem is detected in memory storage. The download will abort if a bad S record is received. A PACK will also be returned upon the initial download request before the first data record is transferred.

The BDM driver returns to command mode when an EOF is encountered. If a download is aborted with a ^C, all download records following the abort will be treated as commands and will be handled as invalid. It is therefore the responsibility of the host to stop transmission of records after issuing the ^C.

Example

In this example, the code was originally located at address ED0000. The display will show the positive acknowledgements and the echoed characters.

```
>DL
```

```
+S00600004844521B
+S213ED0000000700ED005600ED006C00ED007800F7
+S213ED000FED00A400ED00BA00ED00C600ED00EA2E
+S213ED001E00ED010000ED013000ED017700ED0182
+S213ED002D8E00ED01C800ED025800ED026E00EDFD
+S804000000FB
>
```


DM

Function

Display Memory

Syntax

DM[*unit*] *addr count*

unit = B = byte
 W = word (default)
 L = long

addr = The starting address of the memory to display.

count = The number of memory locations (size *unit*) to display.

Description

This command causes the BDM driver to return the contents of the memory location(s) requested. This command will display only the address and the hex data.

Example

In this example, we will display 32 (20 hex) words of memory starting at location 200010.

```
>DM 200010 20
```

```
200010. 3E2E 2E2E 0074 6573 7420 6E75 6D62 6572
200020. 2033 2066 6169 6C65 6420 2D20 4E20 7263
200030. 7664 0020 2020 2020 2020 2020 2020 2020
200040. 2020 2020 2020 2020 2020 2020 2020 2020
>
```

In this example, we will display 5 bytes of memory starting at location 300300.

```
>DMB 300300 5
```

```
00300300. 12 53 12 14 15
>
```

DR

Function

Display Registers

Syntax

DR [*reg_name*]

reg_name = A valid register name for the target.

Description

This command causes the BDM driver to display the contents of a particular register or registers. If no arguments are specified then all the registers will be displayed.

Example

In this example, we will display all the registers of a Motorola 68332 based target system.

>DR

```
D0 = 00000000 D1 = 00000000 D2 = 00000000 D3 = 00000000
D4 = 00000000 D5 = 00000000 D6 = 00000000 D7 = 00000000
A0 = 00000000 A1 = 00000000 A2 = 00000000 A3 = 00000000
A4 = 00000000 A5 = 00000000 A6 = 00000000 A7 = 00E84000
USP = 00E85000 SSP = 00E84000 PC = 00F00086 SR = 2000
VBR = 00E80000 SFC = 0007 DFC = 0007
>
```

In this example, first set some registers with values and then display these registers in different order.

>SRL A0 100000 A3 120000 D1 1234 D3 55

>DR D1 D3 A0 A3

```
D1 = 00001234 D3 = 00000055 A0 = 00100000 A3 = 00120000
>
```



SR – Set Register

DSD

Function

Debug SFC Display

Syntax

DSD

Description

This command displays the current Debug Mode value of the Source Function Code (SFC).

BDM uses the default value of 5 to access most of the target register and memory. This value may be modified to access the special register (i.e., mbar of MC68360).

Example

```
>DSD
>Debugger SFC = 5
>
```



DDD – Debug DFC Display

DDS – Debug DFC Set

DSS – Debug SFC Set

DSS

Function

Debug SFC Set

Syntax

DSS <Value 0-7>

Description

This command modifies the debug Source Function Code (SFC) used by BDM in Command Mode.

The most common reason for changing SFC is to display the mbar register of the MC68360.



The Debug SFC should be returned to the default value of 5 after completion of the operation requiring it to be changed.

Example

```
>DSS 1  
>
```



DDD – Debug DFC Display
DDS – Debug DFC Set
DSD – Debug SFC Display

FR

Function

Check for Freeze

Syntax

FR

Description

This command will check the target status and return either:

! Executing

or:

! Breakpoint or Background Mode entered.

Example

In this example, we will set a breakpoint, cause the target to go into execution, and poll status twice.

```
>SB 1000
```

```
>go
```

```
>FR
```

```
!executing
```

```
>FR
```

```
!Breakpoint or Background Mode entered.
```



GO – Start Executing

SB – Set Breakpoint

SR – Set Register

GO

Function

Start Execution

Syntax

GO

Description

This command causes the BDM driver to go into execution mode. The execution mode will continue until either a breakpoint occurs, an interrupt (^x | ^C) is received from the host, or a double bus fault (as specified in the *Development Support* section of Motorola's *CPU32 Central Processor Unit Reference Manual*). If breakpoints are set, a BGND instruction is inserted into the target code at each breakpoint address. This will allow real-time execution of target code and still allow for breakpoints to be taken.

If the location where execution begins contains a breakpoint BGND, then that breakpoint is temporarily disabled until the program is stepped off the breakpoint. This command will return immediately with the target in the executing mode.

Example

```
>GO  
>
```



- DB** – Display Breakpoints
- FR** – Check for Freeze
- SO** – Single Step
- SB** – Set Breakpoint
- SO** – Step Out of Range

HE

Function

Help

Syntax

HE

Description

This command causes the BDM driver to display the help menu. This menu shows the syntax of the command set.

IN

Function

Initialize

Syntax

IN

Description

This command causes BDM to identify its version number.

Example

>IN

```
Wiggler by Macraigor System Inc.  
Target M6833X M68340 M68360  
Version 1.4  
(c) 1999  TASKING Inc.  
>
```



The revision numbers used in the example are sample numbers for this manual only.

RB

Function

Remove Breakpoint

Syntax

RB [*addr*]

addr = An optional address that specifies the breakpoint to be removed.

If *addr* is omitted, then all breakpoints will be removed.

Description

This command causes the BDM driver to remove a software breakpoint at the address specified. If no arguments are specified, then all breakpoints will be removed. If an address is specified and no breakpoint exists at that address, an error message will be returned.

Example

In this example, first display some breakpoints that were previously setup and then remove a code breakpoint at address 10000. Finally, verify that the breakpoint has been removed. Notice that the second code breakpoint will become the first breakpoint after the RB command is executed.

```
>DB
```

```
1.  010000
2.  010010
```

```
>RB 10000
```

```
>DB
```

```
1.  010010
```

```
>
```



DB – Display Breakpoints
SB – Set Breakpoints

RS

Function

Reset

Syntax

RS

Description

This command causes BDM to reset the target and enable background mode. It is primarily used to reset background mode after the target was reset by other means than through BDM. See the *Development Support* section of Motorola's *CPU32 Central Processor Unit Reference Manual* for information of enabling the BDM mode.

Example

```
>RS
```

```
>
```

```
>RS
```

```
!error! unable to enter background mode
```

SB

Function

Set Breakpoints

Syntax

SB *addr*

Description

This command causes BDM to set a software breakpoint at the address specified.

Example

In this example, we will set a breakpoint to occur when the program counter reaches address 10000. First set the breakpoint and then issue the **GO** command.

```
>SB 10000
```

```
>GO
```

```
>FR
```

```
>!Breakpoint in Background Entered.
```



- DB** – Display Breakpoint
- GO** – Start Execution
- RB** – Remove Breakpoint

SI

Function

Single Step Instruction

Syntax

SI [*count*]

count = An optional number of target instructions to execute. The default is one instruction.

Description

This command causes the BDM driver to go into execution mode for *count* instructions.

Example

In this example, we will single step the target for one instruction.

```
>SI
```

```
>
```

SM

Function

Set Memory

Syntax

SM[*unit*] *addr values*

unit = B = byte
 W = word (default)
 L = long

addr = The start address for the set, in hex.

values = A space-separated list of values to write to memory.

Description

This command causes the BDM driver to fill memory with the hex byte values specified.

Example

In this example, we will display the contents of memory both before and after we set three bytes of memory to the specified values.

```
>DMB 200010 3
```

```
200010: FF 0F 00
```

```
>SMB 200010 0B 7F 34
```

```
>DMB 200010 3
```

```
200010: 0B 7F 34
```

```
>
```



DM – Display Memory

SO

Function

Step Out of Address Range

Syntax

SO *start end*

start = The starting address for the range.

end = The ending address for the range.

Description

This command causes the BDM driver to go into execution mode. The execution mode will continue until any of the following conditions occurs: a breakpoint is encountered, an instruction outside of the specified range is about to be executed, an interrupt is received from the host, or a double bus fault.

Example

In this example, we will single step through the code until the program falls outside the specified range.

```
>SO 0001000 0001fff
```

```
>
```



The BDM driver will execute until the command is complete, but only return a prompt. Do not select a range of values that will cause an infinite loop to occur. If this does occur, the driver will lock out any host interaction. A reboot may be required to abort this condition.

SR

Function

Set Registers

Syntax

SR *reg value* [*reg value ...*]

reg = A valid target register (you may not modify the value of A7).

value = A hex value to write to the register.

Description

This command causes the BDM driver to modify the register or registers specified.

Example

In this example, we will set address register A0 and data register D2 with word values. Then we will display these registers to verify their contents.

```
>DR A0 D2
```

```
A0 = 00000000 D2 = FFFFFFFF
```

```
>SR A0 005B D2 7F34
```

```
>DR A0 D2
```

```
A0 = 0000005B D2 = 00007F34
```

```
>
```



DR – Display Registers

5 TROUBLESHOOTING

- BDM has its own unique set of communication failures. CrossView Pro error messages caused by BDM communication failures fall into three categories:
- Unable to open driver from OpenDriver.
- Open failed.
- Unexpected responses.

5.1 UNABLE TO OPEN DRIVER FROM OPENDRIVER

This failure usually occurs if the device driver is not installed. Refer to the *BDM Software and Hardware Installation* section earlier in this appendix. Another reason for this failure is that the driver is opened by another task. An example of this would be that the control panel is performing setup and CrossView Pro attempts to load a function. This can be corrected by completing one of the tasks and closing the device driver.

5.2 OPEN FAILED FROM CROSSVIEW PRO

If CrossView Pro fails with this message, then the parallel port is unavailable. Refer to the *Configuration Options* section of this manual for information on how to set up communication to the correct port.

5.3 UNEXPECTED RESPONSES

- Unable to read/write memory at xxx

The message is generated by the download request and indicates that the address could not be modified by a write/read test. This is most likely caused by a chip select not being set up correctly.

- Loss of BDM mode

This is a result of the target not being reset via the BDM connection. Use the RS command to regain control.



The RS command causes the target to be reset.

- Bus error

The address specified generated a bus error. It is possible that an application program error occurred or that the chip select was not set up correctly.

- Target communication failure, unable to alter register

This message occurs if CrossView Pro's initial attempt to communicate with the target fails.

- Other unexpected responses

There is no single test to determine what actually failed. The following possibilities should be tested:

- Wiggler cable is incorrectly connected to the target CPU.
- The parallel port is disabled.
- There is a bad cable connection to the Wiggler DB25 connector.
- The parallel port is malfunctioning.
- The target CPU is malfunctioning.
- Failure to follow Device Driver installation instructions. The device driver should not be copied into the CrossView Pro directory.

6 OTHER CONSIDERATIONS

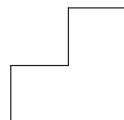
- BDM does not support tracing. Therefore, C-trace is not available in CrossView Pro.
- BDM only supports data breakpoints with assertion mode.
- BDM supports a maximum of 32 breakpoints.
- BDM will return a maximum of 256 memory units (byte, word, long) per display request.

INDEX

INDEX



TASKING



INDEX

Symbols

. (period) operand, 3-18
 ! command, 13-20
 ? command, 5-16, 13-22
 & operator, 3-18
 @format code, 3-13
 --ddeservername, A-28
 --timeout, 9-10
 / command, 5-16, 13-21
 /format code, 3-13
 ^ command, 13-36
 < command, 13-23
 << command, 13-24
 > command, 13-25
 >& command, 13-33
 ># command, 13-29
 >@ command, 13-27
 >* command, 13-35
 >> command, 13-31

A

A command, 13-37
 a command, 13-38
 absolute file, 15-3
 accelerator bar, 4-24
 accelerator button, 4-11, 4-24
 accessing code and data, 6-1
 AddDDMenuEntry, A-28
 adding files to a project, 1-37
 address bias, set, 13-142
 addresses
 in expressions, 3-18
 specifying format of, 6-16
 analysis, 15-3
 window, 15-3
 ANSI C conformity, 1-12
 application
 debugging, 1-27
 executing, 1-24

argument of a function, 3-9
 arrays
 display address of, 6-6
 display character, 3-15, 6-6
 displaying two-dimensional, 6-15
 viewing contents of, 3-16, 6-15
 assembly window
 hexadecimal display, 3-10
 intermixed assembly, 3-10
 source merge limit, 3-10
 assertion mode, 7-28, 15-3
 assertions, 1-5, 7-28, 15-3
 activating, 7-28
 activating and suspending, 7-31
 assertion mode, 7-28
 debugging with, 7-33
 define or modify assertion, 13-38
 defining, 7-29
 deleting, 7-32
 editing, 7-31
 quit assertion mode, 13-178
 statistics, 7-35
 toggle mode, 13-37
 AssertionsChanged, A-11
 autosrc, 6-18

B

B command, 13-40
 b command, 13-41
 background color, 2-5
 background debug mode, Bdm-1,
 Bdm-3-Bdm-32
 command descriptions,
 Bdm-10-Bdm-32
 command interface, Bdm-9-Bdm-32
 configuration options,
 Bdm-7-Bdm-32
 emulator mode, Bdm-9-Bdm-32
 hardware, Bdm-5-Bdm-32

*installation**hardware, Bdm-5**software, Bdm-6–Bdm-7**software, Bdm-5–Bdm-32**software content, Bdm-4**system requirements, Bdm-3–Bdm-4**target environment setup,**Bdm-7–Bdm-9**trouble shooting, Bdm-31–Bdm-32**with CrossView Pro, Bdm-3–Bdm-32*

background mode, 11-28, 15-4

*assertions, 11-33**leaving, 11-31**local and global variables, 11-32**manual refresh, 11-29**refresh limitations, 11-32**running a program, 11-30**stack, 11-32**starting, 11-30**stopping a program, 11-31**updating windows, 11-28**waiting, 11-31*

batch mode, 9-10

batch processing, 9-10

bB command, 13-42

bb command, 13-43

bc command, 13-44

bCYC command, 13-45

bcyc command, 13-46

bD command, 13-47

bd command, 13-49, Rom-67

bdis command, 13-51

BDM. *See* background debug mode

be command, Rom-68

bena command, 13-52

bf command, Rom-69

bI command, 13-53

bi command, 13-54

bias, 15-4

binary constants, 3-5

binary notation, 3-4

bINST command, 13-55

binst command, 13-56

block fill, Rom-69

block move, Rom-71

bm command, Rom-71

break command, 13-57

breakpoint disable, Rom-67

breakpoint enable, Rom-68

breakpoint toggle, 4-23, 7-4

breakpoints, 1-4, 7-1, 13-57, 15-4

*and diagnostic output, 7-27**and multi-line statements, 7-5**and multiple statements, 7-5**and statistical information, 7-27**attaching macros to, 7-21**code, 7-3**commands associated with, 7-18**complex, Rom-16**conditionals, 7-21**count, 15-5**count of, 7-3**cycle count, 7-3, 13-45, 13-46**data, 7-7, Rom-15**data breakpoints over a range of**addresses, 7-11**delete, 13-85**delete all, 13-84**deleting, 7-16**disable, 7-17, 13-51**emulator mode, 7-6**enable, 7-17, 13-52**for loops, 7-6**function, permanent, 13-43**instruction, Rom-15**instruction count, 7-3, 13-55, 13-56**list, 13-40**listing, 7-8**low-level, 15-7**name, 7-3**names, 7-13**patching code with, 7-25**permanent, 7-4**permanent low-level, 13-54**task aware, 13-64**permanent up-level, 13-69*

- probe point*, 1-5, 7-4
 - quiet reporting of*, 7-22
 - reset count*, 7-3, 7-14
 - sequence*, 7-15
 - set at beginning of function*, 13-42
 - set count*, 13-44
 - setting*, 1-25, 7-8, Rom-14-Rom-142
 - from command window*, 7-10
 - from menu*, 7-9
 - from source window*, 7-9
 - from stack window*, 7-10
 - setting the count of*, 7-14
 - strings*, 7-22
 - system startup code*, 7-7
 - task aware*
 - code*, 13-62
 - permanent low-level*, 13-64
 - temporary low-level*, 13-63
 - temporary*, 7-4, 7-12
 - temporary low-level*, 13-53
 - task aware*, 13-63
 - temporary up-level*, 13-67
 - time*, 13-65, 13-66
 - timer*, 7-3
 - up-level*, 7-22
 - while loops*, 7-6
 - without trace mode*, Rom-15
- BreakpointsChanged, A-11
- bt command, 13-62
- btI command, 13-63
- bti command, 13-64
- bTIM command, 13-65, 13-66
- bU command, 13-67
- bu command, 13-69
- bufa command, 13-71
- bufd command, 13-72
- ## C
- C, character constants, 3-6
- C command, 5-12, 13-73
- C trace, 1-5, 13-81
- cache, debugging with, 15-4
- calling functions, 5-14
- case sensitivity, 3-21, 13-179
- casting values, 3-16, 6-15
- CB command, 13-74
- CBRK, B-7
- CCNT, B-7
- cd command, 13-75
- ce command, 13-76
- cf command, Rom-72
- character buffering, Rom-25
- character codes, 6-13
- character codes table, 3-6
- character constants, 3-6
- check for freeze, Bdm-20
- chip select initialization files, Bdm-4
- clear command, 13-77
- close a file I/O stream, 13-117
- CmdAnnotatedOutput, A-11
- cmdannotatedoutput, A-22
- cmdoutput, A-21
- code breakpoints
 - See also breakpoints*
 - set breakpoint*, 13-41
 - task aware*, 13-62
- code coverage, 1-6
- color, windows, 2-5
- color offset, 11-17
- color settings, 2-5
- COM interfaces, A-5
- COM methods
 - Execute*, A-6
 - ExecuteNoWait*, A-7
 - Halt*, A-7
 - Init()*, A-6
- COM object interface, A-3
 - activating*, A-5
 - events*, A-8
 - examples*, A-12
 - methods*, A-6
 - using*, A-3

- command history, displaying recent
 - commands, 9-19
- command language, 3-1
- command line, batch processing, 9-10
- command line options, 4-5
- command mode, Rom-65, Bdm-9
- Command Window, 4-21
 - displaying data in*, 6-8
 - opening*, 1-29
- CommandCanceledByUser, A-9
- CommandInterpreterBusy, A-8
- CommandInterpreterReady, A-9
- commands
 - descriptions*, Rom-66-Rom-142, Bdm-10-Bdm-32
 - multiple*, 3-17
 - syntax*, 4-3
- comments, 3-17
- communication setup, 1-21
- compare application, 1-24, 13-86
- conditional command execution, 13-116
- conditional keywords, 3-19
- configure, Rom-72
- configure CrossView Pro, 1-21
- constants, 3-4
 - binary*, 3-5
 - character*, 3-6
 - character constants in C*, 3-6
 - floating point*, 3-5
 - hexadecimal*, 3-4
 - long integer*, 3-5
 - octal*, 3-5
 - strings*, 3-6
- continue execution, 5-9
- control characters, Rom-64, Bdm-9-Bdm-32
- control operations, 4-38
- coverage, 1-6, 11-6, 15-5
 - disable*, 11-6, 13-75
 - enable*, 11-6, 13-76
 - information*, 13-78
 - marker*, 4-23, 7-4
- memory window*, 4-28
- next covered block*, 13-143
- next not covered block*, 13-144
- previous covered block*, 13-149
- previous not covered block*, 13-154
- source window*, 4-24
- covinfo command, 13-78
- cproinfo command, 13-79
- CPU, reserved variable, 3-10
- CPU selection, 4-7
- cpu selection, 13-95, 13-96
- cpu_type, 1-20
- CRC test, Rom-129
- creating a makefile, 1-38
- CrossView
 - and command line options*, 4-5
 - command files*, 4-6
 - command language*, 3-1
 - command line batch processing*, 9-10
 - command reference*, 13-1
 - commands summary*, 13-4, 13-15
 - customizing*, 4-17
 - desktop*, 4-11
 - features of the execution environment*, Sim-3
 - invoking*, 4-4
 - restrictions of execution environment*, Sim-15
 - sound support*, C-1
 - special features*, 11-1
 - starting*, 4-4
 - state of*, 13-115
 - using*, 4-1
- CrossView Pro
 - background debug mode execution*, Bdm-3-Bdm-32
 - before starting*, 1-17
 - debugging environment*, 1-8
 - documentation*, 1-7
 - exiting*, 1-30
 - features*, 1-3
 - how it works*, 1-10

- invoking*, 1-18
- output*, 1-29
- source level debugging*, 1-8
- target settings*, 1-19
- using windows*, 1-4
- windows*, 1-4
- CrossView Pro workspace, 1-30
- ct command, 13-81
- ct i command, 13-82
- ct r command, 13-83
- cursor, 5-3
- CXL script, 4-37
 - supplied scripts*, 11-13
 - syntax*, 11-19
- CXL syntax, 11-19
 - base types*, 11-21
 - compound types*, 11-22
 - predefined functions*, 11-22
- cycle count, breakpoints, 7-3

D

- D command, 13-84
- d command, 13-85
- data
 - displaying*, 6-1
 - enumerated*, 6-5
 - list data monitors*, 13-130
- data analysis, 11-11
 - add update commands*, 13-107
 - bufa*, 13-71
 - bufd*, 13-72
 - clear sequence of update commands*, 13-108
 - close window*, 13-109
 - create window*, 13-106
 - graph*, 13-106
 - graph debug*, 13-110
 - graph_add_update*, 13-107
 - graph_clear_updates*, 13-108
 - graph_close*, 13-109

- graph_debug*, 13-110
- graphm*, 13-111
- graphmn*, 13-112
- graphp*, 13-113
- memget*, 13-134
- position window*, 13-113
- rawmemget*, 13-158
- supplied scripts*, 11-13
- update*, 13-175
- update window*, 13-175
- Data Analysis Window, 4-36
 - toolbar*, 4-36
- data breakpoints
 - set at an address*, 13-49
 - set over range of addresses*, 13-47
- data coverage, 1-6, 4-28
- data monitoring, 1-5, 15-5
 - removing expressions*, 6-12
- Data Window, 1-5, 4-29, 6-11
 - toolbar*, 4-31
- db command, Rom-73
- dc command, Rom-75
- dcmp command, 13-86
- DDE command line options,
 - ddeservename*, A-28
- DDE commands, AddDDEMenuEntry, A-28
- DDE events, A-27
- DDE items
 - cmdannotatedoutput*, A-22
 - cmdoutput*, A-21
 - event*, A-26
 - exec*, A-24
 - execext*, A-23
 - halt*, A-25
 - Help*, A-20
 - result*, A-27
- DDE server interface, A-20
- debug
 - DFC display*, Bdm-13
 - DFC set*, Bdm-14
 - SFC display*, Bdm-18

SFC set, Bdm-19
 debug instrument, 13-14
 save/restore state, 13-88
 debug instrument I/O, 10-9
 debug_instrument_module, 1-20
 debugger, starting, 1-37
 debugging
 and optimized code, 3-7
 assembly language, 12-3
 code without symbols, 5-14
 environment, 1-8
 multiple programs, 12-3
 notes about, 12-1
 source-level, 1-8
 viewing source while, 1-24
 debugging an application, 1-27
 debugging features, Rom-5
 derivatives, B-7
 description file, Sim-3
 desktop, 4-11
 DestroyedAllSymbols, A-11
 development flow, 1-12
 df command, Rom-77
 di command, Rom-78
 di_state command, 13-88
 diagnostic function, Rom-77
 diagnostic output, and breakpoints,
 7-27
 diagnostics, 15-5, Rom-124-Rom-142
 SmartMON, Rom-124-Rom-142
 user, Rom-135-Rom-142
 dialog boxes, 4-16
 DidAddSymbols, A-11
 DidDownloadImage, A-11
 DidLoadSymbols, A-10
 dis command, 13-89
 disassemble, Rom-78
 disassemble memory, 13-89
 disassembly, 6-17
 window, 15-6
 display, customizing, 4-17
 display breakpoint, Rom-73, Bdm-11

display configuration, Rom-75,
 Bdm-12
 display formats, set default, 13-100
 display memory, Rom-81, Bdm-16
 display registers, Rom-82, Bdm-17
 display trace, Rom-83
 dl command, Rom-79
 dm command, Rom-81
 dn command, 13-90
 documentation, 1-7
 dot operand, 6-11
 download, Rom-79, Bdm-15
 download a file, 13-90
 download image, 13-129
 download mode, Rom-65, Bdm-10
 downloading, files to the execution
 environment, 1-22
 dr command, Rom-82
 dt command, Rom-83
 dump, 3-16, 6-15
 dump command, 13-91
 Dy command, 13-84

E

e command, 5-15, 13-93
 eC command, 13-95
 ec command, 13-96
 echo command, 13-97
 echo string to terminal, 13-19
 EDE, 1-32
 build an application, 1-36
 load files, 1-34
 open a project, 1-34
 select a toolchain, 1-33
 start a new project, 1-37
 starting, 1-32
 edit source, 4-25
 ei command, 13-98

embedded development environment.

See EDE

embedded system, 15–6

emulator, setting up execution

environment, 1–17

emulator communication setup, 1–21

emulator mode, 1–9, Bdm–9–Bdm–32

environment variable

LD_LIBRARY_PATH, 2–3

UIDPATH, 2–3

EPROMS, programming,

Rom–61–Rom–136

error messages, alphabetical listing of,
14–1

Esc key, 4–21

et command, 13–99

evaluate expression, 13–16

event, A–26

events, A–8, A–27

AssertionsChanged, A–11

BreakpointsChanged, A–11

CmdAnnotatedOutput, A–11

CommandCanceledByUser, A–9

CommandInterpreterBusy, A–8

CommandInterpreterReady, A–9

DestroyedAllSymbols, A–11

DidAddSymbols, A–11

DidDownloadImage, A–11

DidLoadSymbols, A–10

HaltButtonPressed, A–9

MenuEntrySelected, A–11

Quit, A–12

Reset, A–10

ResetProgram, A–10

Running, A–9

RunningInBackground, A–9

SourceFileChanged, A–10

Stopped, A–9

ViewedLineNrChanged, A–10

example

starting EDE, 1–32

using EDE, 1–32

exec, A–24

execext, A–23

executable, building for CrossView

Pro, 1–32

Execute, A–6

ExecuteNoWait, A–7

executing an application, 1–24

execution control commands,

summary of, 13–8

execution environment, Sim–1, Rom–1,
Bdm–1

connecting to CrossView, 4–6

downloading files to, 1–22

execution mode, Rom–65, Bdm–10

submodes, Rom–18

execution position, 5–3

changing the, 5–5

definition of, 15–6

sync with viewing position, 5–7

exit, 4–19

exponential notation, 3–5

expression evaluator, 1–4

expressions, 3–3

C character codes, 3–6

character constants, 3–6

evaluating, 6–10

evaluation precision, 3–4

floating point constants, 3–5

format of, 3–13

monitoring, 6–11

removing monitored, 6–12

show, 4–29

special expressions, 3–18

specifying variables in, 3–8

strings, 3–6

watch, 4–29

extension language, 11–19

eye diagram, 11–19

F

f command, 13–100

FFT power spectrum, 11-15
 combined with phase, 11-18
 multi, 11-15
 multi in lines, 11-16
 multi in lines and grid, 11-16
 FFT waterfall, 11-15
 file system simulation, 10-7, 15-6
 close a stream, 13-102
 libraries, 10-8
 redirect output to a file, 13-103
 redirection, 10-7, 13-101
 summary of commands, 13-13
 filenames, 2-3
 floating point constants, 3-5
 format codes, 3-14
 formats, for variables, 6-13
 formatting, Rom-53-Rom-60
 formatting expressions, 3-13
 frame pointer, 3-10
 FSS
 redirection, 10-7
 summary of commands, 13-13
 FSS command, 13-101
 FSS_stdio_close, 13-102
 FSS_stdio_open, 13-103
 functions, 3-20
 calling directly, 5-14
 listing all, 6-8
 listing local variables and parameters
 of, 6-22

G

g command, 5-5, 13-104
 GDI, 1-8, 9-12, 9-13
 logging, 9-13, 9-15, 9-17
 getting started, 1-17
 gi command, 5-6, 13-105
 global variables, 3-8
 glossary, 15-1
 go command, Rom-85
 go to next instruction, Rom-88

gon command, Rom-88
 graph command, 13-106
 graph_add_update command, 13-107
 graph_clear_updates command,
 13-108
 graph_close command, 13-109
 graph_debug command, 13-110
 graphm command, 13-111
 graphmn command, 13-112
 graphp command, 13-113
 GUI update suppress, 13-114
 gus command, 13-114

H

Halt, A-7
 halt, A-25
 halt execution, 5-9
 HaltButtonPressed, A-9
 he command, Rom-90
 Help, A-20
 help, Rom-90, Bdm-22
 on-line, 1-7, 4-39
 summary of help commands, 13-14
 hexadecimal disassembly, 3-10
 hexadecimal notation, 3-4
 history mechanism, 15-7

I

I command, 13-115
 I/O simulation, 1-5
 defined, 15-7
 disable streams, 10-7
 enable streams, 10-7
 file system simulation, 10-7
 redirecting streams, 10-6
 setting up streams, 10-4
 terminal windows, 4-34
 if command, 13-116

image part, 15-7
 in command, Rom-91
 in-situ editing, 6-7, 6-26
 Init(), A-6
 initialize, Rom-91, Bdm-23
 input/output
 interrupt driven, Rom-20-Rom-29
 polled, Rom-23-Rom-29
 system calls, Rom-26
 input/output simulation, 10-1
 defined, 15-7
 summary of commands, 13-12
 instruction count breakpoints, 7-3
 integers, 3-4
 binary, 3-5
 hexadecimal, 3-4
 integral promotion, 3-5
 long, 3-5
 negative, 3-4
 octal, 3-5
 integral promotion, 3-5
 intermixed source and disassembly,
 6-18
 interprocess communication, A-1
 interrupt key, 15-7
 interrupt service routine (ISR),
 Rom-45-Rom-60
 interrupt service routines (ISRs),
 Rom-27
 debugging downloading, Rom-29
 ios_close command, 13-117
 ios_open command, 13-118
 ios_read command, 13-120
 ios_readf command, 13-121
 ios_rewind command, 13-122
 ios_wopen command, 13-123
 ios_write command, 13-124
 ios_wrtf command, 13-125

J

jump to cursor, 5-5

K

kernel support, 1-7, 11-4
 keyboard mappings, 10-10
 keywords, conditional, 3-19-3-22

L

L command, 13-126
 l command, 13-127
 label, in disassembly, 6-17
 language, 3-1
 LD_LIBRARY_PATH, 2-3
 line command, 13-18
 line numbers, 3-10
 listing, 13-127
 load command, 13-129
 load symbol file, 13-129, 13-141
 local variables, 3-7
 and the stack, 3-7
 auto-watch, 4-31
 logging, 9-12
 command window output, 13-31
 commands and screen output, 9-15
 debugger-emulator I/O, 13-33
 debugger-GDI accesses, 13-35
 example, 9-15
 resume, 9-15
 setting up, 9-13
 start, 9-13
 startup options, 9-18
 stop, 9-17
 summary of commands, 13-11
 suspend, 9-15
 long integer constants, 3-5

M

M command, 13-130
 m command, 13-131

macros, 1-7, 8-1, 15-8
 calling other macros, 8-4
 define, 13-163
 defining, 8-3
 delete definition, 13-174
 deleting, 8-8
 echo command, 13-97
 expanding, 8-5
 listing, 8-5
 parameters of, 8-9
 reading from a file, 8-7
 redefining, 8-5, 8-10
 save, 13-162
 saving to a file, 8-6
 summary of commands, 13-12
 using the toolbox, 8-11
 main() function, 15-8
 makefile
 automatic creation of, 1-38
 updating, 1-38
 makepy utility, A-15
 MAU (minimum addressable unit),
 15-8
 mcp command, 13-133
 memget command, 13-134
 memory
 copy, 13-133
 disassembly, 13-89
 displaying, 6-14
 dump, 13-91
 fill, 13-137
 mapping, Sim-3
 search, 13-139
 single fill, 13-136
 memory access, tracing, 1-6
 memory dump, 3-16, 6-15
 memory map, 4-6, 15-8
 Memory Window, 4-27
 setup, 4-28
 toolbar, 4-28
 menu, 4-13
 local popup, 4-14
 menu bar, 4-11

MenuEntrySelected, A-11
 mF command, 13-136
 mf command, 13-137
 minimum addressable unit, 15-8
 mm command, Rom-92
 modify memory, Rom-92
 monitor. *See* ROM Monitor
 monitor data, 13-130
 monitors, 13-131
 more, 3-10
 ms command, 13-139
 multi FFT power spectrum, 11-15
 in lines, 11-16
 in lines and grid, 11-16

N

N command, 13-141
 n command, 13-142
 nC command, 13-143
 nU command, 13-144

O

o command, 13-145
 object modules
 building the demo,
 Rom-57-Rom-60
 linking and locating,
 Rom-52-Rom-60
 octal constants, 3-5
 octal notation, 3-4
 open a file I/O stream, 13-118, 13-123
 operation modes, Rom-64-Rom-142,
 Bdm-9-Bdm-32
 command, Bdm-9
 download, Bdm-10
 execution, Bdm-10
 operational modes
 command, Rom-13

- download, Rom-14*
- execution, Rom-14*
- operators, 3-17
 - order of precedence, 3-17*
 - using addresses, 3-18*
- opt command, 13-146
- optimization, and debugging, 3-7
- options, display or set, 13-146
- OSEK/ORTI, 11-4
- output paging mechanism, 3-10
- overview, 1-1

P

- P command, 13-147
- p command, 13-148
- packet format, A-27
- patches, 15-8
 - and breakpoints, 7-25*
- pC command, 13-149
- pd command, 13-150
- pe command, 13-151
- performing timing analysis, 1-6
- playback, 9-8
 - calling other playback files, 9-9*
 - quitting, 9-10*
 - setting the type of, 9-9*
 - startup options, 9-18*
 - summary of commands, 13-11*
- playback mode, 1-7
 - continuous, 13-23*
 - single step, 13-24*
- pointer, 3-16
 - display character, 3-15, 6-6*
- portinit, Rom-44
- precision, evaluating expresions, 3-4
- print source lines, 13-147, 13-148
- probe point, 1-5, 7-4, 15-9
- problems
 - common, 1-31*
 - communicating with CrossView, 4-9*

- profiling, 1-6, 11-8, 15-9
 - code range, 1-6, 11-10*
 - cumulative, 11-9*
 - cumulative information, 13-79*
 - disable, 11-10, 13-150*
 - enable, 11-10, 13-151*
 - function, 11-9*
 - functions, 1-6*
 - information, 13-152*
- program counter, 3-10, 5-7, 13-73
 - g command (change), 13-104*
 - gi command (change), 13-105*
 - inside function, 3-9*
- program development, 1-12
- program execution
 - controlling, 5-1*
 - notes about, 5-14*
- program reset, 13-153
- proinfo command, 13-152
- project files, adding files, 1-37
- PROMs, programming,
 - Rom-54-Rom-142
- prst command, 13-153
- pseudo-assembly, 6-18
- pU command, 13-154

Q

- Q command, 13-155
- q command, 13-156
- quiet breakpoint recording, 13-155
- Quit, A-12
- quit debugger, 13-156

R

- R command, 5-8, 13-157
- radm, 1-20

RAM, installing RAM based diagnostics,
 Rom-141
 RAM tests, Rom-124
 complete, Rom-126, Rom-128
 simple, Rom-125, Rom-127
 rawmemget command, 13-158
 rb command, Rom-93
 read from an I/O stream, 13-120
 formatted, 13-121
 record
 commands only, 13-25
 CrossView Pro and emulator
 commands, 13-27
 emulator commands only, 13-29
 record and playback, 9-1
 definition of, 15-9
 record mode, 1-7
 recording
 checking status, 9-6
 close file for, 9-6
 entering comments, 9-4
 example, 9-7
 resume, 9-5
 start, 9-3
 startup options, 9-18
 stop, 9-6
 summary of commands, 13-11
 suspend, 9-5
 refresh windows, 13-173
 reg.dat, B-4
 register file syntax, B-4
 register manager, B-1
 register set, fixed, B-6
 Register Window, 4-26, 6-25
 setup, 6-25
 registers, 3-11
 displaying the contents of, 6-8
 special variable, 3-10
 remove breakpoint, Rom-93, Bdm-24
 Reset, A-10
 reset program, 5-8, 13-153
 reset target system, 13-157, 13-159
 reset the target, Bdm-25

ResetProgram, A-10
 resource file, 2-3
 result, A-27
 rewind an I/O stream, 13-122
 RM_INIT, Rom-38
 rm68, B-1
 ROM Monitor, Rom-1
 debugging features, Rom-5
 overview, Rom-3
 ROM monitor, setting up execution
 environment, 1-17
 ROMM_GO, Rom-42
 rst command, 13-159
 RTOS aware debugging, 11-4
 run-time, Rom-27-Rom-142
 Running, A-9
 RunningInBackground, A-9
 RX_CHAR, Rom-46, Rom-48

S

S command, 5-11, 13-160
 s command, 13-161
 save command, 13-162
 save on exit, 4-19
 sb command, Rom-95
 sbc command, Rom-98
 sbd command, Rom-99
 sbr command, Rom-101
 scope loop routines
 read from location, Rom-130
 write and compliment, Rom-132
 write rotating value, Rom-133
 write then read, Rom-134
 write to location, Rom-131
 scoping rules and variables, 3-9
 scroll bar, 4-11
 search
 backward for string, 13-22
 forward for string, 13-21
 summary of commands, 13-14

- search for string, Rom-107
- searching, 5-15-5-18
 - for a function, 5-15*
 - for a source line, 5-17*
 - for a string, 5-16*
- serial ports, 4-6
- set address breakpoint, Rom-98
- set breakpoints, Bdm-26
- set command, 13-163
- set conditional breakpoints, Rom-95
- set data breakpoint, Rom-99
- set data range breakpoint, Rom-101
- set memory, Rom-104, Bdm-28
- set registers, Rom-106, Bdm-30
- Si command, 5-12, 13-165
- si command, 5-12, 13-166, Rom-103
- signal analysis, 4-36
- simulation, I/O, 1-5
- simulator, Sim-1
- single step instruction, Rom-103, Bdm-27
- single stepping, 1-5, 5-9-5-10, Rom-17
 - at machine level, 5-12-5-18*
 - defined, 15-10*
 - into, 5-10*
 - into function calls, 13-161*
 - into functions, 5-10*
 - machine level into functions, 13-166*
 - machine level over functions, 13-165*
 - over, 5-11*
 - over function calls, 13-160*
 - over functions, 5-11*
- sizeof() function, 6-7
- skidding, 15-10
- sm command, Rom-104
- SmartMON
 - addresses, Rom-41*
 - build process, Rom-50-Rom-142*
 - building work code, Rom-54-Rom-142*
 - initialization, Rom-26-Rom-142*
 - linking diagnostics with, Rom-140*
 - operational modes, Rom-13*
 - processing I/O, Rom-19-Rom-142*
 - processing UD commands, Rom-141*
 - required values, Rom-33*
 - resource requirements, Rom-10*
 - setting breakpoints, Rom-14-Rom-142*
 - starting up with CrossView Pro, Rom-58-Rom-142*
 - starting with terminal or emulator, Rom-62*
 - system calls, Rom-111-Rom-142*
 - tracing features, Rom-16-Rom-142*
 - use of interrupts and traps, Rom-10*
- so command, Rom-105
- sound support, C-1
- source directory, change, 13-176
- source level debugging, 1-8
- source line, jump to, 5-17
- source merge limit, 3-10
- source positioning, 5-3
- Source Window, 4-23
 - calling functions, 5-14*
 - change execution position, 5-5*
 - change viewing position, 5-4*
 - controlling program execution, 5-8-5-18*
 - edit source, 4-25*
 - searching in, 5-15-5-18*
 - single stepping, 5-9*
 - sync execution and viewing positions, 5-7*
 - toolbar, 4-24*
- source window, line numbers, 3-10
- SourceFileChanged, A-10
- special function register, 3-11
- special function registers, B-3
- special variables, 3-9, 15-10
 - reserved, 15-9*
 - user-defined, 3-11*
- sr command, Rom-106
- ss command, Rom-107
- st command, 13-167

stack, 6-19
 local variables, 3-7
 organization of, 6-19
 stack pointer, 3-10
 stack trace, 13-168, 13-169
 Stack Window, 4-32, 6-20
 toolbar, 4-33
 start execution, Rom-85, Bdm-21
 startup options, 4-5
 definition, 15-11
 list of, 4-7
 static variables, 3-7
 status bar, 4-11
 step out of address range, Rom-105,
 Bdm-29
 step-out-of-range, Rom-17
 stop target execution, 13-167
 Stopped, A-9
 storage classes, 3-7
 string command, 3-18
 strings, 3-6
 structures
 assignment, 6-8
 viewing, 6-5
 style codes, 3-14
 symbol information, 15-11
 symbolic disassembly, 6-17
 symbols, in disassembly, 3-10
 synchronize execution and viewing
 positions, 5-7, 13-126
 sys_go, Rom-48
 sys_stop, Rom-49
 system calls, Rom-111-Rom-142
 EVT_COPY, Rom-112
 I/O, Rom-26
 IN_CHAR, Rom-113
 IN_STR, Rom-114
 INT_COMP, Rom-115
 INT_ENTER, Rom-116
 INT_RX, Rom-117
 INT_TX, Rom-118
 optional, Rom-13
 OUT_CHAR, Rom-119

OUT_DATA, Rom-120
 OUT_STR, Rom-121
 RD_STR, Rom-122
 required, Rom-13
 ROMM_GO, Rom-123
 system control, Rom-30
 system startup code, 15-11

T

T command, 13-168
 t command, 13-169
 Tab key, 4-21
 target communication, 15-11
 target configuration file, 1-20
 Target Interface Package (TIP), Rom-9,
 Rom-19, Rom-31-Rom-142
 assembling, Rom-51
 description, Rom-31-Rom-142
 initialize and download, Rom-5
 locating, Rom-60
 modules
 diag_tbl.68k, Rom-49-Rom-142
 io_drv.68k, Rom-44-Rom-142
 rmain.68k, Rom-37-Rom-142
 sys_go.68k, Rom-48-Rom-142
 sysstp.68k, Rom-48-Rom-142
 usreq.68k, Rom-32-Rom-142
 programming EPROMS, Rom-61
 required system calls, Rom-13
 target program counter, 13-74
 target settings, 1-19
 target state, 13-14
 target system, 1-8
 task selection, 13-99
 td command, 13-170, Rom-108
 te command, 13-171, Rom-109
 Terminal Window, 4-34
 keyboard mappings, 10-10
 setup, 4-35
 timer breakpoints, 7-3

title, 1-20
 toolbar, 4-11
 data analysis window, 4-36
 data window, 4-31
 memory window, 4-28
 source window, 4-24
 stack window, 4-33
 toolbox, 8-11
 toolchain, 1-12
 trace
 C, 13-81
 disable, 13-170
 disassembled, 13-82
 enable, 13-171
 instruction level, 6-24
 raw, 6-24, 13-83
 source level, 6-23
 trace analysis, 15-12
 trace buffer, 15-12
 trace buffer operation, Rom-17
 trace disable, Rom-108
 trace enable, Rom-109
 trace points, Rom-17
 Trace Window, 4-33, 6-23
 instruction level, 13-82
 raw, 13-83
 source level, 13-81
 traceback mode, 1-5
 transparency mode, 1-9, 11-3, 13-145
 and CrossView startup, 4-6
 defined, 15-12
 entering, 11-3
 one-shot commands, 11-3
 startup options, 11-3
 trigraph sequence, 3-7
 troubleshooting, 1-31, 4-9,
 Rom-59–Rom-142
 TX_CHAR, Rom-45, Rom-47

U

u command, 13-172
 ubgw command, 13-173
 ud command, Rom-110
 UIDPATH, 2-3
 unset command, 13-174
 update command, 13-175
 update windows, 13-172, 13-173
 updating makefile, 1-38
 use command, 13-176
 user defined functions, 1-7
 user diagnostics, Rom-110
 using EDE, 1-32

V

variables, 3-7
 and case sensitivity, 3-21
 and scoping rules, 3-9
 casting, 3-7
 changing, 6-7
 determining the size of, 6-7
 formats of, 6-13
 global, 6-8
 global variables, 3-8
 local, 15-7
 local variables, 3-7
 scope, 15-10
 special, 15-10
 special variables, Pages, 3-9
 specifying in expressions, 3-8
 static variables, 3-7
 user-defined special variables, 3-11
 ViewedLineNrChanged, A-10

viewing position, 3-9, 5-3
 changing the, 5-4-5-7
 defined, 15-12
 establish, 13-93
 establish at address, 13-98
 sync with execution position, 5-7

W

wait for target completion, 13-177
 waiting, 11-31
 window update
 reactivate, 13-114
 suppress, 13-114
 windows, 4-20
 active, 4-15, 15-3
 automatic switching between source and assembly, 3-10
 closing, 4-15
 command window, 4-21
 customizing, 4-17
 data analysis window, 4-36
 data window, 4-29
 help window, 4-37
 memory window, 4-27
 opening, 4-14
 pop-up, 4-37
 register window, 4-26

selecting, 4-15
 source positioning, 5-3
 source window, 4-23
 stack window, 4-32
 terminal windows, 4-34
 toolbox, 4-37
 trace window, 4-33
 workspace file (.cws), 1-30
 write to an I/O stream, 13-124
 formatted, 13-125
 wt command, 13-177

X

x command, 13-178
 X Resources, 2-4
 X Widgets, CrossView Motif, 2-4
 X Windows
 Motif environment, 2-3
 resources, 2-4
 x-t plotting, 11-13
 x-y plotting, 11-14
 xvwedit, 4-25

Z

Z command, 13-179